

# oneAPI

**oneAPI Specification**

*Release 0.85*

**Intel**

**Jul 28, 2020**

# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Target Audience . . . . .	2
1.2	Goals of the Specification . . . . .	2
1.3	Definitions . . . . .	2
1.4	Contribution Guidelines . . . . .	2
1.4.1	Sign your work . . . . .	2
<b>2</b>	<b>Software Architecture</b>	<b>3</b>
2.1	oneAPI Platform . . . . .	4
2.2	API Programming Example . . . . .	5
2.3	Direct Programming Example . . . . .	6
<b>3</b>	<b>Library Interoperability</b>	<b>7</b>
3.1	Queueing . . . . .	7
3.2	API Arguments . . . . .	7
3.3	Asynchronous APIs . . . . .	7
3.4	Exceptions . . . . .	8
<b>4</b>	<b>oneAPI Elements</b>	<b>9</b>
<b>5</b>	<b>DPC++</b>	<b>10</b>
5.1	Overview . . . . .	10
5.2	Detailed API and Language Descriptions . . . . .	11
5.3	Open Source Implementation . . . . .	12
5.4	Testing . . . . .	12
<b>6</b>	<b>oneDPL</b>	<b>13</b>
6.1	Namespaces . . . . .	13
6.2	Supported C++ Standard Library APIs and Algorithms . . . . .	13
6.2.1	Extensions to Parallel STL . . . . .	13
6.2.1.1	DPC++ Execution Policy . . . . .	14
6.2.1.2	Wrappers for SYCL Buffers . . . . .	15
6.2.2	Specific API of oneDPL . . . . .	15
<b>7</b>	<b>oneDNN</b>	<b>18</b>
7.1	Introduction . . . . .	19
7.1.1	General API notes . . . . .	20
7.1.2	Error Handling . . . . .	20
7.2	Conventions . . . . .	21
7.2.1	Variable (Tensor) Names . . . . .	21
7.2.2	RNN-Specific Notation . . . . .	22

7.3	Execution Model . . . . .	22
7.3.1	Engine . . . . .	23
7.3.2	Stream . . . . .	24
7.4	Data model . . . . .	26
7.4.1	Data types . . . . .	26
7.4.1.1	Bfloat16 . . . . .	27
7.4.1.1.1	Workflow . . . . .	27
7.4.1.1.2	Support . . . . .	28
7.4.1.2	Int8 . . . . .	28
7.4.1.2.1	Workflow . . . . .	28
7.4.1.2.2	Support . . . . .	28
7.4.2	Memory . . . . .	28
7.4.2.1	Memory Formats . . . . .	28
7.4.2.1.1	Plain Memory Formats . . . . .	29
7.4.2.1.2	Format Tags . . . . .	30
7.4.2.1.3	Optimized Format ‘any’ . . . . .	30
7.4.2.1.4	Memory Format Propagation . . . . .	31
7.4.2.1.5	API . . . . .	31
7.4.2.2	Memory Descriptors and Objects . . . . .	36
7.4.2.2.1	Descriptors . . . . .	36
7.4.2.2.2	Objects . . . . .	39
7.4.2.2.3	API . . . . .	39
7.5	Primitives . . . . .	42
7.5.1	Common Definitions . . . . .	44
7.5.1.1	Base Class for Primitives . . . . .	44
7.5.1.2	Base Class for Primitives Descriptors . . . . .	46
7.5.1.3	Common Enumerations . . . . .	52
7.5.1.4	Normalization Primitives Flags . . . . .	55
7.5.1.5	Execution argument indices . . . . .	55
7.5.2	Attributes . . . . .	58
7.5.2.1	Post-ops . . . . .	59
7.5.2.1.1	Supported Post-ops . . . . .	59
7.5.2.1.1.1	Eltwise Post-op . . . . .	59
7.5.2.1.1.2	Sum Post-op . . . . .	60
7.5.2.1.1.3	Examples of Chained Post-ops . . . . .	60
7.5.2.1.1.4	Sum -> ReLU . . . . .	60
7.5.2.1.2	API . . . . .	61
7.5.2.2	Scratchpad Mode . . . . .	62
7.5.2.2.1	Examples . . . . .	63
7.5.2.2.1.1	Library Manages Scratchpad . . . . .	63
7.5.2.2.1.2	User Manages Scratchpad . . . . .	64
7.5.2.3	Quantization . . . . .	64
7.5.2.3.1	Quantization Model . . . . .	64
7.5.2.3.1.1	Example: Convolution Quantization Workflow . . . . .	65
7.5.2.3.1.2	Per-Channel Scaling . . . . .	65
7.5.2.3.2	Output Scaling Attribute . . . . .	66
7.5.2.3.2.1	Example 1: weights quantization with per-output-channel-and-group scaling . . . . .	66
7.5.2.3.2.2	Example 2: convolution with groups, with per-output-channel quantization . . . . .	67
7.5.2.3.2.3	Interplay of Output Scales with Post-ops . . . . .	68
7.5.2.4	Attribute Related Error Handling . . . . .	69
7.5.2.5	API . . . . .	69
7.5.3	Batch Normalization . . . . .	72

7.5.3.1	Forward . . . . .	73
7.5.3.1.1	Difference Between Forward Training and Forward Inference . . . . .	73
7.5.3.2	Backward . . . . .	74
7.5.3.3	Execution Arguments . . . . .	74
7.5.3.4	Operation Details . . . . .	75
7.5.3.5	Data Types Support . . . . .	75
7.5.3.6	Data Representation . . . . .	75
7.5.3.6.1	Source, Destination, and Their Gradients . . . . .	75
7.5.3.6.2	Statistics Tensors . . . . .	76
7.5.3.7	Post-ops and Attributes . . . . .	76
7.5.3.8	API . . . . .	76
7.5.4	Binary . . . . .	80
7.5.4.1	Forward and Backward . . . . .	80
7.5.4.2	Execution Arguments . . . . .	80
7.5.4.3	Operation Details . . . . .	80
7.5.4.4	Post-ops and Attributes . . . . .	80
7.5.4.5	Data Types Support . . . . .	81
7.5.4.6	Data Representation . . . . .	81
7.5.4.7	API . . . . .	81
7.5.5	Concat . . . . .	82
7.5.5.1	Forward and Backward . . . . .	83
7.5.5.2	Execution Arguments . . . . .	83
7.5.5.3	Operation Details . . . . .	83
7.5.5.4	Data Types Support . . . . .	83
7.5.5.5	Data Representation . . . . .	83
7.5.5.6	Post-ops and Attributes . . . . .	83
7.5.5.7	API . . . . .	83
7.5.6	Convolution and Deconvolution . . . . .	85
7.5.6.1	Forward . . . . .	85
7.5.6.1.1	Regular Convolution . . . . .	85
7.5.6.1.2	Convolution with Groups . . . . .	86
7.5.6.1.3	Convolution with Dilation . . . . .	86
7.5.6.1.4	Deconvolution (Transposed Convolution) . . . . .	86
7.5.6.1.5	Difference Between Forward Training and Forward Inference . . . . .	86
7.5.6.2	Backward . . . . .	87
7.5.6.3	Execution Arguments . . . . .	87
7.5.6.4	Operation Details . . . . .	87
7.5.6.5	Data Types Support . . . . .	87
7.5.6.6	Data Representation . . . . .	88
7.5.6.7	Post-ops and Attributes . . . . .	89
7.5.6.7.1	Example 1 . . . . .	89
7.5.6.7.2	Example 2 . . . . .	90
7.5.6.8	Algorithms . . . . .	90
7.5.6.9	API . . . . .	90
7.5.7	Elementwise . . . . .	108
7.5.7.1	Forward . . . . .	108
7.5.7.2	Backward . . . . .	109
7.5.7.3	Difference Between Forward Training and Forward Inference . . . . .	110
7.5.7.4	Execution Arguments . . . . .	110
7.5.7.5	Operation Details . . . . .	110
7.5.7.6	Data Type Support . . . . .	110
7.5.7.7	Data Representation . . . . .	111
7.5.7.8	Post-ops and Attributes . . . . .	111
7.5.7.9	API . . . . .	111

7.5.8	Inner Product . . . . .	114
7.5.8.1	Forward . . . . .	114
7.5.8.1.1	Difference Between Forward Training and Forward Inference . . . . .	114
7.5.8.2	Backward . . . . .	114
7.5.8.2.1	Execution Arguments . . . . .	114
7.5.8.3	Operation Details . . . . .	115
7.5.8.4	Data Types Support . . . . .	115
7.5.8.5	Data Representation . . . . .	115
7.5.8.6	Post-ops and Attributes . . . . .	116
7.5.8.7	API . . . . .	116
7.5.9	Layer normalization . . . . .	121
7.5.9.1	Forward . . . . .	122
7.5.9.1.1	Difference Between Forward Training and Forward Inference . . . . .	122
7.5.9.2	Backward . . . . .	122
7.5.9.3	Execution Arguments . . . . .	122
7.5.9.4	Operation Details . . . . .	123
7.5.9.5	Data Types Support . . . . .	123
7.5.9.6	Data Representation . . . . .	124
7.5.9.6.1	Mean and Variance . . . . .	124
7.5.9.6.2	Scale and Shift . . . . .	124
7.5.9.6.3	Source, Destination, and Their Gradients . . . . .	124
7.5.9.7	API . . . . .	124
7.5.10	LogSoftmax . . . . .	128
7.5.10.1	Forward . . . . .	128
7.5.10.1.1	Difference Between Forward Training and Forward Inference . . . . .	129
7.5.10.2	Backward . . . . .	129
7.5.10.3	Execution Arguments . . . . .	129
7.5.10.4	Operation Details . . . . .	129
7.5.10.5	Post-ops and Attributes . . . . .	129
7.5.10.6	Data Type Support . . . . .	129
7.5.10.7	Data Representation . . . . .	130
7.5.10.7.1	Source, Destination, and Their Gradients . . . . .	130
7.5.10.8	API . . . . .	130
7.5.11	Local Response Normalization . . . . .	133
7.5.11.1	Forward . . . . .	133
7.5.11.2	Backward . . . . .	133
7.5.11.2.1	Execution Arguments . . . . .	133
7.5.11.2.1.1	Operation Details . . . . .	133
7.5.11.2.1.2	Data Type Support . . . . .	134
7.5.11.2.1.3	Data Representation . . . . .	134
7.5.11.2.2	Source, Destination, and Their Gradients . . . . .	134
7.5.11.2.2.1	Post-ops and Attributes . . . . .	134
7.5.11.2.2.2	API . . . . .	134
7.5.12	Matrix Multiplication . . . . .	138
7.5.12.1	Execution Arguments . . . . .	138
7.5.12.2	Operation Details . . . . .	138
7.5.12.3	Data Types Support . . . . .	138
7.5.12.4	Data Representation . . . . .	139
7.5.12.5	Attributes and Post-ops . . . . .	139
7.5.12.6	API . . . . .	140
7.5.13	Pooling . . . . .	141
7.5.13.1	Forward . . . . .	142
7.5.13.1.1	Difference Between Forward Training and Forward Inference . . . . .	142
7.5.13.2	Backward . . . . .	142

7.5.13.3	Execution Arguments . . . . .	142
7.5.13.4	Operation Details . . . . .	142
7.5.13.5	Data Type Support . . . . .	143
7.5.13.6	Data Representation . . . . .	143
7.5.13.6.1	Source, Destination, and Their Gradients . . . . .	143
7.5.13.7	Post-ops and Attributes . . . . .	143
7.5.13.8	API . . . . .	143
7.5.14	Reorder . . . . .	147
7.5.14.1	Execution Arguments . . . . .	147
7.5.14.2	Operation Details . . . . .	148
7.5.14.3	Data Types Support . . . . .	148
7.5.14.4	Data Representation . . . . .	148
7.5.14.5	Post-ops and Attributes . . . . .	148
7.5.14.6	API . . . . .	149
7.5.15	Resampling . . . . .	151
7.5.15.1	Forward . . . . .	151
7.5.15.1.1	Nearest Neighbor Resampling . . . . .	151
7.5.15.1.2	Bilinear Resampling . . . . .	152
7.5.15.1.3	Difference Between Forward Training and Forward Inference . . . . .	152
7.5.15.2	Backward . . . . .	152
7.5.15.3	Execution Arguments . . . . .	152
7.5.15.4	Operation Details . . . . .	153
7.5.15.5	Data Types Support . . . . .	153
7.5.15.6	Post-ops and Attributes . . . . .	153
7.5.15.7	API . . . . .	153
7.5.16	RNN . . . . .	157
7.5.16.1	Cell Functions . . . . .	158
7.5.16.1.1	Vanilla RNN . . . . .	158
7.5.16.1.2	LSTM . . . . .	158
7.5.16.1.2.1	LSTM (or Vanilla LSTM) . . . . .	158
7.5.16.1.2.2	LSTM with Peephole . . . . .	159
7.5.16.1.2.3	LSTM with Projection . . . . .	160
7.5.16.1.3	GRU . . . . .	160
7.5.16.1.4	Linear-Before-Reset GRU . . . . .	161
7.5.16.2	Execution Arguments . . . . .	161
7.5.16.3	Operation Details . . . . .	162
7.5.16.4	Data Types Support . . . . .	162
7.5.16.4.1	Data Representation . . . . .	163
7.5.16.4.2	Post-ops and Attributes . . . . .	163
7.5.16.5	API . . . . .	163
7.5.17	Shuffle . . . . .	189
7.5.17.1	Forward . . . . .	189
7.5.17.1.1	Difference Between Forward Training and Forward Inference . . . . .	189
7.5.17.2	Backward . . . . .	189
7.5.17.3	Execution Arguments . . . . .	189
7.5.17.4	Operation Details . . . . .	190
7.5.17.5	Data Types Support . . . . .	190
7.5.17.6	Data Layouts . . . . .	190
7.5.17.7	Post-ops and Attributes . . . . .	190
7.5.17.8	API . . . . .	190
7.5.18	Softmax . . . . .	193
7.5.18.1	Forward . . . . .	193
7.5.18.1.1	Difference Between Forward Training and Forward Inference . . . . .	193
7.5.18.2	Backward . . . . .	193

7.5.18.3	Execution Arguments . . . . .	193
7.5.18.4	Operation Details . . . . .	194
7.5.18.5	Post-ops and Attributes . . . . .	194
7.5.18.6	Data Types Support . . . . .	194
7.5.18.7	Data Representation . . . . .	194
7.5.18.7.1	Source, Destination, and Their Gradients . . . . .	194
7.5.18.8	API . . . . .	194
7.5.19	Sum . . . . .	197
7.5.19.1	Execution Arguments . . . . .	197
7.5.19.2	Operation Details . . . . .	197
7.5.19.3	Post-ops and Attributes . . . . .	198
7.5.19.4	Data Types Support . . . . .	198
7.5.19.5	Data Representation . . . . .	198
7.5.19.5.1	Sources, Destination . . . . .	198
7.5.19.6	API . . . . .	198
7.6	Open Source Implementation . . . . .	199
7.7	Implementation Notes . . . . .	199
7.8	Testing . . . . .	199
<b>8</b>	<b>oneCCL</b> . . . . .	<b>200</b>
8.1	Introduction . . . . .	200
8.2	Definitions . . . . .	200
8.2.1	oneCCL Concepts . . . . .	200
8.2.1.1	Environment . . . . .	201
8.2.1.2	Key-Value Store . . . . .	201
8.2.1.3	Communicator . . . . .	202
8.2.1.4	Device Communicator . . . . .	203
8.2.1.5	Request . . . . .	204
8.2.1.6	Event . . . . .	204
8.2.1.7	Stream . . . . .	205
8.2.1.8	Operation Attributes . . . . .	205
8.2.2	Communication Operations . . . . .	205
8.2.2.1	Datatypes . . . . .	205
8.2.2.1.1	Custom Datatypes . . . . .	206
8.2.2.2	Reductions . . . . .	207
8.2.2.3	Collective Operations . . . . .	207
8.2.2.3.1	Allgatherv . . . . .	208
8.2.2.3.2	Allreduce . . . . .	209
8.2.2.3.3	Alltoall . . . . .	210
8.2.2.3.4	Alltoally . . . . .	212
8.2.2.3.5	Barrier . . . . .	213
8.2.2.3.6	Broadcast . . . . .	213
8.2.2.3.7	Reduce . . . . .	214
8.2.2.3.8	ReduceScatter . . . . .	216
8.2.2.4	Operation Attributes . . . . .	217
8.2.2.5	Operation Progress Tracking . . . . .	219
8.2.2.5.1	Request . . . . .	219
8.2.3	Error Handling . . . . .	219
8.3	Programming Model . . . . .	220
8.3.1	Generic Workflow . . . . .	220
8.3.2	Host Communication Support . . . . .	221
8.3.2.1	Example . . . . .	221
8.3.3	Device Communication Support . . . . .	222
8.3.3.1	Example . . . . .	222

<b>9</b>	<b>Level Zero</b>	<b>225</b>
9.1	Detailed API Descriptions . . . . .	225
<b>10</b>	<b>oneDAL</b>	<b>226</b>
10.1	Introduction . . . . .	226
10.2	Glossary . . . . .	228
10.2.1	Machine learning terms . . . . .	228
10.2.2	oneDAL terms . . . . .	229
10.2.3	Common oneAPI terms . . . . .	230
10.3	Mathematical Notations . . . . .	231
10.4	Programming model . . . . .	231
10.4.1	Basic usage scenario . . . . .	231
10.4.2	Memory objects . . . . .	233
10.4.3	Algorithm Anatomy . . . . .	233
10.4.3.1	Descriptors . . . . .	233
10.4.3.1.1	Floating-point Types . . . . .	235
10.4.3.1.2	Computational Methods . . . . .	235
10.4.3.2	Operations . . . . .	235
10.4.3.2.1	Input . . . . .	235
10.4.3.2.2	Result . . . . .	235
10.4.4	Managing execution context . . . . .	235
10.4.5	Computational modes . . . . .	235
10.4.5.1	Batch . . . . .	235
10.4.5.2	Online . . . . .	236
10.4.5.3	Distributed . . . . .	236
10.5	Common Interface . . . . .	236
10.5.1	Header files . . . . .	236
10.5.2	Namespaces . . . . .	237
10.5.3	Managing object lifetimes . . . . .	237
10.5.4	Error handling . . . . .	237
10.5.4.1	Exception classification . . . . .	237
10.6	Data management . . . . .	239
10.6.1	Key concepts . . . . .	240
10.6.1.1	Dataset . . . . .	240
10.6.1.2	Data source . . . . .	241
10.6.1.3	Table . . . . .	241
10.6.1.4	Metadata . . . . .	242
10.6.1.5	Table builder . . . . .	242
10.6.1.6	Accessor . . . . .	242
10.6.1.7	Use-case example for table, accessor and table builder . . . . .	243
10.6.2	Details . . . . .	245
10.6.2.1	Data Sources . . . . .	245
10.6.2.2	Tables . . . . .	245
10.6.2.2.1	Requirements . . . . .	245
10.6.2.2.2	Table Types . . . . .	246
10.6.2.2.3	Table API . . . . .	246
10.6.2.2.3.1	Homogeneous table . . . . .	247
10.6.2.2.3.2	Structure-of-arrays table . . . . .	249
10.6.2.2.3.3	Arrays-of-structure table . . . . .	249
10.6.2.2.3.4	Compressed-sparse-row table . . . . .	249
10.6.2.2.4	Metadata API . . . . .	249
10.6.2.2.4.1	Data layout . . . . .	250
10.6.2.2.4.2	Data format . . . . .	250
10.6.2.2.4.3	Feature info . . . . .	251

10.6.2.2.4.4	Data type . . . . .	251
10.6.2.2.4.5	Feature type . . . . .	251
10.6.2.3	Table Builders . . . . .	252
10.6.2.3.1	Requirements . . . . .	252
10.6.2.3.2	Table Builder Types . . . . .	252
10.6.2.3.3	Simple Homogeneous Table Builder . . . . .	252
10.6.2.3.4	Simple SOA Table Builder . . . . .	252
10.6.2.4	Accessors . . . . .	252
10.6.2.4.1	Requirements . . . . .	252
10.6.2.4.2	Accessor Types . . . . .	253
10.6.2.4.2.1	Row accessor . . . . .	253
10.6.2.4.2.2	Column accessor . . . . .	253
10.7	Algorithms . . . . .	253
10.7.1	Clustering . . . . .	254
10.7.1.1	K-Means . . . . .	254
10.7.1.1.1	Lloyd's method . . . . .	254
10.7.1.1.2	Usage example . . . . .	255
10.7.1.1.3	API . . . . .	255
10.7.1.1.3.1	Methods . . . . .	255
10.7.1.1.3.2	Descriptor . . . . .	256
10.7.1.1.3.3	Model . . . . .	257
10.7.1.1.3.4	Training <code>onedal::train</code> . . . . .	257
10.7.1.1.3.5	Input . . . . .	257
10.7.1.1.3.6	Result . . . . .	258
10.7.1.1.3.7	Operation semantics . . . . .	259
10.7.1.1.3.8	Inference <code>onedal::infer</code> . . . . .	259
10.7.1.1.3.9	Input . . . . .	259
10.7.1.1.3.10	Result . . . . .	260
10.7.1.1.3.11	Operation semantics . . . . .	261
10.7.2	Nearest Neighbors (kNN) . . . . .	261
10.7.2.1	k-Nearest Neighbors Classification (k-NN) . . . . .	261
10.7.2.1.1	Mathematical formulation . . . . .	261
10.7.2.1.1.1	Training . . . . .	261
10.7.2.1.1.2	Training method: <i>brute-force</i> . . . . .	261
10.7.2.1.1.3	Training method: <i>k-d tree</i> . . . . .	262
10.7.2.1.1.4	Inference . . . . .	262
10.7.2.1.1.5	Inference method: <i>brute-force</i> . . . . .	262
10.7.2.1.1.6	Inference method: <i>k-d tree</i> . . . . .	262
10.7.2.1.2	Programming Interface . . . . .	262
10.7.2.1.2.1	Descriptor . . . . .	262
10.7.2.1.2.2	Computational methods . . . . .	263
10.7.2.1.2.3	Model . . . . .	264
10.7.2.1.2.4	Training <code>train</code> . . . . .	264
10.7.2.1.2.5	Input . . . . .	264
10.7.2.1.2.6	Result . . . . .	265
10.7.2.1.2.7	Operation . . . . .	265
10.7.2.1.2.8	Inference <code>infer</code> . . . . .	266
10.7.2.1.2.9	Input . . . . .	266
10.7.2.1.2.10	Result . . . . .	266
10.7.2.1.2.11	Operation . . . . .	267
10.7.3	Decomposition . . . . .	267
10.7.3.1	Principal Components Analysis (PCA) . . . . .	267
10.7.3.1.1	Covariance-based method . . . . .	268
10.7.3.1.2	SVD-based method . . . . .	268

10.7.3.1.3	Sign-flip technique . . . . .	268
10.7.3.1.4	Usage example . . . . .	268
10.7.3.1.5	API . . . . .	269
10.7.3.1.5.1	Methods . . . . .	269
10.7.3.1.5.2	Descriptor . . . . .	269
10.7.3.1.5.3	Model . . . . .	270
10.7.3.1.5.4	Training <code>onedal::train</code> . . . . .	271
10.7.3.1.5.5	Input . . . . .	271
10.7.3.1.5.6	Result . . . . .	271
10.7.3.1.5.7	Operation semantics . . . . .	272
10.7.3.1.5.8	Inference <code>onedal::infer</code> . . . . .	272
10.7.3.1.5.9	Input . . . . .	272
10.7.3.1.5.10	Result . . . . .	273
10.7.3.1.5.11	Operation semantics . . . . .	274
10.8	Appendix . . . . .	274
10.8.1	k-d Tree . . . . .	274
10.8.1.1	Related terms . . . . .	274
10.9	Bibliography . . . . .	274
<b>11</b>	<b>oneTBB</b>	<b>275</b>
11.1	General Information . . . . .	275
11.1.1	Introduction . . . . .	275
11.1.2	Notational Conventions . . . . .	275
11.1.3	Identifiers . . . . .	277
11.1.3.1	Case . . . . .	277
11.1.3.2	Reserved Identifier Prefixes . . . . .	277
11.1.4	Named Requirements . . . . .	277
11.1.4.1	Algorithms . . . . .	278
11.1.4.1.1	Range . . . . .	278
11.1.4.1.2	Splittable . . . . .	279
11.1.4.1.3	ParallelForBody . . . . .	280
11.1.4.1.4	ParallelReduceBody . . . . .	280
11.1.4.1.5	ParallelReduceFunc . . . . .	280
11.1.4.1.6	ParallelReduceReduction . . . . .	281
11.1.4.1.7	ParallelForEachBody . . . . .	281
11.1.4.1.7.1	ItemType . . . . .	282
11.1.4.1.8	ContainerBasedSequence . . . . .	282
11.1.4.1.9	ParallelScanBody . . . . .	282
11.1.4.1.10	ParallelScanCombine . . . . .	283
11.1.4.1.11	ParallelScanFunc . . . . .	283
11.1.4.1.12	BlockedRangeValue . . . . .	283
11.1.4.1.13	FilterBody . . . . .	284
11.1.4.2	Mutexes . . . . .	284
11.1.4.2.1	Mutex . . . . .	284
11.1.4.2.2	ReaderWriterMutex . . . . .	286
11.1.4.3	Containers . . . . .	288
11.1.4.3.1	HashCompare . . . . .	288
11.1.4.3.2	ContainerRange . . . . .	289
11.1.4.4	Task scheduler . . . . .	289
11.1.4.4.1	SuspendFunc . . . . .	289
11.1.4.5	Flow Graph . . . . .	290
11.1.4.5.1	AsyncNodeBody . . . . .	290
11.1.4.5.2	ContinueNodeBody . . . . .	290
11.1.4.5.3	GatewayType . . . . .	291

11.1.4.5.4	FunctionNodeBody . . . . .	291
11.1.4.5.5	JoinNodeFunctionObject . . . . .	291
11.1.4.5.6	InputNodeBody . . . . .	292
11.1.4.5.7	MultifunctionNodeBody . . . . .	292
11.1.4.5.8	Sequencer . . . . .	293
11.1.5	Thread Safety . . . . .	293
11.2	oneTBB Interfaces . . . . .	293
11.2.1	Configuration . . . . .	293
11.2.1.1	Namespaces . . . . .	294
11.2.1.1.1	tbb Namespace . . . . .	294
11.2.1.1.2	tbb::flow Namespace . . . . .	294
11.2.1.2	Version Information . . . . .	294
11.2.1.2.1	TBB_runtime_interface_version Function . . . . .	295
11.2.1.2.2	TBB_runtime_version Function . . . . .	295
11.2.1.2.3	TBB_VERSION Environment Variable . . . . .	295
11.2.1.3	Enabling Debugging Features . . . . .	295
11.2.1.3.1	TBB_USE_ASSERT Macro . . . . .	296
11.2.1.3.2	TBB_USE_PROFILING_TOOLS Macro . . . . .	296
11.2.1.4	Feature Macros . . . . .	296
11.2.1.4.1	TBB_USE_EXCEPTIONS macro . . . . .	296
11.2.1.4.2	TBB_USE_GLIBCXX_VERSION macro . . . . .	296
11.2.2	Algorithms . . . . .	296
11.2.2.1	Parallel Functions . . . . .	297
11.2.2.1.1	parallel_for . . . . .	297
11.2.2.1.2	parallel_reduce . . . . .	298
11.2.2.1.2.1	Example (Imperative Form) . . . . .	300
11.2.2.1.2.2	Example with Lambda Expressions . . . . .	301
11.2.2.1.3	parallel_deterministic_reduce . . . . .	301
11.2.2.1.4	parallel_scan . . . . .	303
11.2.2.1.4.1	pre_scan and final_scan Classes . . . . .	304
11.2.2.1.4.2	pre_scan_tag and final_scan_tag . . . . .	304
11.2.2.1.4.3	Member functions . . . . .	304
11.2.2.1.4.4	Example (Imperative Form) . . . . .	305
11.2.2.1.4.5	Example with Lambda Expressions . . . . .	306
11.2.2.1.5	parallel_for_each . . . . .	306
11.2.2.1.5.1	feeder Class . . . . .	307
11.2.2.1.5.2	feeder . . . . .	307
11.2.2.1.5.3	Member functions . . . . .	307
11.2.2.1.5.4	Example . . . . .	308
11.2.2.1.6	parallel_invoke . . . . .	308
11.2.2.1.6.1	Example . . . . .	308
11.2.2.1.7	parallel_pipeline . . . . .	309
11.2.2.1.7.1	Example . . . . .	310
11.2.2.1.7.2	filter Class Template . . . . .	310
11.2.2.1.7.3	filter . . . . .	310
11.2.2.1.7.4	filter_mode Enumeration . . . . .	311
11.2.2.1.7.5	filter_mode . . . . .	311
11.2.2.1.7.6	Member functions . . . . .	312
11.2.2.1.7.7	Non-member functions . . . . .	312
11.2.2.1.7.8	Deduction Guides . . . . .	312
11.2.2.1.7.9	flow_control Class . . . . .	312
11.2.2.1.7.10	flow_control . . . . .	312
11.2.2.1.7.11	Member functions . . . . .	313
11.2.2.1.8	parallel_sort . . . . .	313

11.2.2.2	Blocked Ranges . . . . .	314
11.2.2.2.1	blocked_range . . . . .	314
11.2.2.2.1.1	Member functions . . . . .	315
11.2.2.2.2	blocked_range2d . . . . .	316
11.2.2.2.2.1	Member types . . . . .	317
11.2.2.2.2.2	Member functions . . . . .	317
11.2.2.2.3	blocked_range3d . . . . .	319
11.2.2.2.3.1	Member types . . . . .	319
11.2.2.2.3.2	Member functions . . . . .	320
11.2.2.3	Partitioners . . . . .	321
11.2.2.3.1	auto_partitioner . . . . .	321
11.2.2.3.2	affinity_partitioner . . . . .	322
11.2.2.3.3	static_partitioner . . . . .	322
11.2.2.3.4	simple_partitioner . . . . .	323
11.2.2.4	Split Tags . . . . .	324
11.2.2.4.1	proportional split . . . . .	324
11.2.2.4.1.1	Member functions . . . . .	325
11.2.2.4.2	split . . . . .	325
11.2.3	Flow Graph . . . . .	325
11.2.3.1	Graph Class . . . . .	326
11.2.3.1.1	graph . . . . .	326
11.2.3.1.1.1	reset_flags enumeration . . . . .	326
11.2.3.1.1.2	reset_flags Enumeration . . . . .	326
11.2.3.1.1.3	Member functions . . . . .	327
11.2.3.2	Nodes . . . . .	327
11.2.3.2.1	Abstract Interfaces . . . . .	327
11.2.3.2.1.1	graph_node . . . . .	328
11.2.3.2.1.2	sender . . . . .	328
11.2.3.2.1.3	receiver . . . . .	328
11.2.3.2.2	Properties . . . . .	329
11.2.3.2.2.1	Forwarding and Buffering . . . . .	329
11.2.3.2.2.2	Forwarding . . . . .	329
11.2.3.2.2.3	Buffering . . . . .	329
11.2.3.2.3	Functional Nodes . . . . .	330
11.2.3.2.3.1	continue_node . . . . .	330
11.2.3.2.3.2	Member functions . . . . .	331
11.2.3.2.3.3	Deduction Guides . . . . .	333
11.2.3.2.3.4	Example . . . . .	333
11.2.3.2.3.5	function_node . . . . .	333
11.2.3.2.3.6	Member functions . . . . .	334
11.2.3.2.3.7	Deduction Guides . . . . .	335
11.2.3.2.3.8	Example . . . . .	335
11.2.3.2.3.9	input_node . . . . .	335
11.2.3.2.3.10	Member functions . . . . .	336
11.2.3.2.3.11	Deduction Guides . . . . .	337
11.2.3.2.3.12	multifunction_node . . . . .	337
11.2.3.2.3.13	Member types . . . . .	338
11.2.3.2.3.14	Member functions . . . . .	338
11.2.3.2.3.15	async_node . . . . .	339
11.2.3.2.3.16	Member functions . . . . .	340
11.2.3.2.3.17	Member functions . . . . .	340
11.2.3.2.3.18	Example . . . . .	341
11.2.3.2.3.19	Function Nodes Policies . . . . .	342
11.2.3.2.3.20	Queueing . . . . .	343

11.2.3.2.3.21	Rejecting . . . . .	343
11.2.3.2.3.22	Lightweight . . . . .	343
11.2.3.2.3.23	Example . . . . .	343
11.2.3.2.3.24	Nodes Priorities . . . . .	344
11.2.3.2.3.25	Example . . . . .	344
11.2.3.2.3.26	Predefined Concurrency Limits . . . . .	346
11.2.3.2.3.27	copy_body . . . . .	347
11.2.3.2.4	Buffering Nodes . . . . .	347
11.2.3.2.4.1	overwrite_node . . . . .	347
11.2.3.2.4.2	Member functions . . . . .	348
11.2.3.2.4.3	Examples . . . . .	348
11.2.3.2.4.4	write_once_node . . . . .	349
11.2.3.2.4.5	Member functions . . . . .	350
11.2.3.2.4.6	Example . . . . .	351
11.2.3.2.4.7	buffer_node . . . . .	352
11.2.3.2.4.8	Member functions . . . . .	352
11.2.3.2.4.9	queue_node . . . . .	353
11.2.3.2.4.10	Member functions . . . . .	353
11.2.3.2.4.11	Example . . . . .	354
11.2.3.2.4.12	priority_queue_node . . . . .	354
11.2.3.2.4.13	Member functions . . . . .	354
11.2.3.2.4.14	Example . . . . .	355
11.2.3.2.4.15	sequencer_node . . . . .	355
11.2.3.2.4.16	Member functions . . . . .	356
11.2.3.2.4.17	Deduction Guides . . . . .	356
11.2.3.2.4.18	Example . . . . .	356
11.2.3.2.5	Service Nodes . . . . .	357
11.2.3.2.5.1	limiter_node . . . . .	357
11.2.3.2.5.2	Member functions . . . . .	358
11.2.3.2.5.3	broadcast_node . . . . .	359
11.2.3.2.5.4	Member functions . . . . .	359
11.2.3.2.5.5	join_node . . . . .	360
11.2.3.2.5.6	join_node Policies . . . . .	361
11.2.3.2.5.7	Member types . . . . .	362
11.2.3.2.5.8	Member functions . . . . .	362
11.2.3.2.5.9	Non-Member Types . . . . .	363
11.2.3.2.5.10	Deduction Guides . . . . .	363
11.2.3.2.5.11	split_node . . . . .	364
11.2.3.2.5.12	Member functions . . . . .	364
11.2.3.2.5.13	indexer_node . . . . .	365
11.2.3.2.5.14	Member types . . . . .	366
11.2.3.2.5.15	Member functions . . . . .	366
11.2.3.2.5.16	composite_node . . . . .	366
11.2.3.2.5.17	Member functions . . . . .	368
11.2.3.3	Ports and Edges . . . . .	369
11.2.3.3.1	input_port . . . . .	369
11.2.3.3.2	output_port . . . . .	369
11.2.3.3.3	make_edge . . . . .	370
11.2.3.3.4	remove_edge . . . . .	370
11.2.3.4	Special Messages Types . . . . .	371
11.2.3.4.1	continue_msg . . . . .	371
11.2.3.4.2	tagged_msg . . . . .	372
11.2.3.4.2.1	Member functions . . . . .	372
11.2.3.4.2.2	Non-member functions . . . . .	373

11.2.3.5 Examples . . . . .	373
11.2.3.5.1 Dependency Flow Graph Example . . . . .	373
11.2.3.5.2 Message Flow Graph Example . . . . .	376
11.2.4 Task Scheduler . . . . .	377
11.2.4.1 Scheduling controls . . . . .	378
11.2.4.1.1 task_group_context . . . . .	378
11.2.4.1.1.1 Member types and constants . . . . .	378
11.2.4.1.1.2 Member functions . . . . .	379
11.2.4.1.2 global_control . . . . .	379
11.2.4.1.2.1 Member types and constants . . . . .	380
11.2.4.1.2.2 Member functions . . . . .	380
11.2.4.1.3 Resumable tasks . . . . .	381
11.2.4.1.3.1 Example . . . . .	381
11.2.4.2 Task Group . . . . .	382
11.2.4.2.1 task_group . . . . .	382
11.2.4.2.1.1 Member functions . . . . .	382
11.2.4.2.1.2 Non-member functions . . . . .	383
11.2.4.2.2 task_group_status . . . . .	383
11.2.4.2.2.1 Member constants . . . . .	383
11.2.4.3 Task Arena . . . . .	383
11.2.4.3.1 task_arena . . . . .	383
11.2.4.3.1.1 Member types and constants . . . . .	384
11.2.4.3.1.2 Member functions . . . . .	385
11.2.4.3.2 this_task_arena . . . . .	386
11.2.4.3.3 task_scheduler_observer . . . . .	387
11.2.4.3.3.1 Member functions . . . . .	388
11.2.4.3.3.2 Example . . . . .	389
11.2.5 Containers . . . . .	389
11.2.5.1 Sequences . . . . .	389
11.2.5.1.1 concurrent_vector . . . . .	389
11.2.5.1.1.1 Class Template Synopsis . . . . .	390
11.2.5.1.1.2 Requirements . . . . .	392
11.2.5.1.1.3 Description . . . . .	393
11.2.5.1.1.4 Exception Safety . . . . .	393
11.2.5.1.1.5 Member functions . . . . .	393
11.2.5.1.1.6 Construction, destruction, copying . . . . .	393
11.2.5.1.1.7 Empty container constructors . . . . .	393
11.2.5.1.1.8 Constructors from the sequence of elements . . . . .	394
11.2.5.1.1.9 Copying constructors . . . . .	394
11.2.5.1.1.10 Moving constructors . . . . .	395
11.2.5.1.1.11 Destructor . . . . .	395
11.2.5.1.1.12 Assignment operators . . . . .	395
11.2.5.1.1.13 assign . . . . .	396
11.2.5.1.1.14 get_allocator . . . . .	396
11.2.5.1.1.15 Concurrent growth . . . . .	396
11.2.5.1.1.16 grow_by . . . . .	397
11.2.5.1.1.17 grow_to_at_least . . . . .	397
11.2.5.1.1.18 push_back . . . . .	398
11.2.5.1.1.19 emplace_back . . . . .	398
11.2.5.1.1.20 Element access . . . . .	398
11.2.5.1.1.21 Access by index . . . . .	399
11.2.5.1.1.22 Access the first and the last element . . . . .	399
11.2.5.1.1.23 Iterators . . . . .	399
11.2.5.1.1.24 begin and cbegin . . . . .	399

11.2.5.1.1.25	end and cend . . . . .	400
11.2.5.1.1.26	rbegin and crbegin . . . . .	400
11.2.5.1.1.27	rend and crend . . . . .	400
11.2.5.1.1.28	Size and capacity . . . . .	400
11.2.5.1.1.29	size . . . . .	400
11.2.5.1.1.30	empty . . . . .	400
11.2.5.1.1.31	max_size . . . . .	401
11.2.5.1.1.32	capacity . . . . .	401
11.2.5.1.1.33	Concurrently unsafe operations . . . . .	401
11.2.5.1.1.34	Reserving . . . . .	401
11.2.5.1.1.35	Resizing . . . . .	401
11.2.5.1.1.36	shrink_to_fit . . . . .	401
11.2.5.1.1.37	clear . . . . .	402
11.2.5.1.1.38	swap . . . . .	402
11.2.5.1.1.39	Parallel iteration . . . . .	402
11.2.5.1.1.40	range member function . . . . .	402
11.2.5.1.1.41	Non-member functions . . . . .	402
11.2.5.1.1.42	Non-member binary comparisons . . . . .	403
11.2.5.1.1.43	Non-member lexicographical comparisons . . . . .	404
11.2.5.1.1.44	Non-member swap . . . . .	404
11.2.5.1.1.45	Other . . . . .	404
11.2.5.1.1.46	Deduction guides . . . . .	404
11.2.5.2	Queues . . . . .	405
11.2.5.2.1	concurrent_queue . . . . .	405
11.2.5.2.1.1	Class Template Synopsis . . . . .	405
11.2.5.2.1.2	Member functions . . . . .	406
11.2.5.2.1.3	Construction, destruction, copying . . . . .	406
11.2.5.2.1.4	Empty container constructors . . . . .	406
11.2.5.2.1.5	Constructor from the sequence of elements . . . . .	407
11.2.5.2.1.6	Copying constructors . . . . .	407
11.2.5.2.1.7	Moving constructors . . . . .	407
11.2.5.2.1.8	Destructor . . . . .	408
11.2.5.2.1.9	Concurrently safe member functions . . . . .	408
11.2.5.2.1.10	Pushing elements . . . . .	408
11.2.5.2.1.11	Popping elements . . . . .	408
11.2.5.2.1.12	get_allocator . . . . .	409
11.2.5.2.1.13	Concurrently unsafe member functions . . . . .	409
11.2.5.2.1.14	The number of elements . . . . .	409
11.2.5.2.1.15	clear . . . . .	409
11.2.5.2.1.16	Iterators . . . . .	409
11.2.5.2.1.17	unsafe_begin and unsafe_cbegin . . . . .	409
11.2.5.2.1.18	unsafe_end and unsafe_cend . . . . .	410
11.2.5.2.1.19	Other . . . . .	410
11.2.5.2.1.20	Deduction guides . . . . .	410
11.2.5.2.2	concurrent_bounded_queue . . . . .	411
11.2.5.2.2.1	Class Template Synopsis . . . . .	411
11.2.5.2.2.2	Member functions . . . . .	412
11.2.5.2.2.3	Construction, destruction, copying . . . . .	412
11.2.5.2.2.4	Empty container constructors . . . . .	412
11.2.5.2.2.5	Constructor from the sequence of elements . . . . .	413
11.2.5.2.2.6	Copying constructors . . . . .	413
11.2.5.2.2.7	Moving constructors . . . . .	413
11.2.5.2.2.8	Destructor . . . . .	413
11.2.5.2.2.9	Concurrently safe member functions . . . . .	414

11.2.5.2.2.10	Pushing elements . . . . .	414
11.2.5.2.2.11	Popping elements . . . . .	415
11.2.5.2.2.12	abort . . . . .	415
11.2.5.2.2.13	Capacity of the queue . . . . .	415
11.2.5.2.2.14	get_allocator . . . . .	416
11.2.5.2.2.15	Concurrently unsafe member functions . . . . .	416
11.2.5.2.2.16	The number of elements . . . . .	416
11.2.5.2.2.17	clear . . . . .	416
11.2.5.2.2.18	Iterators . . . . .	416
11.2.5.2.2.19	unsafe_begin and unsafe_cbegin . . . . .	416
11.2.5.2.2.20	unsafe_end and unsafe_cend . . . . .	417
11.2.5.2.2.21	Other . . . . .	417
11.2.5.2.2.22	Deduction guides . . . . .	417
11.2.5.2.3	concurrent_priority_queue . . . . .	418
11.2.5.2.3.1	Class Template Synopsis . . . . .	418
11.2.5.2.3.2	Member functions . . . . .	420
11.2.5.2.3.3	Construction, destruction, copying . . . . .	420
11.2.5.2.3.4	Empty container constructors . . . . .	420
11.2.5.2.3.5	Constructors from the sequence of elements . . . . .	420
11.2.5.2.3.6	Copying constructors . . . . .	421
11.2.5.2.3.7	Moving constructors . . . . .	421
11.2.5.2.3.8	Destructor . . . . .	421
11.2.5.2.3.9	Assignment operators . . . . .	421
11.2.5.2.3.10	assign . . . . .	422
11.2.5.2.3.11	Size and capacity . . . . .	422
11.2.5.2.3.12	empty . . . . .	422
11.2.5.2.3.13	size . . . . .	423
11.2.5.2.3.14	Concurrently safe modifiers . . . . .	423
11.2.5.2.3.15	Pushing elements . . . . .	423
11.2.5.2.3.16	Popping elements . . . . .	423
11.2.5.2.3.17	Concurrently unsafe modifiers . . . . .	424
11.2.5.2.3.18	clear . . . . .	424
11.2.5.2.3.19	swap . . . . .	424
11.2.5.2.3.20	Non-member functions . . . . .	424
11.2.5.2.3.21	Non-member swap . . . . .	425
11.2.5.2.3.22	Non-member binary comparisons . . . . .	425
11.2.5.2.3.23	Other . . . . .	425
11.2.5.2.3.24	Deduction guides . . . . .	425
11.2.5.3	Unordered associative containers . . . . .	426
11.2.5.3.1	concurrent_hash_map . . . . .	426
11.2.5.3.1.1	Class Template Synopsis . . . . .	427
11.2.5.3.1.2	Member classes . . . . .	430
11.2.5.3.1.3	accessor and const_accessor . . . . .	430
11.2.5.3.1.4	accessor member class . . . . .	430
11.2.5.3.1.5	const_accessor member class . . . . .	430
11.2.5.3.1.6	Member functions . . . . .	431
11.2.5.3.1.7	Construction and destruction . . . . .	431
11.2.5.3.1.8	Emptiness . . . . .	431
11.2.5.3.1.9	Key-value pair access . . . . .	431
11.2.5.3.1.10	Releasing . . . . .	432
11.2.5.3.1.11	Member functions . . . . .	432
11.2.5.3.1.12	Construction, destruction, copying . . . . .	432
11.2.5.3.1.13	Empty container constructors . . . . .	432
11.2.5.3.1.14	Constructors from the sequence of elements . . . . .	432

11.2.5.3.1.15	Copying constructors . . . . .	433
11.2.5.3.1.16	Moving constructors . . . . .	433
11.2.5.3.1.17	Destructor . . . . .	434
11.2.5.3.1.18	Assignment operators . . . . .	434
11.2.5.3.1.19	get_allocator . . . . .	434
11.2.5.3.1.20	Concurrently unsafe modifiers . . . . .	435
11.2.5.3.1.21	clear . . . . .	435
11.2.5.3.1.22	swap . . . . .	435
11.2.5.3.1.23	Hash policy . . . . .	435
11.2.5.3.1.24	Rehashing . . . . .	435
11.2.5.3.1.25	bucket_count . . . . .	435
11.2.5.3.1.26	Size and capacity . . . . .	435
11.2.5.3.1.27	empty . . . . .	435
11.2.5.3.1.28	size . . . . .	436
11.2.5.3.1.29	max_size . . . . .	436
11.2.5.3.1.30	Lookup . . . . .	436
11.2.5.3.1.31	find . . . . .	436
11.2.5.3.1.32	count . . . . .	436
11.2.5.3.1.33	Concurrently safe modifiers . . . . .	436
11.2.5.3.1.34	Inserting values . . . . .	437
11.2.5.3.1.35	Inserting sequences of elements . . . . .	438
11.2.5.3.1.36	Emplacing elements . . . . .	438
11.2.5.3.1.37	Erasing elements . . . . .	439
11.2.5.3.1.38	Iterators . . . . .	439
11.2.5.3.1.39	begin and cbegin . . . . .	439
11.2.5.3.1.40	end and cend . . . . .	440
11.2.5.3.1.41	equal_range . . . . .	440
11.2.5.3.1.42	Parallel iteration . . . . .	440
11.2.5.3.1.43	range member function . . . . .	440
11.2.5.3.1.44	Non member functions . . . . .	440
11.2.5.3.1.45	Non-member swap . . . . .	441
11.2.5.3.1.46	Non-member binary comparisons . . . . .	441
11.2.5.3.1.47	Other . . . . .	441
11.2.5.3.1.48	Deduction guides . . . . .	441
11.2.5.3.2	concurrent_unordered_map . . . . .	443
11.2.5.3.2.1	Class Template Synopsis . . . . .	443
11.2.5.3.2.2	Description . . . . .	448
11.2.5.3.2.3	Member functions . . . . .	448
11.2.5.3.2.4	Construction, destruction, copying . . . . .	448
11.2.5.3.2.5	Empty container constructors . . . . .	448
11.2.5.3.2.6	Constructors from the sequence of elements . . . . .	449
11.2.5.3.2.7	Copying constructors . . . . .	450
11.2.5.3.2.8	Moving constructors . . . . .	450
11.2.5.3.2.9	Destructor . . . . .	451
11.2.5.3.2.10	Assignment operators . . . . .	451
11.2.5.3.2.11	Iterators . . . . .	452
11.2.5.3.2.12	begin and cbegin . . . . .	452
11.2.5.3.2.13	end and cend . . . . .	452
11.2.5.3.2.14	Size and capacity . . . . .	452
11.2.5.3.2.15	empty . . . . .	452
11.2.5.3.2.16	size . . . . .	452
11.2.5.3.2.17	max_size . . . . .	453
11.2.5.3.2.18	Concurrently safe modifiers . . . . .	453
11.2.5.3.2.19	Emplacing elements . . . . .	453

11.2.5.3.2.20	Inserting values . . . . .	453
11.2.5.3.2.21	Inserting sequences of elements . . . . .	455
11.2.5.3.2.22	Inserting nodes . . . . .	455
11.2.5.3.2.23	Concurrently unsafe modifiers . . . . .	456
11.2.5.3.2.24	Clearing . . . . .	456
11.2.5.3.2.25	Erasing elements . . . . .	456
11.2.5.3.2.26	Erasing sequences . . . . .	457
11.2.5.3.2.27	Extracting nodes . . . . .	457
11.2.5.3.2.28	swap . . . . .	458
11.2.5.3.2.29	Element access . . . . .	458
11.2.5.3.2.30	at . . . . .	458
11.2.5.3.2.31	operator[] . . . . .	459
11.2.5.3.2.32	Lookup . . . . .	459
11.2.5.3.2.33	count . . . . .	459
11.2.5.3.2.34	find . . . . .	460
11.2.5.3.2.35	contains . . . . .	460
11.2.5.3.2.36	equal_range . . . . .	460
11.2.5.3.2.37	Bucket interface . . . . .	461
11.2.5.3.2.38	Bucket begin and bucket end . . . . .	461
11.2.5.3.2.39	The number of buckets . . . . .	461
11.2.5.3.2.40	Size of the bucket . . . . .	462
11.2.5.3.2.41	Bucket number . . . . .	462
11.2.5.3.2.42	Hash policy . . . . .	462
11.2.5.3.2.43	Load factor . . . . .	462
11.2.5.3.2.44	Manual rehashing . . . . .	462
11.2.5.3.2.45	Observers . . . . .	463
11.2.5.3.2.46	get_allocator . . . . .	463
11.2.5.3.2.47	hash_function . . . . .	463
11.2.5.3.2.48	key_eq . . . . .	463
11.2.5.3.2.49	Parallel iteration . . . . .	463
11.2.5.3.2.50	range member function . . . . .	463
11.2.5.3.2.51	Non-member functions . . . . .	463
11.2.5.3.2.52	Non-member swap . . . . .	464
11.2.5.3.2.53	Non-member binary comparisons . . . . .	464
11.2.5.3.2.54	Other . . . . .	465
11.2.5.3.2.55	Deduction guides . . . . .	465
11.2.5.3.3	concurrent_unordered_multimap . . . . .	467
11.2.5.3.3.1	Class Template Synopsis . . . . .	467
11.2.5.3.3.2	Description . . . . .	472
11.2.5.3.3.3	Member functions . . . . .	472
11.2.5.3.3.4	Construction, destruction, copying . . . . .	472
11.2.5.3.3.5	Empty container constructors . . . . .	472
11.2.5.3.3.6	Constructors from the sequence of elements . . . . .	473
11.2.5.3.3.7	Copying constructors . . . . .	474
11.2.5.3.3.8	Moving constructors . . . . .	474
11.2.5.3.3.9	Destructor . . . . .	474
11.2.5.3.3.10	Assignment operators . . . . .	475
11.2.5.3.3.11	Iterators . . . . .	475
11.2.5.3.3.12	begin and cbegin . . . . .	476
11.2.5.3.3.13	end and cend . . . . .	476
11.2.5.3.3.14	Size and capacity . . . . .	476
11.2.5.3.3.15	empty . . . . .	476
11.2.5.3.3.16	size . . . . .	476
11.2.5.3.3.17	max_size . . . . .	476

11.2.5.3.3.18	Concurrently safe modifiers . . . . .	477
11.2.5.3.3.19	Emplacing elements . . . . .	477
11.2.5.3.3.20	Inserting values . . . . .	477
11.2.5.3.3.21	Inserting sequences of elements . . . . .	478
11.2.5.3.3.22	Inserting nodes . . . . .	478
11.2.5.3.3.23	Merging containers . . . . .	479
11.2.5.3.3.24	Concurrently unsafe modifiers . . . . .	479
11.2.5.3.3.25	Clearing . . . . .	480
11.2.5.3.3.26	Erasing elements . . . . .	480
11.2.5.3.3.27	Erasing sequences . . . . .	480
11.2.5.3.3.28	Extracting nodes . . . . .	481
11.2.5.3.3.29	swap . . . . .	482
11.2.5.3.3.30	Lookup . . . . .	482
11.2.5.3.3.31	count . . . . .	482
11.2.5.3.3.32	find . . . . .	482
11.2.5.3.3.33	contains . . . . .	483
11.2.5.3.3.34	equal_range . . . . .	483
11.2.5.3.3.35	Bucket interface . . . . .	484
11.2.5.3.3.36	Bucket begin and bucket end . . . . .	484
11.2.5.3.3.37	The number of buckets . . . . .	484
11.2.5.3.3.38	Size of the bucket . . . . .	485
11.2.5.3.3.39	Bucket number . . . . .	485
11.2.5.3.3.40	Hash policy . . . . .	485
11.2.5.3.3.41	Load factor . . . . .	485
11.2.5.3.3.42	Manual rehashing . . . . .	485
11.2.5.3.3.43	Observers . . . . .	486
11.2.5.3.3.44	get_allocator . . . . .	486
11.2.5.3.3.45	hash_function . . . . .	486
11.2.5.3.3.46	key_eq . . . . .	486
11.2.5.3.3.47	Parallel iteration . . . . .	486
11.2.5.3.3.48	range member function . . . . .	486
11.2.5.3.3.49	Non-member functions . . . . .	486
11.2.5.3.3.50	Non-member swap . . . . .	487
11.2.5.3.3.51	Non-member binary comparisons . . . . .	487
11.2.5.3.3.52	Other . . . . .	488
11.2.5.3.3.53	Deduction guides . . . . .	488
11.2.5.3.4	concurrent_unordered_set . . . . .	490
11.2.5.3.4.1	Class Template Synopsis . . . . .	490
11.2.5.3.4.2	Description . . . . .	495
11.2.5.3.4.3	Member functions . . . . .	495
11.2.5.3.4.4	Construction, destruction, copying . . . . .	495
11.2.5.3.4.5	Empty container constructors . . . . .	495
11.2.5.3.4.6	Constructors from the sequence of elements . . . . .	496
11.2.5.3.4.7	Copying constructors . . . . .	497
11.2.5.3.4.8	Moving constructors . . . . .	497
11.2.5.3.4.9	Destructor . . . . .	497
11.2.5.3.4.10	Assignment operators . . . . .	497
11.2.5.3.4.11	Iterators . . . . .	498
11.2.5.3.4.12	begin and cbegin . . . . .	498
11.2.5.3.4.13	end and cend . . . . .	499
11.2.5.3.4.14	Size and capacity . . . . .	499
11.2.5.3.4.15	empty . . . . .	499
11.2.5.3.4.16	size . . . . .	499
11.2.5.3.4.17	max_size . . . . .	499

11.2.5.3.4.18	Concurrently safe modifiers . . . . .	499
11.2.5.3.4.19	Inserting values . . . . .	499
11.2.5.3.4.20	Inserting sequences of elements . . . . .	500
11.2.5.3.4.21	Inserting nodes . . . . .	501
11.2.5.3.4.22	Emplacing elements . . . . .	501
11.2.5.3.4.23	Merging containers . . . . .	502
11.2.5.3.4.24	Concurrently unsafe modifiers . . . . .	502
11.2.5.3.4.25	Clearing . . . . .	502
11.2.5.3.4.26	Erasing elements . . . . .	502
11.2.5.3.4.27	Erasing sequences . . . . .	503
11.2.5.3.4.28	Extracting nodes . . . . .	503
11.2.5.3.4.29	swap . . . . .	504
11.2.5.3.4.30	Lookup . . . . .	505
11.2.5.3.4.31	count . . . . .	505
11.2.5.3.4.32	find . . . . .	505
11.2.5.3.4.33	contains . . . . .	505
11.2.5.3.4.34	equal_range . . . . .	506
11.2.5.3.4.35	Bucket interface . . . . .	506
11.2.5.3.4.36	Bucket begin and bucket end . . . . .	506
11.2.5.3.4.37	The number of buckets . . . . .	507
11.2.5.3.4.38	Size of the bucket . . . . .	507
11.2.5.3.4.39	Bucket number . . . . .	507
11.2.5.3.4.40	Hash policy . . . . .	507
11.2.5.3.4.41	Load factor . . . . .	507
11.2.5.3.4.42	Manual rehashing . . . . .	508
11.2.5.3.4.43	Observers . . . . .	508
11.2.5.3.4.44	get_allocator . . . . .	508
11.2.5.3.4.45	hash_function . . . . .	508
11.2.5.3.4.46	key_eq . . . . .	508
11.2.5.3.4.47	Parallel iteration . . . . .	508
11.2.5.3.4.48	range member function . . . . .	509
11.2.5.3.4.49	Non-member functions . . . . .	509
11.2.5.3.4.50	Non-member swap . . . . .	509
11.2.5.3.4.51	Non-member binary comparisons . . . . .	510
11.2.5.3.4.52	Other . . . . .	510
11.2.5.3.4.53	Deduction guides . . . . .	510
11.2.5.3.5	concurrent_unordered_multiset . . . . .	512
11.2.5.3.5.1	Class Template Synopsis . . . . .	512
11.2.5.3.5.2	Description . . . . .	517
11.2.5.3.5.3	Member functions . . . . .	517
11.2.5.3.5.4	Construction, destruction, copying . . . . .	517
11.2.5.3.5.5	Empty container constructors . . . . .	517
11.2.5.3.5.6	Constructors from the sequence of elements . . . . .	518
11.2.5.3.5.7	Copying constructors . . . . .	519
11.2.5.3.5.8	Moving constructors . . . . .	519
11.2.5.3.5.9	Destructor . . . . .	519
11.2.5.3.5.10	Assignment operators . . . . .	519
11.2.5.3.5.11	Iterators . . . . .	520
11.2.5.3.5.12	begin and cbegin . . . . .	520
11.2.5.3.5.13	end and cend . . . . .	521
11.2.5.3.5.14	Size and capacity . . . . .	521
11.2.5.3.5.15	empty . . . . .	521
11.2.5.3.5.16	size . . . . .	521
11.2.5.3.5.17	max_size . . . . .	521

11.2.5.3.5.18	Concurrently safe modifiers . . . . .	521
11.2.5.3.5.19	Inserting values . . . . .	521
11.2.5.3.5.20	Inserting sequences of elements . . . . .	522
11.2.5.3.5.21	Inserting nodes . . . . .	522
11.2.5.3.5.22	Emplacing elements . . . . .	523
11.2.5.3.5.23	Merging containers . . . . .	523
11.2.5.3.5.24	Concurrently unsafe modifiers . . . . .	524
11.2.5.3.5.25	Clearing . . . . .	524
11.2.5.3.5.26	Erasing elements . . . . .	524
11.2.5.3.5.27	Erasing sequences . . . . .	525
11.2.5.3.5.28	Extracting nodes . . . . .	525
11.2.5.3.5.29	swap . . . . .	526
11.2.5.3.5.30	Lookup . . . . .	526
11.2.5.3.5.31	count . . . . .	526
11.2.5.3.5.32	find . . . . .	527
11.2.5.3.5.33	contains . . . . .	527
11.2.5.3.5.34	equal_range . . . . .	528
11.2.5.3.5.35	Bucket interface . . . . .	528
11.2.5.3.5.36	Bucket begin and bucket end . . . . .	528
11.2.5.3.5.37	The number of buckets . . . . .	529
11.2.5.3.5.38	Size of the bucket . . . . .	529
11.2.5.3.5.39	Bucket number . . . . .	529
11.2.5.3.5.40	Hash policy . . . . .	529
11.2.5.3.5.41	Load factor . . . . .	529
11.2.5.3.5.42	Manual rehashing . . . . .	530
11.2.5.3.5.43	Observers . . . . .	530
11.2.5.3.5.44	get_allocator . . . . .	530
11.2.5.3.5.45	hash_function . . . . .	530
11.2.5.3.5.46	key_eq . . . . .	530
11.2.5.3.5.47	Parallel iteration . . . . .	530
11.2.5.3.5.48	range member function . . . . .	531
11.2.5.3.5.49	Non-member functions . . . . .	531
11.2.5.3.5.50	Non-member swap . . . . .	531
11.2.5.3.5.51	Non-member binary comparisons . . . . .	532
11.2.5.3.5.52	Other . . . . .	532
11.2.5.3.5.53	Deduction guides . . . . .	532
11.2.5.4	Ordered associative containers . . . . .	534
11.2.5.4.1	concurrent_map . . . . .	534
11.2.5.4.1.1	Class Template Synopsis . . . . .	534
11.2.5.4.1.2	Member classes . . . . .	538
11.2.5.4.1.3	value_compare . . . . .	538
11.2.5.4.1.4	Class Synopsis . . . . .	538
11.2.5.4.1.5	Member objects . . . . .	539
11.2.5.4.1.6	Member functions . . . . .	539
11.2.5.4.1.7	Member functions . . . . .	539
11.2.5.4.1.8	Construction, destruction, copying . . . . .	539
11.2.5.4.1.9	Empty container constructors . . . . .	539
11.2.5.4.1.10	Constructors from the sequence of elements . . . . .	540
11.2.5.4.1.11	Copying constructors . . . . .	540
11.2.5.4.1.12	Moving constructors . . . . .	541
11.2.5.4.1.13	Destructor . . . . .	541
11.2.5.4.1.14	Assignment operators . . . . .	541
11.2.5.4.1.15	Element access . . . . .	542
11.2.5.4.1.16	at . . . . .	542

11.2.5.4.1.17	operator[] . . . . .	542
11.2.5.4.1.18	Iterators . . . . .	543
11.2.5.4.1.19	begin and cbegin . . . . .	543
11.2.5.4.1.20	end and cend . . . . .	543
11.2.5.4.1.21	Size and capacity . . . . .	543
11.2.5.4.1.22	empty . . . . .	543
11.2.5.4.1.23	size . . . . .	543
11.2.5.4.1.24	max_size . . . . .	544
11.2.5.4.1.25	Concurrently safe modifiers . . . . .	544
11.2.5.4.1.26	Inserting values . . . . .	544
11.2.5.4.1.27	Inserting sequences of elements . . . . .	545
11.2.5.4.1.28	Inserting nodes . . . . .	545
11.2.5.4.1.29	Emplacing elements . . . . .	546
11.2.5.4.1.30	Concurrently unsafe modifiers . . . . .	547
11.2.5.4.1.31	Clearing . . . . .	547
11.2.5.4.1.32	Erasing elements . . . . .	547
11.2.5.4.1.33	Erasing sequences . . . . .	548
11.2.5.4.1.34	Extracting nodes . . . . .	548
11.2.5.4.1.35	swap . . . . .	549
11.2.5.4.1.36	Lookup . . . . .	549
11.2.5.4.1.37	count . . . . .	549
11.2.5.4.1.38	find . . . . .	550
11.2.5.4.1.39	contains . . . . .	550
11.2.5.4.1.40	lower_bound . . . . .	550
11.2.5.4.1.41	upper_bound . . . . .	551
11.2.5.4.1.42	equal_range . . . . .	551
11.2.5.4.1.43	Observers . . . . .	552
11.2.5.4.1.44	get_allocator . . . . .	552
11.2.5.4.1.45	key_comp . . . . .	552
11.2.5.4.1.46	value_comp . . . . .	552
11.2.5.4.1.47	Parallel iteration . . . . .	552
11.2.5.4.1.48	range member function . . . . .	552
11.2.5.4.1.49	Non-member functions . . . . .	552
11.2.5.4.1.50	Non-member swap . . . . .	553
11.2.5.4.1.51	Non-member binary comparisons . . . . .	553
11.2.5.4.1.52	Non-member lexicographical comparisons . . . . .	554
11.2.5.4.1.53	Other . . . . .	554
11.2.5.4.1.54	Deduction guides . . . . .	554
11.2.5.4.2	concurrent_multimap . . . . .	556
11.2.5.4.2.1	Class Template Synopsis . . . . .	556
11.2.5.4.2.2	Member classes . . . . .	560
11.2.5.4.2.3	value_compare . . . . .	560
11.2.5.4.2.4	Class Synopsis . . . . .	560
11.2.5.4.2.5	Member objects . . . . .	560
11.2.5.4.2.6	Member functions . . . . .	560
11.2.5.4.2.7	Member functions . . . . .	561
11.2.5.4.2.8	Construction, destruction, copying . . . . .	561
11.2.5.4.2.9	Empty container constructors . . . . .	561
11.2.5.4.2.10	Constructors from the sequence of elements . . . . .	561
11.2.5.4.2.11	Copying constructors . . . . .	562
11.2.5.4.2.12	Moving constructors . . . . .	562
11.2.5.4.2.13	Destructor . . . . .	562
11.2.5.4.2.14	Assignment operators . . . . .	562
11.2.5.4.2.15	Iterators . . . . .	563

11.2.5.4.2.16	begin and cbegin . . . . .	563
11.2.5.4.2.17	end and cend . . . . .	563
11.2.5.4.2.18	Size and capacity . . . . .	564
11.2.5.4.2.19	empty . . . . .	564
11.2.5.4.2.20	size . . . . .	564
11.2.5.4.2.21	max_size . . . . .	564
11.2.5.4.2.22	Concurrently safe modifiers . . . . .	564
11.2.5.4.2.23	Emplacing elements . . . . .	564
11.2.5.4.2.24	Inserting values . . . . .	565
11.2.5.4.2.25	Inserting sequences of elements . . . . .	566
11.2.5.4.2.26	Inserting nodes . . . . .	566
11.2.5.4.2.27	Merging containers . . . . .	567
11.2.5.4.2.28	Concurrently unsafe modifiers . . . . .	567
11.2.5.4.2.29	Clearing . . . . .	567
11.2.5.4.2.30	Erasing elements . . . . .	567
11.2.5.4.2.31	Erasing sequences . . . . .	568
11.2.5.4.2.32	Extracting nodes . . . . .	568
11.2.5.4.2.33	swap . . . . .	569
11.2.5.4.2.34	Lookup . . . . .	570
11.2.5.4.2.35	count . . . . .	570
11.2.5.4.2.36	find . . . . .	570
11.2.5.4.2.37	contains . . . . .	571
11.2.5.4.2.38	lower_bound . . . . .	571
11.2.5.4.2.39	upper_bound . . . . .	571
11.2.5.4.2.40	equal_range . . . . .	572
11.2.5.4.2.41	Observers . . . . .	572
11.2.5.4.2.42	get_allocator . . . . .	572
11.2.5.4.2.43	key_comp . . . . .	572
11.2.5.4.2.44	value_comp . . . . .	573
11.2.5.4.2.45	Parallel iteration . . . . .	573
11.2.5.4.2.46	range member function . . . . .	573
11.2.5.4.2.47	Non-member functions . . . . .	573
11.2.5.4.2.48	Non-member swap . . . . .	574
11.2.5.4.2.49	Non-member binary comparisons . . . . .	574
11.2.5.4.2.50	Non-member lexicographical comparisons . . . . .	574
11.2.5.4.2.51	Other . . . . .	575
11.2.5.4.2.52	Deduction guides . . . . .	575
11.2.5.4.3	concurrent_set . . . . .	576
11.2.5.4.3.1	Class Template Synopsis . . . . .	576
11.2.5.4.3.2	Member functions . . . . .	580
11.2.5.4.3.3	Construction, destruction, copying . . . . .	580
11.2.5.4.3.4	Empty container constructors . . . . .	580
11.2.5.4.3.5	Constructors from the sequence of elements . . . . .	580
11.2.5.4.3.6	Copying constructors . . . . .	581
11.2.5.4.3.7	Moving constructors . . . . .	581
11.2.5.4.3.8	Destructor . . . . .	582
11.2.5.4.3.9	Assignment operators . . . . .	582
11.2.5.4.3.10	Iterators . . . . .	582
11.2.5.4.3.11	begin and cbegin . . . . .	583
11.2.5.4.3.12	end and cend . . . . .	583
11.2.5.4.3.13	Size and capacity . . . . .	583
11.2.5.4.3.14	empty . . . . .	583
11.2.5.4.3.15	size . . . . .	583
11.2.5.4.3.16	max_size . . . . .	583

11.2.5.4.3.17	Concurrently safe modifiers . . . . .	584
11.2.5.4.3.18	Inserting values . . . . .	584
11.2.5.4.3.19	Inserting sequences of elements . . . . .	585
11.2.5.4.3.20	Inserting nodes . . . . .	585
11.2.5.4.3.21	Emplacing elements . . . . .	586
11.2.5.4.3.22	Merging containers . . . . .	586
11.2.5.4.3.23	Concurrently unsafe modifiers . . . . .	587
11.2.5.4.3.24	Clearing . . . . .	587
11.2.5.4.3.25	Erasing elements . . . . .	587
11.2.5.4.3.26	Erasing sequences . . . . .	588
11.2.5.4.3.27	Extracting nodes . . . . .	588
11.2.5.4.3.28	swap . . . . .	589
11.2.5.4.3.29	Lookup . . . . .	589
11.2.5.4.3.30	count . . . . .	589
11.2.5.4.3.31	find . . . . .	589
11.2.5.4.3.32	contains . . . . .	590
11.2.5.4.3.33	lower_bound . . . . .	590
11.2.5.4.3.34	upper_bound . . . . .	590
11.2.5.4.3.35	equal_range . . . . .	591
11.2.5.4.3.36	Observers . . . . .	591
11.2.5.4.3.37	get_allocator . . . . .	591
11.2.5.4.3.38	key_comp . . . . .	591
11.2.5.4.3.39	value_comp . . . . .	592
11.2.5.4.3.40	Parallel iteration . . . . .	592
11.2.5.4.3.41	range member function . . . . .	592
11.2.5.4.3.42	Non-member functions . . . . .	592
11.2.5.4.3.43	Non-member swap . . . . .	593
11.2.5.4.3.44	Non-member binary comparisons . . . . .	593
11.2.5.4.3.45	Non-member lexicographical comparisons . . . . .	593
11.2.5.4.3.46	Other . . . . .	594
11.2.5.4.3.47	Deduction guides . . . . .	594
11.2.5.4.4	concurrent_multiset . . . . .	595
11.2.5.4.4.1	Class Template Synopsis . . . . .	595
11.2.5.4.4.2	Member functions . . . . .	599
11.2.5.4.4.3	Construction, destruction, copying . . . . .	599
11.2.5.4.4.4	Empty container constructors . . . . .	599
11.2.5.4.4.5	Constructors from the sequence of elements . . . . .	599
11.2.5.4.4.6	Copying constructors . . . . .	600
11.2.5.4.4.7	Moving constructors . . . . .	600
11.2.5.4.4.8	Destructor . . . . .	600
11.2.5.4.4.9	Assignment operators . . . . .	601
11.2.5.4.4.10	Iterators . . . . .	601
11.2.5.4.4.11	begin and cbegin . . . . .	601
11.2.5.4.4.12	end and cend . . . . .	602
11.2.5.4.4.13	Size and capacity . . . . .	602
11.2.5.4.4.14	empty . . . . .	602
11.2.5.4.4.15	size . . . . .	602
11.2.5.4.4.16	max_size . . . . .	602
11.2.5.4.4.17	Concurrently safe modifiers . . . . .	602
11.2.5.4.4.18	Inserting values . . . . .	602
11.2.5.4.4.19	Inserting sequences of elements . . . . .	603
11.2.5.4.4.20	Inserting nodes . . . . .	604
11.2.5.4.4.21	Emplacing elements . . . . .	604
11.2.5.4.4.22	Merging containers . . . . .	605

11.2.5.4.4.23	Concurrently unsafe modifiers . . . . .	605
11.2.5.4.4.24	Clearing . . . . .	605
11.2.5.4.4.25	Erasing elements . . . . .	605
11.2.5.4.4.26	Erasing sequences . . . . .	606
11.2.5.4.4.27	Extracting nodes . . . . .	606
11.2.5.4.4.28	swap . . . . .	607
11.2.5.4.4.29	Lookup . . . . .	607
11.2.5.4.4.30	count . . . . .	607
11.2.5.4.4.31	find . . . . .	608
11.2.5.4.4.32	contains . . . . .	608
11.2.5.4.4.33	lower_bound . . . . .	609
11.2.5.4.4.34	upper_bound . . . . .	609
11.2.5.4.4.35	equal_range . . . . .	609
11.2.5.4.4.36	Observers . . . . .	610
11.2.5.4.4.37	get_allocator . . . . .	610
11.2.5.4.4.38	key_comp . . . . .	610
11.2.5.4.4.39	value_comp . . . . .	610
11.2.5.4.4.40	Parallel iteration . . . . .	610
11.2.5.4.4.41	range member function . . . . .	611
11.2.5.4.4.42	Non-member functions . . . . .	611
11.2.5.4.4.43	Non-member swap . . . . .	612
11.2.5.4.4.44	Non-member binary comparisons . . . . .	612
11.2.5.4.4.45	Non-member lexicographical comparisons . . . . .	612
11.2.5.4.4.46	Other . . . . .	613
11.2.5.4.4.47	Deduction guides . . . . .	613
11.2.5.5	Auxiliary classes . . . . .	614
11.2.5.5.1	tbb_hash_compare . . . . .	614
11.2.5.5.1.1	Class Template Synopsis . . . . .	614
11.2.5.5.1.2	Member functions . . . . .	614
11.2.5.5.2	Node handles . . . . .	614
11.2.5.5.2.1	Class synopsis . . . . .	615
11.2.5.5.2.2	Member functions . . . . .	616
11.2.5.5.2.3	Constructors . . . . .	616
11.2.5.5.2.4	Assignment . . . . .	616
11.2.5.5.2.5	Destructor . . . . .	616
11.2.5.5.2.6	Swap . . . . .	616
11.2.5.5.2.7	State . . . . .	617
11.2.5.5.2.8	Access to the stored element . . . . .	617
11.2.5.5.2.9	get_allocator . . . . .	617
11.2.6	Thread Local Storage . . . . .	618
11.2.6.1	combinable . . . . .	618
11.2.6.1.1	Member functions . . . . .	618
11.2.6.2	enumerable_thread_specific . . . . .	620
11.2.6.2.1	Member functions . . . . .	622
11.2.6.2.1.1	Construction, destruction, copying . . . . .	622
11.2.6.2.1.2	Empty container constructors . . . . .	622
11.2.6.2.1.3	Copying constructors . . . . .	623
11.2.6.2.1.4	Moving constructors . . . . .	623
11.2.6.2.1.5	Destructor . . . . .	623
11.2.6.2.1.6	Assignment operators . . . . .	623
11.2.6.2.1.7	Concurrently safe modifiers . . . . .	624
11.2.6.2.1.8	Concurrently unsafe modifiers . . . . .	624
11.2.6.2.1.9	clear . . . . .	624
11.2.6.2.1.10	Size and capacity . . . . .	624

11.2.6.2.1.11	Iteration . . . . .	625
11.2.6.2.1.12	Combining . . . . .	625
11.2.6.2.2	Non-member types and constants . . . . .	626
11.2.6.3	flattened2d . . . . .	626
11.2.6.3.1	Member functions . . . . .	627
11.2.6.3.2	Non-member functions . . . . .	628
11.3	oneTBB Auxiliary Interfaces . . . . .	628
11.3.1	Memory Allocation . . . . .	628
11.3.1.1	Allocators . . . . .	628
11.3.1.1.1	tbb_allocator . . . . .	628
11.3.1.1.1.1	Member Functions . . . . .	629
11.3.1.1.1.2	Non-member Functions . . . . .	629
11.3.1.1.2	scalable_allocator . . . . .	630
11.3.1.1.2.1	Member Functions . . . . .	630
11.3.1.1.2.2	Non-member Functions . . . . .	630
11.3.1.1.3	cache_aligned_allocator . . . . .	631
11.3.1.1.3.1	Member Functions . . . . .	632
11.3.1.1.3.2	Non-member Functions . . . . .	632
11.3.1.2	Memory Resources . . . . .	632
11.3.1.2.1	cache_aligned_resource . . . . .	633
11.3.1.2.1.1	Member Functions . . . . .	633
11.3.1.2.2	scalable_memory_resource . . . . .	634
11.3.1.3	Library Functions . . . . .	634
11.3.1.3.1	C Interface to Scalable Allocator . . . . .	634
11.3.2	Mutual Exclusion . . . . .	637
11.3.2.1	Mutex Classes . . . . .	637
11.3.2.1.1	spin_mutex . . . . .	637
11.3.2.1.1.1	Member classes . . . . .	638
11.3.2.1.1.2	Member functions . . . . .	638
11.3.2.1.2	spin_rw_mutex . . . . .	638
11.3.2.1.2.1	Member classes . . . . .	639
11.3.2.1.2.2	Member functions . . . . .	639
11.3.2.1.3	speculative_spin_mutex . . . . .	639
11.3.2.1.3.1	Member classes . . . . .	640
11.3.2.1.3.2	Member functions . . . . .	640
11.3.2.1.4	speculative_spin_rw_mutex . . . . .	640
11.3.2.1.4.1	Member classes . . . . .	641
11.3.2.1.4.2	Member functions . . . . .	641
11.3.2.1.5	queuing_mutex . . . . .	641
11.3.2.1.5.1	Member classes . . . . .	642
11.3.2.1.5.2	Member functions . . . . .	642
11.3.2.1.6	queuing_rw_mutex . . . . .	642
11.3.2.1.6.1	Member classes . . . . .	643
11.3.2.1.6.2	Member functions . . . . .	643
11.3.2.1.7	null_mutex . . . . .	643
11.3.2.1.7.1	Member classes . . . . .	643
11.3.2.1.7.2	Member functions . . . . .	644
11.3.2.1.8	null_rw_mutex . . . . .	644
11.3.2.1.8.1	Member classes . . . . .	645
11.3.2.1.8.2	Member functions . . . . .	645
11.3.3	Timing . . . . .	645
11.3.3.1	Syntax . . . . .	645
11.3.3.2	Classes . . . . .	646
11.3.3.2.1	tick_count class . . . . .	646

11.3.3.2.2	<code>tick_count::interval_t</code> class . . . . .	646
11.3.3.2.3	Non-member functions . . . . .	647
<b>12</b>	<b>oneVPL</b>	<b>648</b>
12.1	Architecture . . . . .	648
12.1.1	Video Decoding . . . . .	649
12.1.2	Video Encoding . . . . .	650
12.1.3	Video Processing . . . . .	650
12.1.3.1	Color Conversion Support . . . . .	651
12.1.3.2	Deinterlacing/Inverse Telecine Support . . . . .	651
12.1.3.3	Color Format Support . . . . .	652
12.2	Programming Guide . . . . .	653
12.2.1	Status Codes . . . . .	653
12.2.2	SDK Session . . . . .	654
12.2.2.1	Intel® Media Software Development Kit Dispatcher (Legacy) . . . . .	654
12.2.2.2	oneVPL Dispatcher . . . . .	655
12.2.2.3	Multiple Sessions . . . . .	656
12.2.3	Frame and Fields . . . . .	657
12.2.3.1	Frame Surface Locking . . . . .	657
12.2.4	Decoding Procedures . . . . .	658
12.2.4.1	Bitstream Repositioning . . . . .	660
12.2.4.2	Broken Streams Handling . . . . .	660
12.2.4.3	VP8 Specific Details . . . . .	661
12.2.4.4	JPEG . . . . .	661
12.2.4.5	Multi-view Video Decoding . . . . .	663
12.2.5	Encoding Procedures . . . . .	664
12.2.5.1	External Memory . . . . .	664
12.2.5.2	Internal Memory . . . . .	665
12.2.5.3	Configuration Change . . . . .	665
12.2.5.4	External Bitrate Control . . . . .	666
12.2.5.5	JPEG . . . . .	672
12.2.5.6	Multi-view Video Encoding . . . . .	673
12.2.6	Video Processing Procedures . . . . .	674
12.2.6.1	Configuration . . . . .	675
12.2.6.2	Region of Interest . . . . .	677
12.2.6.3	Multi-view Video Processing . . . . .	677
12.2.7	Transcoding Procedures . . . . .	678
12.2.7.1	Asynchronous Pipeline . . . . .	678
12.2.7.2	Surface Pool Allocation . . . . .	679
12.2.7.3	Pipeline Error Reporting . . . . .	680
12.2.8	Working with Hardware Acceleration . . . . .	680
12.2.8.1	Working with Multiple Media Devices . . . . .	680
12.2.8.2	Working with Video Memory . . . . .	683
12.2.8.3	Working with Microsoft DirectX® Applications . . . . .	684
12.2.8.4	Working with VA API Applications . . . . .	686
12.2.9	Memory Allocation and External Allocators . . . . .	687
12.2.9.1	External Memory Management . . . . .	687
12.2.9.2	Internal Memory Management . . . . .	688
12.2.9.3	<code>mfxFrameSurfaceInterface</code> . . . . .	688
12.2.10	Hardware Device Error Handling . . . . .	689
12.3	Mandatory/Optional APIs and Features . . . . .	690
12.3.1	Disclaimer . . . . .	690
12.3.1.1	Functions must be exposed by any implementation . . . . .	690
12.3.1.2	Mandatory API for each implemenation . . . . .	691

12.4 Appendices . . . . .	692
12.4.1 oneVPL for Intel® Media Software Development Kit Users . . . . .	692
12.4.1.1 oneVPL Ease of Use Enhancements . . . . .	692
12.4.1.2 New APIs in oneVPL . . . . .	692
12.4.1.3 Intel® Media Software Development Kit Feature Removals . . . . .	693
12.4.1.4 Intel® Media Software Development Kit API Removals . . . . .	693
12.4.2 Configuration Parameter Constraints . . . . .	695
12.4.2.1 DECODE, ENCODE, and VPP Constraints . . . . .	695
12.4.2.2 DECODE Constraints . . . . .	696
12.4.2.3 ENCODE Constraints . . . . .	696
12.4.2.4 VPP Constraints . . . . .	697
12.4.2.5 Specifying Configuration Parameters . . . . .	698
12.4.3 Multiple-segment Encoding . . . . .	701
12.4.4 Streaming and Video Conferencing Features . . . . .	702
12.4.4.1 Dynamic Bitrate Change . . . . .	702
12.4.4.2 Dynamic Resolution Change . . . . .	703
12.4.4.3 Dynamic Reference Frame Scaling . . . . .	703
12.4.4.4 Forced Keyframe Generation . . . . .	703
12.4.4.5 Reference List Selection . . . . .	704
12.4.4.6 Low Latency Encoding and Decoding . . . . .	704
12.4.4.7 Reference Picture Marking Repetition SEI Message . . . . .	704
12.4.4.8 Long Term Reference Frame . . . . .	705
12.4.4.9 Temporal Scalability . . . . .	705
12.4.5 Switchable Graphics and Multiple Monitors . . . . .	705
12.4.5.1 Switchable Graphics . . . . .	706
12.4.5.2 Multiple Monitors . . . . .	706
12.4.6 Working Directly with VA API for Linux* . . . . .	707
12.4.7 CQP HRD Mode Encoding . . . . .	709
12.5 Acronyms and Abbreviations . . . . .	710
12.6 oneVPL API Versioning . . . . .	711
12.7 oneVPL API Reference . . . . .	712
12.7.1 Types . . . . .	712
12.7.1.1 Basic Types . . . . .	712
12.7.1.2 Typedefs . . . . .	713
12.7.2 Dispatcher API . . . . .	713
12.7.2.1 Defines . . . . .	713
12.7.2.2 Enums . . . . .	713
12.7.2.3 mfxImplType . . . . .	713
12.7.2.4 mfxAccelerationMode . . . . .	713
12.7.2.5 Structures . . . . .	714
12.7.2.6 mfxVariant . . . . .	714
12.7.2.7 mfxDecoderDescription . . . . .	715
12.7.2.8 mfxEncoderDescription . . . . .	717
12.7.2.9 mfxVPPDescription . . . . .	718
12.7.2.10 mfxDeviceDescription . . . . .	719
12.7.2.11 mfxImplDescription . . . . .	720
12.7.2.12 Functions . . . . .	721
12.7.3 Enums . . . . .	725
12.7.3.1 mfxStatus . . . . .	725
12.7.3.2 mfxIMPL . . . . .	727
12.7.3.3 mfxImplCapsDeliveryFormat . . . . .	728
12.7.3.4 mfxPriority . . . . .	728
12.7.3.5 GPUCopy . . . . .	728
12.7.3.6 PlatformCodeName . . . . .	728

12.7.3.7 mfxMediaAdapterType . . . . .	729
12.7.3.8 mfxMemoryFlags . . . . .	729
12.7.3.9 mfxResourceType . . . . .	730
12.7.3.10 ColorFourCC . . . . .	730
12.7.3.11 ChromaFormatIdx . . . . .	732
12.7.3.12 PicStruct . . . . .	733
12.7.3.13 Frame Data Flags . . . . .	733
12.7.3.14 Corruption . . . . .	734
12.7.3.15 TimeStampCalc . . . . .	734
12.7.3.16 IOPattern . . . . .	734
12.7.3.17 CodecFormatFourCC . . . . .	735
12.7.3.18 CodecProfile . . . . .	735
12.7.3.18.1 Multi-view Video Coding Extension Profiles . . . . .	736
12.7.3.18.2 MPEG-2 Profiles . . . . .	736
12.7.3.18.3 VC-1 Profiles . . . . .	736
12.7.3.18.4 HEVC Profiles . . . . .	736
12.7.3.18.5 VP9 Profiles . . . . .	736
12.7.3.18.6 VP9 Profiles . . . . .	737
12.7.3.18.7 JPEG Profiles . . . . .	737
12.7.3.19 CodecLevel . . . . .	737
12.7.3.19.1 H.264 Level 1-1.3 . . . . .	737
12.7.3.19.2 H.264 Level 2-2.2 . . . . .	737
12.7.3.19.3 H.264 Level 3-3.2 . . . . .	737
12.7.3.19.4 H.264 Level 4-4.2 . . . . .	738
12.7.3.19.5 H.264 Level 5-5.2 . . . . .	738
12.7.3.19.6 MPEG2 Levels . . . . .	738
12.7.3.19.7 VC-1 Level Low (Simple and Main Profiles) . . . . .	738
12.7.3.20 VC-1 Advanced Profile Levels . . . . .	738
12.7.3.20.1 HEVC Levels . . . . .	738
12.7.3.21 HEVC Tiers . . . . .	739
12.7.3.22 GopOptFlag . . . . .	739
12.7.3.23 TargetUsage . . . . .	739
12.7.3.24 RateControlMethod . . . . .	740
12.7.3.25 TrellisControl . . . . .	741
12.7.3.26 BRefControl . . . . .	741
12.7.3.27 LookAheadDownSampling . . . . .	742
12.7.3.28 BPSEIControl . . . . .	742
12.7.3.29 SkipFrame . . . . .	742
12.7.3.30 IntraRefreshTypes . . . . .	743
12.7.3.31 WeightedPred . . . . .	743
12.7.3.32 PRefType . . . . .	743
12.7.3.33 ScenarioInfo . . . . .	744
12.7.3.34 ContentInfo . . . . .	744
12.7.3.35 IntraPredBlockSize/InterPredBlockSize . . . . .	744
12.7.3.36 MVPrecision . . . . .	744
12.7.3.37 CodingOptionValue . . . . .	745
12.7.3.38 BitstreamDataFlag . . . . .	745
12.7.3.39 ExtendedBufferID . . . . .	745
12.7.3.40 PayloadCtrlFlags . . . . .	750
12.7.3.41 ExtMemBufferType . . . . .	750
12.7.3.42 ExtMemFrameType . . . . .	750
12.7.3.43 FrameType . . . . .	751
12.7.3.44 MfxNalUnitType . . . . .	752
12.7.3.45 mfxHandleType . . . . .	752

12.7.3.46 mfxSkipMode . . . . .	753
12.7.3.47 FrcAlg . . . . .	753
12.7.3.48 ImageStabMode . . . . .	753
12.7.3.49 InsertHDRPayload . . . . .	754
12.7.3.50 LongTermIdx . . . . .	754
12.7.3.51 TransferMatrix . . . . .	754
12.7.3.52 NominalRange . . . . .	754
12.7.3.53 ROImode . . . . .	754
12.7.3.54 DeinterlacingMode . . . . .	755
12.7.3.55 TelecinePattern . . . . .	755
12.7.3.56 VPPFieldProcessingMode . . . . .	756
12.7.3.57 PicType . . . . .	756
12.7.3.58 MBQPMode . . . . .	756
12.7.3.59 GeneralConstraintFlags . . . . .	756
12.7.3.60 SampleAdaptiveOffset . . . . .	757
12.7.3.61 ErrorTypes . . . . .	757
12.7.3.62 HEVCRegionType . . . . .	757
12.7.3.63 HEVCRegionEncoding . . . . .	758
12.7.3.64 Angle . . . . .	758
12.7.3.65 ScalingMode . . . . .	758
12.7.3.66 InterpolationMode . . . . .	758
12.7.3.67 MirroringType . . . . .	759
12.7.3.68 ChromaSiting . . . . .	759
12.7.3.69 VP9ReferenceFrame . . . . .	759
12.7.3.70 SegmentIdBlockSize . . . . .	760
12.7.3.71 SegmentFeature . . . . .	760
12.7.3.72 mfxComponentType . . . . .	760
12.7.3.73 PartialBitstreamOutput . . . . .	761
12.7.3.74 BRCStatus . . . . .	761
12.7.3.75 Rotation . . . . .	761
12.7.3.76 JPEGColorFormat . . . . .	762
12.7.3.77 JPEGScanType . . . . .	762
12.7.3.78 Protected . . . . .	762
12.7.4 Structs . . . . .	762
12.7.4.1 mfxRange32U . . . . .	762
12.7.4.2 mfxI16Pair . . . . .	763
12.7.4.3 mfxHDLPair . . . . .	763
12.7.4.4 mfxVersion . . . . .	763
12.7.4.5 mfxStructVersion . . . . .	764
12.7.4.6 mfxPlatform . . . . .	764
12.7.4.7 mfxInitParam . . . . .	764
12.7.4.8 mfxInfoMFX . . . . .	765
12.7.4.9 mfxFrameId . . . . .	769
12.7.4.10 mfxFrameInfo . . . . .	769
12.7.4.11 mfxVideoParam . . . . .	772
12.7.4.12 mfxFrameData . . . . .	773
12.7.4.13 mfxFrameSurfaceInterface . . . . .	776
12.7.4.14 mfxFrameSurface1 . . . . .	779
12.7.4.15 mfxBitstream . . . . .	779
12.7.4.16 mfxEncodeStat . . . . .	780
12.7.4.17 mfxDecodeStat . . . . .	780
12.7.4.18 mfxPayload . . . . .	781
12.7.4.19 mfxEncodeCtrl . . . . .	782
12.7.4.20 mfxFrameAllocRequest . . . . .	783

12.7.4.21 mfxFrameAllocResponse . . . . .	783
12.7.4.22 mfxFrameAllocator . . . . .	783
12.7.4.23 mfxComponentInfo . . . . .	785
12.7.4.24 mfxAdapterInfo . . . . .	785
12.7.4.25 mfxAdaptersInfo . . . . .	786
12.7.4.26 mfxQPandMode . . . . .	786
12.7.4.27 VPP Structures . . . . .	786
12.7.4.27.1 mfxInfoVPP . . . . .	786
12.7.4.27.2 mfxVPPStat . . . . .	787
12.7.4.28 Extension buffers structures . . . . .	787
12.7.4.28.1 mfxExtBuffer . . . . .	787
12.7.4.28.2 mfxExtCodingOption . . . . .	787
12.7.4.28.3 mfxExtCodingOption2 . . . . .	789
12.7.4.28.4 mfxExtCodingOption3 . . . . .	793
12.7.4.28.5 mfxExtCodingOptionSPSPPS . . . . .	797
12.7.4.28.6 mfxExtInsertHeaders . . . . .	797
12.7.4.28.7 mfxExtCodingOptionVPS . . . . .	798
12.7.4.28.8 mfxExtThreadsParam . . . . .	798
12.7.4.28.9 mfxExtVideoSignalInfo . . . . .	799
12.7.4.28.10mfxExtAVCRefListCtrl . . . . .	800
12.7.4.28.11mfxExtMasteringDisplayColourVolume . . . . .	801
12.7.4.28.12mfxExtContentLightLevelInfo . . . . .	801
12.7.4.28.13mfxExtPictureTimingSEI . . . . .	802
12.7.4.28.14mfxExtAvTemporalLayers . . . . .	803
12.7.4.28.15mfxExtEncoderCapability . . . . .	803
12.7.4.28.16mfxExtEncoderResetOption . . . . .	804
12.7.4.28.17mfxExtAVCEncodedFrameInfo . . . . .	805
12.7.4.28.18mfxExtEncoderROI . . . . .	806
12.7.4.28.19mfxExtEncoderIPCMArea . . . . .	807
12.7.4.28.20mfxExtAVCRefLists . . . . .	807
12.7.4.28.21mfxExtChromaLocInfo . . . . .	808
12.7.4.28.22mfxExtMBForceIntra . . . . .	808
12.7.4.28.23mfxExtMBQP . . . . .	809
12.7.4.28.24mfxExtHEVCTiles . . . . .	809
12.7.4.28.25mfxExtMBDisableSkipMap . . . . .	810
12.7.4.28.26mfxExtHEVCParam . . . . .	810
12.7.4.28.27mfxExtDecodeErrorReport . . . . .	811
12.7.4.28.28mfxExtDecodedFrameInfo . . . . .	811
12.7.4.28.29mfxExtTimeCode . . . . .	811
12.7.4.28.30mfxExtHEVCRegion . . . . .	812
12.7.4.28.31mfxExtPredWeightTable . . . . .	812
12.7.4.28.32mfxExtAVCRoundingOffset . . . . .	813
12.7.4.28.33mfxExtDirtyRect . . . . .	813
12.7.4.28.34mfxExtMoveRect . . . . .	814
12.7.4.28.35mfxExtMVOverPicBoundaries . . . . .	815
12.7.4.28.36mfxVP9SegmentParam . . . . .	815
12.7.4.28.37mfxExtVP9Segmentation . . . . .	816
12.7.4.28.38mfxVP9TemporalLayer . . . . .	817
12.7.4.28.39mfxExtVP9TemporalLayers . . . . .	817
12.7.4.28.40mfxExtVP9Param . . . . .	818
12.7.4.28.41mfxEncodedUnitInfo . . . . .	819
12.7.4.28.42mfxExtEncodedUnitsInfo . . . . .	819
12.7.4.28.43mfxExtPartialBitstreamParam . . . . .	820
12.7.4.29 VPP Extension Buffers . . . . .	820

12.7.4.29.1	mfxExtVPPDoNotUse . . . . .	820
12.7.4.29.2	mfxExtVPPDoUse . . . . .	821
12.7.4.29.3	mfxExtVPPDenoise . . . . .	821
12.7.4.29.4	mfxExtVPPDetail . . . . .	822
12.7.4.29.5	mfxExtVPPProcAmp . . . . .	822
12.7.4.29.6	mfxExtVPPDeinterlacing . . . . .	823
12.7.4.29.7	mfxExtEncodedSlicesInfo . . . . .	823
12.7.4.29.8	mfxExtVppAuxData . . . . .	824
12.7.4.29.9	mfxExtVPPFrameRateConversion . . . . .	824
12.7.4.29.10	mfxExtVPPIImageStab . . . . .	825
12.7.4.29.11	mfxExtVPPCompInputStream . . . . .	825
12.7.4.29.12	mfxExtVPPComposite . . . . .	826
12.7.4.29.13	mfxExtVPPVideoSignalInfo . . . . .	828
12.7.4.29.14	mfxExtVPPFieldProcessing . . . . .	828
12.7.4.29.15	mfxExtDecVideoProcessing . . . . .	829
12.7.4.29.16	mfxExtVPPRotation . . . . .	830
12.7.4.29.17	mfxExtVPPScaling . . . . .	830
12.7.4.29.18	mfxExtVPPMirroring . . . . .	831
12.7.4.29.19	mfxExtVPPColorFill . . . . .	831
12.7.4.29.20	mfxExtColorConversion . . . . .	831
12.7.4.29.21	mfxExtVppMctf . . . . .	832
12.7.4.30	Bitrate Control Extension Buffers . . . . .	832
12.7.4.30.1	mfxBRCFrameParam . . . . .	832
12.7.4.30.2	mfxBRCFrameCtrl . . . . .	833
12.7.4.30.3	mfxBRCFrameStatus . . . . .	834
12.7.4.30.4	mfxExtBRC . . . . .	834
12.7.4.31	VP8 Extension Buffers . . . . .	836
12.7.4.31.1	mfxExtVP8CodingOption . . . . .	836
12.7.4.32	JPEG Extension Buffers . . . . .	837
12.7.4.32.1	mfxExtJPEGQuantTables . . . . .	837
12.7.4.32.2	mfxExtJPEGHuffmanTables . . . . .	837
12.7.4.33	MVC Extension Buffers . . . . .	838
12.7.4.33.1	mfxMVCViewDependency . . . . .	838
12.7.4.33.2	mfxMVCOperationPoint . . . . .	839
12.7.4.33.3	mfxExtMVCSeqDesc . . . . .	839
12.7.4.33.4	mfxExtMVCTargetViews . . . . .	840
12.7.4.34	PCP Extension Buffers . . . . .	840
12.7.5	Defines . . . . .	841
12.7.6	Functions . . . . .	841
12.7.6.1	Implementation Capabilities . . . . .	841
12.7.6.2	Session Management . . . . .	842
12.7.6.3	VideoCORE . . . . .	845
12.7.6.4	Memory . . . . .	846
12.7.6.5	VideoENCODE . . . . .	847
12.7.6.6	VideoDECODE . . . . .	851
12.7.6.7	VideoVPP . . . . .	856
12.7.6.8	Adapters . . . . .	859
<b>13</b>	<b>oneMKL</b>	<b>861</b>
13.1	oneMKL Architecture . . . . .	861
13.1.1	Execution Model . . . . .	861
13.1.1.1	Use of Queues . . . . .	861
13.1.1.1.1	Non-Member Functions . . . . .	862
13.1.1.1.2	Member Functions . . . . .	862

13.1.1.2	Device Usage . . . . .	862
13.1.1.3	Asynchronous Execution . . . . .	862
13.1.1.3.1	Synchronization When Using Buffers . . . . .	863
13.1.1.3.2	Synchronization When Using USM APIs . . . . .	863
13.1.1.4	Host Thread Safety . . . . .	863
13.1.2	Memory Model . . . . .	863
13.1.2.1	The Buffer Memory Model . . . . .	863
13.1.2.2	Unified Shared Memory Model . . . . .	864
13.1.3	API Design . . . . .	864
13.1.3.1	oneMKL namespaces . . . . .	864
13.1.3.2	Standard C++ datatype usage . . . . .	864
13.1.3.3	DPC++ datatype usage . . . . .	865
13.1.3.4	oneMKL defined datatypes . . . . .	865
13.1.4	Exceptions and Error Handling . . . . .	867
13.1.5	Other Features . . . . .	867
13.1.5.1	oneMKL Specification Versioning Strategy . . . . .	867
13.1.5.2	Current Version of this oneMKL Specification . . . . .	867
13.1.5.3	Pre/Post Condition Checking . . . . .	867
13.2	oneMKL Domains . . . . .	867
13.2.1	Dense Linear Algebra . . . . .	867
13.2.1.1	Matrix Storage . . . . .	868
13.2.1.2	BLAS Routines . . . . .	874
13.2.1.2.1	BLAS Level 1 Routines . . . . .	874
13.2.1.2.1.1	asum . . . . .	875
13.2.1.2.1.2	asum (Buffer Version) . . . . .	875
13.2.1.2.1.3	asum (USM Version) . . . . .	876
13.2.1.2.1.4	axpy . . . . .	876
13.2.1.2.1.5	axpy (Buffer Version) . . . . .	877
13.2.1.2.1.6	axpy (USM Version) . . . . .	877
13.2.1.2.1.7	copy . . . . .	878
13.2.1.2.1.8	copy (Buffer Version) . . . . .	879
13.2.1.2.1.9	copy (USM Version) . . . . .	879
13.2.1.2.1.10	dot . . . . .	880
13.2.1.2.1.11	dot (Buffer Version) . . . . .	880
13.2.1.2.1.12	dot (USM Version) . . . . .	881
13.2.1.2.1.13	sdsdot . . . . .	882
13.2.1.2.1.14	sdsdot (Buffer Version) . . . . .	882
13.2.1.2.1.15	sdsdot (USM Version) . . . . .	883
13.2.1.2.1.16	dotc . . . . .	883
13.2.1.2.1.17	dotc (Buffer Version) . . . . .	884
13.2.1.2.1.18	dotc (USM Version) . . . . .	884
13.2.1.2.1.19	dotu . . . . .	885
13.2.1.2.1.20	dotu (Buffer Version) . . . . .	886
13.2.1.2.1.21	dotu (USM Version) . . . . .	886
13.2.1.2.1.22	nrm2 . . . . .	887
13.2.1.2.1.23	nrm2 (Buffer Version) . . . . .	887
13.2.1.2.1.24	nrm2 (USM Version) . . . . .	888
13.2.1.2.1.25	rot . . . . .	888
13.2.1.2.1.26	rot (Buffer Version) . . . . .	889
13.2.1.2.1.27	rot (USM Version) . . . . .	889
13.2.1.2.1.28	rotg . . . . .	890
13.2.1.2.1.29	rotg (Buffer Version) . . . . .	891
13.2.1.2.1.30	rotg (USM Version) . . . . .	891
13.2.1.2.1.31	rotm . . . . .	892

13.2.1.2.1.32	rotm (Buffer Version) . . . . .	892
13.2.1.2.1.33	rotm (USM Version) . . . . .	894
13.2.1.2.1.34	rotmg . . . . .	895
13.2.1.2.1.35	rotmg (Buffer Version) . . . . .	896
13.2.1.2.1.36	rotmg (USM Version) . . . . .	897
13.2.1.2.1.37	scal . . . . .	898
13.2.1.2.1.38	scal (Buffer Version) . . . . .	899
13.2.1.2.1.39	scal (USM Version) . . . . .	899
13.2.1.2.1.40	swap . . . . .	900
13.2.1.2.1.41	swap (Buffer Version) . . . . .	900
13.2.1.2.1.42	swap (USM Version) . . . . .	901
13.2.1.2.1.43	iamax . . . . .	901
13.2.1.2.1.44	iamax (Buffer Version) . . . . .	902
13.2.1.2.1.45	iamax (USM Version) . . . . .	902
13.2.1.2.1.46	iamin . . . . .	903
13.2.1.2.1.47	iamin (Buffer Version) . . . . .	904
13.2.1.2.1.48	iamin (USM Version) . . . . .	904
13.2.1.2.2	BLAS Level 2 Routines . . . . .	905
13.2.1.2.2.1	gbmv . . . . .	905
13.2.1.2.2.2	gbmv (Buffer Version) . . . . .	906
13.2.1.2.2.3	gbmv (USM Version) . . . . .	907
13.2.1.2.2.4	gemv . . . . .	908
13.2.1.2.2.5	gemv (Buffer Version) . . . . .	908
13.2.1.2.2.6	gemv (USM Version) . . . . .	909
13.2.1.2.2.7	ger . . . . .	910
13.2.1.2.2.8	ger (Buffer Version) . . . . .	911
13.2.1.2.2.9	ger (USM Version) . . . . .	911
13.2.1.2.2.10	gerc . . . . .	912
13.2.1.2.2.11	gerc (Buffer Version) . . . . .	913
13.2.1.2.2.12	gerc (USM Version) . . . . .	913
13.2.1.2.2.13	geru . . . . .	914
13.2.1.2.2.14	geru (Buffer Version) . . . . .	915
13.2.1.2.2.15	geru (USM Version) . . . . .	915
13.2.1.2.2.16	hbmv . . . . .	916
13.2.1.2.2.17	hbmv (Buffer Version) . . . . .	917
13.2.1.2.2.18	hbmv (USM Version) . . . . .	917
13.2.1.2.2.19	hemv . . . . .	918
13.2.1.2.2.20	hemv (Buffer Version) . . . . .	919
13.2.1.2.2.21	hemv (USM Version) . . . . .	920
13.2.1.2.2.22	her . . . . .	921
13.2.1.2.2.23	her (Buffer Version) . . . . .	921
13.2.1.2.2.24	her (USM Version) . . . . .	922
13.2.1.2.2.25	her2 . . . . .	923
13.2.1.2.2.26	her2 (Buffer Version) . . . . .	923
13.2.1.2.2.27	her2 (USM Version) . . . . .	924
13.2.1.2.2.28	hpmv . . . . .	925
13.2.1.2.2.29	hpmv (Buffer Version) . . . . .	925
13.2.1.2.2.30	hpmv (USM Version) . . . . .	926
13.2.1.2.2.31	hpr . . . . .	927
13.2.1.2.2.32	hpr (Buffer Version) . . . . .	927
13.2.1.2.2.33	hpr (USM Version) . . . . .	928
13.2.1.2.2.34	hpr2 . . . . .	929
13.2.1.2.2.35	hpr2 (Buffer Version) . . . . .	929
13.2.1.2.2.36	hpr2 (USM Version) . . . . .	930

13.2.1.2.2.37	sbmv . . . . .	931
13.2.1.2.2.38	sbmv (Buffer Version) . . . . .	931
13.2.1.2.2.39	sbmv (USM Version) . . . . .	932
13.2.1.2.2.40	spmv . . . . .	933
13.2.1.2.2.41	spmv (Buffer Version) . . . . .	934
13.2.1.2.2.42	spmv (USM Version) . . . . .	934
13.2.1.2.2.43	spr . . . . .	935
13.2.1.2.2.44	spr (Buffer Version) . . . . .	936
13.2.1.2.2.45	spr (USM Version) . . . . .	936
13.2.1.2.2.46	spr2 . . . . .	937
13.2.1.2.2.47	spr2 (Buffer Version) . . . . .	937
13.2.1.2.2.48	spr2 (USM Version) . . . . .	938
13.2.1.2.2.49	symv . . . . .	939
13.2.1.2.2.50	symv (Buffer Version) . . . . .	939
13.2.1.2.2.51	symv (USM Version) . . . . .	940
13.2.1.2.2.52	syr . . . . .	941
13.2.1.2.2.53	syr (Buffer Version) . . . . .	941
13.2.1.2.2.54	syr (USM Version) . . . . .	942
13.2.1.2.2.55	syr2 . . . . .	943
13.2.1.2.2.56	syr2 (Buffer Version) . . . . .	943
13.2.1.2.2.57	syr2 (USM Version) . . . . .	944
13.2.1.2.2.58	tbmv . . . . .	945
13.2.1.2.2.59	tbmv (Buffer Version) . . . . .	945
13.2.1.2.2.60	tbmv (USM Version) . . . . .	946
13.2.1.2.2.61	tbsv . . . . .	947
13.2.1.2.2.62	tbsv (Buffer Version) . . . . .	947
13.2.1.2.2.63	tbsv (USM Version) . . . . .	948
13.2.1.2.2.64	tpmv . . . . .	949
13.2.1.2.2.65	tpmv (Buffer Version) . . . . .	949
13.2.1.2.2.66	tpmv (USM Version) . . . . .	950
13.2.1.2.2.67	tpsv . . . . .	951
13.2.1.2.2.68	tpsv (Buffer Version) . . . . .	951
13.2.1.2.2.69	tpsv (USM Version) . . . . .	952
13.2.1.2.2.70	trmv . . . . .	953
13.2.1.2.2.71	trmv (Buffer Version) . . . . .	953
13.2.1.2.2.72	trmv (USM Version) . . . . .	954
13.2.1.2.2.73	trsv . . . . .	955
13.2.1.2.2.74	trsv (Buffer Version) . . . . .	955
13.2.1.2.2.75	trsv (USM Version) . . . . .	956
13.2.1.2.3	BLAS Level 3 Routines . . . . .	957
13.2.1.2.3.1	gemm . . . . .	957
13.2.1.2.3.2	gemm (Buffer Version) . . . . .	958
13.2.1.2.3.3	gemm (USM Version) . . . . .	959
13.2.1.2.3.4	hemm . . . . .	960
13.2.1.2.3.5	hemm (Buffer Version) . . . . .	961
13.2.1.2.3.6	hemm (USM Version) . . . . .	962
13.2.1.2.3.7	herk . . . . .	963
13.2.1.2.3.8	herk (Buffer Version) . . . . .	963
13.2.1.2.3.9	herk (USM Version) . . . . .	964
13.2.1.2.3.10	her2k . . . . .	965
13.2.1.2.3.11	her2k (Buffer Version) . . . . .	966
13.2.1.2.3.12	her2k (USM Version) . . . . .	967
13.2.1.2.3.13	symm . . . . .	968
13.2.1.2.3.14	symm (Buffer Version) . . . . .	968

13.2.1.2.3.15	symm (USM Version) . . . . .	969
13.2.1.2.3.16	syrk . . . . .	971
13.2.1.2.3.17	syrk (Buffer Version) . . . . .	971
13.2.1.2.3.18	syrk (USM Version) . . . . .	972
13.2.1.2.3.19	syr2k . . . . .	973
13.2.1.2.3.20	syr2k (Buffer Version) . . . . .	973
13.2.1.2.3.21	syr2k (USM Version) . . . . .	974
13.2.1.2.3.22	trmm . . . . .	975
13.2.1.2.3.23	trmm (Buffer Version) . . . . .	976
13.2.1.2.3.24	trmm (USM Version) . . . . .	977
13.2.1.2.3.25	trsm . . . . .	978
13.2.1.2.3.26	trsm (Buffer Version) . . . . .	979
13.2.1.2.3.27	trsm (USM Version) . . . . .	980
13.2.1.2.4	BLAS-like Extensions . . . . .	981
13.2.1.2.4.1	axpy_batch . . . . .	981
13.2.1.2.4.2	axpy_batch (Buffer Version) . . . . .	981
13.2.1.2.4.3	axpy_batch (USM Version) . . . . .	982
13.2.1.2.4.4	gemm_batch . . . . .	985
13.2.1.2.4.5	gemm_batch (Buffer Version) . . . . .	985
13.2.1.2.4.6	gemm_batch (USM Version) . . . . .	987
13.2.1.2.4.7	trsm_batch . . . . .	990
13.2.1.2.4.8	trsm_batch (Buffer Version) . . . . .	990
13.2.1.2.4.9	trsm_batch (USM Version) . . . . .	992
13.2.1.2.4.10	gemmmt . . . . .	995
13.2.1.2.4.11	gemmmt (Buffer Version) . . . . .	996
13.2.1.2.4.12	gemmmt (USM Version) . . . . .	997
13.2.1.2.4.13	gemm_bias . . . . .	998
13.2.1.2.4.14	gemm_bias (Buffer Version) . . . . .	999
13.2.1.2.4.15	gemm_bias (USM Version) . . . . .	1000
13.2.1.3	LAPACK Routines . . . . .	1001
13.2.1.3.1	LAPACK Linear Equation Routines . . . . .	1002
13.2.1.3.1.1	geqrf . . . . .	1002
13.2.1.3.1.2	geqrf (BUFFER Version) . . . . .	1003
13.2.1.3.1.3	geqrf (USM Version) . . . . .	1004
13.2.1.3.1.4	geqrf_scratchpad_size . . . . .	1005
13.2.1.3.1.5	geqrf_scratchpad_size . . . . .	1005
13.2.1.3.1.6	getrf . . . . .	1006
13.2.1.3.1.7	getrf (BUFFER Version) . . . . .	1006
13.2.1.3.1.8	getrf (USM Version) . . . . .	1007
13.2.1.3.1.9	getrf_scratchpad_size . . . . .	1008
13.2.1.3.1.10	getrf_scratchpad_size . . . . .	1009
13.2.1.3.1.11	getri . . . . .	1009
13.2.1.3.1.12	getri (BUFFER Version) . . . . .	1010
13.2.1.3.1.13	getri (USM Version) . . . . .	1010
13.2.1.3.1.14	getri_scratchpad_size . . . . .	1011
13.2.1.3.1.15	getri_scratchpad_size . . . . .	1012
13.2.1.3.1.16	getrs . . . . .	1012
13.2.1.3.1.17	getrs (BUFFER Version) . . . . .	1013
13.2.1.3.1.18	getrs (USM Version) . . . . .	1014
13.2.1.3.1.19	getrs_scratchpad_size . . . . .	1015
13.2.1.3.1.20	getrs_scratchpad_size . . . . .	1015
13.2.1.3.1.21	orgqr . . . . .	1016
13.2.1.3.1.22	orgqr (BUFFER Version) . . . . .	1017
13.2.1.3.1.23	orgqr (USM Version) . . . . .	1018

13.2.1.3.1.24	orgqr_scratchpad_size . . . . .	1019
13.2.1.3.1.25	orgqr_scratchpad_size . . . . .	1019
13.2.1.3.1.26	ormqr . . . . .	1020
13.2.1.3.1.27	ormqr (BUFFER Version) . . . . .	1020
13.2.1.3.1.28	ormqr (USM Version) . . . . .	1021
13.2.1.3.1.29	ormqr_scratchpad_size . . . . .	1022
13.2.1.3.1.30	ormqr_scratchpad_size . . . . .	1023
13.2.1.3.1.31	potrf . . . . .	1023
13.2.1.3.1.32	potrf (BUFFER Version) . . . . .	1024
13.2.1.3.1.33	potrf (USM Version) . . . . .	1025
13.2.1.3.1.34	potrf_scratchpad_size . . . . .	1026
13.2.1.3.1.35	potrf_scratchpad_size . . . . .	1027
13.2.1.3.1.36	potri . . . . .	1027
13.2.1.3.1.37	potri (BUFFER Version) . . . . .	1028
13.2.1.3.1.38	potri (USM Version) . . . . .	1029
13.2.1.3.1.39	potri_scratchpad_size . . . . .	1030
13.2.1.3.1.40	potri_scratchpad_size . . . . .	1030
13.2.1.3.1.41	potrs . . . . .	1031
13.2.1.3.1.42	potrs (BUFFER Version) . . . . .	1031
13.2.1.3.1.43	potrs (USM Version) . . . . .	1032
13.2.1.3.1.44	potrs_scratchpad_size . . . . .	1034
13.2.1.3.1.45	potrs_scratchpad_size . . . . .	1034
13.2.1.3.1.46	sytrf . . . . .	1035
13.2.1.3.1.47	sytrf (BUFFER Version) . . . . .	1035
13.2.1.3.1.48	sytrf (USM Version) . . . . .	1037
13.2.1.3.1.49	sytrf_scratchpad_size . . . . .	1038
13.2.1.3.1.50	sytrf_scratchpad_size . . . . .	1038
13.2.1.3.1.51	trtrs . . . . .	1039
13.2.1.3.1.52	trtrs (BUFFER Version) . . . . .	1040
13.2.1.3.1.53	trtrs (USM Version) . . . . .	1041
13.2.1.3.1.54	trtrs_scratchpad_size . . . . .	1042
13.2.1.3.1.55	trtrs_scratchpad_size . . . . .	1042
13.2.1.3.1.56	ungqr . . . . .	1043
13.2.1.3.1.57	ungqr (BUFFER Version) . . . . .	1044
13.2.1.3.1.58	ungqr (USM Version) . . . . .	1045
13.2.1.3.1.59	ungqr_scratchpad_size . . . . .	1046
13.2.1.3.1.60	ungqr_scratchpad_size . . . . .	1046
13.2.1.3.1.61	unmqr . . . . .	1047
13.2.1.3.1.62	unmqr (BUFFER Version) . . . . .	1047
13.2.1.3.1.63	unmqr (USM Version) . . . . .	1048
13.2.1.3.1.64	unmqr_scratchpad_size . . . . .	1050
13.2.1.3.1.65	unmqr_scratchpad_size . . . . .	1050
13.2.1.3.2	LAPACK Singular Value and Eigenvalue Problem Routines . . . . .	1051
13.2.1.3.2.1	gebrd . . . . .	1051
13.2.1.3.2.2	gebrd (BUFFER Version) . . . . .	1052
13.2.1.3.2.3	gebrd (USM Version) . . . . .	1053
13.2.1.3.2.4	gebrd_scratchpad_size . . . . .	1054
13.2.1.3.2.5	gebrd_scratchpad_size . . . . .	1055
13.2.1.3.2.6	gesvd . . . . .	1055
13.2.1.3.2.7	gesvd (BUFFER Version) . . . . .	1056
13.2.1.3.2.8	gesvd (USM Version) . . . . .	1058
13.2.1.3.2.9	gesvd_scratchpad_size . . . . .	1060
13.2.1.3.2.10	gesvd_scratchpad_size . . . . .	1060
13.2.1.3.2.11	heevd . . . . .	1061

13.2.1.3.2.12	heevd (BUFFER Version) . . . . .	1062
13.2.1.3.2.13	heevd (USM Version) . . . . .	1063
13.2.1.3.2.14	heevd_scratchpad_size . . . . .	1064
13.2.1.3.2.15	heevd_scratchpad_size . . . . .	1064
13.2.1.3.2.16	hegvd . . . . .	1065
13.2.1.3.2.17	hegvd (BUFFER Version) . . . . .	1065
13.2.1.3.2.18	hegvd (USM Version) . . . . .	1067
13.2.1.3.2.19	hegvd_scratchpad_size . . . . .	1069
13.2.1.3.2.20	hegvd_scratchpad_size . . . . .	1069
13.2.1.3.2.21	hetrd . . . . .	1070
13.2.1.3.2.22	hetrd (BUFFER Version) . . . . .	1070
13.2.1.3.2.23	hetrd (USM Version) . . . . .	1071
13.2.1.3.2.24	hetrd_scratchpad_size . . . . .	1073
13.2.1.3.2.25	hetrd_scratchpad_size . . . . .	1073
13.2.1.3.2.26	orgbr . . . . .	1074
13.2.1.3.2.27	orgbr (BUFFER Version) . . . . .	1075
13.2.1.3.2.28	orgbr (USM Version) . . . . .	1076
13.2.1.3.2.29	orgbr_scratchpad_size . . . . .	1077
13.2.1.3.2.30	orgbr_scratchpad_size . . . . .	1077
13.2.1.3.2.31	orgtr . . . . .	1078
13.2.1.3.2.32	orgtr (BUFFER Version) . . . . .	1079
13.2.1.3.2.33	orgtr (USM Version) . . . . .	1079
13.2.1.3.2.34	orgtr_scratchpad_size . . . . .	1080
13.2.1.3.2.35	orgtr_scratchpad_size . . . . .	1081
13.2.1.3.2.36	ormtr . . . . .	1081
13.2.1.3.2.37	ormtr (BUFFER Version) . . . . .	1082
13.2.1.3.2.38	ormtr (USM Version) . . . . .	1083
13.2.1.3.2.39	ormtr_scratchpad_size . . . . .	1084
13.2.1.3.2.40	ormtr_scratchpad_size . . . . .	1085
13.2.1.3.2.41	syevd . . . . .	1086
13.2.1.3.2.42	syevd (BUFFER Version) . . . . .	1086
13.2.1.3.2.43	syevd (USM Version) . . . . .	1087
13.2.1.3.2.44	syevd_scratchpad_size . . . . .	1088
13.2.1.3.2.45	syevd_scratchpad_size . . . . .	1089
13.2.1.3.2.46	sygvd . . . . .	1090
13.2.1.3.2.47	sygvd (BUFFER Version) . . . . .	1090
13.2.1.3.2.48	sygvd (USM Version) . . . . .	1092
13.2.1.3.2.49	sygvd_scratchpad_size . . . . .	1094
13.2.1.3.2.50	sygvd_scratchpad_size . . . . .	1094
13.2.1.3.2.51	sytrd . . . . .	1095
13.2.1.3.2.52	sytrd (BUFFER Version) . . . . .	1095
13.2.1.3.2.53	sytrd (USM Version) . . . . .	1096
13.2.1.3.2.54	sytrd_scratchpad_size . . . . .	1098
13.2.1.3.2.55	sytrd_scratchpad_size . . . . .	1098
13.2.1.3.2.56	ungbr . . . . .	1099
13.2.1.3.2.57	ungbr (BUFFER Version) . . . . .	1099
13.2.1.3.2.58	ungbr (USM Version) . . . . .	1101
13.2.1.3.2.59	ungbr_scratchpad_size . . . . .	1102
13.2.1.3.2.60	ungbr_scratchpad_size . . . . .	1102
13.2.1.3.2.61	ungtr . . . . .	1103
13.2.1.3.2.62	ungtr (BUFFER Version) . . . . .	1104
13.2.1.3.2.63	ungtr (USM Version) . . . . .	1104
13.2.1.3.2.64	ungtr_scratchpad_size . . . . .	1105
13.2.1.3.2.65	ungtr_scratchpad_size . . . . .	1106

13.2.1.3.2.66	unmtr . . . . .	1106
13.2.1.3.2.67	unmtr (BUFFER Version) . . . . .	1107
13.2.1.3.2.68	unmtr (USM Version) . . . . .	1108
13.2.1.3.2.69	unmtr_scratchpad_size . . . . .	1109
13.2.1.3.2.70	unmtr_scratchpad_size . . . . .	1110
13.2.1.3.3	LAPACK-like Extensions Routines . . . . .	1111
13.2.1.3.3.1	geqrf_batch . . . . .	1111
13.2.1.3.3.2	geqrf_batch (BUFFER Version) . . . . .	1112
13.2.1.3.3.3	getrf_batch . . . . .	1112
13.2.1.3.3.4	getrf_batch (BUFFER Version) . . . . .	1113
13.2.1.3.3.5	getri_batch . . . . .	1114
13.2.1.3.3.6	getri_batch (BUFFER Version) . . . . .	1114
13.2.1.3.3.7	getrs_batch . . . . .	1115
13.2.1.3.3.8	getrs_batch (BUFFER Version) . . . . .	1115
13.2.1.3.3.9	orgqr_batch . . . . .	1116
13.2.1.3.3.10	orgqr_batch (BUFFER Version) . . . . .	1117
13.2.1.3.3.11	potrf_batch . . . . .	1118
13.2.1.3.3.12	potrf_batch (BUFFER Version) . . . . .	1118
13.2.1.3.3.13	potrs_batch . . . . .	1119
13.2.1.3.3.14	potrs_batch (BUFFER Version) . . . . .	1119
13.2.2	Sparse Linear Algebra . . . . .	1121
13.2.2.1	Sparse BLAS Routines . . . . .	1121
13.2.2.1.1	onemkl::sparse::matrixInit . . . . .	1121
13.2.2.1.2	onemkl::sparse::setCSRstructure . . . . .	1122
13.2.2.1.3	onemkl::sparse::gemm . . . . .	1123
13.2.2.1.4	onemkl::sparse::gemv . . . . .	1125
13.2.2.1.5	onemkl::sparse::gemvdot . . . . .	1127
13.2.2.1.6	onemkl::sparse::gemvOptimize . . . . .	1128
13.2.2.1.7	onemkl::sparse::symv . . . . .	1129
13.2.2.1.8	onemkl::sparse::trmv . . . . .	1131
13.2.2.1.9	onemkl::sparse::trmvOptimize . . . . .	1133
13.2.2.1.10	onemkl::sparse::trsv . . . . .	1134
13.2.2.1.11	onemkl::sparse::trsvOptimize . . . . .	1136
13.2.2.1.12	Supported Types . . . . .	1137
13.2.2.1.13	Exceptions . . . . .	1137
13.2.3	Discrete Fourier Transforms . . . . .	1137
13.2.3.1	Fourier Transform Functions . . . . .	1138
13.2.3.1.1	onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain> . .	1138
13.2.3.1.2	onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::Init	1139
13.2.3.1.3	onemkl::dft::Descriptor<onemkl::dft::Precision,	
onemkl::dft::Domain>::setValue . . . . .	1140	
13.2.3.1.4	onemkl::dft::Descriptor<onemkl::dft::Precision,	
onemkl::dft::Domain>::getValue . . . . .	1141	
13.2.3.1.5	onemkl::dft::Descriptor<onemkl::dft::Precision,	
onemkl::dft::Domain>::commit . . . . .	1143	
13.2.3.1.6	onemkl::dft::Descriptor<onemkl::dft::Precision,	
onemkl::dft::Domain>::computeForward<typename IOType> . . . . .	1144	
13.2.3.1.7	onemkl::dft::Descriptor<onemkl::dft::Precision,	
onemkl::dft::Domain>::computeBackward<typename IOType> . . . . .	1146	
13.2.4	Random Number Generators . . . . .	1147
13.2.4.1	oneMKL RNG Usage Model . . . . .	1148
13.2.4.2	Generate Routine . . . . .	1150
13.2.4.2.1	onemkl::rng::generate . . . . .	1151
13.2.4.3	Engines (Basic Random Number Generators) . . . . .	1152

13.2.4.3.1	onemkl::rng::mrg32k3a . . . . .	1153
13.2.4.3.2	onemkl::rng::philox4x32x10 . . . . .	1154
13.2.4.3.3	onemkl::rng::mcg31m1 . . . . .	1155
13.2.4.3.4	onemkl::rng::mcg59 . . . . .	1155
13.2.4.3.5	onemkl::rng::r250 . . . . .	1156
13.2.4.3.6	onemkl::rng::wichmann_hill . . . . .	1157
13.2.4.3.7	onemkl::rng::mt19937 . . . . .	1158
13.2.4.3.8	onemkl::rng::sfmt19937 . . . . .	1159
13.2.4.3.9	onemkl::rng::mt2203 . . . . .	1159
13.2.4.3.10	onemkl::rng::ars5 . . . . .	1160
13.2.4.3.11	onemkl::rng::sobol . . . . .	1161
13.2.4.3.12	onemkl::rng::niederreiter . . . . .	1162
13.2.4.3.13	onemkl::rng::nondeterministic . . . . .	1163
13.2.4.4	Service Routines . . . . .	1164
13.2.4.4.1	onemkl::rng::leapfrog . . . . .	1164
13.2.4.4.2	onemkl::rng::skip_ahead . . . . .	1165
13.2.4.5	Distributions . . . . .	1168
13.2.4.5.1	Distributions Template Parameter onemkl::rng::method Values . . . . .	1171
13.2.4.5.2	onemkl::rng::uniform (Continuous) . . . . .	1172
13.2.4.5.3	onemkl::rng::gaussian . . . . .	1173
13.2.4.5.4	onemkl::rng::exponential . . . . .	1175
13.2.4.5.5	onemkl::rng::laplace . . . . .	1176
13.2.4.5.6	onemkl::rng::weibull . . . . .	1178
13.2.4.5.7	onemkl::rng::cauchy . . . . .	1179
13.2.4.5.8	onemkl::rng::rayleigh . . . . .	1181
13.2.4.5.9	onemkl::rng::lognormal . . . . .	1182
13.2.4.5.10	onemkl::rng::gumbel . . . . .	1184
13.2.4.5.11	onemkl::rng::gamma . . . . .	1185
13.2.4.5.12	onemkl::rng::beta . . . . .	1186
13.2.4.5.13	onemkl::rng::chi_square . . . . .	1188
13.2.4.5.14	onemkl::rng::uniform (Discrete) . . . . .	1189
13.2.4.5.15	onemkl::rng::uniform_bits . . . . .	1190
13.2.4.5.16	onemkl::rng::bits . . . . .	1191
13.2.4.5.17	onemkl::rng::bernoulli . . . . .	1191
13.2.4.5.18	onemkl::rng::geometric . . . . .	1193
13.2.4.5.19	onemkl::rng::binomial . . . . .	1194
13.2.4.5.20	onemkl::rng::hypergeometric . . . . .	1195
13.2.4.5.21	onemkl::rng::poisson . . . . .	1197
13.2.4.5.22	onemkl::rng::poisson_v . . . . .	1198
13.2.4.5.23	onemkl::rng::negbinomial . . . . .	1199
13.2.4.5.24	onemkl::rng::multinomial . . . . .	1200
13.2.4.6	Bibliography . . . . .	1202
13.2.5	Vector Math . . . . .	1203
13.2.5.1	Special Value Notations . . . . .	1203
13.2.5.2	VM Mathematical Functions . . . . .	1204
13.2.5.2.1	Arithmetic Functions . . . . .	1206
13.2.5.2.1.1	add . . . . .	1206
13.2.5.2.1.2	sub . . . . .	1208
13.2.5.2.1.3	sqr . . . . .	1210
13.2.5.2.1.4	mul . . . . .	1211
13.2.5.2.1.5	mulbyconj . . . . .	1214
13.2.5.2.1.6	conj . . . . .	1215
13.2.5.2.1.7	abs . . . . .	1216
13.2.5.2.1.8	arg . . . . .	1218

13.2.5.2.1.9	linearfrac . . . . .	1220
13.2.5.2.1.10	fmod . . . . .	1222
13.2.5.2.1.11	remainder . . . . .	1224
13.2.5.2.2	Power and Root Functions . . . . .	1226
13.2.5.2.2.1	inv . . . . .	1226
13.2.5.2.2.2	div . . . . .	1228
13.2.5.2.2.3	sqrt . . . . .	1230
13.2.5.2.2.4	invsqrt . . . . .	1232
13.2.5.2.2.5	cbrt . . . . .	1234
13.2.5.2.2.6	invcbrt . . . . .	1235
13.2.5.2.2.7	pow2o3 . . . . .	1237
13.2.5.2.2.8	pow3o2 . . . . .	1238
13.2.5.2.2.9	pow . . . . .	1240
13.2.5.2.2.10	powx . . . . .	1243
13.2.5.2.2.11	powr . . . . .	1245
13.2.5.2.2.12	hypot . . . . .	1247
13.2.5.2.3	Exponential and Logarithmic Functions . . . . .	1249
13.2.5.2.3.1	exp . . . . .	1249
13.2.5.2.3.2	exp2 . . . . .	1251
13.2.5.2.3.3	exp10 . . . . .	1253
13.2.5.2.3.4	expm1 . . . . .	1255
13.2.5.2.3.5	ln . . . . .	1256
13.2.5.2.3.6	log2 . . . . .	1258
13.2.5.2.3.7	log10 . . . . .	1260
13.2.5.2.3.8	log1p . . . . .	1262
13.2.5.2.3.9	logb . . . . .	1264
13.2.5.2.4	Trigonometric Functions . . . . .	1265
13.2.5.2.4.1	cos . . . . .	1266
13.2.5.2.4.2	sin . . . . .	1268
13.2.5.2.4.3	sincos . . . . .	1270
13.2.5.2.4.4	cis . . . . .	1271
13.2.5.2.4.5	tan . . . . .	1273
13.2.5.2.4.6	acos . . . . .	1275
13.2.5.2.4.7	asin . . . . .	1277
13.2.5.2.4.8	atan . . . . .	1279
13.2.5.2.4.9	atan2 . . . . .	1280
13.2.5.2.4.10	cospi . . . . .	1282
13.2.5.2.4.11	sinpi . . . . .	1284
13.2.5.2.4.12	tanpi . . . . .	1286
13.2.5.2.4.13	acosp . . . . .	1288
13.2.5.2.4.14	asinpi . . . . .	1290
13.2.5.2.4.15	atanpi . . . . .	1292
13.2.5.2.4.16	atan2pi . . . . .	1293
13.2.5.2.4.17	cosd . . . . .	1295
13.2.5.2.4.18	sind . . . . .	1297
13.2.5.2.4.19	tand . . . . .	1299
13.2.5.2.5	Hyperbolic Functions . . . . .	1301
13.2.5.2.5.1	cosh . . . . .	1301
13.2.5.2.5.2	sinh . . . . .	1303
13.2.5.2.5.3	tanh . . . . .	1306
13.2.5.2.5.4	acosh . . . . .	1307
13.2.5.2.5.5	asinh . . . . .	1310
13.2.5.2.5.6	atanh . . . . .	1311
13.2.5.2.6	Special Functions . . . . .	1314

13.2.5.2.6.1	erf . . . . .	1314
13.2.5.2.6.2	erfc . . . . .	1317
13.2.5.2.6.3	cdfnorm . . . . .	1320
13.2.5.2.6.4	erfinv . . . . .	1323
13.2.5.2.6.5	erfcinv . . . . .	1326
13.2.5.2.6.6	cdfnorminv . . . . .	1329
13.2.5.2.6.7	lgamma . . . . .	1332
13.2.5.2.6.8	tgamma . . . . .	1333
13.2.5.2.6.9	expint1 . . . . .	1335
13.2.5.2.7	Rounding Functions . . . . .	1337
13.2.5.2.7.1	floor . . . . .	1337
13.2.5.2.7.2	ceil . . . . .	1339
13.2.5.2.7.3	trunc . . . . .	1341
13.2.5.2.7.4	round . . . . .	1342
13.2.5.2.7.5	nearbyint . . . . .	1344
13.2.5.2.7.6	rint . . . . .	1345
13.2.5.2.7.7	modf . . . . .	1347
13.2.5.2.7.8	frac . . . . .	1349
13.2.5.3	VM Service Functions . . . . .	1351
13.2.5.3.1	setmode . . . . .	1351
13.2.5.3.2	get_mode . . . . .	1352
13.2.5.3.3	set_status . . . . .	1353
13.2.5.3.4	get_status . . . . .	1355
13.2.5.3.5	clear_status . . . . .	1356
13.2.5.3.6	create_error_handler . . . . .	1357
13.2.5.4	Miscellaneous VM Functions . . . . .	1361
13.2.5.4.1	copysign . . . . .	1361
13.2.5.4.2	nextafter . . . . .	1362
13.2.5.4.3	fdim . . . . .	1364
13.2.5.4.4	fmax . . . . .	1366
13.2.5.4.5	fmin . . . . .	1367
13.2.5.4.6	maxmag . . . . .	1369
13.2.5.4.7	minmag . . . . .	1370
13.2.5.5	Bibliography . . . . .	1372
14	Contributors	1373
15	HTML and PDF Versions	1374
15.1	Release Notes . . . . .	1374
15.1.1	0.85 . . . . .	1374
15.1.2	0.8 . . . . .	1375
15.1.3	0.7 . . . . .	1375
15.1.4	0.5 . . . . .	1375
16	Legal Notices and Disclaimers	1376
17	Page Not Found	1377
	Bibliography	1378
	Index	1379

## INTRODUCTION

oneAPI is an open, free, and standards-based programming system that provides portability and performance across accelerators and generations of hardware. oneAPI consists of a language and libraries for creating parallel applications:

- *DPC++*: oneAPI’s core language for programming accelerators and multiprocessors. DPCPP allows developers to reuse code across hardware targets (CPUs and accelerators such as GPUs and FPGAs) and tune for a specific architecture
- *oneDPL*: A companion to the DPC++ Compiler for programming oneAPI devices with APIs from C++ standard library, Parallel STL, and extensions.
- *oneDNN*: High performance implementations of primitives for deep learning frameworks
- *oneCCL*: Communication primitives for scaling deep learning frameworks across multiple devices
- *Level Zero*: System interface for oneAPI languages and libraries
- *oneDAL*: Algorithms for accelerated data science
- *oneTBB*: Library for adding thread-based parallelism to complex applications on multiprocessors
- *oneVPL*: Algorithms for accelerated video processing
- *oneMKL*: High performance math routines for science, engineering, and financial applications

oneAPI simplifies software development by providing the same languages and programming models across accelerator architectures. In this section, we introduce the programming model.

Parallel application development is a combination of *API programming*, where the parallel algorithm is hidden behind an API provided by the system, and *direct programming*, where the application programmer writes the parallel algorithm.

When using API programming, a developer implements performance critical sections of the program with library calls. Well-defined and mature problem domains have high-performance solutions packaged as libraries. oneAPI defines a set of APIs for the most used data parallel domains, and oneAPI platforms provide library implementations across a variety of accelerators. Where possible, the API is based on established standards like BLAS. API programming enables a programmer to attain high performance across a diverse set of accelerators with minimal coding & tuning.

Some problem domains are not well suited to API programming because no standard solution exists or because solutions require a level of customization that cannot be easily implemented in a library. In this case, a developer uses direct programming and must explicitly code the parallel algorithm. oneAPI’s programming model is based on data parallelism, where the same computation is performed on each data element, and parallelism of the application scales as the data scales. By allowing the programmer to directly express parallelism, data parallel algorithms make it possible to productively create highly efficient algorithms for parallel architectures.

Data parallel algorithms are used for many of the most computationally demanding problems including scientific computing, artificial intelligence, and visualization. Data parallel algorithms can be efficiently mapped to a diverse set of architectures: multi-core CPUs, GPUs, systolic arrays, and FPGAs.

## 1.1 Target Audience

The expected audience for this specification includes: application developers, middleware developers, system software providers, and hardware providers. As a *contributor* to this specification, you will shape the accelerator software ecosystem. A productive and high performing system must take into account the constraints at all levels of the software stack. As a *user* of this document, you can ensure that your components will inter-operate with applications and system software for the oneAPI platform.

## 1.2 Goals of the Specification

oneAPI seeks to provide:

- *Source-level compatibility*: oneAPI applications and middleware port to a conformant oneAPI platform through recompilation and re-tuning.
- *Performance transparency*: API's and language construct allow the programmer enough control over the mapping to hardware to create an efficient solution
- *Software stack portability*: Platform providers can port a oneAPI software stack by implementing the oneAPI Level Zero interface.

## 1.3 Definitions

This specification uses the definition of must, must not, required, and so on specified in RFC 2119.

## 1.4 Contribution Guidelines

This specification is a continuation of Intel's decades-long history of working with standards groups and industry/academia initiatives such as The Khronos Group, to create and define specifications in an open and fair process to achieve interoperability and interchangeability. oneAPI is intended to be an open specification and we encourage you to help us make it better. Your feedback is optional, but to enable Intel to incorporate any feedback you may provide to this specification, and to further upstream your feedback to other standards bodies, including The Khronos Group SYCL specification, please submit your feedback under the terms and conditions below. Any contribution of your feedback to the oneAPI Specification does not prohibit you from also contributing your feedback directly to other standard bodies, including The Khronos Group under their respective submission policies.

Contribute to the oneAPI Specification by opening issues in the oneAPI Specification [GitHub repository](#).

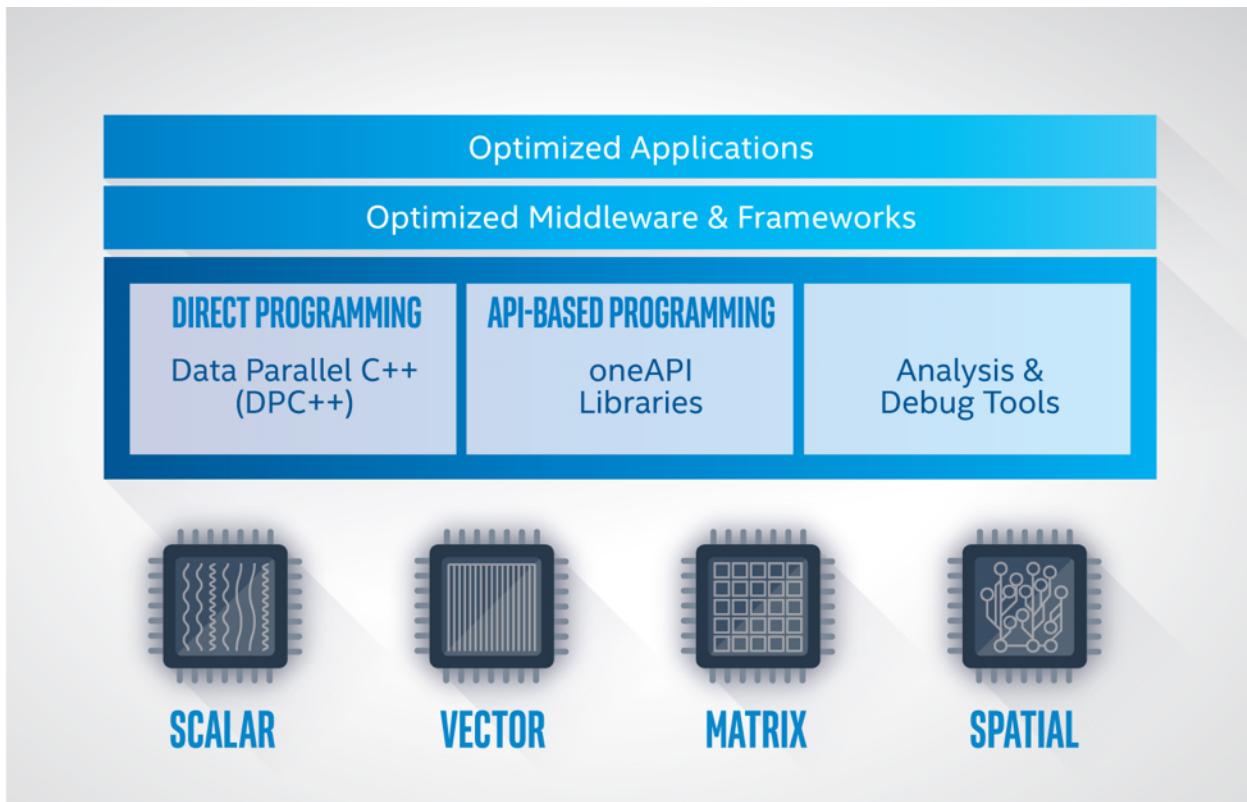
### 1.4.1 Sign your work

Please include a signed-off-by tag in every contribution of your feedback. By including a signed-off-by tag, you agree that: (a) you have a right to license your feedback to Intel; (b) Intel will be free to use, disclose, reproduce, modify, license, or otherwise distribute your feedback at its sole discretion without any obligations or restrictions of any kind, including without limitation, intellectual property rights or licensing obligations; and (c) your feedback will be public and that a record of your feedback may be maintained indefinitely.

If you agree to the above, every contribution of your feedback must include the following line using your real name and email address: Signed-off-by: Joe Smith [joe.smith@email.com](mailto:joe.smith@email.com)

## SOFTWARE ARCHITECTURE

oneAPI provides a common developer interface across a range of data parallel accelerators (see the figure below). Programmers use DPC++ for both API programming and direct programming. The capabilities of a oneAPI platform are determined by the Level Zero interface, which provides system software a common abstraction for a oneAPI device.



## 2.1 oneAPI Platform

A oneAPI platform is comprised of a *host* and a collection of *devices*. The host is typically a multi-core CPU, and the devices are one or more GPUs, FPGAs, and other accelerators. The processor serving as the host can also be targeted as a device by the software.

Each device has an associated command *queue*. A application that employs oneAPI runs on the host, following standard C++ execution semantics. To run a *function object* on a device, the application submits a *command group* containing the function object to the device's queue. A function object contains a function definition together with associated variables. A function object submitted to a queue is also referred to as a *data parallel kernel* or simply a *kernel*.

The application running on the host and the functions running on the devices communicate through *memory*. oneAPI defines several mechanisms for sharing memory across the platform, depending on the capabilities of the devices:

Memory Sharing Mechanism	Description
Buffer objects	<p>An application can create <i>buffer objects</i> to pass data to devices. A buffer is an array of data. A command group will define <i>accessor objects</i> to identify which buffers are accessed in this call to the device. The oneAPI runtime will ensure the data in the buffer is accessible to the function running on the device. The buffer-accessor mechanism is available on all oneAPI platforms</p>
Unified addressing	<p>Unified addressing guarantees that the host and all devices will share a unified address space. Pointer values in the unified address space will always refer to the same location in memory.</p>
Unified shared memory	<p>Unified shared memory enables data to be shared through pointers without using buffers and accessors. There are several levels of support for this feature, depending on the capabilities of the underlying device.</p>

The *scheduler* determines when a command group is run on a device. The following mechanisms are used to determine when a command group is ready to run.

- If the buffer-accessor method is used, the command group is ready when the buffers are defined and copied to the device as necessary.
- If an ordered queue is used for a device, the command group is ready as soon as the prior command groups in the queue are finished.

- If unified shared memory is used, you must specify a set of event objects which the command group depends on, and the command group is ready when all of the events are completed.

The application on the host and the functions on the devices can *synchronize* through *events*, which are objects that can coordinate execution. If the buffer-accessor mechanism is used, the application and device can also synchronize through a *host accessor*, through the destruction of a buffer object, or through other more advanced mechanisms.

## 2.2 API Programming Example

API programming requires the programmer to specify the target device and the memory communication strategy. In the following example, we call the oneMKL matrix multiply routine, GEMM. We are writing in DPC++ and omitting irrelevant details.

We create a queue initialized with a *gpu\_selector* to specify that we want the computation performed on a GPU, and we define buffers to hold the arrays allocated on the host. Compared to a standard C++ GEMM call, we add a parameter to specify the queue, and we replace the references to the arrays with references to the buffers that contain the arrays. Otherwise this is the standard GEMM C++ interface.

```
using namespace cl::sycl;

// declare host arrays
double *A = new double[M*N];
double *B = new double[N*P];
double *C = new double[M*P];

{
    // Initializing the devices queue with a gpu_selector
    queue q{gpu_selector()};

    // Creating 1D buffers for matrices which are bound to host arrays
    buffer<double, 1> a{A, range<1>{M*N}};
    buffer<double, 1> b{B, range<1>{N*P}};
    buffer<double, 1> c{C, range<1>{M*P}};

    mkl::transpose nT = mkl::transpose::nontrans;
    // Syntax
    // void gemm(queue &exec_queue, transpose transa, transpose transb,
    //           int64_t m, int64_t n, int64_t k, T alpha,
    //           buffer<T,1> &a, int64_t lda,
    //           buffer<T,1> &b, int64_t ldb, T beta,
    //           buffer<T,1> &c, int64_t ldc);
    // call gemm
    mkl::blas::gemm(q, nT, nT, M, P, N, 1.0, a, M, b, N, 0.0, c, M);
}
// when we exit the block, the buffer destructor will write result back to C.
```

## 2.3 Direct Programming Example

With direct programming, we specify the target device and the memory communication strategy, as we do for API programming. In addition, we must define and submit a command group to perform the computation. In the following example, we write a simple data parallel matrix multiply. We are writing in DPC++ and omitting irrelevant details.

We create a queue initialized with a *gpu\_selector* to specify that the command group should run on the GPU, and we define buffers to hold the arrays allocated on the host. We then submit the command group to the queue to perform the computation. The command group defines accessors to specify we are reading arrays A and B and writing to C. We then write a C++ lambda to create a function object that computes one element of the resulting matrix multiply. We specify this function object as a parameter to a *parallel\_for* which maps the function across the arrays A and B in parallel. When we leave the scope, the destructor for the buffer object holding C writes the data back to the host array.

```
using namespace cl::sycl;

// declare host arrays
double *Ahost = new double[M*N];
double *Bhost = new double[N*P];
double *Chost = new double[M*P];

{
    // Initializing the devices queue with a gpu_selector
    queue q{gpu_selector()};

    // Creating 2D buffers for matrices which are bound to host arrays
    buffer<double, 2> a{Ahost, range<2>{M,N}};
    buffer<double, 2> b{Bhost, range<2>{N,P}};
    buffer<double, 2> c{Chost, range<2>{M,P}};

    // Submitting command group to queue to compute matrix c=a*b
    q.submit([&](handler &h){
        // Read from a and b, write to c
        auto A = a.get_access<access::mode::read>(h);
        auto B = b.get_access<access::mode::read>(h);
        auto C = c.get_access<access::mode::write>(h);

        int WidthA = a.get_range()[1];

        // Executing kernel
        h.parallel_for<class MatrixMult>(range<2>{M, P}, [=](id<2> index){
            int row = index[0];
            int col = index[1];

            // Compute the result of one element in c
            double sum = 0.0;
            for (int i = 0; i < WidthA; i++) {
                sum += A[row][i] * B[i][col];
            }
            C[index] = sum;
        });
    });
}

// when we exit the block, the buffer destructor will write result back to C.
```

## LIBRARY INTEROPERABILITY

Libraries support API programming. oneAPI libraries meet the following requirements to ensure interoperability and provide full platform performance.

### 3.1 Queueing

APIs that launch a computation on a device allow the programmer to control where the computation is performed by passing a queue. A queue can be passed with every invocation or it can be passed once and retained in a library's internal state.

### 3.2 API Arguments

When an API accepts a buffer as an argument there is an equivalent API that accepts a unified-shared memory (USM) pointer. Some API's accept a USM pointer but do not have variants that accept buffers.

APIs document when an argument is accessed from a device and when an argument is accessed by the host. Behavior is undefined if you pass a device argument that is not accessible by the device (non-USM pointer, for example), or a host argument that is not accessible by the host (`malloc_device`, for example).

### 3.3 Asynchronous APIs

To achieve the best performance on a oneAPI platform, the host and devices should execute concurrently. Concurrent execution is supported via asynchronous APIs that queue one or more kernels and immediately return control to the host. Synchronous APIs stall the host thread until the API is complete.

Scheduling of an asynchronous oneAPI library call is controlled by two mechanisms:

- APIs that accept buffers rely on the buffer/Accessor mechanism in the oneAPI runtime to schedule computation on the host and devices.
- APIs that accept USM pointers accept a vector of prerequisite events and the scheduler waits on the events before submitting the call for execution on the device. The API returns an event that another kernel or library API can wait on to ensure the output data is ready.

## 3.4 Exceptions

This specification recommends that oneAPI library APIs signal errors by throwing C++ exceptions. Some APIs use alternative methods for error reporting, due to legacy requirements.

## ONEAPI ELEMENTS

In the following sections, we define the elements of oneAPI in more detail. They are the DPC++ language and libraries that use DPC++ to offload computation to an accelerator. The libraries provide domain-specific algorithms covering artificial intelligence (AI), high- performance computing (HPC), analytics, video processing, and cross-domain libraries for parallel algorithms and threading.

oneAPI elements include:

- *DPC++*: oneAPI’s core language for programming accelerators and multiprocessors. DPCPP allows developers to reuse code across hardware targets (CPUs and accelerators such as GPUs and FPGAs) and tune for a specific architecture
- *oneDPL*: A companion to the DPC++ Compiler for programming oneAPI devices with APIs from C++ standard library, Parallel STL, and extensions.
- *oneDNN*: High performance implementations of primitives for deep learning frameworks
- *oneCCL*: Communication primitives for scaling deep learning frameworks across multiple devices
- *Level Zero*: System interface for oneAPI languages and libraries
- *oneDAL*: Algorithms for accelerated data science
- *oneTBB*: Library for adding thread-based parallelism to complex applications on multiprocessors
- *oneVPL*: Algorithms for accelerated video processing
- *oneMKL*: High performance math routines for science, engineering, and financial applications

For each element, we give an overview of the functionality, provide detailed information on the APIs, and provide links to tests and open source implementations.

## 5.1 Overview

oneAPI Data Parallel C++ (DPC++) is the direct programming language and associated direct programming APIs of oneAPI. It provides the features needed to define data parallel functions and to launch them on devices. The language is comprised of the following components:

- C++. Every DPC++ program is also a C++ program. A compliant DPC++ implementation must support the C++17 Core Language (as specified in Sections 1-19 of ISO/IEC 14882:2017) or newer. See the [C++ Standard](#).
- SYCL. DPC++ builds on the SYCL specification from The Khronos Group. The SYCL language enables the definition of data parallel functions that can be offloaded to devices and defines runtime APIs and classes that are used to orchestrate the offloaded functions.
- DPC++ Language extensions. A compliant DPC++ implementation must support the specified language features. These include unified shared memory (USM), ordered queues, and reductions. Some extensions are required only when the DPC++ implementation supports a specific class of device, as summarized in the [Extensions Table](#). An implementation supports a class of device if it can target hardware that responds “true” for a DPC++ device type query, either through explicit support built into the implementation, or by using a lower layer that can support those device classes such as the oneAPI Level Zero (Level Zero). A DPC++ implementation must pass the conformance tests for all extensions that are required ([Extensions Table](#)) for the classes of devices that the implementation can support. (See [SYCL Extensions](#).)

This specification requires a minimum of C++17 Core Language support and DPC++ extensions. These version and feature coverage requirements will evolve over time, with specific versions of C++ and SYCL being required, some additional extensions being required, and some DPC++ extensions no longer required if covered by newer C++ or SYCL versions directly.

Table 1: DPC++ Extensions Table: Support requirements for DPC++ implementations supporting specific classes of devices

Extension	CPU	GPU	FPGA	Test <sup>1</sup>
Unified Shared Memory	Required <sup>2</sup>	Required <sup>2</sup>	Required <sup>2</sup>	usm
In-order queues	Required	Required	Required	NA <sup>3</sup>
Optional lambda name	Required	Required	Required	NA <sup>3</sup>
Deduction guides	Required	Required	Required	NA <sup>3</sup>
Reductions	Required	Required	Required	NA <sup>3</sup>
Sub-groups	Required	Required	Not required <sup>4</sup>	sub_group
Sub-group algorithms	Required	Required	Not required <sup>4</sup>	sub_group
Enqueued barriers	Required	Required	Required	NA
Extended atomics	Required	Required	Required	NA
Group algorithms	Required	Required	Required	NA
Group mask	Required	Required	Required	NA
Restrict all arguments	Required	Required	Required	NA
Standard layout relaxed	Required	Required	Required	NA
Queue shortcuts	Required	Required	Required	NA
Reqd work-group size	Required	Required	Required	NA
Data flow pipes	Not required	Not required	Required	fpga_tests

## 5.2 Detailed API and Language Descriptions

The SYCL Specification describes the SYCL APIs and language. DPC++ extensions on top of SYCL are described in the [SYCL Extensions](#) repository.

A brief summary of the extensions is as follows:

- Unified Shared Memory (USM) - defines pointer based memory accesses and management interfaces. Provides the ability to create allocations that are visible and have consistent pointer values across both host and device(s). Different USM capability levels are defined, corresponding to different levels of device and implementation support.
- In-order queues - defines simple in-order semantics for queues, to simplify common coding patterns.
- Optional lambda name - removes requirement to manually name lambdas that define kernels. Simplifies coding and enables composability with libraries. Lambdas can still be manually named, if desired, such as when debugging or interfacing with a `sycl::program` object.
- Deduction guides - simplifies common code patterns and reduces code length and verbosity by enabling Class Template Argument Deduction (CTAD) from modern C++.
- Reductions - provides a reduction abstraction to the ND-range form of `parallel_for`. Improves productivity by providing the common reduction pattern without explicit coding, and enables optimized implementations to exist for combinations of device, runtime, and reduction properties.
- Subgroups - defines a grouping of work-items within a work-group. Synchronization of work-items in a subgroup can occur independently of work-items in other subgroups, and subgroups expose communication operations across work-items in the group. Subgroups commonly map to SIMD hardware where it exists.

<sup>1</sup> Test directory within [extension tests](#)

<sup>2</sup> Minimum of explicit USM support

<sup>3</sup> Not yet available.

<sup>4</sup> Likely to be required in the future

- Subgroup algorithms - defines collective operations across work-items in a sub-group that are available only for sub-groups. Also enables algorithms from the more generic “group algorithms” extension as sub-group collective operations.
- Enqueued barriers - simplifies dependence creation and tracking for some common programming patterns by allowing coarser grained synchronization within a queue without manual creation of fine grained dependencies.
- Extended atomics - provides atomic operations aligned with C++20, including support for floating-point types and shorthand operators
- Group algorithms - defines collective operations that operate across groups of work-items, including broadcast, reduce, and scan. Improves productivity by providing common algorithms without explicit coding, and enables optimized implementations to exist for combinations of device and runtime.
- Group mask - defines a type that can represent a set of work-items from a group, and collective operations that create or operate on that type such as ballot and count.
- Restrict all arguments - defines an attribute that can be applied to kernels (including lambda definitions of kernels) which signals that there will be no memory aliasing between any pointer arguments that are passed to or captured by a kernel. This is an optimization attribute that can have large impact when the developer knows more about the kernel arguments than a compiler can infer or safely assume.
- Standard layout relaxed - removes the requirement that data shared by a host and device(s) must be C++ standard layout types. Requires device compilers to validate layout compatibility.
- Queue shortcuts - defines kernel invocation functions directly on the queue classes, to simplify code patterns where dependencies and/or accessors do not need to be created within the additional command group scope. Reduces code verbosity in some common patterns.
- Required work-group size - defines an attribute that can be applied to kernels (including lambda definitions of kernels) which signals that the kernel will only be invoked with a specific work-group size. This is an optimization attribute that enables optimizations based on additional user-driven information.
- Data flow pipes - enables efficient First-In, First-Out (FIFO) communication in DPC++, a mechanism commonly used when describing algorithms for spatial architectures such as FPGAs.

## 5.3 Open Source Implementation

An open source implementation is available under an LLVM license. Details on incomplete features and known issues are available in the [Release Notes](#) (and the [Getting Started Guide](#) until the release notes are available).

## 5.4 Testing

A DPC++ implementation must pass:

1. The [extension tests](#) for any extension implemented from the [Extensions Table](#). Each extension in the [Extensions Table](#) lists the name of the directory that contains corresponding tests, within the [extension tests](#) tree.

## **ONEDPL**

The oneAPI DPC++ Library (oneDPL) provides the functionality specified in the C++ standard, with extensions to support data parallelism and offloading to devices, and with extensions to simplify its usage for implementing data parallel algorithms.

The library is comprised of the following components:

- The C++ standard library. (See [C++ Standard](#).) oneDPL defines a subset of the C++ standard library which you can use with buffers and data parallel kernels. (See [Supported C++ Standard Library APIs and Algorithms](#).)
- Parallel STL. (See [Supported C++ Standard Library APIs and Algorithms](#).) oneDPL extends Parallel STL with execution policies and companion APIs for running algorithms on oneAPI devices. (See [Extensions to Parallel STL](#).)
- Extensions. An additional set of library classes and functions that are known to be useful in practice but are not (yet) included into C++ or SYCL specifications. (See [Specific API of oneDPL](#).)

### **6.1 Namespaces**

oneDPL uses namespace `std` for the [Supported C++ Standard Library APIs and Algorithms](#) including Parallel STL algorithms and the subset of the standard C++ library for kernels, and uses namespace `dpstd` for its extended functionality.

### **6.2 Supported C++ Standard Library APIs and Algorithms**

For all C++ algorithms accepting execution policies (as defined by C++17), oneDPL provides an implementation supporting `dpstd::execution::device_policy` and SYCL buffers (via `dpstd::begin/end`). (See [Extensions to Parallel STL](#).)

#### **6.2.1 Extensions to Parallel STL**

oneDPL extends Parallel STL with the following APIs:

### 6.2.1.1 DPC++ Execution Policy

```
// Defined in <dpstd/execution>

namespace dpstd {
    namespace execution {

        template <typename BasePolicy, typename KernelName = /*unspecified*/>
        class device_policy;

        template <typename KernelName, typename Arg>
        device_policy<std::execution::parallel_unsequenced_policy, KernelName>
        make_device_policy( const Arg& );

        device_policy<parallel_unsequenced_policy, /*unspecified*/> default_policy;

    }
}
```

A DPC++ execution policy specifies where and how an algorithm runs. It inherits a standard C++ execution policy and allows specification of an optional kernel name as a template parameter.

An object of a `device_policy` type encapsulates a SYCL queue which runs algorithms on a DPC++ compliant device. You can create a policy object from a SYCL queue, device, or device selector, as well as from an existing policy object.

The `make_device_policy` function simplifies `device_policy` creation.

`dpstd::execution::default_policy` is a predefined DPC++ execution policy object that can run algorithms on a default SYCL device.

Examples:

```
using namespace dpstd::execution;

auto policy_a =
    device_policy<parallel_unsequenced_policy, class PolicyA>(cl::sycl::queue{});
std::for_each(policy_a, ...);

auto policy_b = make_device_policy<class PolicyB>(cl::sycl::queue{});
std::for_each(policy_b, ...);

auto policy_c =
    make_device_policy<class PolicyC>(cl::sycl::device{cl::sycl::gpu_selector{}});
std::for_each(policy_c, ...);

auto policy_d = make_device_policy<class PolicyD>(cl::sycl::default_selector{});
std::for_each(policy_d, ...);

// use the predefined dpstd::execution::default_policy policy object
std::for_each(default_policy, ...);
```

### 6.2.1.2 Wrappers for SYCL Buffers

```
// Defined in <dpstd/iterators.h>

namespace dpstd {

    template <cl::sycl::access::mode = cl::sycl::access::mode::read_write, ... >
    /*unspecified*/ begin(cl::sycl::buffer<...>);

    template <cl::sycl::access::mode = cl::sycl::access::mode::read_write, ... >
    /*unspecified*/ end(cl::sycl::buffer<...>);

}
```

`dpstd::begin` and `dpstd::end` are helper functions for passing SYCL buffers to oneDPL algorithms. These functions accept a SYCL buffer and return an object of an unspecified type that satisfies the following requirements:

- Is `CopyConstructible`, `CopyAssignable`, and comparable with operators `==` and `!=`
- The following expressions are valid: `a + n`, `a - n`, `a - b`, where `a` and `b` are objects of the type, and `n` is an integer value
- Provides `get_buffer()` method that returns the SYCL buffer passed to `dpstd::begin` or `dpstd::end` function.

Example:

```
#include <CL/sycl.hpp>
#include <dpstd/execution>
#include <dpstd/algorithms>
#include <dpstd/iterators.h>

int main() {
    cl::sycl::queue q;
    cl::sycl::buffer<int> buf { 1000 };
    auto buf_begin = dpstd::begin(buf);
    auto buf_end = dpstd::end(buf);
    auto policy = dpstd::execution::make_device_policy<class Fill>( q );
    std::fill(policy, buf_begin, buf_end, 42);
    return 0;
}
```

### 6.2.2 Specific API of oneDPL

```
namespace dpstd {

// Declared in <dpstd/iterators.h>

template <typename Integral>
class counting_iterator;

template <typename... Iterators>
class zip_iterator;
template <typename... Iterators>
zip_iterator<Iterators...> make_zip_iterator(Iterators...);
```

(continues on next page)

(continued from previous page)

```

template <typename UnaryFunc, typename Iterator>
class transform_iterator;
template <typename UnaryFunc, typename Iterator>
transform_iterator<UnaryFunc, Iterator> make_transform_iterator(Iterator,_
→UnaryFunc);

// Declared in <dpstd/functional>

struct identity;

// Defined in <dpstd/algorithm>

template<typename Policy, typename InputKeyIt, typename InputValueIt, typename_
→OutputValueIt>
OutputValueIt
exclusive_scan_by_segment(Policy&& policy, InputKeyIt keys_first, InputKeyIt keys_-
→last,
InputValueIt values_first, OutputValueIt values_result);

template<typename Policy, typename InputKeyIt, typename InputValueIt, typename_
→OutputValueIt,
typename T>
OutputValueIt
exclusive_scan_by_segment(Policy&& policy, InputKeyIt keys_first, InputKeyIt keys_-
→last,
InputValueIt values_first, OutputValueIt values_result,
T init);

template<typename Policy, typename InputKeyIt, typename InputValueIt, typename_
→OutputValueIt,
typename T, typename BinaryPredicate>
OutputValueIt
exclusive_scan_by_segment(Policy&& policy, InputKeyIt keys_first, InputKeyIt keys_-
→last,
InputValueIt values_first, OutputValueIt values_result,
T init, BinaryPredicate binary_pred);

template<typename Policy, typename InputKeyIt, typename InputValueIt, typename_
→OutputValueIt,
typename T, typename BinaryPredicate, typename BinaryOp>
OutputValueIt
exclusive_scan_by_segment(Policy&& policy, InputKeyIt keys_first, InputKeyIt keys_-
→last,
InputValueIt values_first, OutputValueIt values_result,
T init, BinaryPredicate binary_pred, BinaryOp binary_op);

template<typename Policy, typename InputKeyIt, typename InputValueIt, typename_
→OutputValueIt>
OutputValueIt
inclusive_scan_by_segment(Policy&& policy, InputKeyIt keys_first, InputKeyIt keys_-
→last,
InputValueIt values_first, OutputValueIt values_result);

template<typename Policy, typename InputKeyIt, typename InputValueIt, typename_
→OutputValueIt,
typename BinaryPred>

```

(continues on next page)

(continued from previous page)

```

OutputValueIt
inclusive_scan_by_segment(Policy&& policy, InputKeyIt keys_first, InputKeyIt keys_
↪last,
InputValueIt values_first, OutputValueIt values_result,
BinaryPred binary_pred);

template<typename Policy, typename InputKeyIt, typename InputValueIt, typename_
↪OutputValueIt,
typename BinaryPred, typename BinaryOp>
OutputValueIt
inclusive_scan_by_segment(Policy&& policy, InputKeyIt keys_first, InputKeyIt keys_
↪last,
InputValueIt values_first, OutputValueIt values_result,
BinaryPred binary_pred, BinaryOp binary_op);

template<typename Policy, typename InputKeyIt, typename InputValueIt, typename_
↪OutputKeyIt,
typename OutputValueIt>
std::pair<OutputKeyIt,OutputValueIt>
reduce_by_segment(Policy&& policy, InputKeyIt keys_first, InputKeyIt keys_last,
InputValueIt values_first, OutputKeyIt keys_result, OutputValueIt_
↪values_result);

template<typename Policy, typename InputKeyIt, typename InputValueIt, typename_
↪OutputKeyIt,
typename OutputValueIt, typename BinaryPred>
std::pair<OutputKeyIt,OutputValueIt>
reduce_by_segment(Policy&& policy, InputKeyIt keys_first, InputKeyIt keys_last,
InputValueIt values_first, OutputKeyIt keys_result, OutputValueIt_
↪values_result,
BinaryPred binary_pred);

template<typename Policy, typename InputKeyIt, typename InputValueIt, typename_
↪OutputKeyIt,
typename OutputValueIt, typename BinaryPred, typename BinaryOp>
std::pair<OutputKeyIt,OutputValueIt>
reduce_by_segment(Policy&& policy, InputKeyIt keys_first, InputKeyIt keys_last,
InputValueIt values_first, OutputKeyIt keys_result, OutputValueIt_
↪values_result,
BinaryPred binary_pred, BinaryOp binary_op);

}

```

## ONEDNN

oneAPI Deep Neural Network Library (oneDNN) is a performance library containing building blocks for deep learning applications and frameworks. oneDNN supports:

- CNN primitives (Convolutions, Inner product, Pooling, etc.)
- RNN primitives (LSTM, Vanilla, RNN, GRU)
- Normalizations (LRN, Batch, Layer)
- Elementwise operations (ReLU, Tanh, ELU, Abs, etc.)
- Softmax, Sum, Concat, Shuffle
- Reorders from/to optimized data layouts
- 8-bit integer, 16-, 32-bit, and bfloat16 floating point data types

```
// Tensor dimensions
int N, C, H, W;

// User-owned DPC++ objects
sycl::device dev; // Device
sycl::context ctx; // Context
sycl::queue queue; // Queue
std::vector<sycl::event> dependencies; // Input events dependencies
sycl::buffer<float, 1> buf_src {sycl::range<1> {N * C * H * W}}; // Source
sycl::buffer<float, 1> buf_dst {sycl::range<1> {N * C * H * W}}; // Results

// Create an engine encapsulating users' DPC++ GPU device and context
dnnl::engine engine {dnnl::engine::kind::gpu, dev, ctx};
// Create a stream encapsulating users' DPC++ GPU queue
dnnl::stream stream {engine, queue};
// Create memory objects that use buf_src and buf_dst as the underlying storage
dnnl::memory mem_src {{N, C, H, W}, dnnl::memory::data_type::f32,
                      dnnl::memory::format_tag::nhwc},
                      engine, buf_src};
dnnl::memory mem_dst {{N, C, H, W}, dnnl::memory::data_type::f32,
                      dnnl::memory::format_tag::nhwc},
                      engine, buf_dst};
// Create a ReLU elementwise primitive
dnnl::eltwise_forward relu {
    {{dnnl::prop_kind::forward_inference, dnnl::algorithm::eltwise_relu,
      mem_src.get_desc(), 0.f, 0.f},
     engine}};
// Execute the ReLU primitive in the stream passing input dependencies and
// retrieving the output dependency
```

(continues on next page)

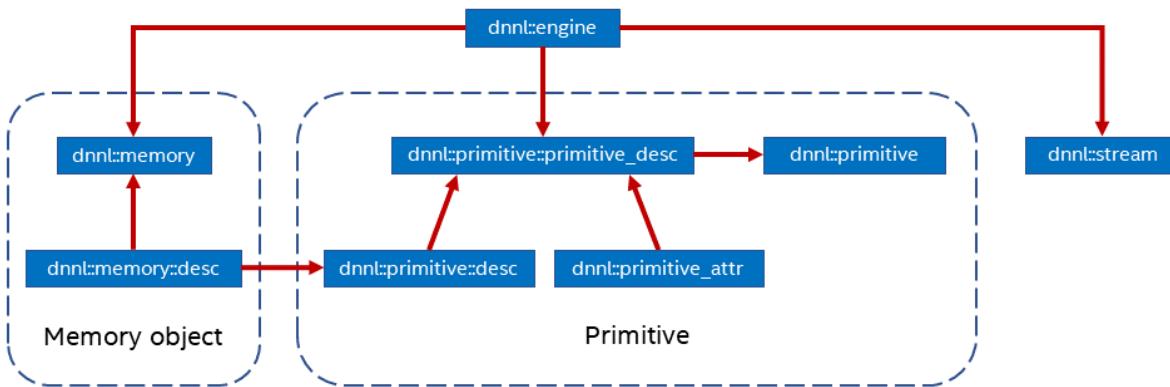
(continued from previous page)

```
sycl::event event = relu.execute_sycl(stream,
    {{DNNL_ARG_SRC, mem_src}, {DNNL_ARG_DST, mem_dst}}, dependencies);
```

## 7.1 Introduction

Although the origins of this specification are in the existing [open source implementation](#), its goal is to define a *portable* set of APIs. To this end, for example, it intentionally omits implementation-specific details like tiled or blocked memory formats (layouts), and instead describes plain multi-dimensional memory formats and defines opaque *optimized* memory format that can be implementation specific.

oneDNN main concepts are *primitives*, *engines* and *streams*.



A *primitive* ([`dnnl::primitive`](#)) is a functor object that encapsulates a particular computation such as forward convolution, backward LSTM computations, or a data transformation operation. A single primitive can sometimes represent more complex *fused* computations such as a forward convolution followed by a ReLU. Fusion, among other things, is controlled via the *primitive attributes* mechanism.

The most important difference between a primitive and a pure function is that a primitive can be specialized for a subset of input parameters.

For example, a convolution primitive stores parameters like tensor shapes and can pre-compute other dependent parameters like cache blocking. This approach allows oneDNN primitives to pre-generate code specifically tailored for the operation to be performed. The oneDNN programming model assumes that the time it takes to perform the pre-computations is amortized by reusing the same primitive to perform computations multiple times.

A primitive may also need a mutable memory buffer that it may use for temporary storage only during computations. Such buffer is called a scratchpad. It can either be owned by a primitive object (which makes that object non-thread safe) or be an execution-time parameter.

Primitive creation is a potentially expensive operation. Users are expected to create primitives once and reuse them multiple times. Alternatively, implementations may reduce the primitive creation cost by caching primitives that have the same parameters. This optimization falls outside of the scope of this specification.

*Engines* ([`dnnl::engine`](#)) are an abstraction of a computational device: a CPU, a specific GPU card in the system, etc. Most primitives are created to execute computations on one specific engine. The only exceptions are reorder primitives that transfer data between two different engines.

*Streams* ([`dnnl::stream`](#)) encapsulate execution context tied to a particular engine. For example, they can correspond to DPC++ command queues.

Memory objects (`dnnl::memory`) encapsulate handles to memory allocated on a specific engine, tensor dimensions, data type, and memory format – the way tensor indices map to offsets in linear memory space. Memory objects are passed to primitives during execution.

## Levels of Abstraction

oneDNN has multiple levels of abstractions for primitives and memory objects in order to expose maximum flexibility to its users.

On the logical level, the library provides the following abstractions:

- Memory descriptors (`dnnl::memory::desc`) define a tensor's logical dimensions, data type, and the format in which the data is laid out in memory. The special format any (`dnnl::memory::format_tag::any`) indicates that the actual format will be defined later.
- Operation descriptors (one for each supported primitive) describe an operation's most basic properties without specifying, for example, which engine will be used to compute them. For example, convolution descriptor describes shapes of source, destination, and weights tensors, propagation kind (forward, backward with respect to data or weights), and other implementation-independent parameters.
- Primitive descriptors (`dnnl::primitive_desc_base`) is the base class and each of the supported primitives have their own version) are at an abstraction level in between operation descriptors and primitives and can be used to inspect details of a specific primitive implementation like expected memory formats via queries to implement memory format propagation (see Memory format propagation) without having to fully instantiate a primitive.

Abstraction level	Memory object	Primitive objects
Logical description	Memory descriptor	Operation descriptor
Intermediate description	N/A	Primitive descriptor
Implementation	Memory object	Primitive

### 7.1.1 General API notes

There are certain assumptions on how oneDNN objects behave:

- Memory and operation descriptors behave similarly to trivial types.
- All other objects behave like shared pointers. Copying is always shallow.

oneDNN objects can be *empty* in which case they are not valid for any use. Memory descriptors are special in this regard, as their empty versions are regarded as *zero* memory descriptors that can be used to indicate absence of a memory descriptor. Empty objects are usually created using default constructors, but also may be a result of an error during object construction (see the next section).

### 7.1.2 Error Handling

All oneDNN functions throw the following exception in case of error.

`struct error : public exception`

The exception class.

Additionally, many oneDNN functions that construct or return oneDNN objects have a boolean `allow_empty` parameter that defaults to `false` and that makes the library to return an empty object (a zero object in case of memory descriptors) when an object cannot be constructed instead of throwing an error.

## 7.2 Conventions

oneDNN documentation relies on a set of standard naming conventions for variables. This section describes these conventions.

### 7.2.1 Variable (Tensor) Names

Neural network models consist of operations of the following form:

$$\text{dst} = f(\text{src}, \text{weights}),$$

where dst and src are activation tensors, and weights are learnable tensors.

The backward propagation consists then in computing the gradients with respect to the srcweights` respectively:

$$\text{diff\_src} = df_{\text{src}}(\text{diff\_dst}, \text{src}, \text{weights}, \text{dst}),$$

and

$$\text{diff\_weights} = df_{\text{weights}}(\text{diff\_dst}, \text{src}, \text{weights}, \text{dst}).$$

While oneDNN uses *src*, *dst*, and *weights* as generic names for the activations and learnable tensors, for a specific operation there might be commonly used and widely known specific names for these tensors. For instance, the *convolution* operation has a learnable tensor called *bias*. For usability reasons, oneDNN primitives use such names in initialization and other functions.

oneDNN uses the following commonly used notations for tensors:

Name	Meaning
src	Source tensor
dst	Destination tensor
weights	Weights tensor
bias	Bias tensor (used in <i>convolution</i> , <i>inner product</i> and other primitives)
scale_shift	Scale and shift tensors (used in <i>Batch Normalization</i> and <i>Layer normalization</i> primitives)
workspace	Workspace tensor that carries additional information from the forward propagation to the backward propagation
scratchpad	Temporary tensor that is required to store the intermediate results
diff_src	Gradient tensor with respect to the source
diff_dst	Gradient tensor with respect to the destination
diff_weights	Gradient tensor with respect to the weights
diff_bias	Gradient tensor with respect to the bias
diff_scale_shif	Gradient tensor with respect to the scale and shift
*_layer	RNN layer data or weights tensors
*_iter	RNN recurrent data or weights tensors

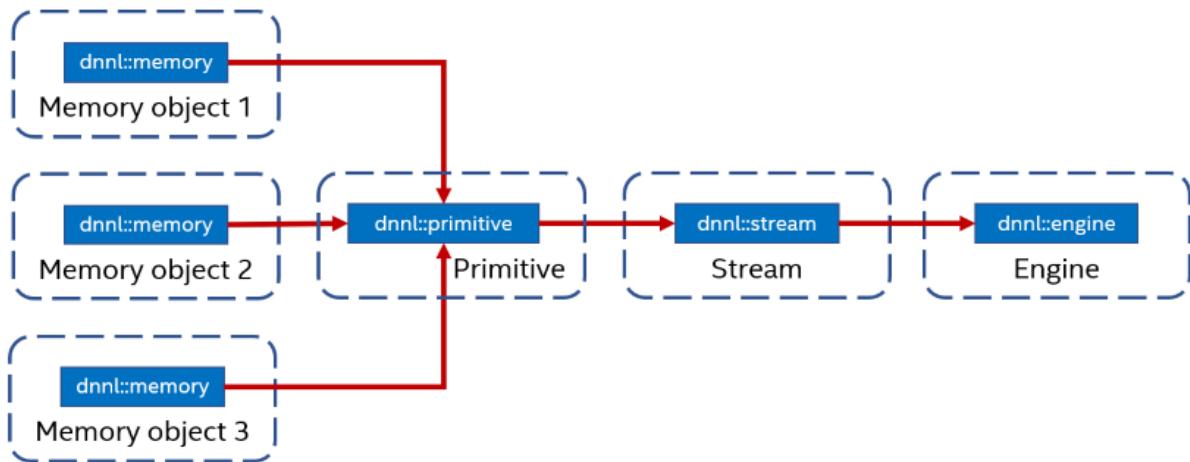
## 7.2.2 RNN-Specific Notation

The following notations are used when describing RNN primitives.

Name	Semantics
.	matrix multiply operator
*	elementwise multiplication operator
W	input weights
U	recurrent weights
$\square^T$	transposition
B	bias
h	hidden state
a	intermediate value
x	input
$\square_t$	timestamp index
$\square_l$	layer index
activation	tanh, relu, logistic
c	cell state
$\tilde{c}$	candidate state
i	input gate
f	forget gate
o	output gate
u	update gate
r	reset gate

## 7.3 Execution Model

To execute a primitive, a user needs to pass memory arguments and a stream to the `dnnl::primitive::execute()` member function.



The primitive's computations are executed on the computational device corresponding to the engine on which the primitive (and memory arguments) were created and happens within the context on the stream.

### 7.3.1 Engine

*Engine* is abstraction of a computational device: a CPU, a specific GPU card in the system, etc. Most primitives are created to execute computations on one specific engine. The only exceptions are reorder primitives that transfer data between two different engines.

Engines correspond to and can be constructed from pairs of the DPC++ `sycl::device` and `sycl::context` objects.

```
struct dnnl::engine
An execution engine.
```

#### Public Types

```
enum kind
Kinds of engines.

Values:

enumerator any
An unspecified engine.

enumerator cpu
CPU engine.

enumerator gpu
GPU engine.
```

#### Public Functions

```
engine()
Constructs an empty engine. An empty engine cannot be used in any operations.

engine(kind akind, size_t index)
Constructs an engine.
```

##### Parameters

- `akind`: The kind of engine to construct.
- `index`: The index of the engine. Must be less than the value returned by `get_count()` for this particular kind of engine.

```
engine(kind akind, const cl::sycl::device &dev, const cl::sycl::context &ctx)
Constructs an engine from SYCL device and context objects.
```

##### Parameters

- `akind`: The kind of engine to construct.
- `dev`: SYCL device.
- `ctx`: SYCL context.

```
kind get_kind() const
Returns the kind of the engine.
```

**Return** The kind of the engine.

`cl::sycl::context get_sycl_context() const`

Returns the underlying SYCL context object.

`cl::sycl::device get_sycl_device() const`

Returns the underlying SYCL device object.

## Public Static Functions

`size_t get_count (kind akind)`

Returns the number of engines of a certain kind.

**Return** The number of engines of the specified kind.

### Parameters

- *akind*: The kind of engines to count.

## 7.3.2 Stream

A *stream* is an encapsulation of execution context tied to a particular engine. They are passed to `dnnl::primitive::execute()` when executing a primitive.

Stream attributes are used to extend stream behavior in an implementation-defined manner.

**struct dnnl::stream\_attr**

A container for stream attributes.

## Public Functions

`stream_attr()`

Constructs default (empty) stream attributes.

`stream_attr (engine::kind akind)`

Constructs stream attributes for a stream that runs on an engine of a particular kind.

### Parameters

- *akind*: Target engine kind.

Streams correspond to and can be constructed from DPC++ `sycl::queue` objects. Alternatively, oneDNN can create and own the corresponding objects itself. Streams are considered to be ephemeral and can be created / destroyed as long these operation do not violate DPC++ synchronization requirements.

Similar to DPC++ queues, streams can be in-order and out-of-order (see the relevant portion of the DPC++ specification for the explanation). The desired behavior can be specified using `dnnl::stream::flags` value. A stream created from a DPC++ queue inherits its behavior.

**struct dnnl::stream**

An execution stream.

## Public Types

### `enum flags`

Stream flags. Can be combined using the bitwise OR operator.

*Values:*

#### `enumerator default_order`

Default order execution. Either in-order or out-of-order depending on the engine runtime.

#### `enumerator in_order`

In-order execution.

#### `enumerator out_of_order`

Out-of-order execution.

#### `enumerator default_flags`

Default stream configuration.

## Public Functions

### `stream()`

Constructs an empty stream. An empty stream cannot be used in any operations.

### `stream(const engine &aengine, flags aflags = flags::default_flags, const stream_attr &attr = stream_attr())`

Constructs a stream for the specified engine and with behavior controlled by the specified flags.

#### Parameters

- `aengine`: Engine to create the stream on.
- `aflags`: Flags controlling stream behavior.
- `attr`: Stream attributes.

### `stream(const engine &aengine, cl::sycl::queue &queue)`

Constructs a stream for the specified engine and the SYCL queue.

#### Parameters

- `aengine`: Engine object to use for the stream.
- `queue`: SYCL queue to use for the stream.

### `cl::sycl::queue get_sycl_queue() const`

Returns the underlying SYCL queue object.

**Return** SYCL queue object.

### `stream &wait()`

Waits for all primitives executing in the stream to finish.

**Return** The stream itself.

## 7.4 Data model

Data in oneDNN is stored in *memory objects* that both store and describe data that can be of various types and be stored in different formats (layouts).

### 7.4.1 Data types

oneDNN supports multiple data types. However, the 32-bit IEEE single-precision floating-point data type is the fundamental type in oneDNN. It is the only data type that must be supported by an implementation. All the other types discussed below are optional.

Primitives operating on the single-precision floating-point data type consume data, produce, and store intermediate results using the same data type.

Moreover, single-precision floating-point data type is often used for intermediate results in the mixed precision computations because it provides better accuracy. For example, the elementwise primitive and elementwise post-ops always use it internally.

oneDNN uses the following enumeration to refer to data types it supports:

`enum dnnl::memory::data_type`

Data type specification.

*Values:*

**enumerator undef**

Undefined data type (used for empty memory descriptors).

**enumerator f16**

16-bit/half-precision floating point.

**enumerator bf16**

non-standard 16-bit floating point with 7-bit mantissa.

**enumerator f32**

32-bit/single-precision floating point.

**enumerator s32**

32-bit signed integer.

**enumerator s8**

8-bit signed integer.

**enumerator u8**

8-bit unsigned integer.

oneDNN supports training and inference with the following data types:

Usage mode	Data types
inference	<code>dnnl::memory::data_type::f32, dnnl::memory::data_type::bf16, dnnl::memory::data_type::f16, dnnl::memory::data_type::s8/dnnl::memory::data_type::u8</code>
training	<code>dnnl::memory::data_type::f32, dnnl::memory::data_type::bf16</code>

---

**Note:** Using lower precision arithmetic may require changes in the deep learning model implementation.

---

Individual primitives may have additional limitations with respect to data type support based on the precision requirements. The list of data types supported by each primitive is included in the corresponding sections of the specification guide.

### 7.4.1.1 Bfloat16

---

**Note:** In this section we abbreviate data types names for readability. For example, `dnnl::memory::data_type::f32` is abbreviated to `f32`.

---

Bfloat16 (`bf16`) is a 16-bit floating point data type based on the IEEE 32-bit single-precision floating point data type (`f32`).

Both `bf16` and `f32` have an 8-bit exponent. However, while `f32` has a 23-bit mantissa, `bf16` has only a 7-bit one, keeping only the most significant bits. As a result, while these data types support a very close numerical range of values, `bf16` has a significantly reduced precision. Therefore, `bf16` occupies a spot between `f32` and the IEEE 16-bit half-precision floating point data type, `f16`. Compared directly to `f16`, which has a 5-bit exponent and a 10-bit mantissa, `bf16` trades increased range for reduced precision.

<b>f32</b>	s	8-bit exp	23 bit mantissa
<b>bf16</b>	s	8-bit exp	7 bit mantissa
<b>f16</b>	s	5-bit exp	10 bit mantissa

More details of the bfloat16 data type can be found [here](#).

One of the advantages of using `bf16` versus `f32` is reduced memory footprint and, hence, increased memory access throughput.

#### 7.4.1.1.1 Workflow

The main difference between implementing training with the `f32` data type and with the `bf16` data type is the way the weights updates are treated. With the `f32` data type, the weights gradients have the same data type as the weights themselves. This is not necessarily the case with the `bf16` data type as oneDNN allows some flexibility here. For example, one could maintain a master copy of all the weights, computing weights gradients in `f32` and converting the result to `bf16` afterwards.

### 7.4.1.1.2 Support

Most of the primitives can support the `bf16` data type for source and weights tensors. Destination tensors can be specified to have either the `bf16` or `f32` data type. The latter is intended for cases in which the output is to be fed to operations that do not support bfloat16 or require better precision.

### 7.4.1.2 Int8

To push higher performance during inference computations, recent work has focused on computations that use activations and weights stored at a lower precision to achieve higher throughput. Int8 computations offer improved performance over higher-precision types because they enable packing more computations into a single instruction, at the cost of reduced (but acceptable) accuracy.

#### 7.4.1.2.1 Workflow

The *Quantization* describes what kind of quantization model oneDNN supports.

#### 7.4.1.2.2 Support

oneDNN supports int8 computations for inference by allowing to specify that primitives input and output memory objects use int8 data types.

## 7.4.2 Memory

There are two levels of abstraction for memory in oneDNN.

1. *Memory descriptor* – engine-agnostic logical description of data (number of dimensions, dimension sizes, *data type*, and *format*).
2. *Memory object* – an engine-specific object combines memory descriptor with storage.

oneDNN defines the following convenience aliases to denote tensor dimensions

`using dnnl::memory::dim = int64_t`  
Integer type for representing dimension sizes and indices.

`using dnnl::memory::dims = std::vector<dim>`  
Vector of dimensions. Implementations are free to force a limit on the vector's length.

#### 7.4.2.1 Memory Formats

In oneDNN memory format is how a multidimensional tensor is stored in 1-dimensional linear memory address space. oneDNN specifies two kinds of memory formats: *plain* which correspond to traditional multidimensional arrays, and *optimized* which are completely opaque.

### 7.4.2.1.1 Plain Memory Formats

Plain memory formats describe how multidimensional tensors are laid out in memory using an array of dimensions and an array of strides both of which have length equal to the rank of the tensor. In oneDNN the order of dimensions is fixed and different dimensions can have certain canonical interpretation depending on the primitive. For example, for CNN primitives the order for activation tensors is  $\{N, C, \dots, D, H, W\}$ , where  $N$  stands for minibatch,  $C$  stands for channels, and  $D, H$ , and  $W$  stand for image spatial dimensions: depth, height and width respectively. Spatial dimensions may be omitted in the order from outermost to innermost; for example, it is not possible to omit  $H$  when  $D$  is present and it is never possible to omit  $W$ . Canonical interpretation is documented for each primitive. However, this means that the strides array plays an important role defining the order in which different dimension are laid out in memory. Moreover, the strides need to agree with dimensions.

More precisely, let  $T$  be a tensor of rank  $n$  and let  $\sigma$  be the permutation of the strides array that sorts it, i.e.  $\text{strides}[i] \geq \text{strides}[j]$  if  $\sigma(i) < \sigma(j)$  for all  $0 \leq i, j < n$ . Then the following must hold:

$$\text{strides}[i] \geq \text{strides}[j] * \text{dimensions}[j] \text{ if } \sigma(i) < \sigma(j) \text{ for all } 0 \leq i, j < n.$$

For an element with coordinates  $(i_0, \dots, i_{n-1})$  such that  $0 \leq i_j < \text{dimensions}[j]$  for  $0 \leq j < n$ , its offset in memory is computed as:

$$\text{offset}(i_0, \dots, i_{n-1}) = \text{offset}_0 + \sum_{j=0}^{n-1} i_j * \text{strides}[j].$$

Here  $\text{offset}_0$  is the offset from the *parent* memory and is non-zero only for *submemory* memory descriptors created using `dnnl::memory::desc::submemory_desc()`. Submemory memory descriptors inherit strides from the parent memory descriptor. Their main purpose is to express in-place concat operations.

As an example, consider an  $M \times N$  matrix  $A$  ( $M$  rows times  $N$  columns). Regardless of whether  $A$  is stored transposed or not,  $\text{dimensions}_A = \{M, N\}$ . However,  $\text{strides}_A = \{LDA, 1\}$  if it is not transposed and  $\text{strides}_A = \{1, LDA\}$  if it is, where  $LDA$  is such that  $LDA \geq N$  if  $A$  is not transposed, and  $LDA \geq M$  if it is. This also shows that  $A$  does not have to be stored *densely* in memory.

---

**Note:** The example above shows that oneDNN assumes data to be stored in row-major order.

---

Code example:

```
int M, N;
dnnl::memory::dims dims {M, N}; // Dimensions always stay the same

// Non-transposed matrix
dnnl::memory::dims strides_non_transposed {N, 1};
dnnl::memory::desc A_non_transposed {dims, dnnl::memory::data_type::f32,
    strides_non_transposed};

// Transposed matrix
dnnl::memory::dims strides_transposed {1, M};
dnnl::memory::desc A_transposed {dims, dnnl::memory::data_type::f32,
    strides_transposed};
```

### 7.4.2.1.2 Format Tags

In addition to strides, oneDNN provides named *format tags* via the `dnnl::memory::format_tag` enum type. The enumerators of this type can be used instead of strides for dense plain layouts.

The format tag names for  $N$ -dimensional memory formats use first  $N$  letters of the English alphabet which can be arbitrarily permuted. This permutation is used to compute strides for tensors with up to 6 dimensions. The resulting strides specify dense storage, in other words, using the nomenclature from the previous section, the following equality holds:

$$\text{stides}[i] = \text{strides}[j] * \text{dimensions}[j] \text{ if } \sigma(i) + 1 = \sigma(j) \text{ for all } 0 \leq i, j < n - 1.$$

In the matrix example, we could have used `dnnl::memory::format_tag::ab` for the non-transposed matrix above, and `dnnl::memory::format_tag::ba` for the transposed:

```
int M, N;
dnnl::memory::dims dims {M, N}; // Dimensions always stay the same

// Non-transposed matrix
dnnl::memory::desc A_non_transposed {dims, dnnl::memory::data_type::f32,
    dnnl::memory::format_tag::ab};

// Transposed matrix
dnnl::memory::desc A_transposed {dims, dnnl::memory::data_type::f32,
    dnnl::memory::format_tag::ba};
```

---

**Note:** In what follows in this section we abbreviate memory format tag names for readability. For example, `dnnl::memory::format_tag::abcd` is abbreviated to `abcd`.

---

In addition to abstract format tag names, oneDNN also provides convenience aliases. Some examples for CNNs and RNNs:

- `nchw` is an alias for `abcd` (see the canonical order order of dimensions for CNNs discussed above).
- `oihw` is an alias for `abcd`.
- `nhwc` is an alias for `acdb`.
- `tnc` is an alias for `abc`.
- `ldio` is an alias for `abcd`.
- `ldoi` is an alias for `abdc`.

### 7.4.2.1.3 Optimized Format ‘any’

Another kind of format that oneDNN supports is an opaque \_optimized\_ memory format that cannot be created directly from strides and dimensions arrays. A memory descriptor for an optimized memory format can only be created by passing `any` when creating certain operation descriptors, using them to create corresponding primitive descriptors and then querying them for memory descriptors. Data in plain memory format should then be reordered into the data in optimized data format before computations. Since reorders are expensive, the optimized memory format needs to be \_propagated\_ through computations graph.

Optimized formats can employ padding, blocking and other data transformations to keep data in layout optimal for a certain architecture. This means that it in general operations like `dnnl::memory::desc::permute_axes()` or `dnnl::memory::desc::submemory_desc()` may fail. It is in general incorrect to use product of dimension

sizes to calculate amount of memory required to store data: `dnnl::memory::desc::get_size()` must be used instead.

#### 7.4.2.1.4 Memory Format Propagation

Memory format propagation is one of the central notions that needs to be well-understood to use oneDNN correctly.

Convolution and inner product primitives choose the memory format when you create them with the placeholder memory format `any` for input or output. The memory format chosen is based on different circumstances such as hardware and convolution parameters. Using the placeholder memory format is the recommended practice for convolutions, since they are the most compute-intensive operations in most topologies where they are present.

Other primitives, such as Elementwise, LRN, batch normalization and other, on forward propagation should use the same memory format as the preceding layer thus propagating the memory format through multiple oneDNN primitives. This avoids unnecessary reorders which may be expensive and should be avoided unless a compute-intensive primitive requires a different format. For performance reasons, backward computations of such primitives requires consistent memory format with the corresponding forward computations. Hence, when initializing there primitives for backward computations you should use `dnnl::memory::format_tag::any` memory format tag as well.

Below is the short summary when to use and not to use memory format `any` during operation description initialization:

Primitive Kinds	Forward Propagation	Backward Propagation	No Propagation
<b>Compute intensive:</b> (De-)convolution, Inner product, RNN	Use <code>any</code>	Use <code>any</code>	N/A
<b>Memory-bandwidth limited:</b> Pooling, Layer and Batch Normalization, Local Response Normalization, Elementwise, Shuffle, Softmax	Use memory format from preceding layer for source tensors, and <code>any</code> for destination tensors	Use <code>any</code> for gradient tensors, and actual memory formats for data tensors	N/A
<b>Memory-bandwidth limited:</b> Re-order, Concat, Sum, Binary	N/A	N/A	Use memory format from preceding layer for source tensors, and <code>any</code> for destination tensors

Additional format synchronization is required between forward and backward propagation when running training workloads. This is achieved via the `hint_pd` arguments of primitive descriptor constructors for primitives that implement backward propagation.

#### 7.4.2.1.5 API

##### `enum dnnl::memory::format_tag`

Memory format tag specification.

Memory format tags can be further divided into two categories:

- Domain-agnostic names, i.e. names that do not depend on the tensor usage in the specific primitive. These names use letters from `a` to `f` to denote logical dimensions and form the order in which the dimensions are laid in memory. For example, `dnnl::memory::format_tag::ab` is used to denote a 2D tensor where the second logical dimension (denoted as `b`) is the innermost, i.e. has stride = 1, and the first logical dimension (`a`) is laid out in memory with stride equal to the size of the second dimension. On the other hand, `dnnl::memory::format_tag::ba` is the transposed version of the same tensor: the outermost dimension (`a`) becomes the innermost one.

- Domain-specific names, i.e. names that make sense only in the context of a certain domain, such as CNN. These names are aliases to the corresponding domain-agnostic tags and used mostly for convenience. For example, `dnnl::memory::format_tag::nc` is used to denote 2D CNN activations tensor memory format, where the channels dimension is the innermost one and the batch dimension is the outermost one. Moreover, `dnnl::memory::format_tag::nc` is an alias for `dnnl::memory::format_tag::ab`, because for CNN primitives the logical dimensions of activations tensors come in order: batch, channels, spatial. In other words, batch corresponds to the first logical dimension (`a`), and channels correspond to the second one (`b`).

The following domain-specific notation applies to memory format tags:

- '`n`' denotes the mini-batch dimension
- '`c`' denotes a channels dimension
- When there are multiple channel dimensions (for example, in convolution weights tensor), '`i`' and '`o`' denote dimensions of input and output channels
- '`g`' denotes a groups dimension for convolution weights
- '`d`', '`h`', and '`w`' denote spatial depth, height, and width respectively

*Values:*

**enumerator undef**

Undefined memory format tag.

**enumerator any**

Placeholder memory format tag. Used to instruct the primitive to select a format automatically.

**enumerator a**

plain 1D tensor

**enumerator ab**

plain 2D tensor

**enumerator ba**

permuted 2D tensor

**enumerator abc**

plain 3D tensor

**enumerator acb**

permuted 3D tensor

**enumerator bac**

permuted 3D tensor

**enumerator bca**

permuted 3D tensor

**enumerator cba**

permuted 3D tensor

**enumerator abcd**

plain 4D tensor

**enumerator abdc**

permuted 4D tensor

**enumerator acdb**

permuted 4D tensor

**enumerator bacd**

permuted 4D tensor

---

```

enumerator bcda
    permuted 4D tensor

enumerator cdba
    permuted 4D tensor

enumerator dcab
    permuted 4D tensor

enumerator abcde
    plain 5D tensor

enumerator abdec
    permuted 5D tensor

enumerator acbde
    permuted 5D tensor

enumerator acdeb
    permuted 5D tensor

enumerator bacde
    permuted 5D tensor

enumerator bcdea
    permuted 5D tensor

enumerator cdeba
    permuted 5D tensor

enumerator decab
    permuted 5D tensor

enumerator abcdef
    plain 6D tensor

enumerator acbdef
    plain 6D tensor

enumerator defcab
    plain 6D tensor

enumerator x = a
    1D tensor; an alias for dnnl::memory::format_tag::a

enumerator nc = ab
    2D CNN activations tensor; an alias for dnnl::memory::format_tag::ab

enumerator cn = ba
    2D CNN activations tensor; an alias for dnnl::memory::format_tag::ba

enumerator tn = ab
    2D RNN statistics tensor; an alias for dnnl::memory::format_tag::ab

enumerator nt = ba
    2D RNN statistics tensor; an alias for dnnl::memory::format_tag::ba

enumerator ncw = abc
    3D CNN activations tensor; an alias for dnnl::memory::format_tag::abc

enumerator nwc = acb
    3D CNN activations tensor; an alias for dnnl::memory::format_tag::acb

```

---

```

enumerator nchw = abcd
    4D CNN activations tensor; an alias for dnnl::memory::format_tag::abcd

enumerator nhwc = acdb
    4D CNN activations tensor; an alias for dnnl::memory::format_tag::acdb

enumerator chwn = bcda
    4D CNN activations tensor; an alias for dnnl::memory::format_tag::bcda

enumerator ncdhw = abcde
    5D CNN activations tensor; an alias for dnnl::memory::format_tag::abcde

enumerator ndhwc = acdeb
    5D CNN activations tensor; an alias for dnnl::memory::format_tag::acdeb

enumerator oi = ab
    2D CNN weights tensor; an alias for dnnl::memory::format_tag::ab

enumerator io = ba
    2D CNN weights tensor; an alias for dnnl::memory::format_tag::ba

enumerator oiw = abc
    3D CNN weights tensor; an alias for dnnl::memory::format_tag::abc

enumerator owi = acb
    3D CNN weights tensor; an alias for dnnl::memory::format_tag::acb

enumerator wio = cba
    3D CNN weights tensor; an alias for dnnl::memory::format_tag::cba

enumerator iwo = bca
    3D CNN weights tensor; an alias for dnnl::memory::format_tag::bca

enumerator oihw = abcd
    4D CNN weights tensor; an alias for dnnl::memory::format_tag::abcd

enumerator hwio = cdba
    4D CNN weights tensor; an alias for dnnl::memory::format_tag::cdba

enumerator ohwi = acdb
    4D CNN weights tensor; an alias for dnnl::memory::format_tag::acdb

enumerator ihwo = bcda
    4D CNN weights tensor; an alias for dnnl::memory::format_tag::bcda

enumerator iohw = bacd
    4D CNN weights tensor; an alias for dnnl::memory::format_tag::bacd

enumerator oidhw = abcde
    5D CNN weights tensor; an alias for dnnl::memory::format_tag::abcde

enumerator dhwio = cdeba
    5D CNN weights tensor; an alias for dnnl::memory::format_tag::cdeba

enumerator odhwi = acdeb
    5D CNN weights tensor; an alias for dnnl::memory::format_tag::acdeb

enumerator iodhw = bacde
    5D CNN weights tensor; an alias for dnnl::memory::format_tag::bacde

enumerator idhwo = bcdea
    5D CNN weights tensor; an alias for dnnl::memory::format_tag::bcdea

```

---

```

enumerator goiw = abcd
    4D CNN weights tensor with groups; an alias for dnnl::memory::format_tag::abcd

enumerator wigo = dcab
    4D CNN weights tensor with groups; an alias for dnnl::memory::format_tag::dcab

enumerator goihw = abcde
    5D CNN weights tensor with groups; an alias for dnnl::memory::format_tag::abcde

enumerator hwigo = decab
    5D CNN weights tensor with groups; an alias for dnnl::memory::format_tag::decab

enumerator giohw = acbde
    5D CNN weights tensor with groups; an alias for dnnl::memory::format_tag::acbde

enumerator goidhw = abcdef
    6D CNN weights tensor with groups; an alias for dnnl::memory::format_tag::abcdef

enumerator giodhw = acbdef
    6D CNN weights tensor with groups; an alias for dnnl::memory::format_tag::abcdef

enumerator dhwigo = defcab
    6D CNN weights tensor with groups; an alias for dnnl::memory::format_tag::defcab

enumerator tnc = abc
    3D RNN data tensor in the format (seq_length, batch, input channels).

enumerator ntc = bac
    3D RNN data tensor in the format (batch, seq_length, input channels).

enumerator ldnc = abcd
    4D RNN states tensor in the format (num_layers, num_directions, batch, state channels).

enumerator ldigo = abcde
    5D RNN weights tensor in the format (num_layers, num_directions, input_channels, num_gates, output_channels).
        • For LSTM cells, the gates order is input, forget, candidate and output gate.
        • For GRU cells, the gates order is update, reset and output gate.

enumerator ldgoi = abdec
    5D RNN weights tensor in the format (num_layers, num_directions, num_gates, output_channels, input_channels).
        • For LSTM cells, the gates order is input, forget, candidate and output gate.
        • For GRU cells, the gates order is update, reset and output gate.

enumerator ldio = abcd
    4D LSTM projection tensor in the format (num_layers, num_directions, num_channels_in_hidden_state, num_channels_in_recurrent_projection).

enumerator ldoi = abdc
    4D LSTM projection tensor in the format (num_layers, num_directions, num_channels_in_recurrent_projection, num_channels_in_hidden_state).

enumerator ldgo = abcd
    4D RNN bias tensor in the format (num_layers, num_directions, num_gates, output_channels).
        • For LSTM cells, the gates order is input, forget, candidate and output gate.

```

- For GRU cells, the gates order is update, reset and output gate.

### 7.4.2.2 Memory Descriptors and Objects

#### 7.4.2.2.1 Descriptors

Memory descriptor is an engine-agnostic logical description of data (number of dimensions, dimension sizes, and data type), and, optionally, the information about the physical format of data in memory. If this information is not known yet, a memory descriptor can be created with format tag set to `dnnl::memory::format_tag::any`. This allows compute-intensive primitives to chose the most appropriate format for the computations. The user is then responsible for reordering their data into the new format if the formats do not match. See [Memory Format Propagation](#).

A memory descriptor can be initialized either by specifying dimensions, and memory format tag or strides for each of them.

User can query amount of memory required by a memory descriptor using the `dnnl::memory::desc::get_size()` function. The size of data in general cannot be computed as the product of dimensions multiplied by the size of the data type. So users are required to use this function for better code portability.

Two memory descriptors can be compared using the equality and inequality operators. The comparison is especially useful when checking whether it is necessary to reorder data from the user's data format to a primitive's format.

Along with ordinary memory descriptors with all dimensions being positive, oneDNN supports *zero-volume* memory descriptors with one or more dimensions set to zero. This is used to support the NumPy\* convention. If a zero-volume memory is passed to a primitive, the primitive typically does not perform any computations with this memory. For example:

- The concatenation primitive would ignore all memory object with zeroes in the concatenation dimension / axis.
- A forward convolution with a source memory object with zero in the minibatch dimension would always produce a destination memory object with a zero in the minibatch dimension and perform no computations.
- However, a forward convolution with a zero in one of the weights dimensions is ill-defined and is considered to be an error by the library because there is no clear definition on what the output values should be.

Data handle of a zero-volume memory is never accessed.

## API

**struct** `dnnl::memory::desc`

A memory descriptor.

### Public Functions

**desc()**

Constructs a zero (empty) memory descriptor. Such a memory descriptor can be used to indicate absence of an argument.

**desc(`const dims &adims, data_type adata_type, format_tag aformat_tag, bool allow_empty = false`)**

Constructs a memory descriptor.

**Note** The logical order of dimensions corresponds to the `abc...` format tag, and the physical meaning of the dimensions depends both on the primitive that would operate on this memory and the operation context.

### Parameters

- `adims`: Tensor dimensions.
- `adata_type`: Data precision/type.
- `aformat_tag`: Memory format tag.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case a zero memory descriptor will be constructed. This flag is optional and defaults to false.

`desc (const dims &adims, data_type adata_type, const dims &strides, bool allow_empty = false)`  
Constructs a memory descriptor by strides.

**Note** The logical order of dimensions corresponds to the `abc...` format tag, and the physical meaning of the dimensions depends both on the primitive that would operate on this memory and the operation context.

#### Parameters

- `adims`: Tensor dimensions.
- `adata_type`: Data precision/type.
- `strides`: Strides for each dimension.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case a zero memory descriptor will be constructed. This flag is optional and defaults to false.

`desc submemory_desc (const dims &adims, const dims &offsets, bool allow_empty = false)`  
Constructs a memory descriptor for a region inside an area described by this memory descriptor.

**Return** A memory descriptor for the region.

#### Parameters

- `adims`: Sizes of the region.
- `offsets`: Offsets to the region from the encompassing memory object in each dimension.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case a zero memory descriptor will be returned. This flag is optional and defaults to false.

`desc reshape (const dims &adims, bool allow_empty = false) const`

Constructs a memory descriptor by reshaping an existing one. The new memory descriptor inherits the data type.

The operation ensures that the transformation of the physical memory format corresponds to the transformation of the logical dimensions. If such transformation is impossible, the function either throws an exception (default) or returns a zero memory descriptor depending on the `allow_empty` flag.

The reshape operation can be described as a combination of the following basic operations:

- Add a dimension of size 1. This is always possible.
- Remove a dimension of size 1.
- Split a dimension into multiple ones. This is possible only if the product of all tensor dimensions stays constant.
- Join multiple consecutive dimensions into a single one. This requires that the dimensions are dense in memory and have the same order as their logical counterparts.

- Here, ‘dense’ means: stride for  $\text{dim}[i] == (\text{stride for } \text{dim}[i + 1]) * \text{dim}[i + 1]$ ;
- And ‘same order’ means:  $i < j$  if and only if stride for  $\text{dim}[j] \leq \text{stride for } \text{dim}[i]$ .

**Note** Reshape may fail for optimized memory formats.

**Return** A new memory descriptor with new dimensions.

#### Parameters

- `adims`: New dimensions. The product of dimensions must remain constant.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case a zero memory descriptor will be returned. This flag is optional and defaults to false.

`desc permute_axes (const std::vector<int> &permutation, bool allow_empty = false) const`  
Constructs a memory descriptor by permuting axes in an existing one.

The physical memory layout representation is adjusted accordingly to maintain the consistency between the logical and physical parts of the memory descriptor. The new memory descriptor inherits the data type.

The logical axes will be permuted in the following manner:

```
for (i = 0; i < ndims(); i++)
    new_desc.dims() [permutation[i]] = dims() [i];
```

Example:

```
std::vector<int> permutation = {1, 0}; // swap the first and
                                         // the second axes
dnnl::memory::desc in_md(
    {2, 3}, data_type, memory::format_tag::ab);
dnnl::memory::desc expect_out_md(
    {3, 2}, data_type, memory::format_tag::ba);

assert(in_md.permute_axes(permutation) == expect_out_md);
```

**Return** A new memory descriptor with new dimensions.

#### Parameters

- `permutation`: Axes permutation.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case a zero memory descriptor will be returned. This flag is optional and defaults to false.

`memory::dims dims () const`

Returns dimensions of the memory descriptor.

Potentially expensive due to the data copy involved.

**Return** A copy of the dimensions vector.

`memory::data_type data_type () const`

Returns the data type of the memory descriptor.

**Return** The data type.

**size\_t get\_size() const**

Returns size of the memory descriptor in bytes.

**Return** The number of bytes required to allocate a memory buffer for the memory object described by this memory descriptor.

**bool is\_zero() const**

Checks whether the memory descriptor is zero (empty).

**Return** `true` if the memory descriptor describes an empty memory and `false` otherwise.

**bool operator==(const desc &other) const**

An equality operator.

**Return** Whether this and the other memory descriptors have the same format tag, dimensions, strides, etc.

#### Parameters

- `other`: Another memory descriptor.

**bool operator!=(const desc &other) const**

An inequality operator.

**Return** Whether this and the other memory descriptors describe different memory.

#### Parameters

- `other`: Another memory descriptor.

### 7.4.2.2 Objects

Memory objects combine memory descriptors with storage for data (a data handle). With USM, the data handle is simply a pointer to `void`. The data handle can be queried using `dnnl::memory::get_data_handle()` and set using `dnnl::memory::set_data_handle()`. The underlying SYCL buffer, when used, can be queried using `dnnl::memory::get_sycl_buffer()` and set using `dnnl::memory::set_sycl_buffer()`. A memory object can also be queried for the underlying memory descriptor and for its engine using `dnnl::memory::get_desc()` and `dnnl::memory::get_engine()`.

### 7.4.2.2.3 API

**struct dnnl::memory**

Memory object.

A memory object encapsulates a handle to a memory buffer allocated on a specific engine, tensor dimensions, data type, and memory format, which is the way tensor indices map to offsets in linear memory space. Memory objects are passed to primitives during execution.

#### Public Functions

**memory()**

Default constructor.

Constructs an empty memory object, which can be used to indicate absence of a parameter.

**memory(const desc &md, const engine &aengine, void \*handle)**

Constructs a memory object.

Unless `handle` is equal to `DNNL_MEMORY_NONE`, the constructed memory object will have the underlying buffer set. In this case, the buffer will be initialized as if `dnnl::memory::set_data_handle()` had been called.

See `memory::set_data_handle()`

#### Parameters

- `md`: Memory descriptor.
- `aengine`: Engine to store the data on.
- `handle`: Handle of the memory buffer to use.
  - A pointer to the user-allocated buffer. In this case the library doesn't own the buffer.
  - The `DNNL_MEMORY_ALLOCATE` special value. Instructs the library to allocate the buffer for the memory object. In this case the library owns the buffer.
  - `DNNL_MEMORY_NONE` to create `dnnl::memory` without an underlying buffer.

```
template<typename T, int ndims = 1>
memory(const desc &md, const engine &aengine, cl::sycl::buffer<T, ndims> &buf)
```

Constructs a memory object from a SYCL buffer.

#### Parameters

- `md`: Memory descriptor.
- `aengine`: Engine to store the data on.
- `buf`: A SYCL buffer.

```
memory(const desc &md, const engine &aengine)
```

Constructs a memory object.

The underlying buffer for the memory will be allocated by the library.

#### Parameters

- `md`: Memory descriptor.
- `aengine`: Engine to store the data on.

```
desc get_desc() const
```

Returns the associated memory descriptor.

```
engine get_engine() const
```

Returns the associated engine.

```
void *get_data_handle() const
```

Returns the underlying memory buffer.

On the CPU engine, or when using USM, this is a pointer to the allocated memory.

```
void set_data_handle(void *handle, const stream &astream) const
```

Sets the underlying memory buffer.

This function may write zero values to the memory specified by the `handle` if the memory object has a zero padding area. This may be time consuming and happens each time this function is called. The operation is always blocking and the `stream` parameter is a hint.

**Note** Even when the memory object is used to hold values that stay constant during the execution of the program (pre-packed weights during inference, for example), the function will still write zeroes to the padding area if it exists. Hence, the `handle` parameter cannot and does not have a `const` qualifier.

### Parameters

- `handle`: Memory buffer to use. On the CPU engine or when USM is used, the memory buffer is a pointer to the actual data. It must have at least `dnnl::memory::desc::get_size()` bytes allocated.
- `astream`: Stream to use to execute padding in.

```
void set_data_handle(void *handle) const
    Sets the underlying memory buffer.
```

See documentation for `dnnl::memory::set_data_handle(void *, const stream &)` `const` for more information.

### Parameters

- `handle`: Memory buffer to use. For the CPU engine, the memory buffer is a pointer to the actual data. It must have at least `dnnl::memory::desc::get_size()` bytes allocated.

```
template<typename T = void>
T *map_data() const
```

Maps a memory object and returns a host-side pointer to a memory buffer with a copy of its contents.

Mapping enables read/write directly from/to the memory contents for engines that do not support direct memory access.

Mapping is an exclusive operation - a memory object cannot be used in other operations until it is unmapped via `dnnl::memory::unmap_data()` call.

**Note** Any primitives working with the memory should be completed before the memory is mapped. Use `dnnl::stream::wait()` to synchronize the corresponding execution stream.

**Note** The `map_data` and `unmap_data` functions are provided mainly for debug and testing purposes and their performance may be suboptimal.

**Return** Pointer to the mapped memory.

### Template Parameters

- `T`: Data type to return a pointer to.

```
void unmap_data(void *mapped_ptr) const
```

Unmaps a memory object and writes back any changes made to the previously mapped memory buffer.

**Note** The `map_data` and `unmap_data` functions are provided mainly for debug and testing purposes and their performance may be suboptimal.

### Parameters

- `mapped_ptr`: A pointer previously returned by `dnnl::memory::map_data()`.

```
template<typename T, int ndims = 1>
```

```
cl::sycl::buffer<T, ndims> get_sycl_buffer(size_t *offset = nullptr) const
```

Returns the underlying SYCL buffer object.

### Template Parameters

- `T`: Type of the requested buffer.

- `ndims`: Number of dimensions of the requested buffer.

#### Parameters

- `offset`: Offset within the returned buffer at which the memory object's data starts. Only meaningful for 1D buffers.

```
template<typename T, int ndims>
void set_sycl_buffer(cl::sycl::buffer<T, ndims> &buf)
```

Sets the underlying buffer to the given SYCL buffer.

#### Template Parameters

- `T`: Type of the buffer.
- `ndims`: Number of dimensions of the buffer.

#### Parameters

- `buf`: SYCL buffer.

#### `DNNL_MEMORY_NONE`

Special pointer value that indicates that a memory object should not have an underlying buffer.

#### `DNNL_MEMORY_ALLOCATE`

Special pointer value that indicates that the library needs to allocate an underlying buffer for a memory object.

## 7.5 Primitives

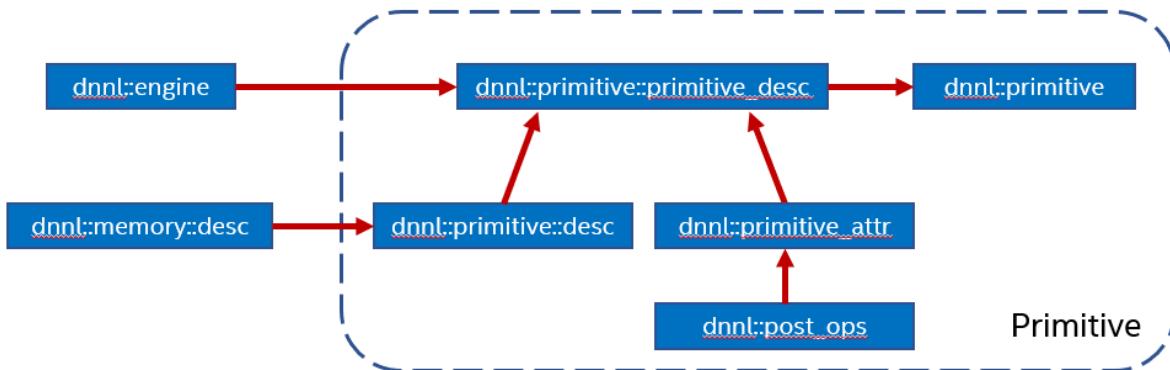
*Primitives* are functor objects that encapsulates a particular computation such as forward convolution, backward LSTM computations, or a data transformation operation. A single primitive can sometimes represent more complex fused computations such as a forward convolution followed by a ReLU.

The most important difference between a primitive and a pure function is that a primitive can store state.

One part of the primitive's state is immutable. For example, convolution primitives store parameters like tensor shapes and can pre-compute other dependent parameters like cache blocking. This approach allows oneDNN primitives to pre-generate code specifically tailored for the operation to be performed. The oneDNN programming model assumes that the time it takes to perform the pre-computations is amortized by reusing the same primitive to perform computations multiple times.

The mutable part of the primitive's state is referred to as a scratchpad. It is a memory buffer that a primitive may use for temporary storage only during computations. The scratchpad can either be owned by a primitive object (which makes that object non-thread safe) or be an execution-time parameter.

Conceptually, oneDNN establishes several layers of how to describe a computation from more abstract to more concrete:



- Operation descriptors (one for each supported primitive) describe an operation's most basic properties without specifying, for example, which engine will be used to compute them. For example, convolution descriptor describes shapes of source, destination, and weights tensors, propagation kind (forward, backward with respect to data or weights), and other implementation-independent parameters. The shapes are usually described as memory descriptors (`dnnl::memory::desc`).
- Primitive descriptors are at the abstraction level in between operation descriptors and primitives. They combine both an operation descriptor and primitive attributes. Primitive descriptors can be used to query various primitive implementation details and, for example, to implement *memory format propagation* by inspecting expected memory formats via queries without having to fully instantiate a primitive. oneDNN may contain multiple implementations for the same primitive that can be used to perform the same particular computation. Primitive descriptors allow one-way iteration which allows inspecting multiple implementations. The library is expected to order the implementations from most to least preferred, so it should always be safe to use the one that is chosen by default.
- Primitives, which are the most concrete, embody actual computations that can be executed.

On the API level:

- Primitives are represented as a class on the top level of the `dnnl` namespace that have `dnnl::primitive` as their base class, for example `dnnl::convolution_forward`
- Operation descriptors are represented as classes named `desc` and nested within the corresponding primitives classes, for example `dnnl::convolution_forward::desc`. The `dnnl::primitive_desc::next_impl()` member function provides a way to iterate over implementations.
- Primitive descriptors are represented as classes named `primitive_desc` and nested within the corresponding primitive classes that have `dnnl::primitive_desc_base` as their base class (except for RNN primitives that derive from `dnnl::rnn_primitive_desc_base`), for example `dnnl::convolution_forward::primitive_desc`

```

namespace dnnl {
    struct something_forward : public primitive {
        struct desc {
            // Primitive-specific constructors.
        }
        struct primitive_desc : public primitive_desc_base {
            // Constructors and primitive-specific memory descriptor queries.
        }
    };
}

```

The sequence of actions to create a primitive is:

1. Create an operation descriptor via, for example, `dnnl::convolution_forward::desc`. The operation descriptor can contain memory descriptors with placeholder `dnnl::memory::format_tag::any` memory formats if the primitive supports it.
2. Create a primitive descriptor based on the operation descriptor, engine and attributes.
3. Create a primitive based on the primitive descriptor obtained in step 1 above.

---

**Note:** Strictly speaking, not all the primitives follow this sequence. For example, the reorder primitive does not have an operation descriptor and thus does not require step 1 above.

---

## 7.5.1 Common Definitions

This section lists common types and definitions used by all or multiple primitives.

### 7.5.1.1 Base Class for Primitives

**struct dnnl::primitive**

Base class for all computational primitives.

Subclassed by `dnnl::batch_normalization_backward`, `dnnl::batch_normalization_forward`,  
`dnnl::binary`, `dnnl::concat`, `dnnl::convolution_backward_data`, `dnnl::convolution_backward_weights`,  
`dnnl::convolution_forward`, `dnnl::deconvolution_backward_data`, `dnnl::deconvolution_backward_weights`,  
`dnnl::deconvolution_forward`, `dnnl::eltwise_backward`, `dnnl::eltwise_forward`, `dnnl::gru_backward`,  
`dnnl::gru_forward`, `dnnl::inner_product_backward_data`, `dnnl::inner_product_backward_weights`,  
`dnnl::inner_product_forward`, `dnnl::layer_normalization_backward`, `dnnl::layer_normalization_forward`,  
`dnnl::lbr_gru_backward`, `dnnl::lbr_gru_forward`, `dnnl::logsoftmax_backward`, `dnnl::logsoftmax_forward`,  
`dnnl::lrn_backward`, `dnnl::lrn_forward`, `dnnl::lstm_backward`, `dnnl::lstm_forward`, `dnnl::matmul`,  
`dnnl::pooling_backward`, `dnnl::pooling_forward`, `dnnl::reorder`, `dnnl::resampling_backward`,  
`dnnl::resampling_forward`, `dnnl::shuffle_backward`, `dnnl::shuffle_forward`, `dnnl::softmax_backward`,  
`dnnl::softmax_forward`, `dnnl::sum`, `dnnl::vanilla_rnn_backward`, `dnnl::vanilla_rnn_forward`

### Public Types

**enum kind**

Kinds of primitives supported by the library.

*Values:*

**enumerator undef**

Undefined primitive.

**enumerator reorder**

A reorder primitive.

**enumerator shuffle**

A shuffle primitive.

**enumerator concat**

A (out-of-place) tensor concatenation primitive.

**enumerator sum**

A summation primitive.

---

```

enumerator convolution
    A convolution primitive.

enumerator deconvolution
    A deconvolution primitive.

enumerator eltwise
    An element-wise primitive.

enumerator softmax
    A softmax primitive.

enumerator pooling
    A pooling primitive.

enumerator lrn
    An LRN primitive.

enumerator batch_normalization
    A batch normalization primitive.

enumerator layer_normalization
    A layer normalization primitive.

enumerator inner_product
    An inner product primitive.

enumerator rnn
    An RNN primitive.

enumerator binary
    A binary primitive.

enumerator logsoftmax
    A logsoftmax primitive.

enumerator matmul
    A matmul (matrix multiplication) primitive.

enumerator resampling
    A resampling primitive.

```

## Public Functions

```

primitive()
    Default constructor. Constructs an empty object.

primitive(const primitive_desc_base &pd)
    Constructs a primitive from a primitive descriptor.

```

### Parameters

- pd: Primitive descriptor.

*kind* **get\_kind() const**  
 Returns the kind of the primitive.

**Return** The primitive kind.

---

```
void execute (const stream &astream, const std::unordered_map<int, memory> &args) const  
Executes computations specified by the primitive in a specified stream.
```

Arguments are passed via an arguments map containing <index, memory object> pairs. The index must be one of the DNNL\_ARG\_\* values such as DNNL\_ARG\_SRC, and the memory must have a memory descriptor matching the one returned by *dnnl::primitive\_desc\_base::query\_md(query::exec\_arg\_md, index)* unless using dynamic shapes (see *DNNL\_RUNTIME\_DIM\_VAL*).

#### Parameters

- *astream*: Stream object. The stream must belong to the same engine as the primitive.
- *args*: Arguments map.

```
cl::sycl::event execute_sycl (const stream &astream, const std::unordered_map<int, memory>  
&args, const std::vector<cl::sycl::event> &deps = {}) const  
Executes computations specified by the primitive in a specified stream.
```

Arguments are passed via an arguments map containing <index, memory object> pairs. The index must be one of the DNNL\_ARG\_\* values such as DNNL\_ARG\_SRC, and the memory must have a memory descriptor matching the one returned by *dnnl::primitive\_desc::query\_md(query::exec\_arg\_md, index)* unless using dynamic shapes (see *DNNL\_RUNTIME\_DIM\_VAL*).

#### Parameters

- *astream*: Stream object. The stream must belong to the same engine as the primitive.
- *args*: Arguments map.
- *deps*: Optional vector with *cl::sycl::event* dependencies.

```
primitive &operator= (const primitive &rhs)  
Assignment operator.
```

### 7.5.1.2 Base Class for Primitives Descriptors

There is no common base class for operation descriptors because they are very different between different primitives. However, there is a common base class for primitive descriptors.

```
struct dnnl::primitive_desc_base
```

Base class for all primitive descriptors.

Subclassed by *dnnl::concat::primitive\_desc*, *dnnl::primitive\_desc*, *dnnl::reorder::primitive\_desc*, *dnnl::sum::primitive\_desc*

#### Public Functions

```
primitive_desc_base()
```

Default constructor. Produces an empty object.

```
engine get_engine() const
```

Returns the engine of the primitive descriptor.

**Return** The engine of the primitive descriptor.

```
const char *impl_info_str() const
```

Returns implementation name.

**Return** The implementation name.

`memory::dim query_s64 (query what) const`  
 Returns a `memory::dim` value (same as `int64_t`).

**Return** The result of the query.

#### Parameters

- `what`: The value to query.

`memory::desc query_md (query what, int idx = 0) const`  
 Returns a memory descriptor.

**Note** There are also convenience methods `dnnl::primitive_desc_base::src_desc()`, `dnnl::primitive_desc_base::dst_desc()`, and others.

**Return** The requested memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a parameter of the specified kind or index.

#### Parameters

- `what`: The kind of parameter to query; can be `dnnl::query::src_md`, `dnnl::query::dst_md`, etc.
- `idx`: Index of the parameter. For example, convolution bias can be queried with `what = dnnl::query::weights_md` and `idx = 1`.

`memory::desc src_desc (int idx) const`  
 Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter with index `pdx`.

#### Parameters

- `idx`: Source index.

`memory::desc dst_desc (int idx) const`  
 Returns a destination memory descriptor.

**Return** Destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination parameter with index `pdx`.

#### Parameters

- `idx`: Destination index.

`memory::desc weights_desc (int idx) const`  
 Returns a weights memory descriptor.

**Return** Weights memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a weights parameter with index `pdx`.

#### Parameters

- `idx`: Weights index.

`memory::desc diff_src_desc (int idx) const`  
 Returns a diff source memory descriptor.

**Return** Diff source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff source parameter with index `pdx`.

**Parameters**

- `idx`: Diff source index.

`memory::desc diff_dst_desc (int idx) const`

Returns a diff destination memory descriptor.

**Return** Diff destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff destination parameter with index `pdx`.

**Parameters**

- `idx`: Diff destination index.

`memory::desc diff_weights_desc (int idx) const`

Returns a diff weights memory descriptor.

**Return** Diff weights memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff weights parameter with index `pdx`.

**Parameters**

- `idx`: Diff weights index.

`memory::desc src_desc () const`

Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter.

`memory::desc dst_desc () const`

Returns a destination memory descriptor.

**Return** Destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination parameter.

`memory::desc weights_desc () const`

Returns a weights memory descriptor.

**Return** Weights memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a weights parameter.

`memory::desc diff_src_desc () const`

Returns a diff source memory descriptor.

**Return** Diff source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff source memory with.

`memory::desc diff_dst_desc () const`

Returns a diff destination memory descriptor.

**Return** Diff destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff destination parameter.

`memory::desc diff_weights_desc () const`

Returns a diff weights memory descriptor.

**Return** Diff weights memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff weights parameter.

`memory::desc workspace_desc() const`  
 Returns the workspace memory descriptor.

**Return** Workspace memory descriptor.

**Return** A zero memory descriptor if the primitive does not require workspace parameter.

`memory::desc scratchpad_desc() const`  
 Returns the scratchpad memory descriptor.

**Return** scratchpad memory descriptor.

**Return** A zero memory descriptor if the primitive does not require scratchpad parameter.

`engine scratchpad_engine() const`

Returns the engine on which the scratchpad memory is located.

**Return** The engine on which the scratchpad memory is located.

`primitive_attr get_primitive_attr() const`

Returns the primitive attributes.

**Return** The primitive attributes.

`dnnl::primitive::kind get_kind() const`

Returns the kind of the primitive descriptor.

**Return** The kind of the primitive descriptor.

It is further derived from to provide base class for all primitives that have operation descriptors.

`struct dnnl::primitive_desc : public dnnl::primitive_desc_base`

A base class for descriptors of all primitives that have an operation descriptor and that support iteration over multiple implementations.

Subclassed by `dnnl::batch_normalization_backward::primitive_desc`, `dnnl::batch_normalization_forward::primitive_desc`,  
`dnnl::binary::primitive_desc`, `dnnl::convolution_backward_data::primitive_desc`,  
`dnnl::convolution_backward_weights::primitive_desc`, `dnnl::convolution_forward::primitive_desc`,  
`dnnl::deconvolution_backward_data::primitive_desc`, `dnnl::deconvolution_backward_weights::primitive_desc`,  
`dnnl::deconvolution_forward::primitive_desc`, `dnnl::eltwise_backward::primitive_desc`,  
`dnnl::eltwise_forward::primitive_desc`, `dnnl::inner_product_backward_data::primitive_desc`,  
`dnnl::inner_product_backward_weights::primitive_desc`, `dnnl::inner_product_forward::primitive_desc`,  
`dnnl::layer_normalization_backward::primitive_desc`, `dnnl::layer_normalization_forward::primitive_desc`,  
`dnnl::logsoftmax_backward::primitive_desc`, `dnnl::logsoftmax_forward::primitive_desc`,  
`dnnl::lrn_backward::primitive_desc`, `dnnl::lrn_forward::primitive_desc`, `dnnl::matmul::primitive_desc`,  
`dnnl::pooling_backward::primitive_desc`, `dnnl::pooling_forward::primitive_desc`,  
`dnnl::resampling_backward::primitive_desc`, `dnnl::resampling_forward::primitive_desc`,  
`dnnl::rnn_primitive_desc_base`, `dnnl::shuffle_backward::primitive_desc`, `dnnl::shuffle_forward::primitive_desc`,  
`dnnl::softmax_backward::primitive_desc`, `dnnl::softmax_forward::primitive_desc`

## Public Functions

`primitive_desc()`

Default constructor. Produces an empty object.

`bool next_impl()`

Advances the primitive descriptor iterator to the next implementation.

**Return** `true` on success, and `false` if the last implementation reached, in which case primitive descriptor is not modified.

The `dnnl::reorder`, `dnnl::sum` and `dnnl::concat` primitives also subclass `dnnl::primitive_desc` to implement their primitive descriptors.

RNN primitives further subclass the `dnnl::primitive_desc_base` to provide utility functions for frequently queried memory descriptors.

```
struct dnnl::rnn_primitive_desc_base : public dnnl::primitive_desc
    Base class for primitive descriptors for RNN primitives.

    Subclassed by dnnl::gru_backward::primitive_desc, dnnl::gru_forward::primitive_desc,
    dnnl::lbr_gru_backward::primitive_desc, dnnl::lbr_gru_forward::primitive_desc,
    dnnl::lstm_backward::primitive_desc, dnnl::lstm_forward::primitive_desc, dnnl::vanilla_rnn_backward::primitive_desc,
    dnnl::vanilla_rnn_forward::primitive_desc
```

## Public Functions

`rnn_primitive_desc_base()`

Default constructor. Produces an empty object.

`memory::desc src_layer_desc() const`

Returns source layer memory descriptor.

**Return** Source layer memory descriptor.

`memory::desc src_iter_desc() const`

Returns source iteration memory descriptor.

**Return** Source iteration memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source iteration parameter.

`memory::desc src_iter_c_desc() const`

Returns source recurrent cell state memory descriptor.

**Return** Source recurrent cell state memory descriptor.

`memory::desc weights_layer_desc() const`

Returns weights layer memory descriptor.

**Return** Weights layer memory descriptor.

`memory::desc weights_iter_desc() const`

Returns weights iteration memory descriptor.

**Return** Weights iteration memory descriptor.

`memory::desc weights_peephole_desc() const`

Returns weights peephole memory descriptor.

**Return** Weights peephole memory descriptor.

`memory::desc weights_projection_desc() const`

Returns weights projection memory descriptor.

**Return** Weights projection memory descriptor.

`memory::desc bias_desc() const`

Returns bias memory descriptor.

**Return** Bias memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a bias parameter.

`memory::desc dst_layer_desc() const`

Returns destination layer memory descriptor.

**Return** Destination layer memory descriptor.

`memory::desc dst_iter_desc() const`

Returns destination iteration memory descriptor.

**Return** Destination iteration memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination iteration parameter.

`memory::desc dst_iter_c_desc() const`

Returns destination recurrent cell state memory descriptor.

**Return** Destination recurrent cell state memory descriptor.

`memory::desc diff_src_layer_desc() const`

Returns diff source layer memory descriptor.

**Return** Diff source layer memory descriptor.

`memory::desc diff_src_iter_desc() const`

Returns diff source iteration memory descriptor.

**Return** Diff source iteration memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff source iteration parameter.

`memory::desc diff_src_iter_c_desc() const`

Returns diff source recurrent cell state memory descriptor.

**Return** Diff source recurrent cell state memory descriptor.

`memory::desc diff_weights_layer_desc() const`

Returns diff weights layer memory descriptor.

**Return** Diff weights layer memory descriptor.

`memory::desc diff_weights_iter_desc() const`

Returns diff weights iteration memory descriptor.

**Return** Diff weights iteration memory descriptor.

`memory::desc diff_weights_peephole_desc() const`

Returns diff weights peephole memory descriptor.

**Return** Diff weights peephole memory descriptor.

`memory::desc diff_weights_projection_desc() const`

Returns diff weights projection memory descriptor.

**Return** Diff weights projection memory descriptor.

`memory::desc diff_bias_desc() const`

Returns diff bias memory descriptor.

**Return** Diff bias memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff bias parameter.

`memory::desc diff_dst_layer_desc() const`

Returns diff destination layer memory descriptor.

**Return** Diff destination layer memory descriptor.

`memory::desc diff_dst_iter_desc() const`  
 Returns diff destination iteration memory descriptor.

**Return** Diff destination iteration memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff destination iteration parameter.

`memory::desc diff_dst_iter_c_desc() const`  
 Returns diff destination recurrent cell state memory descriptor.

**Return** Diff destination recurrent cell state memory descriptor.

### 7.5.1.3 Common Enumerations

`enum dnnl::prop_kind`

Propagation kind.

*Values:*

`enumerator undef`

Undefined propagation kind.

`enumerator forward_training`

Forward data propagation (training mode). In this mode, primitives perform computations necessary for subsequent backward propagation.

`enumerator forward_inference`

Forward data propagation (inference mode). In this mode, primitives perform only computations that are necessary for inference and omit computations that are necessary only for backward propagation.

`enumerator forward_scoring`

Forward data propagation, alias for `dnnl::prop_kind::forward_inference`.

`enumerator forward`

Forward data propagation, alias for `dnnl::prop_kind::forward_training`.

`enumerator backward`

Backward propagation (with respect to all parameters).

`enumerator backward_data`

Backward data propagation.

`enumerator backward_weights`

Backward weights propagation.

`enumerator backward_bias`

Backward bias propagation.

`enum dnnl::algorithm`

Kinds of algorithms.

*Values:*

`enumerator undef`

Undefined algorithm.

`enumerator convolution_auto`

Convolution algorithm that is chosen to be either direct or Winograd automatically

`enumerator convolution_direct`

Direct convolution.

---

```
enumerator convolution_winograd
    Winograd convolution.

enumerator deconvolution_direct
    Direct deconvolution.

enumerator deconvolution_winograd
    Winograd deconvolution.

enumerator eltwise_relu
    Elementwise: rectified linear unit (ReLU)

enumerator eltwise_tanh
    Elementwise: hyperbolic tangent non-linearity (tanh)

enumerator eltwise_elu
    Elementwise: exponential linear unit (ELU)

enumerator eltwise_square
    Elementwise: square.

enumerator eltwise_abs
    Elementwise: abs.

enumerator eltwise_sqrt
    Elementwise: square root.

enumerator eltwise_swish
    Elementwise: swish ( $x \cdot \text{sigmoid}(a \cdot x)$ )

enumerator eltwise_linear
    Elementwise: linear.

enumerator eltwise_bounded_relu
    Elementwise: bounded_relu.

enumerator eltwise_soft_relu
    Elementwise: soft_relu.

enumerator eltwise_logistic
    Elementwise: logistic.

enumerator eltwise_exp
    Elementwise: exponent.

enumerator eltwise_gelu
    Elementwise: gelu alias for dnnl::algorithm::eltwise\_gelu\_tanh

enumerator eltwise_gelu_tanh
    Elementwise: tanh-based gelu.

enumerator eltwise_gelu_erf
    Elementwise: erf-based gelu.

enumerator eltwise_log
    Elementwise: natural logarithm.

enumerator eltwise_clip
    Elementwise: clip.

enumerator eltwise_pow
    Elementwise: pow.
```

---

```

enumerator eltwise_round
    Elementwise: round.

enumerator eltwise_relu_use_dst_for_bwd
    Elementwise: rectified linear unit (ReLU) (dst for backward)

enumerator eltwise_tanh_use_dst_for_bwd
    Elementwise: hyperbolic tangent non-linearity (tanh) (dst for backward)

enumerator eltwise_elu_use_dst_for_bwd
    Elementwise: exponential linear unit (ELU) (dst for backward)

enumerator eltwise_sqrt_use_dst_for_bwd
    Elementwise: square root (dst for backward)

enumerator eltwise_logistic_use_dst_for_bwd
    Elementwise: logistic (dst for backward)

enumerator eltwise_exp_use_dst_for_bwd
    Elementwise: exponent (dst for backward)

enumerator lrn_across_channels
    Local response normalization (LRN) across multiple channels.

enumerator lrn_within_channel
    LRN within a single channel.

enumerator pooling_max
    Max pooling.

enumerator pooling_avg
    Average pooling exclude padding, alias for dnnl::algorithm::pooling\_avg\_include\_padding

enumerator pooling_avg_include_padding
    Average pooling include padding.

enumerator pooling_avg_exclude_padding
    Average pooling exclude padding.

enumerator vanilla_rnn
    RNN cell.

enumerator vanilla_lstm
    LSTM cell.

enumerator vanilla_gru
    GRU cell.

enumerator lbr_gru
    GRU cell with linear before reset. Differs from original GRU in how the new memory gate is calculated:  

 $c_t = \tanh(W_c * x_t + b_{c_x} + r_t * (U_c * h_{t-1} + b_{c_h}))$  LRB GRU expects 4 bias tensors on input:  $[b_u, b_r, b_{c_x}, b_{c_h}]$ 

enumerator binary_add
    Binary add.

enumerator binary_mul
    Binary mul.

enumerator binary_max
    Binary max.

enumerator binary_min
    Binary min.

```

---

```
enumerator resampling_nearest
    Nearest Neighbor resampling method.

enumerator resampling_linear
    Linear (Bilinear, Trilinear) resampling method.
```

#### 7.5.1.4 Normalization Primitives Flags

**enum dnnl::normalization\_flags**  
Flags for normalization primitives (can be combined via ‘|’)

*Values:*

**enumerator none**

Use no normalization flags. If specified, the library computes mean and variance on forward propagation for training and inference, outputs them on forward propagation for training, and computes the respective derivatives on backward propagation.

**enumerator use\_global\_stats**

Use global statistics. If specified, the library uses mean and variance provided by the user as an input on forward propagation and does not compute their derivatives on backward propagation. Otherwise, the library computes mean and variance on forward propagation for training and inference, outputs them on forward propagation for training, and computes the respective derivatives on backward propagation.

**enumerator use\_scale\_shift**

Use scale and shift parameters. If specified, the user is expected to pass scale and shift as inputs on forward propagation. On backward propagation of type *dnnl::prop\_kind::backward*, the library computes their derivatives. If not specified, the scale and shift parameters are not used by the library in any way.

**enumerator fuse\_norm\_relu**

Fuse normalization with ReLU. On training, normalization will require the workspace to implement backward propagation. On inference, the workspace is not required and behavior is the same as when normalization is fused with ReLU using the post-ops API.

#### 7.5.1.5 Execution argument indices

**DNNL\_ARG\_SRC\_0**

Source argument #0.

**DNNL\_ARG\_SRC**

A special mnemonic for source argument for primitives that have a single source. An alias for *DNNL\_ARG\_SRC\_0*.

**DNNL\_ARG\_SRC\_LAYER**

A special mnemonic for RNN input vector. An alias for *DNNL\_ARG\_SRC\_0*.

**DNNL\_ARG\_FROM**

A special mnemonic for reorder source argument. An alias for *DNNL\_ARG\_SRC\_0*.

**DNNL\_ARG\_SRC\_1**

Source argument #1.

**DNNL\_ARG\_SRC\_ITER**

A special mnemonic for RNN input recurrent hidden state vector. An alias for *DNNL\_ARG\_SRC\_1*.

**DNNL\_ARG\_SRC\_2**

Source argument #2.

**DNNL\_ARG\_SRC\_ITER\_C**

A special mnemonic for RNN input recurrent cell state vector. An alias for [DNNL\\_ARG\\_SRC\\_2](#).

**DNNL\_ARG\_DST\_0**

Destination argument #0.

**DNNL\_ARG\_DST**

A special mnemonic for destination argument for primitives that have a single destination. An alias for [DNNL\\_ARG\\_DST\\_0](#).

**DNNL\_ARG\_TO**

A special mnemonic for reorder destination argument. An alias for [DNNL\\_ARG\\_DST\\_0](#).

**DNNL\_ARG\_DST\_LAYER**

A special mnemonic for RNN output vector. An alias for [DNNL\\_ARG\\_DST\\_0](#).

**DNNL\_ARG\_DST\_1**

Destination argument #1.

**DNNL\_ARG\_DST\_ITER**

A special mnemonic for RNN input recurrent hidden state vector. An alias for [DNNL\\_ARG\\_DST\\_1](#).

**DNNL\_ARG\_DST\_2**

Destination argument #2.

**DNNL\_ARG\_DST\_ITER\_C**

A special mnemonic for LSTM output recurrent cell state vector. An alias for [DNNL\\_ARG\\_DST\\_2](#).

**DNNL\_ARG\_WEIGHTS\_0**

Weights argument #0.

**DNNL\_ARG\_WEIGHTS**

A special mnemonic for primitives that have a single weights argument. Alias for [DNNL\\_ARG\\_WEIGHTS\\_0](#).

**DNNL\_ARG\_SCALE\_SHIFT**

A special mnemonic for scale and shift argument of normalization primitives. Alias for [DNNL\\_ARG\\_WEIGHTS\\_0](#).

**DNNL\_ARG\_WEIGHTS\_LAYER**

A special mnemonic for RNN weights applied to the layer input. An alias for [DNNL\\_ARG\\_WEIGHTS\\_0](#).

**DNNL\_ARG\_WEIGHTS\_1**

Weights argument #1.

**DNNL\_ARG\_WEIGHTS\_ITER**

A special mnemonic for RNN weights applied to the recurrent input. An alias for [DNNL\\_ARG\\_WEIGHTS\\_1](#).

**DNNL\_ARG\_BIAS**

Bias tensor argument.

**DNNL\_ARG\_MEAN**

Mean values tensor argument.

**DNNL\_ARG\_VARIANCE**

Variance values tensor argument.

**DNNL\_ARG\_WORKSPACE**

Workspace tensor argument. Workspace is used to pass information from forward propagation to backward propagation computations.

**DNNL\_ARG\_SCRATCHPAD**

Scratchpad (temporary storage) tensor argument.

**DNNL\_ARG\_DIFF\_SRC\_0**

Gradient (diff) of the source argument #0.

**DNNL\_ARG\_DIFF\_SRC**

A special mnemonic for primitives that have a single diff source argument. An alias for [DNNL\\_ARG\\_DIFF\\_SRC\\_0](#).

**DNNL\_ARG\_DIFF\_SRC\_LAYER**

A special mnemonic for gradient (diff) of RNN input vector. An alias for [DNNL\\_ARG\\_DIFF\\_SRC\\_0](#).

**DNNL\_ARG\_DIFF\_SRC\_1**

Gradient (diff) of the source argument #1.

**DNNL\_ARG\_DIFF\_SRC\_ITER**

A special mnemonic for gradient (diff) of RNN input recurrent hidden state vector. An alias for [DNNL\\_ARG\\_DIFF\\_SRC\\_1](#).

**DNNL\_ARG\_DIFF\_SRC\_2**

Gradient (diff) of the source argument #2.

**DNNL\_ARG\_DIFF\_SRC\_ITER\_C**

A special mnemonic for gradient (diff) of RNN input recurrent cell state vector. An alias for [DNNL\\_ARG\\_DIFF\\_SRC\\_1](#).

**DNNL\_ARG\_DIFF\_DST\_0**

Gradient (diff) of the destination argument #0.

**DNNL\_ARG\_DIFF\_DST**

A special mnemonic for primitives that have a single diff destination argument. An alias for [DNNL\\_ARG\\_DIFF\\_DST\\_0](#).

**DNNL\_ARG\_DIFF\_DST\_LAYER**

A special mnemonic for gradient (diff) of RNN output vector. An alias for [DNNL\\_ARG\\_DIFF\\_DST\\_0](#).

**DNNL\_ARG\_DIFF\_DST\_1**

Gradient (diff) of the destination argument #1.

**DNNL\_ARG\_DIFF\_DST\_ITER**

A special mnemonic for gradient (diff) of RNN input recurrent hidden state vector. An alias for [DNNL\\_ARG\\_DIFF\\_DST\\_1](#).

**DNNL\_ARG\_DIFF\_DST\_2**

Gradient (diff) of the destination argument #2.

**DNNL\_ARG\_DIFF\_DST\_ITER\_C**

A special mnemonic for gradient (diff) of RNN input recurrent cell state vector. An alias for [DNNL\\_ARG\\_DIFF\\_DST\\_2](#).

**DNNL\_ARG\_DIFF\_WEIGHTS\_0**

Gradient (diff) of the weights argument #0.

**DNNL\_ARG\_DIFF\_WEIGHTS**

A special mnemonic for primitives that have a single diff weights argument. Alias for [DNNL\\_ARG\\_DIFF\\_WEIGHTS\\_0](#).

**DNNL\_ARG\_DIFF\_SCALE\_SHIFT**

A special mnemonic for diff of scale and shift argument of normalization primitives. Alias for [DNNL\\_ARG\\_DIFF\\_WEIGHTS\\_0](#).

**DNNL\_ARG\_DIFF\_WEIGHTS\_LAYER**

A special mnemonic for diff of RNN weights applied to the layer input. An alias for [DNNL\\_ARG\\_DIFF\\_WEIGHTS\\_0](#).

**DNNL\_ARG\_DIFF\_WEIGHTS\_1**

Gradient (diff) of the weights argument #1.

**DNNL\_ARG\_DIFF\_WEIGHTS\_ITER**

A special mnemonic for diff of RNN weights applied to the recurrent input. An alias for [DNNL\\_ARG\\_DIFF\\_WEIGHTS\\_1](#).

**DNNL\_ARG\_DIFF\_BIAS**

Gradient (diff) of the bias tensor argument.

**DNNL\_ARG\_ATTR\_OUTPUT\_SCALES**

Output scaling factors provided at execution time.

**DNNL\_ARG\_MULTIPLE\_SRC**

Starting index for source arguments for primitives that take a variable number of source arguments.

**DNNL\_ARG\_MULTIPLE\_DST**

Starting index for destination arguments for primitives that produce a variable number of destination arguments.

**DNNL\_ARG\_ATTR\_ZERO\_POINTS**

Zero points provided at execution time.

**DNNL\_RUNTIME\_DIM\_VAL**

A wildcard value for dimensions that are unknown at a primitive creation time.

**DNNL\_RUNTIME\_SIZE\_VAL**

A size\_t counterpart of the [DNNL\\_RUNTIME\\_DIM\\_VAL](#). For instance, this value is returned by [dnnl::memory::desc::get\\_size\(\)](#) if either of the dimensions or strides equal to [DNNL\\_RUNTIME\\_DIM\\_VAL](#).

**DNNL\_RUNTIME\_F32\_VAL**

A wildcard value for floating point values that are unknown at a primitive creation time.

**DNNL\_RUNTIME\_S32\_VAL**

A wildcard value for int32\_t values that are unknown at a primitive creation time.

## 7.5.2 Attributes

The parameters passed to create a primitive descriptor specify the problem. An engine specifies where the primitive will be executed. An operation descriptor specifies the basics: the operation kind; the propagation kind; the source, destination, and other tensors; the strides (if applicable); and so on.

*Attributes* specify some extra properties of the primitive. Users must create them before use and must set required specifics using the corresponding setters. The attributes are copied during primitive descriptor creation, so users can change or destroy attributes right after that.

If not modified, attributes can stay empty, which is equivalent to the default attributes. Primitive descriptors' constructors have empty attributes as default parameters, so unless required users can simply omit them.

Attributes can also contain *post-ops*, which are computations executed after the primitive.

### 7.5.2.1 Post-ops

*Post-ops* are operations that are appended after a primitive. They are implemented using the *Attributes* mechanism. If there are multiple post-ops, they are executed in the order they have been appended.

The post-ops are represented by `dnnl::post_ops` which is copied once it is attached to the attributes using `dnnl::primitive_attr::set_post_ops()` function. The attributes then need to be passed to a primitive descriptor creation function to take effect. Below is a simple sketch:

```
dnnl::post_ops po; // default empty post-ops
assert(po.len() == 0); // no post-ops attached

po.append_SOMETHING(params); // append some particular post-op
po.append_SOMETHING_ELSE(other_params); // append one more post-op

// (!) Note that the order in which post-ops are appended matters!
assert(po.len() == 2);

dnnl::primitive_attr attr; // default attributes
attr.set_post_ops(po); // attach the post-ops to the attr
// any changes to po after this point don't affect the value stored in attr

primitive::primitive_desc op_pd(params, attr); // create a pd with the attr
```

---

**Note:** Different primitives may have different post-ops support. Moreover, the support might also depend on the actual implementation of a primitive. So robust code should be able to handle errors accordingly. See the [Attribute Related Error Handling](#).

---

**Note:** Post-ops do not change memory format of the operation destination memory object.

---

The post-op objects can be inspected using the `dnnl::post_ops::kind()` function that takes an index of the post-op to inspect (that must be less than the value returned by `dnnl::post_ops::len()`), and returns its kind.

#### 7.5.2.1.1 Supported Post-ops

##### 7.5.2.1.1.1 Eltwise Post-op

The eltwise post-op is appended using `dnnl::post_ops::append_eltwise()` function. The `dnnl::post_ops::kind()` returns `dnnl::primitive::kind::eltwise` for such a post-op.

The eltwise post-op replaces:

$$\text{dst}[:] = \text{Op}(...)$$

with

$$\text{dst}[:] = \text{scale} \cdot \text{eltwise}(\text{Op}(...))$$

The intermediate result of the `Op(...)` is not preserved.

The `scale` factor is supported in `int8` inference only. For all other cases the scale must be `1.0`.

### 7.5.2.1.1.2 Sum Post-op

The sum post-op accumulates the result of a primitive with the existing data and is appended using `dnnl::post_ops::append_sum()` function. The `dnnl::post_ops::kind()` returns `dnnl::primitive::kind::sum` for such a post-op.

Prior to accumulating the result, the existing value is multiplied by scale. The scale parameter can be used in The `scale` factor is supported in `int8` inference only and should be used only when the result and the existing data have different magnitudes. For all other cases the scale must be `1.0`.

Additionally, the sum post-op can reinterpret the destination values as a different data type of the same size. This may be used to, for example, reinterpret 8-bit signed data as unsigned or vice versa (which requires that values fall within a common range to work).

The sum post-op replaces

$$\text{dst}[:] = \text{Op}(...)$$

with

$$\text{dst}[:] = \text{scale} \cdot \text{as}_\text{datatype}(\text{dst}[:]) + \text{Op}(...)$$

### 7.5.2.1.1.3 Examples of Chained Post-ops

Post-ops can be chained together by appending one after another. Note that the order matters: the post-ops are executed in the order they have been appended.

### 7.5.2.1.1.4 Sum -> ReLU

This pattern is pretty common for the CNN topologies of the ResNet family.

```
dnnl::post_ops po;
po.append_sum(
    /* scale = */ 1.f);
po.append_eltwise(
    /* scale      = */ 1.f,
    /* algorithm = */ dnnl::algorithm::eltwise_relu,
    /* neg slope = */ 0.f,
    /* unused for ReLU */ 0.f);

dnnl::primitive_attr attr;
attr.set_post_ops(po);

convolution_forward::primitive_desc(conv_d, attr, engine);
```

This will lead to the following computations:

$$\text{dst}[:] = \text{ReLU}(\text{dst}[:] + \text{conv}(\text{src}[:], \text{weights}[:]))$$

### 7.5.2.1.2 API

**struct** `dnnl::post_ops`

Post-ops.

Post-ops are computations executed after the main primitive computations and are attached to the primitive via primitive attributes.

#### Public Functions

**post\_ops()**

Constructs an empty sequence of post-ops.

**int len() const**

Returns the number of post-ops entries.

*primitive::kind kind(int index) const*

Returns the primitive kind of post-op at entry with a certain index.

**Return** Primitive kind of the post-op at the specified index.

#### Parameters

- `index`: Index of the post-op to return the kind for.

**void append\_sum(float scale = 1.f, memory::data\_type data\_type = memory::data\_type::undef)**

Appends an accumulation (sum) post-op. Prior to accumulating the result, the previous value would be multiplied by a scaling factor `scale`.

The kind of this post-op is `dnnl::primitive::kind::sum`.

This feature may improve performance for cases like residual learning blocks, where the result of convolution is accumulated to the previously computed activations. The parameter `scale` may be used for the integer-based computations when the result and previous activations have different logical scaling factors.

In the simplest case when the accumulation is the only post-op, the computations would be `dst[:] := scale * dst[:] + op(...)` instead of `dst[:] := op(...)`.

If `data_type` is specified, the original `dst` tensor will be reinterpreted as a tensor with the provided data type. Because it is a reinterpretation, `data_type` and `dst` data type should have the same size. As a result, computations would be `dst[:] <- scale * as_data_type(dst[:]) + op(...)` instead of `dst[:] <- op(...)`.

**Note** This post-op executes in-place and does not change the destination layout.

#### Parameters

- `scale`: Scaling factor.
- `data_type`: Data type.

**void get\_params\_sum(int index, float &scale) const**

Returns the parameters of an accumulation (sum) post-op.

#### Parameters

- `index`: Index of the sum post-op.
- `scale`: Scaling factor of the sum post-op.

---

```
void get_params_sum(int index, float &scale, memory::data_type &data_type) const
    Returns the parameters of an accumulation (sum) post-op.
```

#### Parameters

- **index**: Index of the sum post-op.
- **scale**: Scaling factor of the sum post-op.
- **data\_type**: Data type of the sum post-op.

```
void append_eltwise(float scale, algorithm aalgorithm, float alpha, float beta)
    Appends an elementwise post-op.
```

The kind of this post-op is *dnnl::primitive::kind::eltwise*.

In the simplest case when the elementwise is the only post-op, the computations would be `dst[ :] := scale * eltwise_op(op(...))` instead of `dst[ :] <- op(...)`, where `eltwise_op` is configured with the given parameters.

#### Parameters

- **scale**: Scaling factor.
- **aalgorithm**: Elementwise algorithm.
- **alpha**: Alpha parameter for the elementwise algorithm.
- **beta**: Beta parameter for the elementwise algorithm.

```
void get_params_eltwise(int index, float &scale, algorithm &aalgorithm, float &alpha, float
    &beta) const
    Returns parameters of an elementwise post-up.
```

#### Parameters

- **index**: Index of the post-op.
- **scale**: Output scaling factor.
- **aalgorithm**: Output elementwise algorithm kind.
- **alpha**: Output alpha parameter for the elementwise algorithm.
- **beta**: Output beta parameter for the elementwise algorithm.

### 7.5.2.2 Scratchpad Mode

Some primitives might require a temporary buffer while performing their computations. For instance, the operations that do not have enough independent work to utilize all cores on a system might use parallelization over the reduction dimension (the K dimension in the GEMM notation). In this case different threads compute partial results in private temporary buffers, and then the private results are added to produce the final result. Another example is using matrix multiplication (GEMM) to implement convolution. Before calling GEMM, the source activations need to be transformed using the `im2col` operation. The transformation result is written to a temporary buffer that is then used as an input for the GEMM.

In both of these examples, the temporary buffer is no longer required once the primitive computation is completed. oneDNN refers to such kind of a memory buffer as a *scratchpad*.

Both types of implementation might need extra space for the reduction in case there are too few independent tasks. The amount of memory required by the `im2col` transformation is proportional to the size of the source image multiplied

by the weights spatial size. The size of a buffer for reduction is proportional to the tensor size to be reduced (e.g., `diff_weights` in the case of backward by weights) multiplied by the number of threads in the reduction groups (the upper bound is the total number of threads).

By contrast, some other primitives might require very little extra space. For instance, one of the implementation of the `dnnl::sum` primitive requires temporary space only to store the pointers to data for each and every input array (that is, the size of the scratchpad is `n * sizeof(void *)`, where `n` is the number of summands).

oneDNN supports two modes for handling scratchpads:

```
enum dnnl::scratchpad_mode
    Scratchpad mode.
```

*Values:*

#### **enumerator library**

The library manages the scratchpad allocation. There may be multiple implementation-specific policies that can be configured via mechanisms that fall outside of the scope of this specification.

#### **enumerator user**

The user manages the scratchpad allocation by querying and providing the scratchpad memory to primitives. This mode is thread-safe as long as the scratchpad buffers are not used concurrently by two primitive executions.

The scratchpad mode is controlled though the `dnnl::primitive_attr::set_scratchpad_mode()` primitive attributes.

If the user provides scratchpad memory to a primitive, this memory must be created using the same engine that the primitive uses.

All primitives support both scratchpad modes.

---

**Note:** Primitives are not thread-safe by default. The only way to make the primitive execution fully thread-safe is to use the `dnnl::scratchpad_mode::user` mode and not pass the same scratchpad memory to two primitives that are executed concurrently.

---

### 7.5.2.2.1 Examples

#### 7.5.2.2.1.1 Library Manages Scratchpad

As mentioned above, this is a default behavior. We only want to highlight how a user can query the amount of memory consumed by a primitive due to a scratchpad.

```
// Use default attr, hence the library allocates scratchpad
dnnl::primitive::primitive_desc op_pd(params, /* other arguments */);

// Print how much memory would be hold by a primitive due to scratchpad
std::cout << "primitive will use "
    << op_pd.query_s64(dnnl::query::memory_consumption_s64)
    << " bytes" << std::endl;

// In this case scratchpad is internal, hence user visible scratchpad memory
// descriptor should be empty:
auto zero_md = dnnl::memory::desc();
```

### 7.5.2.2.1.2 User Manages Scratchpad

```

// Create an empty (default) attributes
dnnl::primitive_attr attr;

// Default scratchpad mode is `library`:
assert(attr.get_scratchpad_mode() == dnnl::scratchpad_mode::library);

// Set scratchpad mode to `user`
attr.set_scratchpad_mode(dnnl::scratchpad_mode::user);

// Create a primitive descriptor with custom attributes
dnnl::primitive::primitive_desc op_pd(op_d, attr, engine);

// Query the scratchpad memory descriptor
dnnl::memory::desc scratchpad_md = op_pd.scratchpad_desc();

// Note, that a primitive doesn't consume memory in this configuration:
assert(op_pd.query_s64(dnnl::query::memory_consumption_s64) == 0);

// Create a primitive
dnnl::primitive prim(op_pd);

// ... more code ...

// Create a scratchpad memory
// NOTE: if scratchpad is not required for a particular primitive the
//       scratchpad_md.get_size() will return 0. It is fine to have
//       scratchpad_ptr == nullptr in this case.
void *scratchpad_ptr = user_memory_manager::allocate(scratchpad_md.get_size());
// NOTE: engine here must match the engine of the primitive
dnnl::memory scratchpad(scratchpad_md, engine, scratchpad_ptr);

// Pass a scratchpad memory to a primitive
prim.execute(stream, { /* other arguments */,
    {DNNL_ARG_SCRATCHPAD, scratchpad}});

```

### 7.5.2.3 Quantization

Primitives may support reduced precision computations which require quantization.

#### 7.5.2.3.1 Quantization Model

The primary quantization model that the library assumes is the following:

$$x_{f32}[:] = scale_{f32} \cdot (x_{int8}[:] - 0_{x_{int8}})$$

where  $scale_{f32}$  is a *scaling factor* that is somehow known in advance and  $[:]$  is used to denote elementwise application of the formula to the arrays. Typically, the process of computing scale factors is called *calibration*. The library cannot compute any of the scale factors at run-time dynamically. Hence, the model is sometimes called a *static* quantization model. The main rationale to support only *static* quantization out-of-the-box is higher performance. To use *dynamic* quantization:

1. Compute the result in higher precision, like `dnnl::memory::data_type::s32`.

2. Find the required characteristics, like min and max values, and derive the scale factor.
3. Re-quantize to the lower precision data type.

oneDNN assumes a fixed zero position. For most of the primitives, the real zero value is mapped to the zero for quantized values; that is,  $0_{x_{int8}} = 0$ . For example, this is the only model that *Convolution and Deconvolution* and *Inner Product* currently support. The *RNN* primitives have limited support of shifted zero.

For the rest of this section we that  $0_{x_{int8}} = 0$ .

#### 7.5.2.3.1.1 Example: Convolution Quantization Workflow

Consider a convolution without bias. The tensors are represented as:

- $\text{src}_{f32}[:] = \text{scale}_{\text{src}} \cdot \text{src}_{int8}[:]$
- $\text{weights}_{f32}[:] = \text{scale}_{\text{weights}} \cdot \text{weights}_{int8}[:]$
- $\text{dst}_{f32}[:] = \text{scale}_{\text{dst}} \cdot \text{dst}_{int8}[:]$

Here the  $\text{src}_{f32}$ ,  $\text{weights}_{f32}$ ,  $\text{dst}_{f32}$  are not computed at all, the whole work happens with  $int8$  tensors. As mentioned above, we also somehow know all the scaling factors:  $\text{scale}_{\{\text{src}\}}$ ,  $\text{scale}_{\{\text{weights}\}}$ ,  $\text{scale}_{\{\text{dst}\}}$ .

So the task is to compute the  $\text{dst}_{int8}$  tensor.

Mathematically, the computations are:

$$\text{dst}_{int8}[:] = \text{f32\_to\_int8}(\text{output\_scale} \cdot \text{conv}_{s32}(\text{src}_{int8}, \text{weights}_{int8})),$$

where

- $\text{output\_scale} := \frac{\text{scale}_{\text{src}} \cdot \text{scale}_{\text{weights}}}{\text{scale}_{\text{dst}}};$
- $\text{conv}_{s32}$  is just a regular convolution which takes source and weights with  $int8$  data type and compute the result in  $int32$  data type ( $int32$  is chosen to avoid overflows during the computations);
- $\text{f32\_to\_s8}()$  converts an  $f32$  value to  $s8$  with potential saturation if the values are out of the range of the  $int8$  data type.

Note that in order to perform the operation, one doesn't need to know the exact scaling factors for all the tensors; it is enough to know only the  $\text{output\_scale}$ . The library utilizes this fact: a user needs to provide only this one extra parameter to the convolution primitive (see the *Output Scaling Attribute* section below).

#### 7.5.2.3.1.2 Per-Channel Scaling

Primitives may have limited support of multiple scales for a quantized tensor. The most popular use case is the *Convolution and Deconvolution* primitives that support per-output-channel scaling factors for the weights, meaning that the actual convolution computations would need to scale different output channels differently.

Let  $\alpha$  denote scales:

- $\text{src}_{f32}(n, ic, ih, iw) = \alpha_{\text{src}} \cdot \text{src}_{int8}(n, ic, ih, iw)$
- $\text{weights}_{f32}(oc, ic, kh, kw) = \alpha_{\text{weights}}(oc) \cdot \text{weights}_{int8}(oc, ic, kh, kw)$
- $\text{dst}_{f32}(n, oc, oh, ow) = \text{scale}_{\text{dst}} \cdot \text{dst}_{int8}(n, oc, oh, ow)$

Note that now the weights' scaling factor depends on the  $oc$ .

To compute the  $\text{dst}_{int8}$  we need to perform the following:

$$\text{dst}_{int8}(n, oc, oh, ow) = \text{f32\_to\_int8}(\text{output\_scale}(oc) \cdot \text{conv}_{s32}(\text{src}_{int8}, \text{weights}_{int8})|_{(n, oc, oh, ow)}),$$

where

$$\text{output\_scale}(oc) := \frac{\alpha_{\text{src}} \cdot \alpha_{\text{weights}}(oc)}{\alpha_{\text{dst}}}.$$

The user is responsible for preparing quantized weights accordingly. To do that, oneDNN provides reorders that can perform per-channel scaling:

$$\text{weights}_{\text{int8}}(oc, ic, kh, kw) = \text{f32\_to\_int8}(\text{output\_scale}(oc) \cdot \text{weights}_{\text{f32}}(oc, ic, kh, kw)),$$

where

$$\text{output\_scale}(oc) := \frac{1}{\alpha_{\text{weights}}(oc)}.$$

### 7.5.2.3.2 Output Scaling Attribute

oneDNN provides `dnnl::primitive_attr::set_output_scales()` for setting scaling factors for most of the primitives.

The primitives may not support output scales if source (and weights) tensors are not of the int8 data type. In other words, convolution operating on the single precision floating point data type may not scale the output result.

In the simplest case, when there is only one common scale the attribute changes the op behavior from

$$\text{dst}[:] = \text{Op}(\dots)$$

to

$$\text{dst}[:] = \text{scale} \cdot \text{Op}(\dots).$$

To support scales per one or several dimensions, users must set the appropriate mask.

Say the primitive destination is a  $D_0 \times \dots \times D_{n-1}$  tensor and we want to have output scales per  $d_i$  dimension (where  $0 \leq d_i < n$ ).

Then  $\text{mask} = \sum_{d_i} 2^{d_i}$  and the number of scales should be  $\text{scales.size()} = \prod_{d_i} D_{d_i}$ .

The scaling happens in the single precision floating point data type (`dnnl::memory::data_type::f32`). Before it is stored, the result is converted to the destination data type with saturation if required. The rounding happens according to the current hardware setting.

#### 7.5.2.3.2.1 Example 1: weights quantization with per-output-channel-and-group scaling

```
// weights dimensions
const int G, OC, IC, KH, KW;

// original f32 weights in plain format
dnnl::memory::desc wei_plain_f32_md(
    {G, OC/G, IC/G, KH, KW},           // dims
    dnnl::memory::data_type::f32,       // the data originally in f32
    dnnl::memory::format_tag::hwigo   // the plain memory format
);

// the scaling factors for quantized weights
// An unique scale for each group and output-channel.
```

(continues on next page)

(continued from previous page)

```

std::vector<float> wei_scales(G * OC/G) = { /* values */ };

// int8 convolution primitive descriptor
dnnl::convolution_forward::primitive_desc conv_pd(/* see the next example */);

// query the convolution weights memory descriptor
dnnl::memory::desc wei_conv_s8_md = conv_pd.weights_desc();

// prepare the inverse of the scales
// (f32 = scale * int8 --> int8 = 1/scale * f32)
std::vector<float> inv_wei_scales(wei_scales.size());
for (size_t i = 0; i < wei_scales.size(); ++i)
    inv_wei_scales[i] = 1.f / wei_scales[i];

// prepare the attributes for the reorder
dnnl::primitive_attr attr;
const int mask = 0
    | (1 << 0) // scale per G dimension, which is the dim #0
    | (1 << 1); // scale per OC dimension, which is the dim #1
attr.set_output_scales(mask, inv_wei_scales);

// create reorder that would perform:
// wei_s8(g, oc, ic, kh, kw) <- 1/scale(g, oc) * wei_f32(g, oc, ic, kh, kw)
// including the data format transformation.
auto wei_reorder_pd = dnnl::reorder::primitive_desc(
    wei_plain_f32_md, engine, // source
    wei_conv_s8_md, engine, // destination,
    attr);
auto wei_reorder = dnnl::reorder(wei_reorder_pd);

```

### 7.5.2.3.2.2 Example 2: convolution with groups, with per-output-channel quantization

This example is complementary to the previous example (which should ideally be the first one). Let's say we want to create an int8 convolution with per-output channel scaling.

```

const float src_scale; // src_f32[:] = src_scale * src_s8[:]
const float dst_scale; // dst_f32[:] = dst_scale * dst_s8[:]

// the scaling factors for quantized weights (as declared above)
// An unique scale for each group and output-channel.
std::vector<float> wei_scales(G * OC/G) = {...};

// Src, weights, and dst memory descriptors for convolution,
// with memory format tag == any to allow a convolution implementation
// to chose the appropriate memory format

dnnl::memory::desc src_conv_s8_any_md(
    {BATCH, IC, IH, IW}, // dims
    dnnl::memory::data_type::s8, // the data originally in s8
    dnnl::memory::format_tag::any // let convolution to choose
);

dnnl::memory::desc wei_conv_s8_any_md(
    {G, OC/G, IC/G, KH, KW}, // dims

```

(continues on next page)

(continued from previous page)

```

dnnl::memory::data_type::s8, // the data originally in s8
dnnl::memory::format_tag::any // let convolution to choose
);

dnnl::memory::desc dst_conv_s8_any_md(...); // ditto

// Create a convolution operation descriptor
dnnl::convolution_forward::desc conv_d(
    dnnl::prop_kind::forward_inference,
    dnnl::algorithm::convolution_direct,
    src_conv_s8_any_md, // what's important is that
    wei_conv_s8_any_md, // we specified that we want
    dst_conv_s8_any_md, // computations in s8
    strides, padding_l, padding_r,
    dnnl::padding_kind::zero
);

// prepare the attributes for the convolution
dnnl::primitive_attr attr;
const int mask = 0
| (1 << 1); // scale per OC dimension, which is the dim #1 on dst tensor:
    // (BATCH, OC, OH, OW)
    // 0   1   2   3
std::vector<float> conv_output_scales(G * OC/G);
for (int g_oc = 0; G * OC/G; ++g_oc)
    conv_output_scales[g_oc] = src_scale * wei_scales(g_oc) / dst_scale;
attr.set_output_scales(mask, conv_output_scales);

// create a convolution primitive descriptor with the scaling factors
auto conv_pd = dnnl::convolution_forward::primitive_desc(
    conv_d, // general (non-customized) operation descriptor
    attr, // the attributes contain the output scaling
    engine);

```

### 7.5.2.3.2.3 Interplay of Output Scales with Post-ops

In general, the *Post-ops* are independent from the output scales. The output scales are applied to the result first; then post-ops will take effect.

That has an implication on the scaling factors passed to the library, however. Consider the following example of a convolution with tanh post-op:

$$\text{dst}_{s8}[:] = \frac{1}{\text{scale}_{\text{dst}}} \cdot \tanh(\text{scale}_{\text{src}} \cdot \text{scale}_{\text{weights}} \cdot \text{conv}_{s32}(\text{src}_{s8}, \text{wei}_{s8}))$$

- The convolution output scales are  $\text{conv\_output\_scale} = \text{scale}_{\text{src}} \cdot \text{scale}_{\text{weights}}$ , i.e. there is no division by  $\text{scale}_{\text{dst}}$ .
- And the post-ops scale for tanh is set to  $\text{scale}_{\text{tanh\_post\_op}} = \frac{1}{\text{scale}_{\text{dst}}}$ .

### 7.5.2.4 Attribute Related Error Handling

Since the attributes are created separately from the corresponding primitive descriptor, consistency checks are delayed. Users can successfully set attributes in whatever configuration they want. However, when they try to create a primitive descriptor with the attributes they set, it might happen that there is no primitive implementation that supports such a configuration. In this case the library will throw the `dnnl::error` exception.

### 7.5.2.5 API

```
struct dnnl::primitive_attr
```

Primitive attributes.

#### Public Functions

##### `primitive_attr()`

Constructs default (empty) primitive attributes.

##### `scratchpad_mode get_scratchpad_mode() const`

Returns the scratchpad mode.

##### `void set_scratchpad_mode(scratchpad_mode mode)`

Sets scratchpad mode.

#### Parameters

- mode: Specified scratchpad mode.

##### `void get_output_scales(int &mask, std::vector<float> &scales) const`

Returns output scaling factors correspondence mask and values.

#### Parameters

- mask: Scaling factors correspondence mask that defines the correspondence between the output tensor dimensions and the scales vector. The set i-th bit indicates that a dedicated output scaling factor is used for each index along that dimension. The mask value of 0 implies a common output scaling factor for the whole output tensor.
- scales: Vector of output scaling factors.

##### `void set_output_scales(int mask, const std::vector<float> &scales)`

Sets output scaling factors correspondence mask and values.

Example usage:

```
int mb = 32, oc = 32,
    oh = 14, ow = 14; // convolution output params
// unique output scales per output channel
vector<float> scales = { ... };
int oc_dim = 1; // mb_dim = 0, channel_dim = 1, height_dim = 2, ...

// construct a convolution descriptor
dnnl::convolution::desc conv_d;

dnnl::primitive_attr attr;
attr.set_output_scales(attr, oc, 1 << oc_dim, scales);
```

(continues on next page)

(continued from previous page)

```
dnnl::primitive_desc conv_pd(conv_d, attr, engine);
```

**Note** The order of dimensions does not depend on how elements are laid out in memory. For example:

- for a 2D CNN activations tensor the order is always (n, c)
- for a 4D CNN activations tensor the order is always (n, c, h, w)
- for a 5D CNN weights tensor the order is always (g, oc, ic, kh, kw)

### Parameters

- **mask**: Defines the correspondence between the output tensor dimensions and the **scales** vector. The set i-th bit indicates that a dedicated scaling factor is used for each index along that dimension. Set the mask to 0 to use a common output scaling factor for the whole output tensor.
- **scales**: Constant vector of output scaling factors. If the scaling factors are known at the time of this call, the following equality must hold:  $scales.size() = \prod_{d \in mask} output.dims[d]$ . Violations can only be detected when the attributes are used to create a primitive descriptor. If the scaling factors are not known at the time of the call, this vector must contain a single **DNNL\_RUNTIME\_F32\_VAL** value and the output scaling factors must be passed at execution time as an argument with index **DNNL\_ARG\_ATTR\_OUTPUT\_SCALES**.

```
void get_scales (int arg, int &mask, std::vector<float> &scales) const
```

Returns scaling factors correspondence mask and values for a given memory argument.

### Parameters

- **arg**: Parameter argument index as passed to the [\*primitive::execute\(\)\*](#) call.
- **mask**: Scaling factors correspondence mask that defines the correspondence between the output tensor dimensions and the **scales** vector. The set i-th bit indicates that a dedicated scaling factor is used for each index along that dimension. Set the mask to 0 to use a common scaling factor for the whole output tensor.
- **scales**: Output vector of scaling factors.

```
void set_scales (int arg, int mask, const std::vector<float> &scales)
```

Sets scaling factors for primitive operations for a given memory argument.

**See** [\*dnnl::primitive\\_attr::set\\_output\\_scales\*](#)

### Parameters

- **arg**: Parameter argument index as passed to the [\*primitive::execute\(\)\*](#) call.
- **mask**: Scaling factors correspondence mask that defines the correspondence between the tensor dimensions and the **scales** vector. The set i-th bit indicates that a dedicated scaling factor is used for each index along that dimension. Set the mask to 0 to use a common scaling factor for the whole output tensor.
- **scales**: Constant vector of scaling factors. The following equality must hold:  $scales.size() = \prod_{d \in mask} argument.dims[d]$ .

```
void get_zero_points (int arg, int &mask, std::vector<int32_t> &zero_points) const
```

Returns zero points correspondence mask and values.

## Parameters

- `arg`: Parameter argument index as passed to the `primitive::execute()` call.
- `mask`: Zero points correspondence mask that defines the correspondence between the output tensor dimensions and the `zero_points` vector. The set i-th bit indicates that a dedicated zero point is used for each index along that dimension. Set the mask to 0 to use a common zero point for the whole output tensor.
- `zero_points`: Output vector of zero points.

`void set_zero_points (int arg, int mask, const std::vector<int32_t> &zero_points)`  
Sets zero points for primitive operations for a given memory argument.

See [dnnl::primitive\\_attr::set\\_output\\_scales](#)

## Parameters

- `arg`: Parameter argument index as passed to the `primitive::execute()` call.
- `mask`: Zero point correspondence mask that defines the correspondence between the tensor dimensions and the `zero_points` vector. The set i-th bit indicates that a dedicated zero point is used for each index along that dimension. Set the mask to 0 to use a common zero point for the whole output tensor.
- `zero_points`: Constant vector of zero points. If the zero points are known at the time of this call, the following equality must hold:  $\text{zero\_points.size()} = \prod_{d \in \text{mask}} \text{argument.dims}[d]$ . If the zero points are not known at the time of the call, this vector must contain a single `DNNL_RUNTIME_F32_VAL` value and the zero points must be passed at execution time as an argument with index `DNNL_ARG_ATTR_ZERO_POINTS`.

`const post_ops get_post_ops () const`  
Returns post-ops previously set via `set_post_ops()`.

**Return** Post-ops.

`void set_post_ops (const post_ops ops)`  
Sets post-ops.

**Note** There is no way to check whether the post-ops would be supported by the target primitive. Any error will be reported by the respective primitive descriptor constructor.

## Parameters

- `ops`: Post-ops object to copy post-ops from.

`void set_rnn_data_qparams (float scale, float shift)`  
Sets quantization scale and shift parameters for RNN data tensors.

For performance reasons, the low-precision configuration of the RNN primitives expect input activations to have the unsigned 8-bit integer data type. The scale and shift parameters are used to quantize floating-point data to unsigned integer and must be passed to the RNN primitive using attributes.

The quantization formula is `scale * (data + shift)`.

Example usage:

```
// RNN parameters
int l = 2, t = 2, mb = 32, sic = 32, slc = 32, dic = 32, dlc = 32;
// Activations quantization parameters
float scale = 2.0f, shift = 0.5f;

primitive_attr attr;

// Set scale and shift for int8 quantization of activation
attr.set_rnn_data_qparams(scale, shift);

// Create and configure rnn op_desc
vanilla_rnn_forward::desc rnn_d(/* arguments */);
vanilla_rnn_forward::primitive_desc rnn_d(rnn_d, attr, engine);
```

**Note** Quantization scale and shift are common for src\_layer, src\_iter, dst\_iter, and dst\_layer.

#### Parameters

- **scale**: The value to scale the data by.
- **shift**: The value to shift the data by.

```
void set_rnn_weights_qparams (int mask, const std::vector<float> &scales)
```

Sets quantization scaling factors for RNN weights tensors. The low-precision configuration of the RNN primitives expect input weights to use the signed 8-bit integer data type. The scaling factors are used to quantize floating-point data to signed integer and must be passed to RNN primitives using attributes.

**Note** The dimension order is always native and does not depend on the actual layout used. For example, five-dimensional weights always have (l, d, i, g, o) logical dimension ordering.

**Note** Quantization scales are common for weights\_layer and weights\_iteration

#### Parameters

- **mask**: Scaling factors correspondence mask that defines the correspondence between the output tensor dimensions and the **scales** vector. The set i-th bit indicates that a dedicated scaling factor should be used each index along that dimension. Set the mask to 0 to use a common scaling factor for the whole output tensor.
- **scales**: Constant vector of output scaling factors. The following equality must hold:  

$$\text{scales.size()} = \prod_{d \in \text{mask}} \text{weights.dims}[d]$$
. Violations can only be detected when the attributes are used to create a primitive descriptor.

### 7.5.3 Batch Normalization

The batch normalization primitive performs a forward or backward batch normalization operation on tensors with number of dimensions equal to 2 or more. Variable names follow the standard *Conventions*.

The batch normalization operation is defined by the following formulas. We show formulas only for 2D spatial data which are straightforward to generalize to cases of higher and lower dimensions.

The different flavors of the primitive are controlled by the **flags** parameter that is passed to the operation descriptor initialization function like `dnnl::batch_normalization_forward::desc::desc()`. Multiple flags can be combined using the bitwise OR operator (`|`).

### 7.5.3.1 Forward

$$\text{dst}(n, c, h, w) = \gamma(c) \cdot \frac{\text{src}(n, c, h, w) - \mu(c)}{\sqrt{\sigma^2(c) + \epsilon}} + \beta(c),$$

where

- $\gamma(c)$  and  $\beta(c)$  are optional scale and shift for a channel (controlled using the `use_scaleshift` flag),
- $\mu(c)$  and  $\sigma^2(c)$  are mean and variance for a channel (controlled using the `use_global_stats` flag), and
- $\epsilon$  is a constant to improve numerical stability.

Mean and variance are computed at runtime or provided by a user. When mean and variance are computed at runtime, the following formulas are used:

- $\mu(c) = \frac{1}{NHW} \sum_{nhw} \text{src}(n, c, h, w),$
- $\sigma^2(c) = \frac{1}{NHW} \sum_{nhw} (\text{src}(n, c, h, w) - \mu(c))^2.$

The  $\gamma(c)$  and  $\beta(c)$  tensors are considered learnable.

In the training mode, the primitive also optionally supports fusion with ReLU activation with zero negative slope applied to the result (see `fuse_norm_relu` flag).

---

**Note:** The batch normalization primitive computes population mean and variance and not the sample or unbiased versions that are typically used to compute running mean and variance. \* Using the mean and variance computed by the batch normalization primitive, running mean and variance  $\hat{\mu}_i$  and  $\hat{\sigma}_i^2$  where  $i$  is iteration number, can be computed as:

$$\begin{aligned}\hat{\mu}_{i+1} &= \alpha \cdot \hat{\mu}_i + (1 - \alpha) \cdot \mu, \\ \hat{\sigma}_{i+1}^2 &= \alpha \cdot \hat{\sigma}_i^2 + (1 - \alpha) \cdot \sigma^2.\end{aligned}$$


---

#### 7.5.3.1.1 Difference Between Forward Training and Forward Inference

- If mean and variance are computed at runtime (i.e., `use_global_stats` is not set), they become outputs for the propagation kind `forward_training` (because they would be required during the backward propagation) and are not exposed for the propagation kind `forward_inference`.
- If batch normalization is created with ReLU fusion (i.e., `fuse_norm_relu` is set), for the propagation kind `forward_training` the primitive would produce a workspace memory as one extra output. This memory is required to compute the backward propagation. When the primitive is executed with propagation kind `forward_inference`, the workspace is not produced. Behavior would be the same as creating a batch normalization primitive with ReLU as a post-op (see section below).

### 7.5.3.2 Backward

The backward propagation computes  $\text{diff\_src}(n, c, h, w)$ ,  $\text{diff\_}\gamma(c)^*$ , and  $\text{diff\_}\beta(c)^*$  based on  $\text{diff\_dst}(n, c, h, w)$ ,  $\text{src}(n, c, h, w)$ ,  $\mu(c)$ ,  $\sigma^2(c)$ ,  $\gamma(c)^*$ , and  $\beta(c)^*$ .

The tensors marked with an asterisk are used only when the primitive is configured to use  $\gamma(c)$  and  $\beta(c)$  (i.e., `use_scaleshift` is set).

### 7.5.3.3 Execution Arguments

Depending on the flags and propagation kind, the batch normalization primitive requires different inputs and outputs. For clarity, a summary is shown below.

	<code>forward_infer</code>	<code>backward_training</code>	<code>backward</code>	<code>backward_data</code>
<code>none</code>	<i>In:</i> src; <i>Out:</i> dst	<i>In:</i> src; <i>Out:</i> dst, $\mu$ , $\sigma^2$	<i>In:</i> diff_dst, src, $\mu$ , $\sigma^2$ ; <i>Out:</i> diff_src	Same as for <code>backward</code>
<code>use_global</code>	<i>In:</i> src, $\mu$ , $\sigma^2$ ; <i>Out:</i> dst	<i>In:</i> src, $\mu$ , $\sigma^2$ ; <i>Out:</i> dst	<i>In:</i> diff_dst, src, $\mu$ , $\sigma^2$ ; <i>Out:</i> diff_src	Same as for <code>backward</code>
<code>use_scaleshift</code>	<i>In:</i> src, $\gamma$ , $\beta$ ; <i>Out:</i> dst	<i>In:</i> src, $\gamma$ , $\beta$ ; <i>Out:</i> dst, $\mu$ , $\sigma^2$	<i>In:</i> diff_dst, src, $\mu$ , $\sigma^2$ , $\gamma$ , $\beta$ ; <i>Out:</i> diff_src, diff_γ, diff_β	Not supported
<code>use_global</code>   <code>use_scaleshift</code>	<i>In:</i> src, $\mu$ , $\sigma^2$ , $\gamma$ , $\beta$ ; <i>Out:</i> dst	<i>In:</i> src, $\mu$ , $\sigma^2$ , $\gamma$ , $\beta$ ; <i>Out:</i> dst	<i>In:</i> diff_dst, src, $\mu$ , $\sigma^2$ , $\gamma$ , $\beta$ ; <i>Out:</i> diff_src, diff_γ, diff_β	Not supported
<code>flags</code>   <code>fuse_norm_reflags</code>	<i>In:</i> same as with <code>flags</code> ; <i>Out:</i> same as with <code>flags</code>	<i>In:</i> same as with <code>flags</code> ; <i>Out:</i> same as with <code>flags</code> , workspace	<i>In:</i> same as with <code>flags</code> , workspace; <i>Out:</i> same as with <code>flags</code>	Same as for <code>backward</code> if <code>flags</code> do not contain <code>use_scaleshift</code> ; not supported otherwise

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	<code>DNNL_ARG_SRC</code>
$\gamma, \beta$	<code>DNNL_ARG_SCALE_SHIFT</code>
mean ( $\mu$ )	<code>DNNL_ARG_MEAN</code>
variance ( $\sigma$ )	<code>DNNL_ARG_VARIANCE</code>
dst	<code>DNNL_ARG_DST</code>
workspace	<code>DNNL_ARG_WORKSPACE</code>
diff_dst	<code>DNNL_ARG_DIFF_DST</code>
diff_src	<code>DNNL_ARG_DIFF_SRC</code>
diff_γ, diff_β	<code>DNNL_ARG_DIFF_SCALE_SHIFT</code>

#### 7.5.3.4 Operation Details

1. For forward propagation, the mean and variance might be either computed at runtime (in which case they are outputs of the primitive) or provided by a user (in which case they are inputs). In the latter case, a user must set the `use_global_stats` flag. For the backward propagation, the mean and variance are always input parameters.
2. The memory format and data type for `src` and `dst` are assumed to be the same, and in the API they are typically referred to as `data` (e.g., see `data_desc` in `dnnl::batch_normalization_forward::desc::desc()`). The same is true for `diff_src` and `diff_dst`. The corresponding memory descriptors are referred to as `diff_data_desc`.
3. Both forward and backward propagation support in-place operations, meaning that `src` can be used as input and output for forward propagation, and `diff_dst` can be used as input and output for backward propagation. In case of an in-place operation, the original data will be overwritten. Note, however, that backward propagation requires original `src`, hence the corresponding forward propagation should not be performed in-place.
4. As mentioned above, the batch normalization primitive can be fused with ReLU activation even in the training mode. In this case, on the forward propagation the primitive has one additional output, `workspace`, that should be passed during the backward propagation.

#### 7.5.3.5 Data Types Support

The operation supports the following combinations of data types.

---

**Note:** Here we abbreviate data types names for readability. For example, `dnnl::memory::data_type::f32` is abbreviated to `f32`.

---

Propagation	Source / Destination	Mean / Variance / ScaleShift
forward / backward	<code>f32, bf16</code>	<code>f32</code>
forward	<code>f16</code>	<code>f32</code>
forward	<code>s8</code>	<code>f32</code>

#### 7.5.3.6 Data Representation

##### 7.5.3.6.1 Source, Destination, and Their Gradients

Like other CNN primitives, the batch normalization primitive expects data to be  $N \times C \times SP_n \times \dots \times SP_0$  tensor.

The batch normalization primitive is optimized for the following memory formats:

Spatial	Logical tensor	Implementations optimized for memory formats
0D	NC	<code>nc(ab)</code>
1D	NCW	<code>ncw(abc), nwc(acb), optimized</code>
2D	NCHW	<code>nchw(abcd), nhwc(acdb), optimized</code>
3D	NCDHW	<code>ncdhw(abcde), ndhwc(acdeb), optimized</code>

Here *optimized* means the format chosen by the preceding compute-intensive primitive.

### 7.5.3.6.2 Statistics Tensors

The mean ( $\mu$ ) and variance ( $\sigma^2$ ) are separate 1D tensors of size  $C$ .

The format of the corresponding memory object must be `x` (`a`).

If used, the scale ( $\gamma$ ) and shift ( $\beta$ ) are combined in a single 2D tensor of shape  $2 \times C$ .

The format of the corresponding memory object must be `nc` (`ab`).

### 7.5.3.7 Post-ops and Attributes

Propagation	Type	Operation	Description
forward	post-op	eltwise	Applies an eltwise operation to the output.

---

**Note:** Using ReLU as a post-op does not produce additional output in the workspace that is required to compute backward propagation correctly. Hence, one should use the `fuse_norm_relu` flag for training.

---

### 7.5.3.8 API

```
struct dnnl::batch_normalization_forward : public dnnl::primitive
    Batch normalization forward propagation primitive.
```

#### Public Functions

**batch\_normalization\_forward()**

Default constructor. Produces an empty object.

**batch\_normalization\_forward(const primitive\_desc &pd)**

Constructs a batch normalization forward propagation primitive.

#### Parameters

- pd: Primitive descriptor for a batch normalization forward propagation primitive.

**struct desc**

Descriptor for a batch normalization forward propagation primitive.

#### Public Functions

```
desc(prop_kind aprop_kind, const memory::desc &data_desc, float epsilon, normalization_flags
flags)
```

Constructs a batch normalization descriptor for forward propagation.

**Note** In-place operation is supported: the dst can refer to the same memory as the src.

#### Parameters

- aprop\_kind: Propagation kind. Possible values are `dnnl::prop_kind::forward_training` and `dnnl::prop_kind::forward_inference`.
- data\_desc: Source and destination memory descriptors.
- epsilon: Batch normalization epsilon parameter.
- flags: Batch normalization flags (`dnnl::normalization_flags`).

---

```
struct primitive_desc : public dnnl::primitive_desc
```

Primitive descriptor for a batch normalization forward propagation primitive.

## Public Functions

```
primitive_desc()
```

Default constructor. Produces an empty object.

```
primitive_desc(const desc &adesc, const engine &aengine, bool allow_empty = false)
```

Constructs a primitive descriptor for a batch normalization forward propagation primitive.

### Parameters

- adesc: Descriptor for a batch normalization forward propagation primitive.
- aengine: Engine to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
primitive_desc(const desc &adesc, const primitive_attr &attr, const engine &aengine,
```

```
bool allow_empty = false)
```

Constructs a primitive descriptor for a batch normalization forward propagation primitive.

### Parameters

- adesc: Descriptor for a batch normalization forward propagation primitive.
- attr: Primitive attributes to use.
- aengine: Engine to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
memory::desc src_desc() const
```

Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter.

```
memory::desc dst_desc() const
```

Returns a destination memory descriptor.

**Return** Destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination parameter.

```
memory::desc weights_desc() const
```

Returns a weights memory descriptor.

**Return** Weights memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a weights parameter.

```
memory::desc workspace_desc() const
```

Returns the workspace memory descriptor.

**Return** Workspace memory descriptor.

**Return** A zero memory descriptor if the primitive does not require workspace parameter.

```
memory::desc mean_desc() const
```

Returns memory descriptor for mean.

**Return** Memory descriptor for mean.

```
memory::desc variance_desc() const
```

Returns memory descriptor for variance.

**Return** Memory descriptor for variance.

---

```
struct dnnl::batch_normalization_backward : public dnnl::primitive
```

Batch normalization backward propagation primitive.

### Public Functions

```
batch_normalization_backward()
```

Default constructor. Produces an empty object.

```
batch_normalization_backward(const primitive_desc &pd)
```

Constructs a batch normalization backward propagation primitive.

#### Parameters

- pd: Primitive descriptor for a batch normalization backward propagation primitive.

```
struct desc
```

Descriptor for a batch normalization backward propagation primitive.

### Public Functions

```
desc(prop_kind aprop_kind, const memory::desc &diff_data_desc, const memory::desc  
&data_desc, float epsilon, normalization_flags flags)
```

Constructs a batch normalization descriptor for backward propagation.

#### Parameters

- aprop\_kind: Propagation kind. Possible values are `dnnl::prop_kind::backward_data` and `dnnl::prop_kind::backward` (diffs for all parameters are computed in this case).
- diff\_data\_desc: Diff source and diff destination memory descriptor.
- data\_desc: Source memory descriptor.
- epsilon: Batch normalization epsilon parameter.
- flags: Batch normalization flags (`dnnl::normalization_flags`).

```
struct primitive_desc : public dnnl::primitive_desc
```

Primitive descriptor for a batch normalization backward propagation primitive.

### Public Functions

```
primitive_desc()
```

Default constructor. Produces an empty object.

```
primitive_desc(const desc &adesc, const engine &aengine, const  
batch_normalization_forward::primitive_desc &hint_fwd_pd, bool al-  
low_empty = false)
```

Constructs a primitive descriptor for a batch normalization backward propagation primitive.

#### Parameters

- adesc: Descriptor for a batch normalization backward propagation primitive.
- aengine: Engine to use.
- hint\_fwd\_pd: Primitive descriptor for a batch normalization forward propagation primitive. It is used as a hint for deciding which memory format to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
primitive_desc(const desc &adesc, const primitive_attr &attr, const engine &aengine,
const batch_normalization_forward::primitive_desc &hint_fwd_pd, bool allow_empty = false)
```

Constructs a primitive descriptor for a batch normalization backward propagation primitive.

#### Parameters

- adesc: Descriptor for a batch normalization backward propagation primitive.
- attr: Primitive attributes to use.
- aengine: Engine to use.
- hint\_fwd\_pd: Primitive descriptor for a batch normalization forward propagation primitive. It is used as a hint for deciding which memory format to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**memory::desc src\_desc() const**

Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter.

**memory::desc weights\_desc() const**

Returns a weights memory descriptor.

**Return** Weights memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a weights parameter.

**memory::desc dst\_desc() const**

Returns a destination memory descriptor.

**Return** Destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination parameter.

**memory::desc diff\_src\_desc() const**

Returns a diff source memory descriptor.

**Return** Diff source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff source memory with.

**memory::desc diff\_dst\_desc() const**

Returns a diff destination memory descriptor.

**Return** Diff destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff destination parameter.

**memory::desc diff\_weights\_desc() const**

Returns a diff weights memory descriptor.

**Return** Diff weights memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff weights parameter.

**memory::desc mean\_desc() const**

Returns memory descriptor for mean.

**Return** Memory descriptor for mean.

**memory::desc variance\_desc() const**

Returns memory descriptor for variance.

**Return** Memory descriptor for variance.

**memory::desc workspace\_desc() const**

Returns the workspace memory descriptor.

**Return** Workspace memory descriptor.

**Return** A zero memory descriptor if the primitive does not require workspace parameter.

## 7.5.4 Binary

The binary primitive computes a result of a binary elementwise operation between tensors source 0 and source 1.

$$\text{dst}(\bar{x}) = \text{src}_0(\bar{x}) \text{ op } \text{src}_1(\bar{x}),$$

where  $\bar{x} = (x_0, \dots, x_n)$  and  $\text{op}$  is an operator like addition, multiplication, maximum or minimum. Variable names follow the standard [Conventions](#).

### 7.5.4.1 Forward and Backward

The binary primitive does not have a notion of forward or backward propagations.

### 7.5.4.2 Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
<code>src<sub>0</sub></code>	<code>DNNL_ARG_SRC_0</code>
<code>src<sub>1</sub></code>	<code>DNNL_ARG_SRC_1</code>
<code>dst</code>	<code>DNNL_ARG_DST</code>

### 7.5.4.3 Operation Details

- The binary primitive requires all source and destination tensors to have the same number of dimensions.
- The binary primitive supports implicit broadcast semantics for source 1. It means that if some dimension has value of one, this value will be used to compute an operation with each point of source 0 for this dimension.
- The dst memory format can be either specified explicitly or by `dnnl::memory::format_tag::any` (recommended), in which case the primitive will derive the most appropriate memory format based on the format of the source 0 tensor.
- Destination memory descriptor should completely match source 0 memory descriptor.
- The binary primitive supports in-place operations, meaning that source 0 tensor may be used as the destination, in which case its data will be overwritten.

### 7.5.4.4 Post-ops and Attributes

The following attributes should be supported:

Type	Op-eration	Description	Restrictions
At-tribute	<code>Scale</code> s	Scales the corresponding input tensor by the given scale factor(s).	The corresponding tensor has integer data type. Only one scale per tensor is supported. Input tensors only.
Post-op	<code>Sum</code>	Adds the operation result to the destination tensor instead of overwriting it.	Must precede eltwise post-op.
Post-op	<code>Eltwi</code> s	Applies an elementwise operation to the result.	

### 7.5.4.5 Data Types Support

The source and destination tensors may have `dnnl::memory::data_type::f32`, `dnnl::memory::data_type::bf16`, `dnnl::memory::data_type::s8` or `dnnl::memory::data_type::u8` data types.

### 7.5.4.6 Data Representation

The binary primitive works with arbitrary data tensors. There is no special meaning associated with any of tensors dimensions.

### 7.5.4.7 API

```
struct dnnl::binary : public dnnl::primitive
```

Elementwise binary operator primitive.

#### Public Functions

##### **binary()**

Default constructor. Produces an empty object.

##### **binary(**const** primitive\_desc &pd)**

Constructs an elementwise binary operation primitive.

#### Parameters

- pd: Primitive descriptor for an elementwise binary operation primitive.

##### **struct desc**

Descriptor for an elementwise binary operator primitive.

#### Public Functions

```
desc(algorithm aalgorithm, const memory::desc &src0, const memory::desc &src1, const memory::desc &dst)
```

Constructs a descriptor for an elementwise binary operator primitive.

#### Parameters

- aalgorithm: Elementwise algorithm.
- src0: Memory descriptor for source tensor #0.
- src1: Memory descriptor for source tensor #1.
- dst: Memory descriptor for destination tensor.

##### **struct primitive\_desc : **public** dnnl::primitive\_desc**

Primitive descriptor for an elementwise binary operator primitive.

## Public Functions

**primitive\_desc()**

Default constructor. Produces an empty object.

**primitive\_desc(const desc &adesc, const engine &aengine, bool allow\_empty = false)**

Constructs a primitive descriptor for an elementwise binary operator primitive.

### Parameters

- adesc: Descriptor for an elementwise binary operator primitive.
- aengine: Engine to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**primitive\_desc(const desc &adesc, const primitive\_attr &attr, const engine &aengine, bool allow\_empty = false)**

Constructs a primitive descriptor for an elementwise binary operator primitive.

### Parameters

- adesc: Descriptor for an elementwise binary operator primitive.
- aengine: Engine to use.
- attr: Primitive attributes to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**memory::desc src\_desc(int idx = 0) const**

Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter with index pdx.

### Parameters

- idx: Source index.

**memory::desc src0\_desc() const**

Returns the memory descriptor for source #0.

**memory::desc src1\_desc() const**

Returns the memory descriptor for source #1.

**memory::desc dst\_desc() const**

Returns a destination memory descriptor.

**Return** Destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination parameter.

## 7.5.5 Concat

A primitive to concatenate data by arbitrary dimension.

The concat primitive concatenates  $N$  tensors over concat\_dimension (here denoted as  $C$ ), and is defined as

$$\text{dst}(\overline{ou}, c, \overline{in}) = \text{src}_i(\overline{ou}, c', \overline{in}),$$

where

- $c = C_1 + \dots + C_{i-1} + c'$ ,
- $\overline{ou}$  is the outermost indices (to the left from concat axis),
- $\overline{in}$  is the innermost indices (to the right from concat axis), and

Variable names follow the standard *Conventions*.

#### 7.5.5.1 Forward and Backward

The concat primitive does not have a notion of forward or backward propagations. The backward propagation for the concatenation operation is simply an identity operation.

#### 7.5.5.2 Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	<i>DNNL_ARG_MULTIPLE_SRC</i>
dst	<i>DNNL_ARG_DST</i>

#### 7.5.5.3 Operation Details

1. The dst memory format can be either specified by a user or derived by the primitive. The recommended way is to allow the primitive to choose the most appropriate format.
2. The concat primitive requires all source and destination tensors to have the same shape except for the `concat_dimension`. The destination dimension for the `concat_dimension` must be equal to the sum of the `concat_dimension` dimensions of the sources (i.e.  $C = \sum_i C_i$ ). Implicit broadcasting is not supported.

#### 7.5.5.4 Data Types Support

The concat primitive supports arbitrary data types for source and destination tensors. However, it is required that all source tensors are of the same data type (but not necessarily matching the data type of the destination tensor).

#### 7.5.5.5 Data Representation

The concat primitive does not assign any special meaning associated with any logical dimensions.

#### 7.5.5.6 Post-ops and Attributes

The concat primitive does not support any post-ops or attributes.

#### 7.5.5.7 API

```
struct dnnl::concat : public dnnl::primitive
    Tensor concatenation (concat) primitive.
```

## Public Functions

**concat()**

Default constructor. Produces an empty object.

**concat (const primitive\_desc &pd)**

Constructs a concatenation primitive.

### Parameters

- pd: Primitive descriptor for concatenation primitive.

**struct primitive\_desc : public dnnl::primitive\_desc\_base**

Primitive descriptor for a concat primitive.

## Public Functions

**primitive\_desc()**

Default constructor. Produces an empty object.

**primitive\_desc (const memory::desc &dst, int concat\_dimension, const std::vector<memory::desc> &srcs, const engine &aengine, const primitive\_attr &attr = primitive\_attr())**

Constructs a primitive descriptor for an out-of-place concatenation primitive.

### Parameters

- dst: Destination memory descriptor.
- concat\_dimension: Source tensors will be concatenated over dimension with this index. Note that order of dimensions does not depend on memory format.
- srcs: Vector of source memory descriptors.
- aengine: Engine to perform the operation on.
- attr: Primitive attributes to use (optional).

**primitive\_desc (int concat\_dimension, const std::vector<memory::desc> &srcs, const engine &aengine, const primitive\_attr &attr = primitive\_attr())**

Constructs a primitive descriptor for an out-of-place concatenation primitive.

This version derives the destination memory descriptor automatically.

### Parameters

- concat\_dimension: Source tensors will be concatenated over dimension with this index. Note that order of dimensions does not depend on memory format.
- srcs: Vector of source memory descriptors.
- aengine: Engine to perform the operation on.
- attr: Primitive attributes to use (optional).

**memory::desc src\_desc (int idx = 0) const**

Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter with index pdx.

### Parameters

- idx: Source index.

**memory::desc dst\_desc () const**

Returns a destination memory descriptor.

**Return** Destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination parameter.

## 7.5.6 Convolution and Deconvolution

The convolution and deconvolution primitives compute forward, backward, or weight update for a batched convolution or deconvolution operations on 1D, 2D, or 3D spatial data with bias.

The operations are defined by the following formulas. We show formulas only for 2D spatial data which are straightforward to generalize to cases of higher and lower dimensions. Variable names follow the standard *Conventions*.

### 7.5.6.1 Forward

Let src, weights and dst be  $N \times IC \times IH \times IW$ ,  $OC \times IC \times KH \times KW$ , and  $N \times OC \times OH \times OW$  tensors respectively. Let bias be a 1D tensor with  $OC$  elements.

Furthermore, let the remaining convolution parameters be:

Parameter	Depth	Height	Width	Comment
Padding: Front, top, and left	$PD_L$	$PH_L$	$PW_L$	In the API padding_l indicates the corresponding vector of paddings (_l in the name stands for <b>left</b> )
Padding: Back, bottom, and right	$PD_R$	$PH_R$	$PW_R$	In the API padding_r indicates the corresponding vector of paddings (_r in the name stands for <b>right</b> )
Stride	$SD$	$SH$	$SW$	Convolution without strides is defined by setting the stride parameters to 1
Dilation	$DD$	$DH$	$DW$	Non-dilated convolution is defined by setting the dilation parameters to 0

The following formulas show how oneDNN computes convolutions. They are broken down into several types to simplify the exposition, but in reality the convolution types can be combined.

To further simplify the formulas, we assume that  $\text{src}(n, ic, ih, iw) = 0$  if  $ih < 0$ , or  $ih \geq IH$ , or  $iw < 0$ , or  $iw \geq IW$ .

#### 7.5.6.1.1 Regular Convolution

$$\begin{aligned} \text{dst}(n, oc, oh, ow) &= \text{bias}(oc) \\ &+ \sum_{ic=0}^{IC-1} \sum_{kh=0}^{KH-1} \sum_{kw=0}^{KW-1} \text{src}(n, ic, oh', ow') \cdot \text{weights}(oc, ic, kh, kw). \end{aligned}$$

Here:

- $oh' = oh \cdot SH + kh - PH_L$ ,
- $ow' = ow \cdot SW + kw - PW_L$ ,
- $OH = \lfloor \frac{IH - KH + PH_L + PH_R}{SH} \rfloor + 1$ ,
- $OW = \lfloor \frac{IW - KW + PW_L + PW_R}{SW} \rfloor + 1$ .

### 7.5.6.1.2 Convolution with Groups

oneDNN adds a separate groups dimension to memory objects representing weights tensors and represents weights as  $G \times OC_G \times IC_G \times KH \times KW$  5D tensors for 2D convolutions with groups.

$$\begin{aligned} \text{dst}(n, g \cdot OC_G + oc_g, oh, ow) &= \text{bias}(g \cdot OC_G + oc_g) \\ &+ \sum_{ic_g=0}^{IC_G-1} \sum_{kh=0}^{KH-1} \sum_{kw=0}^{KW-1} \text{src}(n, g \cdot IC_G + ic_g, oh', ow') \cdot \text{weights}(g, oc_g, ic_g, kh, kw), \end{aligned}$$

where

- $IC_G = \frac{IC}{G}$ ,
- $OC_G = \frac{OC}{G}$ , and
- $oc_g \in [0, OC_G]$ .

The case when  $OC_G = IC_G = 1$  is also known as *a depthwise convolution*.

### 7.5.6.1.3 Convolution with Dilation

$$\begin{aligned} \text{dst}(n, oc, oh, ow) &= \text{bias}(oc) + \\ &+ \sum_{ic=0}^{IC-1} \sum_{kh=0}^{KH-1} \sum_{kw=0}^{KW-1} \text{src}(n, ic, oh'', ow'') \cdot \text{weights}(oc, ic, kh, kw). \end{aligned}$$

Here:

- $oh'' = oh \cdot SH + kh \cdot (DH + 1) - PH_L$ ,
- $ow'' = ow \cdot SW + kw \cdot (DW + 1) - PW_L$ ,
- $OH = \lfloor \frac{IH - DKH + PH_L + PH_R}{SH} \rfloor + 1$ , where  $DKH = 1 + (KH - 1) \cdot (DH + 1)$ , and
- $OW = \lfloor \frac{IW - DKW + PW_L + PW_R}{SW} \rfloor + 1$ , where  $DKW = 1 + (KW - 1) \cdot (DW + 1)$ .

### 7.5.6.1.4 Deconvolution (Transposed Convolution)

Deconvolutions (also called fractionally-strided convolutions or transposed convolutions) can be defined by swapping the forward and backward passes of a convolution. One way to put it is to note that the weights define a convolution, but whether it is a direct convolution or a transposed convolution is determined by how the forward and backward passes are computed.

### 7.5.6.1.5 Difference Between Forward Training and Forward Inference

There is no difference between the `forward_training` and `forward_inference` propagation kinds.

### 7.5.6.2 Backward

The backward propagation computes diff\_src based on diff\_dst and weights.

The weights update computes diff\_weights and diff\_bias based on diff\_dst and src.

---

**Note:** The *optimized* memory formats src and weights might be different on forward propagation, backward propagation, and weights update.

---

### 7.5.6.3 Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	<i>DNNL_ARG_SRC</i>
weights	<i>DNNL_ARG_WEIGHTS</i>
bias	<i>DNNL_ARG_BIAS</i>
dst	<i>DNNL_ARG_DST</i>
diff_src	<i>DNNL_ARG_DIFF_SRC</i>
diff_weights	<i>DNNL_ARG_DIFF_WEIGHTS</i>
diff_bias	<i>DNNL_ARG_DIFF_BIAS</i>
diff_dst	<i>DNNL_ARG_DIFF_DST</i>

### 7.5.6.4 Operation Details

N/A

### 7.5.6.5 Data Types Support

Convolution primitive supports the following combination of data types for source, destination, and weights memory objects.

---

**Note:** Here we abbreviate data types names for readability. For example, *dnnl::memory::data\_type::f32* is abbreviated to *f32*.

---

Propagation	Source	Weights	Destination	Bias
forward / backward	<i>f32</i>	<i>f32</i>	<i>f32</i>	<i>f32</i>
forward	<i>f16</i>	<i>f16</i>	<i>f16</i>	<i>f16</i>
forward	<i>u8, s8</i>	<i>s8</i>	<i>u8, s8, s32, f32</i>	<i>u8, s8, s32, f32</i>
forward	<i>bf16</i>	<i>bf16</i>	<i>f32, bf16</i>	<i>f32, bf16</i>
backward	<i>f32, bf16</i>	<i>bf16</i>	<i>bf16</i>	
weights update	<i>bf16</i>	<i>f32, bf16</i>	<i>bf16</i>	<i>f32, bf16</i>

### 7.5.6.6 Data Representation

Like other CNN primitives, the convolution primitive expects the following tensors:

Spatial	Source / Destination	Weights
1D	$N \times C \times W$	$[G \times] OC \times IC \times KW$
2D	$N \times C \times H \times W$	$[G \times] OC \times IC \times KH \times KW$
3D	$N \times C \times D \times H \times W$	$[G \times] OC \times IC \times KD \times KH \times KW$

Memory format of data and weights memory objects is critical for convolution primitive performance. In the oneDNN programming model, convolution is one of the few primitives that support the placeholder memory format tag [any](#) and can define data and weight memory objects format based on the primitive parameters. When using [any](#) it is necessary to first create a convolution primitive descriptor and then query it for the actual data and weight memory objects formats.

While convolution primitives can be created with memory formats specified explicitly, the performance is likely to be suboptimal.

The table below shows the combinations for which *plain* memory formats the convolution primitive is optimized for.

Spatial	Convolution Type	Data / Weights logical tensor	Implementation optimized for memory formats
1D, 2D, 3D		<a href="#">any</a>	<i>optimized</i>
1D	f32, bf16	NCW / OIW, GOIW	<i>ncw (abc) / oiw (abc), goiw (abcd)</i>
1D	f32, bf16	NCW / OIW, GOIW	<i>nwc (acb) / wio (cba), wigo (dcab)</i>
1D	int8	NCW / OIW	<i>nwc (acb) / wio (cba)</i>
2D	f32, bf16	NCHW / OIHW, GOIHW	<i>nchw (abcd) / oihw (abcd), goihw (abcde)</i>
2D	f32, bf16	NCHW / OIHW, GOIHW	<i>nhwc (acdb) / hwio (cdba), hwigo (decab)</i>
2D	int8	NCHW / OIHW, GOIHW	<i>nhwc (acdb) / hwio (cdba), hwigo (decab)</i>
3D	f32, bf16	NCDHW / OIDHW, GOIDHW	<i>ncdhw (abcde) / oidhw (abcde), goidhw (abcdef)</i>
3D	f32, bf16	NCDHW / OIDHW, GOIDHW	<i>ndhwc (acdeb) / dhwio (cdeba), dhwigo (defcab)</i>
3D	int8	NCDHW / OIDHW	<i>ndhwc (acdeb) / dhwio (cdeba)</i>

### 7.5.6.7 Post-ops and Attributes

Post-ops and attributes enable you to modify the behavior of the convolution primitive by applying the output scale to the result of the primitive and by chaining certain operations after the primitive. The following attributes and post-ops are supported:

Propa-gation	Type	Operation	Description	Restrictions
forward	at-tribute	<i>Output scale</i>	Scales the result of convolution by given scale factor(s)	int8 convolu-tions only
forward	post-op	<i>Eltwise</i>	Applies an elementwise operation to the result	
forward	post-op	<i>Sum</i>	Adds the operation result to the destination tensor instead of overwriting it	

The primitive supports dynamic quantization via run-time output scales. That means a user could configure attributes with output scales set to the `DNNL_RUNTIME_F32_VAL` wildcard value instead of the actual scales, if the scales are not known at the primitive descriptor creation stage. In this case, the user must provide the scales as an additional input memory object with argument `DNNL_ARG_ATTR_OUTPUT_SCALES` during the execution stage.

---

**Note:** The library does not prevent using post-ops in training, but note that not all post-ops are feasible for training usage. For instance, using ReLU with non-zero negative slope parameter as a post-op would not produce an additional output workspace that is required to compute backward propagation correctly. Hence, in this particular case one should use separate convolution and eltwise primitives for training.

---

The following post-ops chaining should be supported by the library:

Type of convolutions	Post-ops sequence supported
f32 and bf16 convolution	eltwise, sum, sum -> eltwise
int8 convolution	eltwise, sum, sum -> eltwise, eltwise -> sum

The attributes and post-ops take effect in the following sequence:

- Output scale attribute,
- Post-ops, in order they were attached.

The operations during attributes and post-ops applying are done in single precision floating point data type. The conversion to the actual destination data type happens just before the actual storing.

#### 7.5.6.7.1 Example 1

Consider the following pseudo code:

```
attribute attr;
attr.set_output_scale(alpha);
attr.set_post_ops({
    { sum={scale=beta} },
    { eltwise={scale=gamma, type=tanh, alpha=ignore, beta=ignored} }
});
convolution_forward(src, weights, dst, attr)
```

The would lead to the following:

$$\text{dst}(\bar{x}) = \gamma \cdot \tanh(\alpha \cdot \text{conv}(\text{src}, \text{weights}) + \beta \cdot \text{dst}(\bar{x}))$$

### 7.5.6.7.2 Example 2

The following pseudo code:

```
attribute attr;
attr.set_output_scale(alpha);
attr.set_post_ops({
    {eltwise={scale=gamma, type=relu, alpha=eta, beta=ignored}}
    {sum={scale=beta}},
});
convolution_forward(src, weights, dst, attr)
```

That would lead to the following:

$$\text{dst}(\bar{x}) = \beta \cdot \text{dst}(\bar{x}) + \gamma \cdot \text{ReLU}(\alpha \cdot \text{conv}(\text{src}, \text{weights}), \eta)$$

### 7.5.6.8 Algorithms

oneDNN implementations may implement convolution primitives using several different algorithms which can be chosen by the user.

- *Direct* (`dnnl::algorithm::convolution_direct`). The convolution operation is computed directly using SIMD instructions. This also includes implicit GEMM formulations which notably may require workspace.
- *Winograd* (`dnnl::algorithm::convolution_winograd`). This algorithm reduces computational complexity of convolution at the expense of accuracy loss and additional memory operations. The implementation is based on the [Fast Algorithms for Convolutional Neural Networks by A. Lavin and S. Gray](#). The Winograd algorithm often results in the best performance, but it is applicable only to particular shapes. Moreover, Winograd only supports int8 and f32 data types.
- *Auto* (`dnnl::algorithm::convolution_auto`). In this case the library should automatically select the *best* algorithm based on the heuristics that take into account tensor shapes and the number of logical processors available.

### 7.5.6.9 API

```
struct dnnl::convolution_forward : public dnnl::primitive
    Convolution forward propagation primitive.
```

## Public Functions

### `convolution_forward()`

Default constructor. Produces an empty object.

### `convolution_forward(const primitive_desc &pd)`

Constructs a convolution forward propagation primitive.

#### Parameters

- pd: Primitive descriptor for a convolution forward propagation primitive.

### `struct desc`

Descriptor for a convolution forward propagation primitive.

## Public Functions

### `desc(prop_kind aprop_kind, algorithm aalgorithm, const memory::desc &src_desc, const memory::desc &weights_desc, const memory::desc &bias_desc, const memory::desc &dst_desc, const memory::dims &strides, const memory::dims &padding_l, const memory::dims &padding_r)`

Constructs a descriptor for a convolution forward propagation primitive with bias.

Arrays strides, padding\_l, and padding\_r contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

**Note** All the memory descriptors may be initialized with the `dnnl::memory::format_tag::any` value of `format_tag`.

#### Parameters

- aprop\_kind: Propagation kind. Possible values are `dnnl::prop_kind::forward_training`, and `dnnl::prop_kind::forward_inference`.
- aalgorithm: Convolution algorithm. Possible values are `dnnl::algorithm::convolution_direct`, `dnnl::algorithm::convolution_winograd`, and `dnnl::algorithm::convolution_auto`.
- src\_desc: Source memory descriptor.
- weights\_desc: Weights memory descriptor.
- bias\_desc: Bias memory descriptor. Passing zero memory descriptor disables the bias term.
- dst\_desc: Destination memory descriptor.
- strides: Strides for each spatial dimension.
- padding\_l: Vector of padding values for low indices for each spatial dimension ([ [front,] top,] left).
- padding\_r: Vector of padding values for high indices for each spatial dimension ([ [back,] bottom,] right).

### `desc(prop_kind aprop_kind, algorithm aalgorithm, const memory::desc &src_desc, const memory::desc &weights_desc, const memory::desc &dst_desc, const memory::dims &strides, const memory::dims &padding_l, const memory::dims &padding_r)`

Constructs a descriptor for a convolution forward propagation primitive without bias.

Arrays strides, padding\_l, and padding\_r contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

**Note** All the memory descriptors may be initialized with the `dnnl::memory::format_tag::any` value of `format_tag`.

#### Parameters

- `aprop_kind`: Propagation kind. Possible values are `dnnl::prop_kind::forward_training`, and `dnnl::prop_kind::forward_inference`.
- `aalgorithm`: Convolution algorithm. Possible values are `dnnl::algorithm::convolution_direct`, `dnnl::algorithm::convolution_winograd`, and `dnnl::algorithm::convolution_auto`.
- `src_desc`: Source memory descriptor.
- `weights_desc`: Weights memory descriptor.
- `dst_desc`: Destination memory descriptor.
- `strides`: Strides for each spatial dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension ([ [front,] top,] left).
- `padding_r`: Vector of padding values for high indices for each spatial dimension ([ [back,] bottom,] right).

```
desc (prop_kind aprop_kind, algorithm aalgorithm, const memory::desc &src_desc, const memory::desc &weights_desc, const memory::desc &bias_desc, const memory::desc &dst_desc, const memory::dims &strides, const memory::dims &dilates, const memory::dims &padding_l, const memory::dims &padding_r)
```

Constructs a descriptor for a dilated convolution forward propagation primitive with bias.

Arrays `strides`, `dilates`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

**Note** All the memory descriptors may be initialized with the `dnnl::memory::format_tag::any` value of `format_tag`.

#### Parameters

- `aprop_kind`: Propagation kind. Possible values are `dnnl::prop_kind::forward_training`, and `dnnl::prop_kind::forward_inference`.
- `aalgorithm`: Convolution algorithm. Possible values are `dnnl::algorithm::convolution_direct`, `dnnl::algorithm::convolution_winograd`, and `dnnl::algorithm::convolution_auto`.
- `src_desc`: Source memory descriptor.
- `weights_desc`: Weights memory descriptor.
- `bias_desc`: Bias memory descriptor. Passing zero memory descriptor disables the bias term.
- `dst_desc`: Destination memory descriptor.
- `strides`: Strides for each spatial dimension.
- `dilates`: Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension ([ [front,] top,] left).
- `padding_r`: Vector of padding values for high indices for each spatial dimension ([ [back,] bottom,] right).

```
desc (prop_kind aprop_kind, algorithm aalgorithm, const memory::desc &src_desc, const memory::desc &weights_desc, const memory::desc &dst_desc, const memory::dims &strides, const memory::dims &dilates, const memory::dims &padding_l, const memory::dims &padding_r)
```

Constructs a descriptor for a dilated convolution forward propagation primitive without bias.

Arrays `strides`, `dilates`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

**Note** All the memory descriptors may be initialized with the `dnnl::memory::format_tag::any` value of `format_tag`.

#### Parameters

- aprop\_kind: Propagation kind. Possible values are `dnnl::prop_kind::forward_training`, and `dnnl::prop_kind::forward_inference`.
- aalgorithm: Convolution algorithm. Possible values are `dnnl::algorithm::convolution_direct`, `dnnl::algorithm::convolution_winograd`, and `dnnl::algorithm::convolution_auto`.
- src\_desc: Source memory descriptor.
- weights\_desc: Weights memory descriptor.
- dst\_desc: Destination memory descriptor.
- strides: Strides for each spatial dimension.
- dilates: Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- padding\_l: Vector of padding values for low indices for each spatial dimension ( [ [front,] top,] left).
- padding\_r: Vector of padding values for high indices for each spatial dimension ( [ [back,] bottom,] right).

**struct primitive\_desc : public dnnl::primitive\_desc**

Primitive descriptor for a convolution forward propagation primitive.

## Public Functions

**primitive\_desc()**

Default constructor. Produces an empty object.

**primitive\_desc(const desc &adesc, const engine &aengine, bool allow\_empty = false)**

Constructs a primitive descriptor for a convolution forward propagation primitive.

### Parameters

- adesc: Descriptor for a convolution forward propagation primitive.
- aengine: Engine to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**primitive\_desc(const desc &adesc, const primitive\_attr &attr, const engine &aengine, bool allow\_empty = false)**

Constructs a primitive descriptor for a convolution forward propagation primitive.

### Parameters

- adesc: Descriptor for a convolution forward propagation primitive.
- aengine: Engine to use.
- attr: Primitive attributes to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**memory::desc src\_desc() const**

Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter.

**memory::desc weights\_desc() const**

Returns a weights memory descriptor.

**Return** Weights memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a weights parameter.

`memory::desc dst_desc() const`  
 Returns a destination memory descriptor.  
**Return** Destination memory descriptor.  
**Return** A zero memory descriptor if the primitive does not have a destination parameter.

`memory::desc bias_desc() const`  
 Returns the bias memory descriptor.  
**Return** The bias memory descriptor.  
**Return** A zero memory descriptor of the primitive does not have a bias parameter.

**struct** `dnnl::convolution_backward_data : public dnnl::primitive`  
 Convolution backward propagation primitive.

## Public Functions

`convolution_backward_data()`  
 Default constructor. Produces an empty object.

`convolution_backward_data(const primitive_desc &pd)`  
 Constructs a convolution backward propagation primitive.

### Parameters

- pd: Primitive descriptor for a convolution backward propagation primitive.

**struct desc**

Descriptor for a convolution backward propagation primitive.

## Public Functions

`desc(algorithm aalgorithm, const memory::desc &diff_src_desc, const memory::desc &weights_desc, const memory::desc &diff_dst_desc, const memory::dims &strides, const memory::dims &padding_l, const memory::dims &padding_r)`  
 Constructs a descriptor for a convolution backward propagation primitive.

Arrays strides, padding\_l, and padding\_r contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

**Note** All the memory descriptors may be initialized with the `dnnl::memory::format_tag::any` value of `format_tag`.

### Parameters

- aalgorithm: Convolution algorithm. Possible values are `dnnl::algorithm::convolution_direct`, `dnnl::algorithm::convolution_winograd`, and `dnnl::algorithm::convolution_auto`.
- diff\_src\_desc: Diff source memory descriptor.
- weights\_desc: Weights memory descriptor.
- diff\_dst\_desc: Diff destination memory descriptor.
- strides: Strides for each spatial dimension.
- padding\_l: Vector of padding values for low indices for each spatial dimension ([ [front,] top,] left).
- padding\_r: Vector of padding values for high indices for each spatial dimension ([ [back,] bottom,] right).

```
desc(algorithm aalgorithm, const memory::desc &diff_src_desc, const memory::desc &weights_desc, const memory::desc &diff_dst_desc, const memory::dims &strides, const memory::dims &dilates, const memory::dims &padding_l, const memory::dims &padding_r)
```

Constructs a descriptor for dilated convolution backward propagation primitive.

Arrays *strides*, *dilates*, *padding\_l*, and *padding\_r* contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

**Note** All the memory descriptors may be initialized with the *dnnl::memory::format\_tag::any* value of *format\_tag*.

#### Parameters

- *aalgorithm*: Convolution algorithm. Possible values are *dnnl::algorithm::convolution\_direct*, *dnnl::algorithm::convolution\_winograd*, and *dnnl::algorithm::convolution\_auto*.
- *diff\_src\_desc*: Diff source memory descriptor.
- *weights\_desc*: Weights memory descriptor.
- *diff\_dst\_desc*: Diff destination memory descriptor.
- *strides*: Strides for each spatial dimension.
- *dilates*: Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- *padding\_l*: Vector of padding values for low indices for each spatial dimension ( [ [front,] top,] left).
- *padding\_r*: Vector of padding values for high indices for each spatial dimension ( [ [back,] bottom,] right).

```
struct primitive_desc : public dnnl::primitive_desc
```

Primitive descriptor for a convolution backward propagation primitive.

### Public Functions

```
primitive_desc()
```

Default constructor. Produces an empty object.

```
primitive_desc(const desc &adesc, const engine &aengine, const convolu-
```

```
tion_forward::primitive_desc &hint_fwd_pd, bool allow_empty = false)
```

Constructs a primitive descriptor for a convolution backward propagation primitive.

#### Parameters

- *adesc*: Descriptor for a convolution backward propagation primitive.
- *aengine*: Engine to perform the operation on.
- *hint\_fwd\_pd*: Primitive descriptor for a convolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- *allow\_empty*: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to *false*.

```
primitive_desc(const desc &adesc, const primitive_attr &attr, const engine &aengine, const convolution_forward::primitive_desc &hint_fwd_pd, bool allow_empty = false)
```

Constructs a primitive descriptor for a convolution backward propagation primitive.

#### Parameters

- *adesc*: Descriptor for a convolution backward propagation primitive.
- *aengine*: Engine to perform the operation on.
- *attr*: Primitive attributes to use.

- `hint_fwd_pd`: Primitive descriptor for a convolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

`memory::desc diff_src_desc() const`

Returns a diff source memory descriptor.

**Return** Diff source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff source memory with.

`memory::desc weights_desc() const`

Returns a weights memory descriptor.

**Return** Weights memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a weights parameter.

`memory::desc diff_dst_desc() const`

Returns a diff destination memory descriptor.

**Return** Diff destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff destination parameter.

**struct** `dnnl::convolution_backward_weights : public dnnl::primitive`

Convolution weights gradient primitive.

## Public Functions

`convolution_backward_weights()`

Default constructor. Produces an empty object.

`convolution_backward_weights(const primitive_desc &pd)`

Constructs a convolution weights gradient primitive.

### Parameters

- `pd`: Primitive descriptor for a convolution weights gradient primitive.

**struct desc**

Descriptor for a convolution weights gradient primitive.

## Public Functions

`desc(algorithm aalgorithm, const memory::desc &src_desc, const memory::desc &diff_weights_desc, const memory::desc &diff_bias_desc, const memory::desc &diff_dst_desc, const memory::dims &strides, const memory::dims &padding_l, const memory::dims &padding_r)`

Constructs a descriptor for a convolution weights gradient primitive with bias.

Arrays `strides`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

**Note** All the memory descriptors may be initialized with the `dnnl::memory::format_tag::any` value of `format_tag`.

### Parameters

- `aalgorithm`: Convolution algorithm. Possible values are `dnnl::algorithm::convolution_direct`, `dnnl::algorithm::convolution_winograd`, and `dnnl::algorithm::convolution_auto`.
- `src_desc`: Source memory descriptor.

- `diff_weights_desc`: Diff weights memory descriptor.
- `diff_bias_desc`: Diff bias memory descriptor. Passing zero memory descriptor disables the bias term.
- `diff_dst_desc`: Diff destination memory descriptor.
- `strides`: Strides for each spatial dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension (`[[front,] top,] left`).
- `padding_r`: Vector of padding values for high indices for each spatial dimension (`[[back,] bottom,] right`).

```
desc(algorithm aalgorithm, const memory::desc &src_desc, const memory::desc &diff_weights_desc, const memory::desc &diff_bias_desc, const memory::dims &diff_dst_desc, const memory::dims &strides,
```

```
const memory::dims &padding_l, const memory::dims &padding_r)
```

Constructs a descriptor for a convolution weights gradient primitive without bias.

Arrays `strides`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

**Note** All the memory descriptors may be initialized with the `dnnl::memory::format_tag::any` value of `format_tag`.

#### Parameters

- `aalgorithm`: Convolution algorithm. Possible values are `dnnl::algorithm::convolution_direct`, `dnnl::algorithm::convolution_winograd`, and `dnnl::algorithm::convolution_auto`.
- `src_desc`: Source memory descriptor.
- `diff_weights_desc`: Diff weights memory descriptor.
- `diff_dst_desc`: Diff destination memory descriptor.
- `strides`: Strides for each spatial dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension (`[[front,] top,] left`).
- `padding_r`: Vector of padding values for high indices for each spatial dimension (`[[back,] bottom,] right`).

```
desc(algorithm aalgorithm, const memory::desc &src_desc, const memory::desc &diff_weights_desc, const memory::desc &diff_bias_desc, const memory::desc &diff_dst_desc, const memory::dims &strides, const memory::dims &dilates, const memory::dims &padding_l, const memory::dims &padding_r)
```

Constructs a descriptor for a dilated convolution weights gradient primitive with bias.

Arrays `strides`, `dilates`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

**Note** All the memory descriptors may be initialized with the `dnnl::memory::format_tag::any` value of `format_tag`.

#### Parameters

- `aalgorithm`: Convolution algorithm. Possible values are `dnnl::algorithm::convolution_direct`, `dnnl::algorithm::convolution_winograd`, and `dnnl::algorithm::convolution_auto`.
- `src_desc`: Source memory descriptor.
- `diff_weights_desc`: Diff weights memory descriptor.
- `diff_bias_desc`: Diff bias memory descriptor. Passing zero memory descriptor disables the bias term.
- `diff_dst_desc`: Diff destination memory descriptor.
- `strides`: Strides for each spatial dimension.
- `dilates`: Dilations for each spatial dimension. A zero value means no dilation in the corre-

sponding dimension.

- padding\_l: Vector of padding values for low indices for each spatial dimension ([[front,] top,] left).
- padding\_r: Vector of padding values for high indices for each spatial dimension ([ [back,] bottom,] right).

```
desc(algorithm aalgorithm, const memory::desc &src_desc, const memory::desc &diff_weights_desc, const memory::desc &diff_dst_desc, const memory::dims &strides, const memory::dims &dilates, const memory::dims &padding_l, const memory::dims &padding_r)
```

Constructs a descriptor for a dilated convolution weights gradient primitive without bias.

Arrays strides, dilates, padding\_l, and padding\_r contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

**Note** All the memory descriptors may be initialized with the *dnnl::memory::format\_tag::any* value of format\_tag.

#### Parameters

- *aalgorithm*: Convolution algorithm. Possible values are *dnnl::algorithm::convolution\_direct*, *dnnl::algorithm::convolution\_winograd*, and *dnnl::algorithm::convolution\_auto*.
- *src\_desc*: Source memory descriptor.
- *diff\_weights\_desc*: Diff weights memory descriptor.
- *diff\_dst\_desc*: Diff destination memory descriptor.
- *strides*: Strides for each spatial dimension.
- *dilates*: Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- *padding\_l*: Vector of padding values for low indices for each spatial dimension ([[front,] top,] left).
- *padding\_r*: Vector of padding values for high indices for each spatial dimension ([ [back,] bottom,] right).

```
struct primitive_desc : public dnnl::primitive_desc
```

Primitive descriptor for a convolution weights gradient primitive.

### Public Functions

```
primitive_desc()
```

Default constructor. Produces an empty object.

```
primitive_desc(const desc &adesc, const engine &aengine, const convolution_forward::primitive_desc &hint_fwd_pd, bool allow_empty = false)
```

Constructs a primitive descriptor for a convolution weights gradient primitive.

#### Parameters

- *adesc*: Descriptor for a convolution weights gradient primitive.
- *aengine*: Engine to use.
- *hint\_fwd\_pd*: Primitive descriptor for a convolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- *allow\_empty*: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
primitive_desc(const desc &adesc, const primitive_attr &attr, const engine &aengine, const convolution_forward::primitive_desc &hint_fwd_pd, bool allow_empty = false)
```

Constructs a primitive descriptor for a convolution weights gradient primitive.

#### Parameters

- adesc: Descriptor for a convolution weights gradient primitive.
- attr: Primitive attributes to use.
- aengine: Engine to use.
- hint\_fwd\_pd: Primitive descriptor for a convolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

#### `memory::desc src_desc() const`

Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter.

#### `memory::desc diff_weights_desc() const`

Returns a diff weights memory descriptor.

**Return** Diff weights memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff weights parameter.

#### `memory::desc diff_dst_desc() const`

Returns a diff destination memory descriptor.

**Return** Diff destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff destination parameter.

#### `memory::desc diff_bias_desc() const`

Returns the diff bias memory descriptor.

**Return** The diff bias memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff bias parameter.

### `struct dnnl::deconvolution_forward : public dnnl::primitive`

Deconvolution forward propagation primitive.

#### Public Functions

##### `deconvolution_forward()`

Default constructor. Produces an empty object.

##### `deconvolution_forward(const primitive_desc &pd)`

Constructs a deconvolution forward propagation primitive.

#### Parameters

- pd: Primitive descriptor for a deconvolution forward propagation primitive.

### `struct desc`

Descriptor for a deconvolution forward propagation primitive.

## Public Functions

```
desc (prop_kind aprop_kind, algorithm aalgorithm, const memory::desc &src_desc, const memory::desc &weights_desc, const memory::desc &bias_desc, const memory::desc &dst_desc, const memory::dims &strides, const memory::dims &padding_l, const memory::dims &padding_r)
```

Constructs a descriptor for a deconvolution forward propagation primitive with bias.

Arrays strides, padding\_l, and padding\_r contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

**Note** All the memory descriptors may be initialized with the *dnnl::memory::format\_tag::any* value of format\_tag.

### Parameters

- *aprop\_kind*: Propagation kind. Possible values are *dnnl::prop\_kind::forward\_training*, and *dnnl::prop\_kind::forward\_inference*.
- *aalgorithm*: Deconvolution algorithm: *dnnl::algorithm::deconvolution\_direct*, and *dnnl::algorithm::deconvolution\_winograd*.
- *src\_desc*: Source memory descriptor.
- *weights\_desc*: Weights memory descriptor.
- *bias\_desc*: Bias memory descriptor. Passing zero memory descriptor disables the bias term.
- *dst\_desc*: Destination memory descriptor.
- *strides*: Vector of strides for spatial dimension.
- *padding\_l*: Vector of padding values for low indices for each spatial dimension ([[front,] top,] left).
- *padding\_r*: Vector of padding values for high indices for each spatial dimension ([ [back,] bottom,] right).

```
desc (prop_kind aprop_kind, algorithm aalgorithm, const memory::desc &src_desc, const memory::desc &weights_desc, const memory::desc &dst_desc, const memory::dims &strides, const memory::dims &padding_l, const memory::dims &padding_r)
```

Constructs a descriptor for a deconvolution forward propagation primitive without bias.

Arrays strides, padding\_l, and padding\_r contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

**Note** All the memory descriptors may be initialized with the *dnnl::memory::format\_tag::any* value of format\_tag.

### Parameters

- *aprop\_kind*: Propagation kind. Possible values are *dnnl::prop\_kind::forward\_training*, and *dnnl::prop\_kind::forward\_inference*.
- *aalgorithm*: Deconvolution algorithm: *dnnl::algorithm::deconvolution\_direct*, and *dnnl::algorithm::deconvolution\_winograd*.
- *src\_desc*: Source memory descriptor.
- *weights\_desc*: Weights memory descriptor.
- *dst\_desc*: Destination memory descriptor.
- *strides*: Vector of strides for spatial dimension.
- *padding\_l*: Vector of padding values for low indices for each spatial dimension ([[front,] top,] left).
- *padding\_r*: Vector of padding values for high indices for each spatial dimension ([ [back,] bottom,] right).

```
desc (prop_kind aprop_kind, algorithm aalgorithm, const memory::desc &src_desc, const memory::desc &weights_desc, const memory::desc &bias_desc, const memory::desc &dst_desc, const memory::dims &strides, const memory::dims &dilates, const memory::dims &padding_l, const memory::dims &padding_r)
```

Constructs a descriptor for a dilated deconvolution forward propagation primitive with bias.

Arrays `strides`, `dilates`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

**Note** All the memory descriptors may be initialized with the `dnnl::memory::format_tag::any` value of `format_tag`.

#### Parameters

- `aprop_kind`: Propagation kind. Possible values are `dnnl::prop_kind::forward_training`, and `dnnl::prop_kind::forward_inference`.
- `aalgorithm`: Deconvolution algorithm: `dnnl::algorithm::deconvolution_direct`, and `dnnl::algorithm::deconvolution_winograd`.
- `src_desc`: Source memory descriptor.
- `weights_desc`: Weights memory descriptor.
- `bias_desc`: Bias memory descriptor. Passing zero memory descriptor disables the bias term.
- `dst_desc`: Destination memory descriptor.
- `strides`: Vector of strides for spatial dimension.
- `dilates`: Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension ( [ [front,] top,] left).
- `padding_r`: Vector of padding values for high indices for each spatial dimension ( [ [back,] bottom,] right).

```
desc (prop_kind aprop_kind, algorithm aalgorithm, const memory::desc &src_desc, const memory::desc &weights_desc, const memory::desc &dst_desc, const memory::dims &strides, const memory::dims &dilates, const memory::dims &padding_l, const memory::dims &padding_r)
```

Constructs a descriptor for a dilated deconvolution forward propagation primitive without bias.

Arrays `strides`, `dilates`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

**Note** All the memory descriptors may be initialized with the `dnnl::memory::format_tag::any` value of `format_tag`.

#### Parameters

- `aprop_kind`: Propagation kind. Possible values are `dnnl::prop_kind::forward_training`, and `dnnl::prop_kind::forward_inference`.
- `aalgorithm`: Deconvolution algorithm: `dnnl::algorithm::deconvolution_direct`, and `dnnl::algorithm::deconvolution_winograd`.
- `src_desc`: Source memory descriptor.
- `weights_desc`: Weights memory descriptor.
- `dst_desc`: Destination memory descriptor.
- `strides`: Vector of strides for spatial dimension.
- `dilates`: Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension ( [ [front,] top,] left).
- `padding_r`: Vector of padding values for high indices for each spatial dimension ( [ [back,] bottom,] right).

---

```
struct primitive_desc : public dnnl::primitive_desc
```

Primitive descriptor for a deconvolution forward propagation primitive.

## Public Functions

```
primitive_desc()
```

Default constructor. Produces an empty object.

```
primitive_desc(const desc &adesc, const engine &aengine, bool allow_empty = false)
```

Constructs a primitive descriptor for a deconvolution forward propagation primitive.

### Parameters

- **adesc**: Descriptor for a deconvolution forward propagation primitive.
- **aengine**: Engine to use.
- **allow\_empty**: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
primitive_desc(const desc &adesc, const primitive_attr &attr, const engine &aengine,  
                  bool allow_empty = false)
```

Constructs a primitive descriptor for a deconvolution forward propagation primitive.

### Parameters

- **adesc**: Descriptor for a deconvolution forward propagation primitive.
- **aengine**: Engine to use.
- **attr**: Primitive attributes to use.
- **allow\_empty**: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
memory::desc src_desc() const
```

Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter.

```
memory::desc weights_desc() const
```

Returns a weights memory descriptor.

**Return** Weights memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a weights parameter.

```
memory::desc dst_desc() const
```

Returns a destination memory descriptor.

**Return** Destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination parameter.

```
memory::desc bias_desc() const
```

Returns the bias memory descriptor.

**Return** The bias memory descriptor.

**Return** A zero memory descriptor of the primitive does not have a bias parameter.

```
struct dnnl::deconvolution_backward_data : public dnnl::primitive
```

Deconvolution backward propagation primitive.

## Public Functions

### `deconvolution_backward_data()`

Default constructor. Produces an empty object.

### `deconvolution_backward_data(const primitive_desc &pd)`

Constructs a deconvolution backward propagation primitive.

#### Parameters

- pd: Primitive descriptor for a deconvolution backward propagation primitive.

### `struct desc`

Descriptor for a deconvolution backward propagation primitive.

## Public Functions

### `desc(algorithm aalgorithm, const memory::desc &diff_src_desc, const memory::desc &weights_desc, const memory::desc &diff_dst_desc, const memory::dims &strides, const memory::dims &padding_l, const memory::dims &padding_r)`

Constructs a descriptor for a deconvolution backward propagation primitive.

Arrays strides, padding\_l, and padding\_r contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

**Note** All the memory descriptors may be initialized with the `dnnl::memory::format_tag::any` value of `format_tag`.

#### Parameters

- aalgorithm: Deconvolution algorithm (`dnnl::algorithm::convolution_direct`, `dnnl::algorithm::convolution_winograd`).
- diff\_src\_desc: Diff source memory descriptor.
- weights\_desc: Weights memory descriptor.
- diff\_dst\_desc: Diff destination memory descriptor.
- strides: Strides for each spatial dimension.
- padding\_l: Vector of padding values for low indices for each spatial dimension ([ [front,] top,] left).
- padding\_r: Vector of padding values for high indices for each spatial dimension ([ [back,] bottom,] right).

### `desc(algorithm aalgorithm, const memory::desc &diff_src_desc, const memory::desc &weights_desc, const memory::desc &diff_dst_desc, const memory::dims &strides, const memory::dims &dilates, const memory::dims &padding_l, const memory::dims &padding_r)`

Constructs a descriptor for a dilated deconvolution backward propagation primitive.

Arrays strides, dilates, padding\_l, and padding\_r contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

**Note** All the memory descriptors may be initialized with the `dnnl::memory::format_tag::any` value of `format_tag`.

#### Parameters

- aalgorithm: Deconvolution algorithm (`dnnl::algorithm::convolution_direct`, `dnnl::algorithm::convolution_winograd`).
- diff\_src\_desc: Diff source memory descriptor.
- weights\_desc: Weights memory descriptor.
- diff\_dst\_desc: Diff destination memory descriptor.

- `strides`: Strides for each spatial dimension.
- `dilates`: Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension ([ [front,] top,] left).
- `padding_r`: Vector of padding values for high indices for each spatial dimension ([ [back,] bottom,] right).

**struct primitive\_desc : public dnnl::primitive\_desc**

Primitive descriptor for a deconvolution backward propagation primitive.

## Public Functions

**primitive\_desc()**

Default constructor. Produces an empty object.

**primitive\_desc(const desc &adesc, const engine &aengine, const deconvolution\_forward::primitive\_desc &hint\_fwd\_pd, bool allow\_empty = false)**

Constructs a primitive descriptor for a deconvolution backward propagation primitive.

### Parameters

- `adesc`: Descriptor for a deconvolution backward propagation primitive.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for a deconvolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**primitive\_desc(const desc &adesc, const primitive\_attr &attr, const engine &aengine, const deconvolution\_forward::primitive\_desc &hint\_fwd\_pd, bool allow\_empty = false)**

Constructs a primitive descriptor for a deconvolution backward propagation primitive.

### Parameters

- `adesc`: Descriptor for a deconvolution backward propagation primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for a deconvolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**memory::desc diff\_src\_desc() const**

Returns a diff source memory descriptor.

**Return** Diff source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff source memory with.

**memory::desc weights\_desc() const**

Returns a weights memory descriptor.

**Return** Weights memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a weights parameter.

**memory::desc diff\_dst\_desc() const**

Returns a diff destination memory descriptor.

**Return** Diff destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff destination parameter.

---

```
struct dnnl::deconvolution_backward_weights : public dnnl::primitive
Deconvolution weights gradient primitive.
```

## Public Functions

**deconvolution\_backward\_weights()**

Default constructor. Produces an empty object.

**deconvolution\_backward\_weights(const primitive\_desc &pd)**

Constructs a deconvolution weights gradient primitive.

### Parameters

- pd: Primitive descriptor for a deconvolution weights gradient primitive.

**struct desc**

Descriptor for a deconvolution weights gradient primitive.

## Public Functions

```
desc(algorithm aalgorithm, const memory::desc &src_desc, const memory::desc
&diff_weights_desc, const memory::desc &diff_bias_desc, const memory::desc
&diff_dst_desc, const memory::dims &strides, const memory::dims &padding_l,
const memory::dims &padding_r)
```

Constructs a descriptor for a deconvolution weights gradient primitive with bias.

Arrays strides, padding\_l, and padding\_r contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

**Note** All the memory descriptors may be initialized with the *dnnl::memory::format\_tag::any* value of format\_tag.

### Parameters

- aalgorithm: Deconvolution algorithm. Possible values are *dnnl::algorithm::deconvolution\_direct*, and *dnnl::algorithm::deconvolution\_winograd*.
- src\_desc: Source memory descriptor.
- diff\_weights\_desc: Diff weights memory descriptor.
- diff\_bias\_desc: Diff bias memory descriptor. Passing zero memory descriptor disables the bias term.
- diff\_dst\_desc: Diff destination memory descriptor.
- strides: Strides for each spatial dimension.
- padding\_l: Vector of padding values for low indices for each spatial dimension ([[front,] top,] left)).
- padding\_r: Vector of padding values for high indices for each spatial dimension ([ [back,] bottom,] right)).

```
desc(algorithm aalgorithm, const memory::desc &src_desc, const memory::desc
&diff_weights_desc, const memory::desc &diff_dst_desc, const memory::dims &strides,
const memory::dims &padding_l, const memory::dims &padding_r)
```

Constructs a descriptor for a deconvolution weights gradient primitive without bias.

Arrays strides, padding\_l, and padding\_r contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

**Note** All the memory descriptors may be initialized with the *dnnl::memory::format\_tag::any* value of format\_tag.

**Parameters**

- `aalgorithm`: Deconvolution algorithm. Possible values are `dnnl::algorithm::deconvolution_direct`, and `dnnl::algorithm::deconvolution_winograd`.
- `src_desc`: Source memory descriptor.
- `diff_weights_desc`: Diff weights memory descriptor.
- `diff_dst_desc`: Diff destination memory descriptor.
- `strides`: Strides for each spatial dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension ( [ [front,] top,] left).
- `padding_r`: Vector of padding values for high indices for each spatial dimension ( [ [back,] bottom,] right).

```
desc(algorithm aalgorithm, const memory::desc &src_desc, const memory::desc &diff_weights_desc, const memory::desc &diff_bias_desc, const memory::desc &diff_dst_desc, const memory::dims &strides, const memory::dims &dilates, const memory::dims &padding_l, const memory::dims &padding_r)
```

Constructs a descriptor for a dilated deconvolution weights gradient primitive with bias.

Arrays `strides`, `dilates`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

**Note** All the memory descriptors may be initialized with the `dnnl::memory::format_tag::any` value of `format_tag`.

**Parameters**

- `aalgorithm`: Deconvolution algorithm. Possible values are `dnnl::algorithm::deconvolution_direct`, and `dnnl::algorithm::deconvolution_winograd`.
- `src_desc`: Source memory descriptor.
- `diff_weights_desc`: Diff weights memory descriptor.
- `diff_bias_desc`: Diff bias memory descriptor. Passing zero memory descriptor disables the bias term.
- `diff_dst_desc`: Diff destination memory descriptor.
- `strides`: Strides for each spatial dimension.
- `dilates`: Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- `padding_l`: Vector of padding values for low indices for each spatial dimension ( [ [front,] top,] left).
- `padding_r`: Vector of padding values for high indices for each spatial dimension ( [ [back,] bottom,] right).

```
desc(algorithm aalgorithm, const memory::desc &src_desc, const memory::desc &diff_weights_desc, const memory::desc &diff_dst_desc, const memory::dims &strides, const memory::dims &dilates, const memory::dims &padding_l, const memory::dims &padding_r)
```

Constructs a descriptor for a dilated deconvolution weights gradient primitive without bias.

Arrays `strides`, `dilates`, `padding_l`, and `padding_r` contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

**Note** All the memory descriptors may be initialized with the `dnnl::memory::format_tag::any` value of `format_tag`.

**Parameters**

- `aalgorithm`: Deconvolution algorithm. Possible values are `dnnl::algorithm::deconvolution_direct`, and `dnnl::algorithm::deconvolution_winograd`.
- `src_desc`: Source memory descriptor.
- `diff_weights_desc`: Diff weights memory descriptor.

- diff\_dst\_desc: Diff destination memory descriptor.
- strides: Strides for each spatial dimension.
- dilates: Dilations for each spatial dimension. A zero value means no dilation in the corresponding dimension.
- padding\_l: Vector of padding values for low indices for each spatial dimension ( [[front,] top,] left ).
- padding\_r: Vector of padding values for high indices for each spatial dimension ( [ [back,] bottom,] right ).

**struct primitive\_desc : public dnnl::primitive\_desc**

Primitive descriptor for a deconvolution weights gradient primitive.

## Public Functions

**primitive\_desc()**

Default constructor. Produces an empty object.

**primitive\_desc(const desc &adesc, const engine &aengine, const deconvolution\_forward::primitive\_desc &hint\_fwd\_pd, bool allow\_empty = false)**

Constructs a primitive descriptor for a deconvolution weights update primitive.

### Parameters

- adesc: Descriptor for a deconvolution weights gradient primitive.
- aengine: Engine to use.
- hint\_fwd\_pd: Primitive descriptor for a deconvolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**primitive\_desc(const desc &adesc, const primitive\_attr &attr, const engine &aengine, const deconvolution\_forward::primitive\_desc &hint\_fwd\_pd, bool allow\_empty = false)**

Constructs a primitive descriptor for a deconvolution weights update primitive.

### Parameters

- adesc: Descriptor for a deconvolution weights gradient primitive.
- attr: Primitive attributes to use.
- aengine: Engine to use.
- hint\_fwd\_pd: Primitive descriptor for a deconvolution forward propagation primitive. It is used as a hint for deciding which memory format to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**memory::desc src\_desc() const**

Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter.

**memory::desc diff\_weights\_desc() const**

Returns a diff weights memory descriptor.

**Return** Diff weights memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff weights parameter.

**memory::desc diff\_dst\_desc() const**

Returns a diff destination memory descriptor.

**Return** Diff destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff destination parameter.

`memory::desc diff_bias_desc() const`

Returns the diff bias memory descriptor.

**Return** The diff bias memory descriptor.

**Return** A zero memory descriptor of the primitive does not have a diff bias parameter.

## 7.5.7 Elementwise

The elementwise primitive applies an operation to every element of the tensor. Variable names follow the standard *Conventions*.

$$\text{dst}(\bar{x}) = \text{Operation}(\text{src}(\bar{x})),$$

for  $\bar{x} = (x_0, \dots, x_n)$ .

### 7.5.7.1 Forward

The following forward operations are supported. Here  $s$  and  $d$  denote src and dst, tensor values respectively.

Elementwise algorithm	Forward formula
<code>eltwise_abs</code>	$d = \begin{cases} s & \text{if } s > 0 \\ -s & \text{if } s \leq 0 \end{cases}$
<code>eltwise_bounded_relu</code>	$d = \begin{cases} \alpha & \text{if } s > \alpha \geq 0 \\ s & \text{if } 0 < s \leq \alpha \\ 0 & \text{if } s \leq 0 \end{cases}$
<code>eltwise_clip</code>	$d = \begin{cases} \beta & \text{if } s > \beta \geq \alpha \\ s & \text{if } \alpha < s \leq \beta \\ \alpha & \text{if } s \leq \alpha \end{cases}$
<code>eltwise_elu, eltwise_elu_use_dst_for_bwd</code>	$d = \begin{cases} s & \text{if } s > 0 \\ \alpha(e^s - 1) & \text{if } s \leq 0 \end{cases}$
<code>eltwise_exp, eltwise_exp_use_dst_for_bwd</code>	$d = e^s$
<code>eltwise_gelu_erf</code>	$d = 0.5s(1 + \text{erf}[\frac{s}{\sqrt{2}}])$
<code>eltwise_gelu_tanh</code>	$d = 0.5s(1 + \tanh[\sqrt{\frac{2}{\pi}}(s + 0.044715s^3)])$
<code>eltwise_linear</code>	$d = \alpha s + \beta$
<code>eltwise_log</code>	$d = \log_e s$
<code>eltwise_logistic, eltwise_logistic_use_dst_for_bwd</code>	$d = \frac{1}{1+e^{-s}}$
<code>eltwise_pow</code>	$d = \alpha s^\beta$
<code>eltwise_relu, eltwise_relu_use_dst_for_bwd</code>	$d = \begin{cases} s & \text{if } s > 0 \\ \alpha s & \text{if } s \leq 0 \end{cases}$
<code>eltwise_round</code>	$d = \text{round}(s)$
<code>eltwise_soft_relu</code>	$d = \log_e(1 + e^s)$
<code>eltwise_sqrt, eltwise_sqrt_use_dst_for_bwd</code>	$d = \sqrt{s}$
<code>eltwise_square</code>	$d = s^2$
<code>eltwise_swish</code>	$d = \frac{s}{1+e^{-\alpha s}}$
<code>eltwise_tanh, eltwise_tanh_use_dst_for_bwd</code>	$d = \tanh s$

### 7.5.7.2 Backward

The backward propagation computes  $\text{diff\_src}(\bar{s})$ , based on  $\text{diff\_dst}(\bar{s})$  and  $\text{src}(\bar{s})$ . However, some operations support a computation using  $\text{dst}(\bar{s})$  memory produced during forward propagation. Refer to the table above for a list of operations supporting destination as input memory and the corresponding formulas.

The following backward operations are supported. Here  $s$ ,  $d$ ,  $ds$  and  $dd$  denote  $\text{src}$ ,  $\text{dst}$ ,  $\text{diff\_src}$ , and a  $\text{diff\_dst}$  tensor values respectively.

Elementwise algorithm	Backward formula
<code>eltwise_abs</code>	$ds = \begin{cases} dd & \text{if } s > 0 \\ -dd & \text{if } s < 0 \\ 0 & \text{if } s = 0 \end{cases}$
<code>eltwise_bounded_relu</code>	$ds = \begin{cases} dd & \text{if } 0 < s \leq \alpha, \\ 0 & \text{otherwise} \end{cases}$
<code>eltwise_clip</code>	$ds = \begin{cases} dd & \text{if } \alpha < s \leq \beta \\ 0 & \text{otherwise} \end{cases}$
<code>eltwise_elu</code>	$ds = \begin{cases} dd & \text{if } s > 0 \\ dd \cdot \alpha e^s & \text{if } s \leq 0 \end{cases}$
<code>eltwise_elu_use_dst_for_bwd</code>	$ds = \begin{cases} dd & \text{if } d > 0 \\ dd \cdot (d + \alpha) & \text{if } d \leq 0 \end{cases} \text{ only if } \alpha \geq 0$
<code>eltwise_exp</code>	$ds = dd \cdot e^s$
<code>eltwise_exp_use_dst_for_bwd</code>	$ds = dd \cdot d$
<code>eltwise_gelu_erf</code>	$ds = dd \cdot \left( 0.5 + 0.5 \operatorname{erf} \left( \frac{s}{\sqrt{2}} \right) + \frac{s}{\sqrt{2}\pi} e^{-0.5s^2} \right)$
<code>eltwise_gelu_tanh</code>	$ds = dd \cdot 0.5(1 + \tanh[\sqrt{\frac{2}{\pi}}(s + 0.044715s^3)]) \cdot (1 + \sqrt{\frac{2}{\pi}}(s + 0.134145s^3)) \cdot (1 - \tanh[\sqrt{\frac{2}{\pi}}(s + 0.044715s^3)])$
<code>eltwise_linear</code>	$ds = \alpha \cdot dd$
<code>eltwise_log</code>	$ds = \frac{dd}{s}$
<code>eltwise_logistic</code>	$ds = \frac{dd}{1+e^{-s}} \cdot (1 - \frac{1}{1+e^{-s}})$
<code>eltwise_logistic_use_dst_for_bwd</code>	$ds = dd \cdot d \cdot (1 - d)$
<code>eltwise_pow</code>	$ds = dd \cdot \alpha \beta s^{\beta-1}$
<code>eltwise_relu</code>	$ds = \begin{cases} dd & \text{if } s > 0 \\ \alpha \cdot dd & \text{if } s \leq 0 \end{cases}$
<code>eltwise_relu_use_dst_for_bwd</code>	$ds = \begin{cases} dd & \text{if } d > 0 \\ \alpha \cdot dd & \text{if } d \leq 0 \end{cases} \text{ only if } \alpha \geq 0$
<code>eltwise_soft_relu</code>	$ds = \frac{dd}{1+e^{-s}}$
<code>eltwise_sqrt</code>	$ds = \frac{dd}{2\sqrt{s}}$
<code>eltwise_sqrt_use_dst_for_bwd</code>	$ds = \frac{dd}{2d}$
<code>eltwise_square</code>	$ds = dd \cdot 2s$
<code>eltwise_swish</code>	$ds = \frac{dd}{1+e^{-\alpha s}} (1 + \alpha s (1 - \frac{1}{1+e^{-\alpha s}}))$
<code>eltwise_tanh</code>	$ds = dd \cdot (1 - \tanh^2 s)$
<code>eltwise_tanh_use_dst_for_bwd</code>	$ds = dd \cdot (1 - d^2)$

### 7.5.7.3 Difference Between Forward Training and Forward Inference

There is no difference between the #dnnl\_forward\_training and #dnnl\_forward\_inference propagation kinds.

### 7.5.7.4 Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	<i>DNNL_ARG_SRC</i>
dst	<i>DNNL_ARG_DST</i>
diff_src	<i>DNNL_ARG_DIFF_SRC</i>
diff_dst	<i>DNNL_ARG_DIFF_DST</i>

### 7.5.7.5 Operation Details

1. The `dnnl::eltwise_forward::desc::desc()` and `dnnl::eltwise_backward::desc::desc()` constructors take both parameters  $\alpha$ , and  $\beta$ . These parameters are ignored if they are unused by the algorithm.
2. The memory format and data type for src and dst are assumed to be the same, and in the API are typically denoted as data (for example `dnnl::eltwise_forward::desc::desc()` has a `data_desc` argument). The same holds for diff\_src and diff\_dst. The corresponding memory descriptors are denoted as `diff_data_desc`.
3. Both forward and backward propagation support in-place operations, meaning that src can be used as input and output for forward propagation, and diff\_dst can be used as input and output for backward propagation. In case of an in-place operation, the original data will be overwritten. Note, however, that some algorithms for backward propagation require original src, hence the corresponding forward propagation should not be performed in-place for those algorithms. Algorithms that use dst for backward propagation can be safely done in-place.
4. For some operations it might be beneficial to compute backward propagation based on  $dst(\bar{s})$ , rather than on  $src(\bar{s})$ , for improved performance.

---

**Note:** For operations supporting destination memory as input, dst can be used instead of src when backward propagation is computed. This enables several performance optimizations (see the tips below).

---

### 7.5.7.6 Data Type Support

The eltwise primitive should support the following combinations of data types.

---

**Note:** Here we abbreviate data types names for readability. For example, `dnnl::memory::data_type::f32` is abbreviated to `f32`.

---

Propagation	Source / Destination	Intermediate data type
forward / backward	<i>f32, bf16</i>	<i>f32</i>
forward	<i>f16</i>	<i>f16</i>
forward	<i>s32 / s8 / u8</i>	<i>f32</i>

Here the intermediate data type means that the values coming in are first converted to the intermediate data type, then the operation is applied, and finally the result is converted to the output data type.

### 7.5.7.7 Data Representation

The eltwise primitive works with arbitrary data tensors. There is no special meaning associated with any logical dimensions.

### 7.5.7.8 Post-ops and Attributes

The eltwise primitive does not have to support any post-ops or attributes.

### 7.5.7.9 API

```
struct dnnl::eltwise_forward : public dnnl::primitive
```

Elementwise unary operation forward propagation primitive.

#### Public Functions

##### **eltwise\_forward()**

Default constructor. Produces an empty object.

##### **eltwise\_forward(**const** primitive\_desc &pd)**

Constructs an eltwise forward propagation primitive.

#### Parameters

- pd: Primitive descriptor for an eltwise forward propagation primitive.

##### **struct desc**

Descriptor for an elementwise forward propagation primitive.

#### Public Functions

```
desc (prop_kind aprop_kind, algorithm aalgorithm, const memory::desc &data_desc, float alpha  
= 0, float beta = 0)
```

Constructs a descriptor for an elementwise forward propagation primitive.

#### Parameters

- aprop\_kind: Propagation kind. Possible values are *dnnl::prop\_kind::forward\_training*, and *dnnl::prop\_kind::forward\_inference*.
- aalgorithm: Elementwise algorithm kind.
- data\_desc: Source and destination memory descriptors.
- alpha: The alpha parameter for the elementwise operation. Specific meaning depends on the algorithm.
- beta: The beta parameter for the elementwise operation. Specific meaning depends on the algorithm.

##### **struct primitive\_desc : **public** dnnl::primitive\_desc**

Primitive descriptor for an elementwise forward propagation primitive.

## Public Functions

**primitive\_desc()**

Default constructor. Produces an empty object.

**primitive\_desc(const desc &adesc, const engine &aengine, bool allow\_empty = false)**

Constructs a primitive descriptor for an elementwise forward propagation primitive.

### Parameters

- adesc: Descriptor for an elementwise forward propagation primitive.
- aengine: Engine to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**primitive\_desc(const desc &adesc, const primitive\_attr &attr, const engine &aengine, bool allow\_empty = false)**

Constructs a primitive descriptor for an elementwise forward propagation primitive.

### Parameters

- adesc: Descriptor for an elementwise forward propagation primitive.
- aengine: Engine to use.
- attr: Primitive attributes to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**memory::desc src\_desc() const**

Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter.

**memory::desc dst\_desc() const**

Returns a destination memory descriptor.

**Return** Destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination parameter.

**struct dnnl::eltwise\_backward : public dnnl::primitive**

Elementwise unary operation backward propagation primitive.

**See** [eltwise\\_forward](#)

## Public Functions

**eltwise\_backward()**

Default constructor. Produces an empty object.

**eltwise\_backward(const primitive\_desc &pd)**

Constructs an eltwise backward propagation primitive.

### Parameters

- pd: Primitive descriptor for an eltwise backward propagation primitive.

**struct desc**

Descriptor for an elementwise backward propagation primitive.

## Public Functions

```
desc(algorithm aalgorithm, const memory::desc &diff_data_desc, const memory::desc &data_desc, float alpha = 0, float beta = 0)
```

Constructs a descriptor for an elementwise backward propagation primitive.

### Parameters

- *aalgorithm*: Elementwise algorithm kind.
- *diff\_data\_desc*: Diff source and destination memory descriptors.
- *data\_desc*: Source memory descriptor.
- *alpha*: The alpha parameter for the elementwise operation. Specific meaning depends on the algorithm.
- *beta*: The beta parameter for the elementwise operation. Specific meaning depends on the algorithm.

```
struct primitive_desc : public dnnl::primitive_desc
```

Primitive descriptor for eltwise backward propagation.

## Public Functions

```
primitive_desc()
```

Default constructor. Produces an empty object.

```
primitive_desc(const desc &adesc, const engine &aengine, const eltwise_forward::primitive_desc &hint_fwd_pd, bool allow_empty = false)
```

Constructs a primitive descriptor for an elementwise backward propagation primitive.

### Parameters

- *adesc*: Descriptor for an elementwise backward propagation primitive.
- *aengine*: Engine to use.
- *hint\_fwd\_pd*: Primitive descriptor for an elementwise forward propagation primitive. It is used as a hint for deciding which memory format to use.
- *allow\_empty*: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
primitive_desc(const desc &adesc, const primitive_attr &attr, const engine &aengine, const eltwise_forward::primitive_desc &hint_fwd_pd, bool allow_empty = false)
```

Constructs a primitive descriptor for an elementwise backward propagation primitive.

### Parameters

- *adesc*: Descriptor for an elementwise backward propagation primitive.
- *attr*: Primitive attributes to use.
- *aengine*: Engine to use.
- *hint\_fwd\_pd*: Primitive descriptor for an elementwise forward propagation primitive. It is used as a hint for deciding which memory format to use.
- *allow\_empty*: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
memory::desc src_desc() const
```

Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter.

```
memory::desc diff_src_desc() const
```

Returns a diff source memory descriptor.

**Return** Diff source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff source memory with.

`memory::desc diff_dst_desc() const`

Returns a diff destination memory descriptor.

**Return** Diff destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff destination parameter.

## 7.5.8 Inner Product

The inner product primitive (sometimes called *fully connected layer*) treats each activation in the minibatch as a vector and computes its product with a weights 2D tensor producing a 2D tensor as an output.

### 7.5.8.1 Forward

Let src, weights, bias and dst be  $N \times IC$ ,  $OC \times IC$ ,  $OC$ , and  $N \times OC$  tensors, respectively. Variable names follow the standard *Conventions*. Then:

$$dst(n, oc) = bias(oc) + \sum_{ic=0}^{IC-1} src(n, ic) \cdot weights(oc, ic)$$

In cases where the src and weights tensors have spatial dimensions, they are flattened to 2D. For example, if they are 4D  $N \times IC' \times IH \times IW$  and  $OC \times IC' \times KH \times KW$  tensors, then the formula above is applied with  $IC = IC' \cdot IH \cdot IW$ . In such cases, the src and weights tensors must have equal spatial dimensions (e.g.  $KH = IH$  and  $KW = IW$  for 4D tensors).

#### 7.5.8.1.1 Difference Between Forward Training and Forward Inference

There is no difference between the `forward_training` and `forward_inference` propagation kinds.

### 7.5.8.2 Backward

The backward propagation computes diff\_src based on diff\_dst and weights.

The weights update computes diff\_weights and diff\_bias based on diff\_dst and src.

---

**Note:** The *optimized* memory formats src and weights might be different on forward propagation, backward propagation, and weights update.

---

#### 7.5.8.2.1 Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	<i>DNNL_ARG_SRC</i>
weights	<i>DNNL_ARG_WEIGHTS</i>
bias	<i>DNNL_ARG_BIAS</i>
dst	<i>DNNL_ARG_DST</i>
diff_src	<i>DNNL_ARG_DIFF_SRC</i>
diff_weights	<i>DNNL_ARG_DIFF_WEIGHTS</i>
diff_bias	<i>DNNL_ARG_DIFF_BIAS</i>
diff_dst	<i>DNNL_ARG_DIFF_DST</i>

### 7.5.8.3 Operation Details

N/A

### 7.5.8.4 Data Types Support

Inner product primitive supports the following combination of data types for source, destination, weights, and bias.

---

**Note:** Here we abbreviate data types names for readability. For example, `dnnl::memory::data_type::f32` is abbreviated to `f32`.

---

Propagation	Source	Weights	Destination	Bias
forward / backward	<i>f32</i>	<i>f32</i>	<i>f32</i>	<i>f32</i>
forward	<i>f16</i>	<i>f16</i>	<i>f16</i>	<i>f16</i>
forward	<i>u8, s8</i>	<i>s8</i>	<i>u8, s8, s32, f32</i>	<i>u8, s8, s32, f32</i>
forward	<i>bf16</i>	<i>bf16</i>	<i>f32, bf16</i>	<i>f32, bf16</i>
backward	<i>f32, bf16</i>	<i>bf16</i>	<i>bf16</i>	
weights update	<i>bf16</i>	<i>f32, bf16</i>	<i>bf16</i>	<i>f32, bf16</i>

### 7.5.8.5 Data Representation

Like other CNN primitives, the inner product primitive expects the following tensors:

Spatial	Source	Destination	Weights
1D	$N \times C \times W$	$N \times C$	$OC \times IC \times KW$
2D	$N \times C \times H \times W$	$N \times C$	$OC \times IC \times KH \times KW$
3D	$N \times C \times D \times H \times W$	$N \times C$	$OC \times IC \times KD \times KH \times KW$

Memory format of data and weights memory objects is critical for inner product primitive performance. In the oneDNN programming model, inner product primitive is one of the few primitives that support the placeholder format `any` and can define data and weight memory objects formats based on the primitive parameters. When using `any` it is necessary to first create an inner product primitive descriptor and then query it for the actual data and weight memory objects formats.

The table below shows the combinations for which **plain** memory formats the inner product primitive is optimized for. For the destination tensor (which is always  $N \times C$ ) the memory format is always `nc(ab)`.

Spatial	Source / Weights logical tensor	Implementation optimized for memory formats
0D	NC / OI	<i>nc(ab)</i> / <i>oi(ab)</i>
0D	NC / OI	<i>nc(ab)</i> / <i>io(ba)</i>
1D	NCW / OIW	<i>ncw(abc)</i> / <i>oiw(abc)</i>
1D	NCW / OIW	<i>nwc(acb)</i> / <i>wio(cba)</i>
2D	NCHW / OIHW	<i>nchw(abcd)</i> / <i>oihw(abcd)</i>
2D	NCHW / OIHW	<i>nhwc(acdb)</i> / <i>hwio(cdba)</i>
3D	NCDHW / OIDHW	<i>ncdhw(abcd)</i> / <i>oidhw(abcd)</i>
3D	NCDHW / OIDHW	<i>ndhwc(acdeb)</i> / <i>dhwio(cdeba)</i>

### 7.5.8.6 Post-ops and Attributes

The following post-ops should be supported by inner product primitives:

Propagation	Type	Operation	Description	Restrictions
forward	at-tribute	<i>Output scale</i>	Scales the result of inner product by given scale factor(s)	int8 inner products only
forward	post-op	<i>Eltwise</i>	Applies an elementwise operation to the result	
forward	post-op	<i>Sum</i>	Adds the operation result to the destination tensor instead of overwriting it	

### 7.5.8.7 API

```
struct dnnl::inner_product_forward : public dnnl::primitive
    Inner product forward propagation primitive.
```

#### Public Functions

**inner\_product\_forward()**

Default constructor. Produces an empty object.

**inner\_product\_forward(const primitive\_desc &pd)**

Constructs an inner product forward propagation primitive.

#### Parameters

- pd: Primitive descriptor for an inner product forward propagation primitive.

**struct desc**

Descriptor for an inner product forward propagation primitive.

## Public Functions

```
desc(prop_kind aprop_kind, const memory::desc &src_desc, const memory::desc
&weights_desc, const memory::desc &bias_desc, const memory::desc &dst_desc)
```

Constructs a descriptor for an inner product forward propagation primitive with bias.

**Note** All the memory descriptors may be initialized with the *dnnl::memory::format\_tag::any* value of *format\_tag*.

### Parameters

- *aprop\_kind*: Propagation kind. Possible values are *dnnl::prop\_kind::forward\_training*, and *dnnl::prop\_kind::forward\_inference*.
- *src\_desc*: Memory descriptor for src.
- *weights\_desc*: Memory descriptor for diff weights.
- *bias\_desc*: Memory descriptor for diff bias.
- *dst\_desc*: Memory descriptor for diff dst.

```
desc(prop_kind aprop_kind, const memory::desc &src_desc, const memory::desc
&weights_desc, const memory::desc &dst_desc)
```

Constructs a descriptor for an inner product forward propagation primitive without bias.

**Note** All the memory descriptors may be initialized with the *dnnl::memory::format\_tag::any* value of *format\_tag*.

### Parameters

- *aprop\_kind*: Propagation kind. Possible values are *dnnl::prop\_kind::forward\_training*, and *dnnl::prop\_kind::forward\_inference*.
- *src\_desc*: Memory descriptor for src.
- *weights\_desc*: Memory descriptor for diff weights.
- *dst\_desc*: Memory descriptor for dst.

```
struct primitive_desc : public dnnl::primitive_desc
```

Primitive descriptor for an inner product forward propagation primitive.

## Public Functions

```
primitive_desc()
```

Default constructor. Produces an empty object.

```
primitive_desc(const desc &adesc, const engine &aengine, bool allow_empty = false)
```

Constructs a primitive descriptor for an inner product forward propagation primitive.

### Parameters

- *adesc*: Descriptor for an inner product forward propagation primitive.
- *aengine*: Engine to use.
- *allow\_empty*: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
primitive_desc(const desc &adesc, const primitive_attr &attr, const engine &aengine,
bool allow_empty = false)
```

Constructs a primitive descriptor for an inner product forward propagation primitive.

### Parameters

- *adesc*: Descriptor for an inner product forward propagation primitive.
- *attr*: Primitive attributes to use.
- *aengine*: Engine to use.
- *allow\_empty*: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

`memory::desc src_desc() const`

Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter.

`memory::desc weights_desc() const`

Returns a weights memory descriptor.

**Return** Weights memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a weights parameter.

`memory::desc dst_desc() const`

Returns a destination memory descriptor.

**Return** Destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination parameter.

`memory::desc bias_desc() const`

Returns the bias memory descriptor.

**Return** The bias memory descriptor.

**Return** A zero memory descriptor of the primitive does not have a bias parameter.

**struct** `dnnl::inner_product_backward_data : public dnnl::primitive`

Inner product backward propagation primitive.

## Public Functions

`inner_product_backward_data()`

Default constructor. Produces an empty object.

`inner_product_backward_data(const primitive_desc &pd)`

Constructs an inner product backward propagation primitive.

### Parameters

- pd: Primitive descriptor for an inner product backward propagation primitive.

**struct desc**

Descriptor for an inner product backward propagation primitive.

## Public Functions

`desc(const memory::desc &diff_src_desc, const memory::desc &weights_desc, const memory::desc &diff_dst_desc)`

Constructs a descriptor for an inner product backward propagation primitive.

**Note** All the memory descriptors may be initialized with the `dnnl::memory::format_tag::any` value of `format_tag`.

### Parameters

- diff\_src\_desc: Memory descriptor for diff src.
- weights\_desc: Memory descriptor for weights.
- diff\_dst\_desc: Memory descriptor for diff dst.

**struct primitive\_desc : public dnnl::primitive\_desc**

Primitive descriptor for an inner product backward propagation primitive.

## Public Functions

**primitive\_desc()**

Default constructor. Produces an empty object.

**primitive\_desc(const desc &adesc, const engine &aengine, const inner\_product\_forward::primitive\_desc &hint\_fwd\_pd, bool allow\_empty = false)**

Constructs a primitive descriptor for an inner product backward propagation primitive.

### Parameters

- adesc: Descriptor for an inner product backward propagation primitive.
- aengine: Engine to use.
- hint\_fwd\_pd: Primitive descriptor for an inner product forward propagation primitive. It is used as a hint for deciding which memory format to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**primitive\_desc(const desc &adesc, const primitive\_attr &attr, const engine &aengine, const inner\_product\_forward::primitive\_desc &hint\_fwd\_pd, bool allow\_empty = false)**

Constructs a primitive descriptor for an inner product backward propagation primitive.

### Parameters

- adesc: Descriptor for an inner product backward propagation primitive.
- attr: Primitive attributes to use.
- aengine: Engine to use.
- hint\_fwd\_pd: Primitive descriptor for an inner product forward propagation primitive. It is used as a hint for deciding which memory format to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**memory::desc diff\_src\_desc() const**

Returns a diff source memory descriptor.

**Return** Diff source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff source memory with.

**memory::desc weights\_desc() const**

Returns a weights memory descriptor.

**Return** Weights memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a weights parameter.

**memory::desc diff\_dst\_desc() const**

Returns a diff destination memory descriptor.

**Return** Diff destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff destination parameter.

**struct dnnl::inner\_product\_backward\_weights : public dnnl::primitive**

Inner product weights gradient primitive.

## Public Functions

**inner\_product\_backward\_weights()**

Default constructor. Produces an empty object.

**inner\_product\_backward\_weights(const primitive\_desc &pd)**

Constructs an inner product weights gradient primitive.

### Parameters

- pd: Primitive descriptor for an inner product weights gradient primitive.

**struct desc**

Descriptor for an inner product weights gradient primitive.

## Public Functions

**desc(const memory::desc &src\_desc, const memory::desc &diff\_weights\_desc, const memory::desc &diff\_bias\_desc, const memory::desc &diff\_dst\_desc)**

Constructs a descriptor for an inner product descriptor weights update primitive with bias.

**Note** All the memory descriptors may be initialized with the `dnnl::memory::format_tag::any` value of `format_tag`.

### Parameters

- src\_desc: Memory descriptor for src.
- diff\_weights\_desc: Memory descriptor for diff weights.
- diff\_bias\_desc: Memory descriptor for diff bias.
- diff\_dst\_desc: Memory descriptor for diff dst.

**desc(const memory::desc &src\_desc, const memory::desc &diff\_weights\_desc, const memory::desc &diff\_dst\_desc)**

Constructs a descriptor for an inner product descriptor weights update primitive without bias.

**Note** All the memory descriptors may be initialized with the `dnnl::memory::format_tag::any` value of `format_tag`.

### Parameters

- src\_desc: Memory descriptor for src.
- diff\_weights\_desc: Memory descriptor for diff weights.
- diff\_dst\_desc: Memory descriptor for diff dst.

**struct primitive\_desc : public dnnl::primitive\_desc**

Primitive descriptor for an inner product weights gradient primitive.

## Public Functions

**primitive\_desc()**

Default constructor. Produces an empty object.

**primitive\_desc(const desc &adesc, const engine &aengine, const inner\_product\_forward::primitive\_desc &hint\_fwd\_pd, bool allow\_empty = false)**

Constructs a primitive descriptor for an inner product weights update primitive.

### Parameters

- adesc: Descriptor for an inner product weights gradient primitive.
- aengine: Engine to use.
- hint\_fwd\_pd: Primitive descriptor for an inner product forward propagation primitive. It is used as a hint for deciding which memory format to use.

- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
primitive_desc (const desc &adesc, const primitive_attr &attr, const engine &aengine,  

const inner_product_forward::primitive_desc &hint_fwd_pd, bool allow_empty = false)
```

Constructs a primitive descriptor for an inner product weights update primitive.

#### Parameters

- *adesc*: Descriptor for an inner product weights gradient primitive.
- *attr*: Primitive attributes to use.
- *aengine*: Engine to use.
- *hint\_fwd\_pd*: Primitive descriptor for an inner product forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

*memory::desc* **src\_desc** () **const**

Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter.

*memory::desc* **diff\_weights\_desc** () **const**

Returns a diff weights memory descriptor.

**Return** Diff weights memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff weights parameter.

*memory::desc* **diff\_dst\_desc** () **const**

Returns a diff destination memory descriptor.

**Return** Diff destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff destination parameter.

*memory::desc* **diff\_bias\_desc** () **const**

Returns the diff bias memory descriptor.

**Return** The diff bias memory descriptor.

**Return** A zero memory descriptor of the primitive does not have a diff bias parameter.

## 7.5.9 Layer normalization

The layer normalization primitive performs a forward or backward layer normalization operation on a 2-5D data tensor.

The layer normalization operation performs normalization over the last logical axis of the data tensor and is defined by the following formulas. We show formulas only for 3D data, which are straightforward to generalize to cases of higher dimensions. Variable names follow the standard *Conventions*.

### 7.5.9.1 Forward

$$\text{dst}(t, n, c) = \gamma(c) \cdot \frac{\text{src}(t, n, c) - \mu(t, n)}{\sqrt{\sigma^2(t, n) + \varepsilon}} + \beta(c),$$

where

- $\gamma(c), \beta(c)$  are optional scale and shift for a channel (see the `use_scaleshift` flag),
- $\mu(t, n), \sigma^2(t, n)$  are mean and variance (see `use_global_stats` flag), and
- $\varepsilon$  is a constant to improve numerical stability.

Mean and variance are computed at runtime or provided by a user. When mean and variance are computed at runtime, the following formulas are used:

- $\mu(t, n) = \frac{1}{C} \sum_c \text{src}(t, n, c),$
- $\sigma^2(t, n) = \frac{1}{C} \sum_c (\text{src}(t, n, c) - \mu(t, n))^2.$

The  $\gamma(c)$  and  $\beta(c)$  tensors are considered learnable.

#### 7.5.9.1.1 Difference Between Forward Training and Forward Inference

If mean and variance are computed at runtime (i.e., `use_global_stats` is not set), they become outputs for the propagation kind `forward_training` (because they would be required during the backward propagation). Data layout for mean and variance must be specified during initialization of the layer normalization descriptor by passing the memory descriptor for statistics (e.g., by passing `stat_desc` in `dnnl::layer_normalization_forward::desc::desc()`). Mean and variance are not exposed for the propagation kind `forward_inference`.

### 7.5.9.2 Backward

The backward propagation computes  $\text{diff\_src}(t, n, c)$ ,  $\text{diff\_}\gamma(c)^*$ , and  $\text{diff\_}\beta(c)^*$  based on  $\text{diff\_dst}(t, n, c)$ ,  $\text{src}(t, n, c)$ ,  $\mu(t, n)$ ,  $\sigma^2(t, n)$ ,  $\gamma(c)^*$ , and  $\beta(c)^*$ .

The tensors marked with an asterisk are used only when the primitive is configured to use  $\gamma(c)$ , and  $\beta(c)$  (i.e., `use_scaleshift` is set).

### 7.5.9.3 Execution Arguments

Depending on the flags and propagation kind, the layer normalization primitive requires different inputs and outputs. For clarity, a summary is shown below.

	<code>forward_inference</code>	<code>forward_training</code>	<code>backward</code>	<code>backward_data</code>
<code>none</code>	<i>In:</i> src <i>Out:</i> dst	<i>In:</i> src <i>Out:</i> dst, $\mu, \sigma^2$	<i>In:</i> diff_dst, src, $\mu, \sigma^2$ <i>Out:</i> diff_src	Same as for <code>backward</code>
<code>use_global_stats</code>	<i>In:</i> src, $\mu, \sigma^2$ <i>Out:</i> dst	<i>In:</i> src, $\mu, \sigma^2$ <i>Out:</i> dst	<i>In:</i> diff_dst, src, $\mu, \sigma^2$ <i>Out:</i> diff_src	Same as for <code>backward</code>
<code>use_scaleshift</code>	<i>In:</i> src, $\gamma, \beta$ <i>Out:</i> dst	<i>In:</i> src, $\gamma, \beta$ <i>Out:</i> dst, $\mu, \sigma^2$	<i>In:</i> diff_dst, src, $\mu, \sigma^2, \gamma, \beta$ <i>Out:</i> diff_src, diff_γ, diff_β	Not supported
<code>use_global_stats   use_scaleshift</code>	<i>In:</i> src, $\mu, \sigma^2, \gamma, \beta$ <i>Out:</i> dst	<i>In:</i> src, $\mu, \sigma^2, \gamma, \beta$ <i>Out:</i> dst	<i>In:</i> diff_dst, src, $\mu, \sigma^2, \gamma, \beta$ <i>Out:</i> diff_src, diff_γ, diff_β	Not supported

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	<i>DNNL_ARG_SRC</i>
$\gamma, \beta$	<i>DNNL_ARG_SCALE_SHIFT</i>
mean ( $\mu$ )	<i>DNNL_ARG_MEAN</i>
variance ( $\sigma$ )	<i>DNNL_ARG_VARIANCE</i>
dst	<i>DNNL_ARG_DST</i>
diff_dst	<i>DNNL_ARG_DIFF_DST</i>
diff_src	<i>DNNL_ARG_DIFF_SRC</i>
diff_ $\gamma$ , diff_ $\beta$	<i>DNNL_ARG_DIFF_SCALE_SHIFT</i>

#### 7.5.9.4 Operation Details

1. The different flavors of the primitive are partially controlled by the flags parameter that is passed to the operation descriptor initialization function (e.g., `dnnl::layer_normalization_forward::desc::desc()`). Multiple flags can be combined using the bitwise OR operator (`|`).
2. For forward propagation, the mean and variance might be either computed at runtime (in which case they are outputs of the primitive) or provided by a user (in which case they are inputs). In the latter case, a user must set the `use_global_stats` flag. For the backward propagation, the mean and variance are always input parameters.
3. The memory format and data type for `src` and `dst` are assumed to be the same, and in the API they are typically referred to as `data` (e.g., see `data_desc` in `dnnl::layer_normalization_forward::desc::desc()`). The same is true for `diff_src` and `diff_dst`. The corresponding memory descriptors are referred to as `diff_data_desc`.
4. Both forward and backward propagation support in-place operations, meaning that `src` can be used as input and output for forward propagation, and `diff_dst` can be used as input and output for backward propagation. In case of an in-place operation, the original data will be overwritten. Note, however, that backward propagation requires original `src`, hence the corresponding forward propagation should not be performed in-place.

#### 7.5.9.5 Data Types Support

The layer normalization supports the following combinations of data types.

---

**Note:** Here we abbreviate data types names for readability. For example, `dnnl::memory::data_type::f32` is abbreviated to `f32`.

---

Propagation	Source / Destination	Mean / Variance / ScaleShift
forward / backward	<code>f32</code>	<code>f32</code>
forward	<code>f16</code>	<code>f32</code>

## 7.5.9.6 Data Representation

### 7.5.9.6.1 Mean and Variance

The mean ( $\mu$ ) and variance ( $\sigma^2$ ) are separate tensors with number of dimensions equal to ( $data\_ndims - 1$ ) and size ( $data\_dim[0], data\_dim[1], \dots, data\_dim[ndims - 2]$ ).

The corresponding memory object can have an arbitrary memory format. Unless mean and variance are computed at runtime and not exposed (i.e., propagation kind is `forward_inference` and `use_global_stats` is not set), the user should provide a memory descriptor for statistics when initializing the layer normalization descriptor. For best performance, it is advised to use the memory format that follows the data memory format; i.e., if the data format is `tnc`, the best performance can be expected for statistics with the `tn` format and suboptimal for statistics with the `nt` format.

### 7.5.9.6.2 Scale and Shift

If used, the scale ( $\gamma$ ) and shift ( $\beta$ ) are combined in a single 2D tensor of shape  $2 \times C$ .

The format of the corresponding memory object must be `nc(ab)`.

### 7.5.9.6.3 Source, Destination, and Their Gradients

The layer normalization primitive works with an arbitrary data tensor; however, it was designed for RNN data tensors (i.e., `nc`, `tnc`, `ldnc`). Unlike CNN data tensors, RNN data tensors have a single feature dimension. Layer normalization performs normalization over the last logical dimension (feature dimension for RNN tensors) across non-feature dimensions.

The layer normalization primitive is optimized for the following memory formats:

Logical tensor	Implementations optimized for memory formats
NC	<code>nc(ab)</code>
TNC	<code>tnc(abc)</code> , <code>ntc(bac)</code>
LDNC	<code>ldnc(abcd)</code>

## 7.5.9.7 API

```
struct dnnl::layer_normalization_forward : public dnnl::primitive
    Layer normalization forward propagation primitive.
```

### Public Functions

`layer_normalization_forward()`

Default constructor. Produces an empty object.

`layer_normalization_forward(const primitive_desc &pd)`

Constructs a layer normalization forward propagation primitive.

#### Parameters

- pd: Primitive descriptor for a layer normalization forward propagation primitive.

`struct desc`

Descriptor for a layer normalization forward propagation primitive.

## Public Functions

```
desc (prop_kind aprop_kind, const memory::desc &data_desc, const memory::desc &stat_desc,  
float epsilon, normalization_flags flags)
```

Constructs a descriptor for layer normalization forward propagation primitive.

### Parameters

- *aprop\_kind*: Propagation kind. Possible values are *dnnl::prop\_kind::forward\_training*, and *dnnl::prop\_kind::forward\_inference*.
- *data\_desc*: Source and destination memory descriptor.
- *stat\_desc*: Statistics memory descriptors.
- *epsilon*: Layer normalization epsilon parameter.
- *flags*: Layer normalization flags (*dnnl::normalization\_flags*).

```
desc (prop_kind aprop_kind, const memory::desc &data_desc, float epsilon, normalization_flags  
flags)
```

Constructs a descriptor for layer normalization forward propagation primitive.

### Parameters

- *aprop\_kind*: Propagation kind. Possible values are *dnnl::prop\_kind::forward\_training*, and *dnnl::prop\_kind::forward\_inference*.
- *data\_desc*: Source and destination memory descriptor.
- *epsilon*: Layer normalization epsilon parameter.
- *flags*: Layer normalization flags (*dnnl::normalization\_flags*).

```
struct primitive_desc : public dnnl::primitive_desc
```

Primitive descriptor for a layer normalization forward propagation primitive.

## Public Functions

```
primitive_desc()
```

Default constructor. Produces an empty object.

```
primitive_desc (const desc &adesc, const engine &aengine, bool allow_empty = false)
```

Constructs a primitive descriptor for a layer normalization forward propagation primitive.

### Parameters

- *adesc*: Descriptor for a layer normalization forward propagation primitive.
- *aengine*: Engine to use.
- *allow\_empty*: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
primitive_desc (const desc &adesc, const primitive_attr &attr, const engine &aengine,  
bool allow_empty = false)
```

Constructs a primitive descriptor for a layer normalization forward propagation primitive.

### Parameters

- *adesc*: Descriptor for a layer normalization forward propagation primitive.
- *attr*: Primitive attributes to use.
- *aengine*: Engine to use.
- *allow\_empty*: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
memory::desc src_desc () const
```

Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter.

`memory::desc dst_desc() const`

Returns a destination memory descriptor.

**Return** Destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination parameter.

`memory::desc weights_desc() const`

Returns a weights memory descriptor.

**Return** Weights memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a weights parameter.

`memory::desc workspace_desc() const`

Returns the workspace memory descriptor.

**Return** Workspace memory descriptor.

**Return** A zero memory descriptor if the primitive does not require workspace parameter.

`memory::desc mean_desc() const`

Returns memory descriptor for mean.

**Return** Memory descriptor for mean.

`memory::desc variance_desc() const`

Returns memory descriptor for variance.

**Return** Memory descriptor for variance.

**struct** `dnnl::layer_normalization_backward : public dnnl::primitive`

Layer normalization backward propagation primitive.

## Public Functions

`layer_normalization_backward()`

Default constructor. Produces an empty object.

`layer_normalization_backward(const primitive_desc &pd)`

Constructs a layer normalization backward propagation primitive.

### Parameters

- pd: Primitive descriptor for a layer normalization backward propagation primitive.

**struct desc**

Descriptor for a layer normalization backward propagation primitive.

## Public Functions

`desc(prop_kind aprop_kind, const memory::desc &diff_data_desc, const memory::desc &data_desc, const memory::desc &stat_desc, float epsilon, normalization_flags flags)`

Constructs a descriptor for layer normalization backward propagation primitive.

### Parameters

- aprop\_kind: Propagation kind. Possible values are `dnnl::prop_kind::backward_data` and `dnnl::prop_kind::backward` (diffs for all parameters are computed in this case).
- diff\_data\_desc: Diff source and diff destination memory descriptor.
- data\_desc: Source memory descriptor.
- stat\_desc: Statistics memory descriptors.
- epsilon: Layer normalization epsilon parameter.
- flags: Layer normalization flags (`dnnl::normalization_flags`).

```
desc (prop_kind a_prop_kind, const memory::desc &diff_data_desc, const memory::desc &data_desc, float epsilon, normalization_flags flags)
```

Constructs a descriptor for layer normalization backward propagation primitive.

**Parameters**

- *a\_prop\_kind*: Propagation kind. Possible values are *dnnl::prop\_kind::backward\_data* and *dnnl::prop\_kind::backward* (diffs for all parameters are computed in this case).
- *diff\_data\_desc*: Diff source and diff destination memory descriptor.
- *data\_desc*: Source memory descriptor.
- *epsilon*: Layer normalization epsilon parameter.
- *flags*: Layer normalization flags (*dnnl::normalization\_flags*).

```
struct primitive_desc : public dnnl::primitive_desc
```

Primitive descriptor for a layer normalization backward propagation primitive.

**Public Functions**

```
primitive_desc()
```

Default constructor. Produces an empty object.

```
primitive_desc (const desc &adesc, const engine &aengine, const layer_normalization_forward::primitive_desc &hint_fwd_pd, bool allow_empty = false)
```

Constructs a primitive descriptor for a layer normalization backward propagation primitive.

**Parameters**

- *adesc*: Descriptor for a layer normalization backward propagation primitive.
- *aengine*: Engine to use.
- *hint\_fwd\_pd*: Primitive descriptor for a layer normalization forward propagation primitive. It is used as a hint for deciding which memory format to use.
- *allow\_empty*: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
primitive_desc (const desc &adesc, const primitive_attr &attr, const engine &aengine, const layer_normalization_forward::primitive_desc &hint_fwd_pd, bool allow_empty = false)
```

Constructs a primitive descriptor for a layer normalization backward propagation primitive.

**Parameters**

- *adesc*: Descriptor for a layer normalization backward propagation primitive.
- *attr*: Primitive attributes to use.
- *aengine*: Engine to use.
- *hint\_fwd\_pd*: Primitive descriptor for a layer normalization forward propagation primitive. It is used as a hint for deciding which memory format to use.
- *allow\_empty*: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
memory::desc src_desc() const
```

Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter.

```
memory::desc weights_desc() const
```

Returns a weights memory descriptor.

**Return** Weights memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a weights parameter.

---

`memory::desc dst_desc() const`  
 Returns a destination memory descriptor.  
**Return** Destination memory descriptor.

`memory::desc diff_src_desc() const`  
 Returns a diff source memory descriptor.  
**Return** Diff source memory descriptor.  
**Return** A zero memory descriptor if the primitive does not have a diff source memory with.

`memory::desc diff_dst_desc() const`  
 Returns a diff destination memory descriptor.  
**Return** Diff destination memory descriptor.  
**Return** A zero memory descriptor if the primitive does not have a diff destination parameter.

`memory::desc diff_weights_desc() const`  
 Returns a diff weights memory descriptor.  
**Return** Diff weights memory descriptor.  
**Return** A zero memory descriptor if the primitive does not have a diff weights parameter.

`memory::desc mean_desc() const`  
 Returns memory descriptor for mean.  
**Return** Memory descriptor for mean.

`memory::desc variance_desc() const`  
 Returns memory descriptor for variance.  
**Return** Memory descriptor for variance.

`memory::desc workspace_desc() const`  
 Returns the workspace memory descriptor.  
**Return** Workspace memory descriptor.  
**Return** A zero memory descriptor if the primitive does not require workspace parameter.

## 7.5.10 LogSoftmax

The logsoftmax primitive performs softmax along a particular axis on data with arbitrary dimensions followed by the logarithm function. All other axes are treated as independent (batch).

In general form, the operation is defined by the following formulas. Variable names follow the standard *Conventions*.

### 7.5.10.1 Forward

The second form is used as more numerically stable:

$$\begin{aligned} \text{dst}(\overline{ou}, c, \overline{in}) &= \ln \left( \frac{e^{\text{src}(\overline{ou}, c, \overline{in}) - \nu(\overline{ou}, \overline{in})}}{\sum_{ic} e^{\text{src}(\overline{ou}, ic, \overline{in}) - \nu(\overline{ou}, \overline{in})}} \right) \\ &= (\text{src}(\overline{ou}, c, \overline{in}) - \nu(\overline{ou}, \overline{in})) - \ln \left( \sum_{ic} e^{\text{src}(\overline{ou}, ic, \overline{in}) - \nu(\overline{ou}, \overline{in})} \right), \end{aligned}$$

where

- $c$  axis over which the logsoftmax computation is computed on,
- $\overline{ou}$  is the outermost index (to the left of logsoftmax axis),
- $\overline{in}$  is the innermost index (to the right of logsoftmax axis), and
- $\nu$  is used to produce more accurate results and defined as:

$$\nu(\overline{ou}, \overline{in}) = \max_{ic} \text{src}(\overline{ou}, ic, \overline{in})$$

#### 7.5.10.1.1 Difference Between Forward Training and Forward Inference

There is no difference between the *forward\_training* and *forward\_inference* propagation kinds.

#### 7.5.10.2 Backward

The backward propagation computes  $\text{diff\_src}(ou, c, in)$ , based on  $\text{diff\_dst}(ou, c, in)$  and  $\text{dst}(ou, c, in)$ .

#### 7.5.10.3 Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	<i>DNNL_ARG_SRC</i>
dst	<i>DNNL_ARG_DST</i>
diff_src	<i>DNNL_ARG_DIFF_SRC</i>
diff_dst	<i>DNNL_ARG_DIFF_DST</i>

#### 7.5.10.4 Operation Details

Both forward and backward propagation support in-place operations, meaning that `src` can be used as input and output for forward propagation, and `diff_dst` can be used as input and output for backward propagation. In case of in-place operation, the original data will be overwritten.

#### 7.5.10.5 Post-ops and Attributes

The logsoftmax primitive does not support any post-ops or attributes.

#### 7.5.10.6 Data Type Support

The logsoftmax primitive supports the following combinations of data types.

---

**Note:** Here we abbreviate data types names for readability. For example, `dnnl::memory::data_type::f32` is abbreviated to `f32`.

---

Propagation	Source / Destination
forward / backward	<i>bf16, f32</i>

## 7.5.10.7 Data Representation

### 7.5.10.7.1 Source, Destination, and Their Gradients

The logsoftmax primitive works with arbitrary data tensors. There is no special meaning associated with any logical dimensions. However, the logsoftmax axis is typically referred to as channels (hence in formulas we use  $c$ ).

## 7.5.10.8 API

```
struct dnnl::logsoftmax_forward : public dnnl::primitive
```

Logsoftmax forward propagation primitive.

### Public Functions

**logsoftmax\_forward()**

Default constructor. Produces an empty object.

**logsoftmax\_forward(const primitive\_desc &pd)**

Constructs a logsoftmax forward propagation primitive.

#### Parameters

- pd: Primitive descriptor for a logsoftmax forward propagation primitive.

**struct desc**

Descriptor for a logsoftmax forward propagation primitive.

### Public Functions

**desc()**

Default constructor. Produces an empty object.

**desc(prop\_kind aprop\_kind, const memory::desc &data\_desc, int logsoftmax\_axis)**

Constructs a descriptor for a logsoftmax forward propagation primitive.

#### Parameters

- aprop\_kind: Propagation kind. Possible values are `dnnl::prop_kind::forward_training`, and `dnnl::prop_kind::forward_inference`.
- data\_desc: Source and destination memory descriptor.
- logsoftmax\_axis: Axis over which softmax is computed.

**struct primitive\_desc : public dnnl::primitive\_desc**

Primitive descriptor for a logsoftmax forward propagation primitive.

### Public Functions

**primitive\_desc()**

Default constructor. Produces an empty object.

**primitive\_desc(const desc &adesc, const engine &aengine, bool allow\_empty = false)**

Constructs a primitive descriptor for a logsoftmax forward propagation primitive.

#### Parameters

- adesc: descriptor for a logsoftmax forward propagation primitive.
- aengine: Engine to use.

- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
primitive_desc (const desc &adesc, const primitive_attr &attr, const engine &aengine,  
                  bool allow_empty = false)
```

Constructs a primitive descriptor for a logsoftmax forward propagation primitive.

#### Parameters

- *adesc*: Descriptor for a logsoftmax forward propagation primitive.
- *aengine*: Engine to use.
- *attr*: Primitive attributes to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
memory::desc src_desc () const
```

Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter.

```
memory::desc dst_desc () const
```

Returns a destination memory descriptor.

**Return** Destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination parameter.

```
struct dnnl::logsoftmax_backward : public dnnl::primitive
```

Logsoftmax backward propagation primitive.

### Public Functions

```
logsoftmax_backward ()
```

Default constructor. Produces an empty object.

```
logsoftmax_backward (const primitive_desc &pd)
```

Constructs a logsoftmax backward propagation primitive.

#### Parameters

- *pd*: Primitive descriptor for a logsoftmax backward propagation primitive.

```
struct desc
```

Descriptor for a logsoftmax backward propagation primitive.

### Public Functions

```
desc ()
```

Default constructor. Produces an empty object.

```
desc (const memory::desc &diff_data_desc, const memory::desc &data_desc, int logsoftmax_axis)
```

Constructs a descriptor for a logsoftmax backward propagation primitive.

#### Parameters

- *diff\_data\_desc*: Diff source and diff destination memory descriptors.
- *data\_desc*: Destination memory descriptor.
- *logsoftmax\_axis*: Axis over which softmax is computed.

---

```
struct primitive_desc : public dnnl::primitive_desc
```

Primitive descriptor for a logsoftmax backward propagation primitive.

## Public Functions

```
primitive_desc()
```

Default constructor. Produces an empty object.

```
primitive_desc(const desc &adesc, const engine &aengine, const logsoft-
```

```
max_forward::primitive_desc &hint_fwd_pd, bool allow_empty = false)
```

Constructs a primitive descriptor for a logsoftmax backward propagation primitive.

### Parameters

- **adesc**: Descriptor for a logsoftmax backward propagation primitive.
- **aengine**: Engine to use.
- **hint\_fwd\_pd**: Primitive descriptor for a logsoftmax forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **allow\_empty**: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
primitive_desc(const desc &adesc, const primitive_attr &attr, const engine &aengine,
```

```
const logsoftmax_forward::primitive_desc &hint_fwd_pd, bool allow_empty
```

```
= false)
```

Constructs a primitive descriptor for a logsoftmax backward propagation primitive.

### Parameters

- **adesc**: Descriptor for a logsoftmax backward propagation primitive.
- **attr**: Primitive attributes to use.
- **aengine**: Engine to use.
- **hint\_fwd\_pd**: Primitive descriptor for a logsoftmax forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **allow\_empty**: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
memory::desc dst_desc() const
```

Returns a destination memory descriptor.

**Return** Destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination parameter.

```
memory::desc diff_src_desc() const
```

Returns a diff source memory descriptor.

**Return** Diff source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff source memory with.

```
memory::desc diff_dst_desc() const
```

Returns a destination memory descriptor.

**Return** Destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination parameter.

## 7.5.11 Local Response Normalization

The LRN primitive performs a forward or backward local response normalization operation defined by the following formulas. Variable names follow the standard *Conventions*.

### 7.5.11.1 Forward

LRN *across channels*:

$$\text{dst}(n, c, h, w) = \left\{ k + \frac{\alpha}{n_l} \sum_{i=-(n_l-1)/2}^{(n_l+1)/2-1} (\text{src}(n, c+i, h, w))^2 \right\}^{-\beta} \cdot \text{src}(n, c, h, w),$$

LRN *within channel*:

$$\text{dst}(n, c, h, w) = \left\{ k + \frac{\alpha}{n_l} \sum_{i=-(n_l-1)/2}^{(n_l+1)/2-1} \sum_{j=-(n_l-1)/2}^{(n_l+1)/2-1} (\text{src}(n, c, h+i, w+j))^2 \right\}^{-\beta} \cdot \text{src}(n, c, h, w),$$

where  $n_l$  is the `local_size`. Formulas are provided for 2D spatial data case.

### 7.5.11.2 Backward

The backward propagation computes  $\text{diff\_src}(n, c, h, w)$ , based on  $\text{diff\_dst}(n, c, h, w)$  and  $\text{src}(n, c, h, w)$ .

#### 7.5.11.2.1 Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
<code>src</code>	<code>DNNL_ARG_SRC</code>
<code>dst</code>	<code>DNNL_ARG_DST</code>
<code>workspace</code>	<code>DNNL_ARG_WORKSPACE</code>
<code>diff_src</code>	<code>DNNL_ARG_DIFF_SRC</code>
<code>diff_dst</code>	<code>DNNL_ARG_DIFF_DST</code>

##### 7.5.11.2.1.1 Operation Details

1. During training, LRN might or might not require a workspace on forward and backward passes. The behavior is implementation specific. Optimized implementations typically require a workspace and use it to save some intermediate results from the forward pass that accelerate computations on the backward pass. To check whether a workspace is required, query the LRN primitive descriptor for the workspace. Success indicates that the workspace is required and its description will be returned.
2. The memory format and data type for `src` and `dst` are assumed to be the same, and in the API are typically referred to as `data` (e.g., see `data_desc` in `dnnl::lrn_forward::desc::desc()`). The same holds for `diff_src` and `diff_dst`. The corresponding memory descriptors are referred to as `diff_data_desc`.

### 7.5.11.2.1.2 Data Type Support

The LRN primitive supports the following combinations of data types.

---

**Note:** Here we abbreviate data types names for readability. For example, `dnnl::memory::data_type::f32` is abbreviated to `f32`.

---

Propagation	Source / Destination
forward / backward	<code>f32, bf16</code>
forward	<code>f16</code>

### 7.5.11.2.1.3 Data Representation

#### 7.5.11.2.2 Source, Destination, and Their Gradients

Like most other primitives, the LRN primitive expects the following tensors:

Spatial	Source / Destination
0D	$N \times C$
1D	$N \times C \times W$
2D	$N \times C \times H \times W$
3D	$N \times C \times D \times H \times W$

The LRN primitive is optimized for the following memory formats:

Spatial	Logical tensor	Implementations optimized for memory formats
2D	NCHW	<code>nchw (abcd), nhwc (acdb), optimized</code>

Here *optimized* means the format chosen by the preceding compute-intensive primitive.

#### 7.5.11.2.2.1 Post-ops and Attributes

The LRN primitive does not support any post-ops or attributes.

#### 7.5.11.2.2.2 API

```
struct dnnl::lrn_forward : public dnnl::primitive
    Local response normalization (LRN) forward propagation primitive.
```

## Public Functions

```
lrn_forward()
    Default constructor. Produces an empty object.

lrn_forward(const primitive_desc &pd)
    Constructs an LRN forward propagation primitive.
```

### Parameters

- pd: Primitive descriptor for an LRN forward propagation primitive.

### struct desc

Descriptor for an LRN forward propagation primitive.

## Public Functions

```
desc(prop_kind aprop_kind, algorithm aalgorithm, const memory::desc &data_desc, memory::dim local_size, float alpha, float beta, float k = 1.f)
    Constructs a descriptor for a LRN forward propagation primitive.
```

### Parameters

- aprop\_kind: Propagation kind. Possible values are `dnnl::prop_kind::forward_training`, and `dnnl::prop_kind::forward_inference`.
- aalgorithm: LRN algorithm kind: either `dnnl::algorithm::lrn_across_channels`, or `dnnl::algorithm::lrn_within_channel`.
- data\_desc: Source and destination memory descriptors.
- local\_size: Regularization local size.
- alpha: The alpha regularization parameter.
- beta: The beta regularization parameter.
- k: The k regularization parameter.

### struct primitive\_desc : public dnnl::primitive\_desc

Primitive descriptor for an LRN forward propagation primitive.

## Public Functions

```
primitive_desc()
    Default constructor. Produces an empty object.

primitive_desc(const desc &adesc, const engine &aengine, bool allow_empty = false)
    Constructs a primitive descriptor for an LRN forward propagation primitive.
```

### Parameters

- adesc: Descriptor for an LRN forward propagation primitive.
- aengine: Engine to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
primitive_desc(const desc &adesc, const primitive_attr &attr, const engine &aengine, bool allow_empty = false)
    Constructs a primitive descriptor for an LRN forward propagation primitive.
```

### Parameters

- adesc: Descriptor for an LRN forward propagation primitive.
- aengine: Engine to use.
- attr: Primitive attributes to use.

- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

`memory::desc src_desc() const`

Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter.

`memory::desc dst_desc() const`

Returns a destination memory descriptor.

**Return** Destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination parameter.

`memory::desc workspace_desc() const`

Returns the workspace memory descriptor.

**Return** Workspace memory descriptor.

**Return** A zero memory descriptor if the primitive does not require workspace parameter.

`struct dnnl::lrn_backward : public dnnl::primitive`

Local response normalization (LRN) backward propagation primitive.

## Public Functions

`lrn_backward()`

Default constructor. Produces an empty object.

`lrn_backward(const primitive_desc &pd)`

Constructs an LRN backward propagation primitive.

### Parameters

- `pd`: Primitive descriptor for an LRN backward propagation primitive.

`struct desc`

Descriptor for an LRN backward propagation primitive.

## Public Functions

`desc(algorithm aalgorithm, const memory::desc &data_desc, const memory::desc &diff_data_desc, memory::dim local_size, float alpha, float beta, float k = 1.f)`

Constructs a descriptor for an LRN backward propagation primitive.

### Parameters

- `aalgorithm`: LRN algorithm kind: either `dnnl::algorithm::lrn_across_channels`, or `dnnl::algorithm::lrn_within_channel`.
- `diff_data_desc`: Diff source and diff destination memory descriptor.
- `data_desc`: Source memory descriptor.
- `local_size`: Regularization local size.
- `alpha`: The alpha regularization parameter.
- `beta`: The beta regularization parameter.
- `k`: The k regularization parameter.

`struct primitive_desc : public dnnl::primitive_desc`

Primitive descriptor for an LRN backward propagation primitive.

## Public Functions

**primitive\_desc()**

Default constructor. Produces an empty object.

**primitive\_desc(const desc &adesc, const engine &aengine, const lrn\_forward::primitive\_desc &hint\_fwd\_pd, bool allow\_empty = false)**

Constructs a primitive descriptor for an LRN backward propagation primitive.

### Parameters

- adesc: Descriptor for an LRN backward propagation primitive.
- aengine: Engine to use.
- hint\_fwd\_pd: Primitive descriptor for an LRN forward propagation primitive. It is used as a hint for deciding which memory format to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**primitive\_desc(const desc &adesc, const primitive\_attr &attr, const engine &aengine, const lrn\_forward::primitive\_desc &hint\_fwd\_pd, bool allow\_empty = false)**

Constructs a primitive descriptor for an LRN backward propagation primitive.

### Parameters

- adesc: Descriptor for an LRN backward propagation primitive.
- attr: Primitive attributes to use.
- aengine: Engine to use.
- hint\_fwd\_pd: Primitive descriptor for an LRN forward propagation primitive. It is used as a hint for deciding which memory format to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**memory::desc diff\_src\_desc() const**

Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter.

**memory::desc diff\_dst\_desc() const**

Returns a diff destination memory descriptor.

**Return** Diff destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff destination parameter.

**memory::desc workspace\_desc() const**

Returns the workspace memory descriptor.

**Return** Workspace memory descriptor.

**Return** A zero memory descriptor if the primitive does not require workspace parameter.

## 7.5.12 Matrix Multiplication

The matrix multiplication (MatMul) primitive computes the product of two 2D tensors with optional bias addition. Variable names follow the standard *Conventions*.

$$\text{dst}(m, n) = \sum_{k=0}^K (\text{src}(m, k) \cdot \text{weights}(k, n)) + \text{bias}(m, n)$$

The MatMul primitive also supports batching multiple independent matrix multiplication operations, in which case the tensors must be 3D:

$$\text{dst}(mb, m, n) = \sum_{k=0}^K (\text{src}(mb, m, k) \cdot \text{weights}(mb, k, n)) + \text{bias}(mb, m, n)$$

The bias tensor is optional and supports implicit broadcast semantics: any of its dimensions can be 1 and the same value would be used across the corresponding dimension. However, bias must have the same number of dimensions as the dst.

### 7.5.12.1 Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	<i>DNNL_ARG_SRC</i>
weights	<i>DNNL_ARG_WEIGHTS</i>
bias	<i>DNNL_ARG_BIAS</i>
dst	<i>DNNL_ARG_DST</i>

### 7.5.12.2 Operation Details

The MatMul primitive supports input and output tensors with run-time specified shapes and memory formats. The run-time specified dimensions or strides are specified using the *DNNL\_RUNTIME\_DIM\_VAL* wildcard value during the primitive initialization and creation stage. At the execution stage, the user must pass fully specified memory objects so that the primitive is able to perform the computations. Note that the less information about shapes or format is available at the creation stage, the less performant execution will be. In particular, if the shape is not known at creation stage, one cannot use the special format tag *any* to enable an implementation to choose the most appropriate memory format for the corresponding input or output shapes. On the other hand, run-time specified shapes enable users to create a primitive once and use it in different situations.

### 7.5.12.3 Data Types Support

The MatMul primitive supports the following combinations of data types for source, destination, weights, and bias tensors.

---

**Note:** Here we abbreviate data types names for readability. For example, *dnnl::memory::data\_type::f32* is abbreviated to *f32*.

---

Source	Weights	Destination	Bias
<i>f32</i>	<i>f32</i>	<i>f32</i>	<i>f32</i>
<i>f16</i>	<i>f16</i>	<i>f16</i>	<i>f16</i>
<i>bf16</i>	<i>bf16</i>	<i>bf16</i>	<i>bf16, f32</i>
<i>u8, s8</i>	<i>s8, u8</i>	<i>u8, s8, s32, f32</i>	<i>u8, s8, s32, f32</i>

#### 7.5.12.4 Data Representation

The MatMul primitive expects the following tensors:

Dims	Source	Weights	Destination	Bias (optional)
2D	$M \times K$	$K \times N$	$M \times N$	$(M \text{ or } 1) \times (N \text{ or } 1)$
3D	$MB \times M \times K$	$MB \times K \times N$	$MB \times M \times N$	$(MB \text{ or } 1) \times (M \text{ or } 1) \times (N \text{ or } 1)$

The MatMul primitive is generally optimized for the case in which memory objects use plain memory formats (with some restrictions; see the table below). However, it is recommended to use the placeholder memory format *any* if an input tensor is reused across multiple executions. In this case, the primitive will set the most appropriate memory format for the corresponding input tensor.

The table below shows the combinations of memory formats for which the MatMul primitive is optimized. The memory format of the destination tensor should always be *ab* for the 2D case and *abc* for the 3D one.

Dims	Logical tensors	MatMul is optimized for the following memory formats
2D	Source: $M \times K$ , Weights: $K \times N$	Source: <i>ab</i> or <i>ba</i> , Weights: <i>ab</i> or <i>ba</i>
3D	Source: $MB \times M \times K$ , Weights: $MB \times K \times N$	Source: <i>abc</i> or <i>acb</i> , Weights: <i>abc</i> or <i>acb</i>

#### 7.5.12.5 Attributes and Post-ops

Attributes and post-ops enable modifying the behavior of the MatMul primitive. The following attributes and post-ops are supported:

Type	Operation	Description	Restrictions
At-tribute	<i>Output scales</i>	Scales the result by given scale factor(s)	
At-tribute	<i>Zero points</i>	Sets zero point(s) for the corresponding tensors	Int8 computations only
Post-op	<i>Eltwise</i>	Applies an elementwise operation to the result	
Post-op	<i>Sum</i>	Adds the operation result to the destination tensor instead of overwriting it	

To facilitate dynamic quantization, the primitive should support run-time output scales. That means a user could configure attributes with output scales set to the *DNNL\_RUNTIME\_F32\_VAL* wildcard value instead of the actual scales, if the scales are not known at the primitive descriptor creation stage. In this case, the user must provide the scales as an additional input memory object with argument *DNNL\_ARG\_ATTR\_OUTPUT\_SCALES* during the execution stage.

Similarly to run-time output scales, the primitive supports run-time zero points. The wildcard value for zero points is *DNNL\_RUNTIME\_S32\_VAL*. During the execution stage, the corresponding memory object needs to be passed in the argument with index set to (*DNNL\_ARG\_ATTR\_ZERO\_POINTS* | *DNNL\_ARG\_\${MEMORY}*). For instance,

source tensor zero points memory argument would be passed with index (DNNL\_ARG\_ATTR\_ZERO\_POINTS | DNNL\_ARG\_SRC).

### 7.5.12.6 API

**struct** dnnl::**matmul** : **public** dnnl::*primitive*  
Matrix multiplication (matmul) primitive.

#### Public Functions

**matmul()**

Default constructor. Produces an empty object.

**matmul(**const** *primitive\_desc* &*pd*)**

Constructs a matmul primitive.

##### Parameters

- *pd*: Primitive descriptor for a matmul primitive.

**struct desc**

Descriptor for a matmul primitive.

#### Public Functions

**desc(**const** *memory::desc* &*src\_desc*, **const** *memory::desc* &*weights\_desc*, **const** *memory::desc* &*dst\_desc*)**

Constructs a descriptor for a matmul primitive.

##### Parameters

- *src\_desc*: Memory descriptor for source (matrix A).
- *weights\_desc*: Memory descriptor for weights (matrix B).
- *dst\_desc*: Memory descriptor for destination (matrix C).

**desc(**const** *memory::desc* &*src\_desc*, **const** *memory::desc* &*weights\_desc*, **const** *memory::desc* &*bias\_desc*, **const** *memory::desc* &*dst\_desc*)**

Constructs a descriptor for a matmul primitive.

##### Parameters

- *src\_desc*: Memory descriptor for source (matrix A).
- *weights\_desc*: Memory descriptor for weights (matrix B).
- *dst\_desc*: Memory descriptor for destination (matrix C).
- *bias\_desc*: Memory descriptor for bias.

**struct primitive\_desc : public dnnl::*primitive\_desc***

Primitive descriptor for a matmul primitive.

## Public Functions

### `primitive_desc()`

Default constructor. Produces an empty object.

### `primitive_desc(const desc &adesc, const engine &aengine, bool allow_empty = false)`

Constructs a primitive descriptor for a matmul primitive.

#### Parameters

- `adesc`: Descriptor for a matmul primitive.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

### `primitive_desc(const desc &adesc, const primitive_attr &attr, const engine &aengine, bool allow_empty = false)`

Constructs a primitive descriptor for a matmul primitive.

#### Parameters

- `adesc`: Descriptor for a matmul primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

### `memory::desc src_desc() const`

Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter.

### `memory::desc weights_desc() const`

Returns a weights memory descriptor.

**Return** Weights memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a weights parameter.

### `memory::desc bias_desc() const`

Returns the bias memory descriptor.

**Return** The bias memory descriptor.

**Return** A zero memory descriptor of the primitive does not have a bias parameter.

### `memory::desc dst_desc() const`

Returns a destination memory descriptor.

**Return** Destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination parameter.

## 7.5.13 Pooling

The pooling primitive performs forward or backward max or average pooling operation on 1D, 2D, or 3D spatial data.

The pooling operation is defined by the following formulas. We show formulas only for 2D spatial data which are straightforward to generalize to cases of higher and lower dimensions. Variable names follow the standard *Conventions*.

### 7.5.13.1 Forward

Max pooling:

$$\text{dst}(n, c, oh, ow) = \max_{kh, kw} (\text{src}(n, c, oh \cdot SH + kh - PH_L, ow \cdot SW + kw - PW_L))$$

Average pooling:

$$\text{dst}(n, c, oh, ow) = \frac{1}{DENOM} \sum_{kh, kw} \text{src}(n, c, oh \cdot SH + kh - PH_L, ow \cdot SW + kw - PW_L)$$

Here output spatial dimensions are calculated similarly to how they are done for *Convolution and Deconvolution*.

Average pooling supports two algorithms:

- *pooling\_avg\_include\_padding*, in which case  $DENOM = KH \cdot KW$ ,
- *pooling\_avg\_exclude\_padding*, in which case  $DENOM$  equals to the size of overlap between an averaging window and images.

#### 7.5.13.1.1 Difference Between Forward Training and Forward Inference

Max pooling requires a workspace for the *forward\_training* propagation kind, and does not require it for *forward\_inference* (see details below).

### 7.5.13.2 Backward

The backward propagation computes  $\text{diff_src}(n, c, h, w)$ , based on  $\text{diff_dst}(n, c, h, w)$  and, in case of max pooling, workspace.

### 7.5.13.3 Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	<i>DNNL_ARG_SRC</i>
dst	<i>DNNL_ARG_DST</i>
workspace	<i>DNNL_ARG_WORKSPACE</i>
diff_src	<i>DNNL_ARG_DIFF_SRC</i>
diff_dst	<i>DNNL_ARG_DIFF_DST</i>

### 7.5.13.4 Operation Details

1. During training, max pooling requires a workspace on forward (*forward\_training*) and backward passes to save indices where a maximum was found. The workspace format is opaque, and the indices cannot be restored from it. However, one can use backward pooling to perform up-sampling (used in some detection topologies). The workspace can be created via `dnnl::pooling_forward::primitive_desc::workspace_desc()`.
2. A user can use memory format tag *any* for dst memory descriptor when creating pooling forward propagation. The library would derive the appropriate format from the `src` memory descriptor. However, the `src` itself must be defined. Similarly, a user can use memory format tag *any* for the `diff_src` memory descriptor when creating pooling backward propagation.

### 7.5.13.5 Data Type Support

The pooling primitive supports the following combinations of data types.

---

**Note:** Here we abbreviate data types names for readability. For example, `dnnl::memory::data_type::f32` is abbreviated to `f32`.

---

Propagation	Source / Destination	Accumulation data type (used for average pooling only)
forward / backward	<code>f32, bf16</code>	<code>f32</code>
forward	<code>f16</code>	<code>f16</code>
forward	<code>s8, u8, s32</code>	<code>s32</code>

### 7.5.13.6 Data Representation

#### 7.5.13.6.1 Source, Destination, and Their Gradients

Like other CNN primitives, the pooling primitive expects data to be an  $N \times C \times W$  tensor for the 1D spatial case, an  $N \times C \times H \times W$  tensor for the 2D spatial case, and an  $N \times C \times D \times H \times W$  tensor for the 3D spatial case.

The pooling primitive is optimized for the following memory formats:

Spatial	Logical tensor	Data type	Implementations optimized for memory formats
1D	NCW	f32	<code>ncw(abc), nwc(acb), optimized^</code>
1D	NCW	s32, s8, u8	<code>nwc(acb), optimized^</code>
2D	NCHW	f32	<code>nchw(abcd), nhwc(acdb), optimized^</code>
2D	NCHW	s32, s8, u8	<code>nhwc(acdb), optimized^</code>
3D	NCDHW	f32	<code>ncdhw(abcde), ndhwc(acdeb), optimized^</code>
3D	NCDHW	s32, s8, u8	<code>ndhwc(acdeb), optimized^</code>

Here `optimized^` means the format that comes out of any preceding compute-intensive primitive.

#### 7.5.13.7 Post-ops and Attributes

The pooling primitive does not support any post-ops or attributes.

#### 7.5.13.8 API

```
struct dnnl::pooling_forward : public dnnl::primitive
    Pooling forward propagation primitive.
```

## Public Functions

### `pooling_forward()`

Default constructor. Produces an empty object.

### `pooling_forward(const primitive_desc &pd)`

Constructs a pooling forward propagation primitive.

#### Parameters

- pd: Primitive descriptor for a pooling forward propagation primitive.

### `struct desc`

Descriptor for a pooling forward propagation primitive.

## Public Functions

### `desc(prop_kind aprop_kind, algorithm aalgorithm, const memory::desc &src_desc, const memory::desc &dst_desc, const memory::dims &strides, const memory::dims &kernel, const memory::dims &padding_l, const memory::dims &padding_r)`

Constructs a descriptor for pooling forward propagation primitive.

Arrays strides, kernel, padding\_l, and padding\_r contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

#### Parameters

- aprop\_kind: Propagation kind. Possible values are `dnnl::prop_kind::forward_training`, and `dnnl::prop_kind::forward_inference`.
- aalgorithm: Pooling algorithm kind: either `dnnl::algorithm::pooling_max`, `dnnl::algorithm::pooling_avg_include_padding`, or `dnnl::algorithm::pooling_avg` (same as `dnnl::algorithm::pooling_avg_exclude_padding`).
- src\_desc: Source memory descriptor.
- dst\_desc: Destination memory descriptor.
- strides: Vector of strides for spatial dimension.
- kernel: Vector of kernel spatial dimensions.
- padding\_l: Vector of padding values for low indices for each spatial dimension ([[front,] top,] left)).
- padding\_r: Vector of padding values for high indices for each spatial dimension ([ [back,] bottom,] right)).

### `struct primitive_desc : public dnnl::primitive_desc`

Primitive descriptor for a pooling forward propagation primitive.

## Public Functions

### `primitive_desc()`

Default constructor. Produces an empty object.

### `primitive_desc(const desc &adesc, const engine &aengine, bool allow_empty = false)`

Constructs a primitive descriptor for a pooling forward propagation primitive.

#### Parameters

- adesc: Descriptor for a pooling forward propagation primitive.
- aengine: Engine to use.

- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
primitive_desc (const desc &adesc, const primitive_attr &attr, const engine &aengine,  
                  bool allow_empty = false)
```

Constructs a primitive descriptor for a pooling forward propagation primitive.

#### Parameters

- *adesc*: Descriptor for a pooling forward propagation primitive.
- *aengine*: Engine to use.
- *attr*: Primitive attributes to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
memory::desc src_desc () const
```

Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter.

```
memory::desc dst_desc () const
```

Returns a destination memory descriptor.

**Return** Destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination parameter.

```
memory::desc workspace_desc () const
```

Returns the workspace memory descriptor.

**Return** Workspace memory descriptor.

**Return** A zero memory descriptor if the primitive does not require workspace parameter.

```
struct dnnl::pooling_backward : public dnnl::primitive
```

Pooling backward propagation primitive.

## Public Functions

```
pooling_backward ()
```

Default constructor. Produces an empty object.

```
pooling_backward (const primitive_desc &pd)
```

Constructs a pooling backward propagation primitive.

#### Parameters

- *pd*: Primitive descriptor for a pooling backward propagation primitive.

```
struct desc
```

Descriptor for a pooling backward propagation primitive.

## Public Functions

```
desc(algorithm aalgorithm, const memory::desc &diff_src_desc, const memory::desc &diff_dst_desc, const memory::dims &strides, const memory::dims &kernel, const memory::dims &padding_l, const memory::dims &padding_r)
```

Constructs a descriptor for pooling backward propagation primitive.

Arrays *strides*, *kernel*, *padding\_l*, and *padding\_r* contain values for spatial dimensions only and hence must have the same number of elements as there are spatial dimensions. The order of values is the same as in the tensor: depth (for 3D tensors), height (for 3D and 2D tensors), and width.

### Parameters

- *aalgorithm*: Pooling algorithm kind: either *dnnl::algorithm::pooling\_max*, *dnnl::algorithm::pooling\_avg\_include\_padding*, or *dnnl::algorithm::pooling\_avg* (same as *dnnl::algorithm::pooling\_avg\_exclude\_padding*).
- *diff\_src\_desc*: Diff source memory descriptor.
- *diff\_dst\_desc*: Diff destination memory descriptor.
- *strides*: Vector of strides for spatial dimension.
- *kernel*: Vector of kernel spatial dimensions.
- *padding\_l*: Vector of padding values for low indices for each spatial dimension ([ [front,] top,] left).
- *padding\_r*: Vector of padding values for high indices for each spatial dimension ([ [back,] bottom,] right).

```
struct primitive_desc : public dnnl::primitive_desc
```

Primitive descriptor for a pooling backward propagation primitive.

## Public Functions

```
primitive_desc()
```

Default constructor. Produces an empty object.

```
primitive_desc(const desc &adesc, const engine &aengine, const pooling_forward::primitive_desc &hint_fwd_pd, bool allow_empty = false)
```

Constructs a primitive descriptor for a pooling backward propagation primitive.

### Parameters

- *adesc*: Descriptor for a pooling backward propagation primitive.
- *aengine*: Engine to use.
- *hint\_fwd\_pd*: Primitive descriptor for a pooling forward propagation primitive. It is used as a hint for deciding which memory format to use.
- *allow\_empty*: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to *false*.

```
primitive_desc(const desc &adesc, const primitive_attr &attr, const engine &aengine, const pooling_forward::primitive_desc &hint_fwd_pd, bool allow_empty = false)
```

Constructs a primitive descriptor for a pooling backward propagation primitive.

### Parameters

- *adesc*: Descriptor for a pooling backward propagation primitive.
- *attr*: Primitive attributes to use.
- *aengine*: Engine to use.
- *hint\_fwd\_pd*: Primitive descriptor for a pooling forward propagation primitive. It is used as a hint for deciding which memory format to use.

- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

`memory::desc diff_src_desc() const`

Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter.

`memory::desc diff_dst_desc() const`

Returns a diff destination memory descriptor.

**Return** Diff destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff destination parameter.

`memory::desc workspace_desc() const`

Returns the workspace memory descriptor.

**Return** Workspace memory descriptor.

**Return** A zero memory descriptor if the primitive does not require workspace parameter.

## 7.5.14 Reorder

A primitive to copy data between two memory objects. This primitive is typically used to change the way that the data is laid out in memory.

The reorder primitive copies data between different memory formats but does not change the tensor from mathematical perspective. Variable names follow the standard [Conventions](#).

$$\text{dst}(\bar{x}) = \text{src}(\bar{x})$$

for  $\bar{x} = (x_0, \dots, x_n)$ .

As described in [Introduction](#) in order to achieve the best performance some primitives (such as convolution) require special memory format which is typically referred to as an *optimized* memory format. The *optimized* memory format may match or may not match memory format that data is currently kept in. In this case a user can use reorder primitive to copy (reorder) the data between the memory formats.

Using the attributes and post-ops users can also use reorder primitive to quantize the data (and if necessary change the memory format simultaneously).

### 7.5.14.1 Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	<code>DNNL_ARG_FROM</code>
dst	<code>DNNL_ARG_TO</code>

### 7.5.14.2 Operation Details

1. The reorder primitive requires the source and destination tensors to have the same shape. Implicit broadcasting is not supported.
2. While in most of the cases the reorder should be able to handle arbitrary source and destination memory formats and data types, it might happen than some combinations are not implemented. For instance:
  - Reorder implementations between weights in non-plain memory formats might be limited (but if encountered in real practice should be treated as a bug and reported to oneDNN team);
  - Weights in one Winograd format cannot be reordered to the weights of the other Winograd format;
  - Quantized weights for convolution with #dnnl\_s8 source data type cannot be dequantized back to the #dnnl\_f32 data type;
3. To alleviate the problem a user may rely on fact that the reorder from original plain memory format and user's data type to the *optimized* format with chosen data type should be always implemented.

### 7.5.14.3 Data Types Support

The reorder primitive supports arbitrary data types for the source and destination.

When converting the data from one data type to a smaller one saturation is used. For instance:

```
reorder(src={1024, data_type=f32}, dst={}, data_type=s8)
// dst == {127}

reorder(src={-124, data_type=f32}, dst={}, data_type=u8)
// dst == {0}
```

### 7.5.14.4 Data Representation

The reorder primitive works with arbitrary data tensors. There is no special meaning associated with any logical dimensions.

### 7.5.14.5 Post-ops and Attributes

The reorder primitive should support the following attributes and post-ops:

Type	Operation	Description	Restrictions
At-tribute	<i>Output scales</i>	Scales the result by given scale factor(s)	
Post-op	<i>Sum</i>	Adds the operation result to the destination tensor instead of overwriting it	

For instance, the following pseudo-code

```
reorder(
    src = {dims={N, C, H, W}, data_type=dt_src, memory_format=fmt_src},
    dst = {dims={N, C, H, W}, data_type=dt_dst, memory_format=fmt_dst},
    attr ={
        output_scale=alpha,
```

(continues on next page)

(continued from previous page)

```
post-ops = { sum={scale=beta} },
})
```

would lead to the following operation:

$$\text{dst}(\bar{x}) = \alpha \cdot \text{src}(\bar{x}) + \beta \cdot \text{dst}(\bar{x})$$

---

**Note:** The intermediate operations are being done using single precision floating point data type.

---

### 7.5.14.6 API

**struct dnnl::reorder : public dnnl::primitive**  
Reorder primitive.

#### Public Functions

##### **reorder()**

Default constructor. Produces an empty object.

##### **reorder(const primitive\_desc &pd)**

Constructs a reorder primitive.

###### Parameters

- pd: Primitive descriptor for reorder primitive.

##### **reorder(const memory &src, const memory &dst, const primitive\_attr &attr = primitive\_attr())**

Constructs a reorder primitive that would reorder data between memory objects having the same memory descriptors as memory objects src and dst.

###### Parameters

- src: Source memory object.
- dst: Destination memory object.
- attr: Primitive attributes to use (optional).

##### **void execute(const stream &astream, memory &src, memory &dst) const**

Executes the reorder primitive.

###### Parameters

- astream: Stream object. The stream must belong to the same engine as the primitive.
- src: Source memory object.
- dst: Destination memory object.

##### **cl::sycl::event execute\_sycl(const stream &astream, memory &src, memory &dst, const std::vector<cl::sycl::event> &deps = {}) const**

Executes the reorder primitive (SYCL-aware version)

**Return** SYCL event that corresponds to the SYCL queue underlying the astream.

## Parameters

- `astream`: Stream object. The stream must belong to the same engine as the primitive.
- `src`: Source memory object.
- `dst`: Destination memory object.
- `deps`: Vector of SYCL events that the execution should depend on.

```
struct primitive_desc : public dnnl::primitive_desc_base
```

Primitive descriptor for a reorder primitive.

## Public Functions

**primitive\_desc()**

Default constructor. Produces an empty object.

```
primitive_desc(const engine &src_engine, const memory::desc &src_md, const engine &dst_engine, const memory::desc &dst_md, const primitive_attr &attr = primitive_attr(), bool allow_empty = false)
```

Constructs a primitive descriptor for reorder primitive.

**Note** If `allow_empty` is true, the constructor does not throw if a primitive descriptor cannot be created.

### Parameters

- `src_engine`: Engine on which the source memory object will be located.
- `src_md`: Source memory descriptor.
- `dst_engine`: Engine on which the destination memory object will be located.
- `dst_md`: Destination memory descriptor.
- `attr`: Primitive attributes to use (optional).
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
primitive_desc(const memory &src, const memory &dst, const primitive_attr &attr = primitive_attr(), bool allow_empty = false)
```

Constructs a primitive descriptor for reorder primitive.

### Parameters

- `src`: Source memory object. It is used to obtain the source memory descriptor and engine.
- `dst`: Destination memory object. It is used to obtain the destination memory descriptor and engine.
- `attr`: Primitive attributes to use (optional).
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**engine get\_src\_engine() const**

Returns the engine on which the source memory is allocated.

**Return** The engine on which the source memory is allocated.

**engine get\_dst\_engine() const**

Returns the engine on which the destination memory is allocated.

**Return** The engine on which the destination memory is allocated.

**memory::desc src\_desc() const**

Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter.

`memory::desc dst_desc() const`

Returns a destination memory descriptor.

**Return** Destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination parameter.

## 7.5.15 Resampling

The resampling primitive computes forward or backward resampling operation on 1D, 2D, or 3D spatial data. Resampling performs spatial scaling of original tensor using one of the supported interpolation algorithms:

- Nearest Neighbor
- Linear (or Bilinear for 2D spatial tensor, Trilinear for 3D spatial tensor).

Resampling operation is defined by the source tensor and scaling factors in each spatial dimension. Upsampling and downsampling are the alternative terms for resampling that are used when all scaling factors are greater (upsampling) or less (downsampling) than one.

The resampling operation is defined by the following formulas. We show formulas only for 2D spatial data which are straightforward to generalize to cases of higher and lower dimensions. Variable names follow the standard [Conventions](#).

Let src and dst be  $N \times C \times IH \times IW$  and  $N \times C \times OH \times OW$  tensors respectively. Let  $F_h = \frac{OH}{IH}$  and  $F_w = \frac{OW}{IW}$  define scaling factors in each spatial dimension.

The following formulas show how oneDNN computes resampling for nearest neighbor and bilinear interpolation methods. To further simplify the formulas, we assume the following:

- $\text{src}(n, ic, ih, iw) = 0$  if  $ih < 0$  or  $iw < 0$ ,
- $\text{src}(n, ic, ih, iw) = \text{src}(n, ic, IH - 1, iw)$  if  $ih \geq IH$ ,
- $\text{src}(n, ic, ih, iw) = \text{src}(n, ic, ih, IW - 1)$  if  $iw \geq IW$ .

### 7.5.15.1 Forward

#### 7.5.15.1.1 Nearest Neighbor Resampling

$$\text{dst}(n, c, oh, ow) = \text{src}(n, c, ih, iw)$$

where

- $ih = [\frac{oh+0.5}{F_h} - 0.5]$ ,
- $iw = [\frac{ow+0.5}{F_w} - 0.5]$ .

### 7.5.15.1.2 Bilinear Resampling

$$\begin{aligned} \text{dst}(n, c, oh, ow) = & \text{src}(n, c, ih_0, iw_0) \cdot W_{ih} \cdot W_{iw} + \\ & \text{src}(n, c, ih_1, iw_0) \cdot (1 - W_{ih}) \cdot W_{iw} + \\ & \text{src}(n, c, ih_0, iw_1) \cdot W_{ih} \cdot (1 - W_{iw}) + \\ & \text{src}(n, c, ih_1, iw_1) \cdot (1 - W_{ih}) \cdot (1 - W_{iw}) \end{aligned}$$

where

- $ih_0 = \left\lfloor \frac{oh+0.5}{F_h} - 0.5 \right\rfloor$ ,
- $ih_1 = \left\lceil \frac{oh+0.5}{F_h} - 0.5 \right\rceil$ ,
- $iw_0 = \left\lfloor \frac{ow+0.5}{F_w} - 0.5 \right\rfloor$ ,
- $iw_1 = \left\lceil \frac{ow+0.5}{F_w} - 0.5 \right\rceil$ ,
- $W_{ih} = \frac{oh+0.5}{F_h} - 0.5 - ih_0$ ,
- $W_{iw} = \frac{ow+0.5}{F_w} - 0.5 - iw_0$ .

### 7.5.15.1.3 Difference Between Forward Training and Forward Inference

There is no difference between the *forward\_training* and *forward\_inference* propagation kinds.

## 7.5.15.2 Backward

The backward propagation computes diff\_src based on diff\_dst.

### 7.5.15.3 Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	<i>DNNL_ARG_SRC</i>
dst	<i>DNNL_ARG_DST</i>
diff_src	<i>DNNL_ARG_DIFF_SRC</i>
diff_dst	<i>DNNL_ARG_DIFF_DST</i>

#### 7.5.15.4 Operation Details

1. Resampling implementation supports data with arbitrary data tag (*nchw*, *nhwc*, etc.) but memory tags for *src* and *dst* are expected to be the same. Resampling primitive supports *dst* and *diff\_src* memory tag *any* and can define destination format based on source format.
2. Resampling descriptor can be created by specifying the source and destination memory descriptors, only the source descriptor and floating point factors, or the source and destination memory descriptors and factors. In case when user does not provide the destination descriptor, the destination dimensions are deduced using the factors:  $output\_spatial\_size = \left\lfloor \frac{input\_spatial\_size}{F} \right\rfloor$ .

---

**Note:** Resampling algorithm uses factors as defined by the relation  $F = \frac{output\_spatial\_size}{input\_spatial\_size}$  that do not necessarily equal to the ones passed by the user.

---

#### 7.5.15.5 Data Types Support

Resampling primitive supports the following combination of data types for source and destination memory objects.

---

**Note:** Here we abbreviate data types names for readability. For example, *dnnl::memory::data\_type::f32* is abbreviated to *f32*.

---

Propagation	Source / Destination
forward / backward	<i>f32, bf16</i>
forward	<i>f16, s8, u8</i>

#### 7.5.15.6 Post-ops and Attributes

The resampling primitive does not support any post-ops or attributes.

#### 7.5.15.7 API

```
struct dnnl::resampling_forward : public dnnl::primitive
    Resampling forward propagation.
```

##### Public Functions

```
resampling_forward()
    Default constructor. Produces an empty object.

resampling_forward(const primitive_desc &pd)
    Constructs a resampling forward propagation primitive.
```

##### Parameters

- pd: Primitive descriptor for a resampling forward propagation primitive.

```
struct desc
    Descriptor for resampling forward propagation.
```

## Public Functions

**desc** (*prop\_kind aprop\_kind, algorithm aalgorithm, const memory::desc &src\_desc, const memory::desc &dst\_desc*)

Constructs a descriptor for a resampling forward propagation primitive using source and destination memory descriptors.

**Note** The destination memory descriptor may be initialized with *dnnl::memory::format\_tag::any* value of *format\_tag*.

### Parameters

- *aprop\_kind*: Propagation kind. Possible values are *dnnl::prop\_kind::forward\_training*, and *dnnl::prop\_kind::forward\_inference*.
- *aalgorithm*: resampling algorithm kind: either *dnnl::algorithm::resampling\_nearest*, or *dnnl::algorithm::resampling\_linear*
- *src\_desc*: Source memory descriptor.
- *dst\_desc*: Destination memory descriptor.

**desc** (*prop\_kind aprop\_kind, algorithm aalgorithm, const std::vector<float> &factors, const memory::desc &src\_desc*)

Constructs a descriptor for a resampling forward propagation primitive using source memory descriptor and factors.

### Parameters

- *aprop\_kind*: Propagation kind. Possible values are *dnnl::prop\_kind::forward\_training*, and *dnnl::prop\_kind::forward\_inference*.
- *aalgorithm*: resampling algorithm kind: either *dnnl::algorithm::resampling\_nearest*, or *dnnl::algorithm::resampling\_linear*
- *factors*: Vector of scaling factors for spatial dimension.
- *src\_desc*: Source memory descriptor.

**desc** (*prop\_kind aprop\_kind, algorithm aalgorithm, const std::vector<float> &factors, const memory::desc &src\_desc, const memory::desc &dst\_desc*)

Constructs a descriptor for a resampling forward propagation primitive.

**Note** The destination memory descriptor may be initialized with *dnnl::memory::format\_tag::any* value of *format\_tag*.

### Parameters

- *aprop\_kind*: Propagation kind. Possible values are *dnnl::prop\_kind::forward\_training*, and *dnnl::prop\_kind::forward\_inference*.
- *aalgorithm*: resampling algorithm kind: either *dnnl::algorithm::resampling\_nearest*, or *dnnl::algorithm::resampling\_linear*
- *factors*: Vector of scaling factors for spatial dimension.
- *src\_desc*: Source memory descriptor.
- *dst\_desc*: Destination memory descriptor.

**struct primitive\_desc : public dnnl::primitive\_desc**

Primitive descriptor for a resampling forward propagation primitive.

## Public Functions

**primitive\_desc()**

Default constructor. Produces an empty object.

**primitive\_desc(const desc &adesc, const engine &aengine, bool allow\_empty = false)**

Constructs a primitive descriptor for a resampling forward propagation primitive.

### Parameters

- adesc: Descriptor for a resampling forward propagation primitive.
- aengine: Engine to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**primitive\_desc(const desc &adesc, const primitive\_attr &attr, const engine &aengine, bool allow\_empty = false)**

Constructs a primitive descriptor for a resampling forward propagation primitive.

### Parameters

- adesc: Descriptor for a resampling forward propagation primitive.
- aengine: Engine to use.
- attr: Primitive attributes to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**memory::desc src\_desc() const**

Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter.

**memory::desc dst\_desc() const**

Returns a destination memory descriptor.

**Return** Destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination parameter.

**struct dnnl::resampling\_backward : public dnnl::primitive**

Resampling backward propagation primitive.

## Public Functions

**resampling\_backward()**

Default constructor. Produces an empty object.

**resampling\_backward(const primitive\_desc &pd)**

Constructs a resampling backward propagation primitive.

### Parameters

- pd: Primitive descriptor for a resampling backward propagation primitive.

**struct desc**

Descriptor for a resampling backward propagation primitive.

## Public Functions

```
desc(algorithm aalgorithm, const memory::desc &diff_src_desc, const memory::desc &diff_dst_desc)
```

Constructs a descriptor for a resampling backward propagation primitive using source and destination memory descriptors.

### Parameters

- *aalgorithm*: resampling algorithm kind: either *dnnl::algorithm::resampling\_nearest*, or *dnnl::algorithm::resampling\_linear*
- *diff\_src\_desc*: Diff source memory descriptor.
- *diff\_dst\_desc*: Diff destination memory descriptor.

```
desc(algorithm aalgorithm, const std::vector<float> &factors, const memory::desc &diff_src_desc, const memory::desc &diff_dst_desc)
```

Constructs a descriptor for resampling backward propagation primitive.

### Parameters

- *aalgorithm*: resampling algorithm kind: either *dnnl::algorithm::resampling\_nearest*, or *dnnl::algorithm::resampling\_linear*
- *factors*: Vector of scaling factors for spatial dimension.
- *diff\_src\_desc*: Diff source memory descriptor.
- *diff\_dst\_desc*: Diff destination memory descriptor.

```
struct primitive_desc : public dnnl::primitive_desc
```

Primitive descriptor for resampling backward propagation primitive.

## Public Functions

```
primitive_desc()
```

Default constructor. Produces an empty object.

```
primitive_desc(const desc &adesc, const engine &aengine, const resampling_forward::primitive_desc &hint_fwd_pd, bool allow_empty = false)
```

Constructs a primitive descriptor for a resampling backward propagation primitive.

### Parameters

- *adesc*: Descriptor for a resampling backward propagation primitive.
- *aengine*: Engine to use.
- *hint\_fwd\_pd*: Primitive descriptor for a resampling forward propagation primitive. It is used as a hint for deciding which memory format to use.
- *allow\_empty*: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
primitive_desc(const desc &adesc, const primitive_attr &attr, const engine &aengine, const resampling_forward::primitive_desc &hint_fwd_pd, bool allow_empty = false)
```

Constructs a primitive descriptor for a resampling backward propagation primitive.

### Parameters

- *adesc*: Descriptor for a resampling backward propagation primitive.
- *attr*: Primitive attributes to use.
- *aengine*: Engine to use.
- *hint\_fwd\_pd*: Primitive descriptor for a resampling forward propagation primitive. It is used as a hint for deciding which memory format to use.

- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

`memory::desc diff_src_desc() const`

Returns a diff source memory descriptor.

**Return** Diff source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff source memory with.

`memory::desc diff_dst_desc() const`

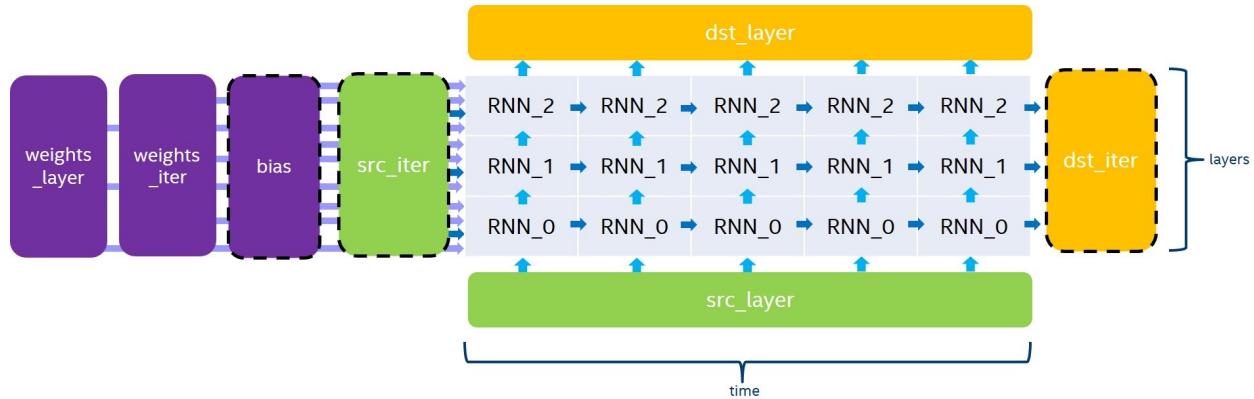
Returns a diff destination memory descriptor.

**Return** Diff destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff destination parameter.

## 7.5.16 RNN

The RNN primitive computes a stack of unrolled recurrent cells, as depicted in Figure 1. `bias`, `src_iter` and `dst_iter` are optional parameters. If not provided, `bias` and `src_iter` default to 0. Variable names follow the standard [Conventions](#).



The RNN primitive supports four modes for evaluation direction:

- `left2right` will process the input data timestamps by increasing order,
- `right2left` will process the input data timestamps by decreasing order,
- `bidirectional_concat` will process all the stacked layers from `left2right` and from `right2left` independently, and will concatenate the output in `dst_layer` over the channel dimension,
- `bidirectional_sum` will process all the stacked layers from `left2right` and from `right2left` independently, and will sum the two outputs to `dst_layer`.

Even though the RNN primitive supports passing a different number of channels for `src_layer`, `src_iter`, `dst_layer`, and `dst_iter`, we always require the following conditions in order for the dimension to be consistent:

- $\text{channels}(\text{dst\_layer}) = \text{channels}(\text{dst\_iter})$ ,
- when  $T > 1$ ,  $\text{channels}(\text{src\_iter}) = \text{channels}(\text{dst\_iter})$ ,
- when  $L > 1$ ,  $\text{channels}(\text{src\_layer}) = \text{channels}(\text{dst\_layer})$ ,
- when using the `bidirectional_concat` direction,  $\text{channels}(\text{dst\_layer}) = 2 * \text{channels}(\text{dst\_iter})$ .

The general formula for the execution of a stack of unrolled recurrent cells depends on the current iteration of the previous layer ( $h_{t,l-1}$  and  $c_{t,l-1}$ ) and the previous iteration of the current layer ( $h_{t-1,l}$ ). Here is the exact equation for

non-LSTM cells:

$$h_{t,l} = \text{Cell}(h_{t,l-1}, h_{t-1,l})$$

where  $t, l$  are the indices of the timestamp and the layer of the cell being executed.

And here is the equation for LSTM cells:

$$(h_{t,l}, c_{t,l}) = \text{Cell}(h_{t,l-1}, h_{t-1,l}, c_{t-1,l})$$

where  $t, l$  are the indices of the timestamp and the layer of the cell being executed.

### 7.5.16.1 Cell Functions

The RNN API provides four cell functions:

- *Vanilla RNN*, a single-gate recurrent cell,
- *LSTM*, a four-gate long short-term memory cell,
- *GRU*, a three-gate gated recurrent unit cell,
- *Linear-before-reset GRU*, a three-gate recurrent unit cell with the linear layer before the reset gate.

#### 7.5.16.1.1 Vanilla RNN

A single-gate recurrent cell initialized with `dnnl::vanilla_rnn_forward::desc` or `dnnl::vanilla_rnn_backward::desc` as in the following example.

```
auto vanilla_rnn_desc = dnnl::vanilla_rnn_forward::desc(
    aprop, activation, direction, src_layer_desc, src_iter_desc,
    weights_layer_desc, weights_iter_desc, bias_desc,
    dst_layer_desc, dst_iter_desc);
```

The Vanilla RNN cell should support the ReLU, Tanh and Sigmoid activation functions. The following equations defines the mathematical operation performed by the Vanilla RNN cell for the forward pass:

$$\begin{aligned} a_t &= W \cdot h_{t,l-1} + U \cdot h_{t-1,l} + B \\ h_t &= \text{activation}(a_t) \end{aligned}$$

#### 7.5.16.1.2 LSTM

##### 7.5.16.1.2.1 LSTM (or Vanilla LSTM)

A four-gate long short-term memory recurrent cell initialized with `dnnl::lstm_forward::desc` or `dnnl::lstm_backward::desc` as in the following example.

```
auto lstm_desc = dnnl::lstm_forward::desc(
    aprop, direction, src_layer_desc, src_iter_h_desc, src_iter_c_desc,
    weights_layer_desc, weights_iter_desc, bias_desc, dst_layer_desc,
    dst_iter_h_desc, dst_iter_c_desc);
```

Note that for all tensors with a dimension depending on the gates number, we implicitly require the order of these gates to be  $i$ ,  $f$ ,  $\tilde{c}$ , and  $o$ . The following equation gives the mathematical description of these gates and output for the forward pass:

$$\begin{aligned} i_t &= \sigma(W_i \cdot h_{t,l-1} + U_i \cdot h_{t-1,l} + B_i) \\ f_t &= \sigma(W_f \cdot h_{t,l-1} + U_f \cdot h_{t-1,l} + B_f) \end{aligned}$$

$$\begin{aligned} \tilde{c}_t &= \tanh(W_{\tilde{c}} \cdot h_{t,l-1} + U_{\tilde{c}} \cdot h_{t-1,l} + B_{\tilde{c}}) \\ c_t &= f_t * c_{t-1} + i_t * \tilde{c}_t \end{aligned}$$

$$\begin{aligned} o_t &= \sigma(W_o \cdot h_{t,l-1} + U_o \cdot h_{t-1,l} + B_o) \\ h_t &= \tanh(c_t) * o_t \end{aligned}$$

where  $W_*$  are stored in weights\_layer,  $U_*$  are stored in weights\_iter and  $B_*$  are stored in bias.

---

**Note:** In order for the dimensions to be consistent, we require  $\text{channels}(\text{src\_iter\_c}) = \text{channels}(\text{dst\_iter\_c}) = \text{channels}(\text{dst\_iter})$ .

---

### 7.5.16.1.2.2 LSTM with Peephole

A four-gate long short-term memory recurrent cell with peephole initialized with `dnnl::lstm_forward::desc` or `dnnl::lstm_backward::desc` as in the following example.

```
auto lstm_desc = dnnl::lstm_forward::desc(
    aprop, direction, src_layer_desc, src_iter_h_desc, src_iter_c_desc,
    weights_layer_desc, weights_iter_desc, weights_peephole_desc,
    bias_desc, dst_layer_desc, dst_iter_h_desc, dst_iter_c_desc);
```

Similarly to vanilla LSTM, we implicitly require the order of these gates to be  $i$ ,  $f$ ,  $\tilde{c}$ , and  $o$ . For peephole weights, the gates order is:  $i, f, o$ . The following equation gives the mathematical description of these gates and output for the forward pass:

$$\begin{aligned} i_t &= \sigma(W_i \cdot h_{t,l-1} + U_i \cdot h_{t-1,l} + P_i \cdot c_{t-1} + B_i) \\ f_t &= \sigma(W_f \cdot h_{t,l-1} + U_f \cdot h_{t-1,l} + P_f \cdot c_{t-1} + B_f) \end{aligned}$$

$$\begin{aligned} \tilde{c}_t &= \tanh(W_{\tilde{c}} \cdot h_{t,l-1} + U_{\tilde{c}} \cdot h_{t-1,l} + B_{\tilde{c}}) \\ c_t &= f_t * c_{t-1} + i_t * \tilde{c}_t \end{aligned}$$

$$\begin{aligned} o_t &= \sigma(W_o \cdot h_{t,l-1} + U_o \cdot h_{t-1,l} + P_o \cdot c_t + B_o) \\ h_t &= \tanh(c_t) * o_t \end{aligned}$$

where  $P_*$  are stored in weights\_peephole, and the other parameters are the same as in vanilla LSTM.

---

**Note:** If the `weights_peephole_desc` passed to the operation descriptor constructor is a zero memory descriptor, the primitive will behave the same as in LSTM primitive without peephole.

---

### 7.5.16.1.2.3 LSTM with Projection

A four-gate long short-term memory recurrent cell with projection initialized with `dnnl::lstm_forward::desc` or `dnnl::lstm_backward::desc` as in the following example.

```
auto lstm_desc = dnnl::lstm_forward::desc(
    aprop, direction, src_layer_desc, src_iter_h_desc, src_iter_c_desc,
    weights_layer_desc, weights_iter_desc, weights_peephole_desc,
    weights_projection_desc, bias_desc, dst_layer_desc, dst_iter_h_desc,
    dst_iter_c_desc);
```

Similarly to vanilla LSTM, we implicitly require the order of the gates to be  $i$ ,  $f$ ,  $\tilde{c}$ , and  $o$  for all tensors with a dimension depending on the gates. The following equation gives the mathematical description of these gates and output for the forward pass (for simplicity, LSTM without peephole is shown):

$$\begin{aligned} i_t &= \sigma(W_i \cdot h_{t,l-1} + U_i \cdot h_{t-1,l} + B_i) \\ f_t &= \sigma(W_f \cdot h_{t,l-1} + U_f \cdot h_{t-1,l} + B_f) \end{aligned}$$

$$\begin{aligned} \tilde{c}_t &= \tanh(W_{\tilde{c}} \cdot h_{t,l-1} + U_{\tilde{c}} \cdot h_{t-1,l} + B_{\tilde{c}}) \\ c_t &= f_t * c_{t-1} + i_t * \tilde{c}_t \end{aligned}$$

$$\begin{aligned} o_t &= \sigma(W_o \cdot h_{t,l-1} + U_o \cdot h_{t-1,l} + B_o) \\ h_t &= R \cdot (\tanh(c_t) * o_t) \end{aligned}$$

where  $R$  is stored in `weights_projection`, and the other parameters are the same as in vanilla LSTM.

---

**Note:** If the `weights_projection_desc` passed to the operation descriptor constructor is a zero memory descriptor, the primitive will behave the same as in LSTM primitive without projection.

---

### 7.5.16.1.3 GRU

A three-gate gated recurrent unit cell, initialized with `dnnl::gru_forward::desc` or `dnnl::gru_backward::desc` as in the following example.

```
auto gru_desc = dnnl::gru_forward::desc(
    aprop, direction, src_layer_desc, src_iter_desc,
    weights_layer_desc, weights_iter_desc, bias_desc,
    dst_layer_desc, dst_iter_desc);
```

Note that for all tensors with a dimension depending on the gates number, we implicitly require the order of these gates to be:  $u$ ,  $r$ , and  $o$ . The following equation gives the mathematical definition of these gates.

$$\begin{aligned} u_t &= \sigma(W_u \cdot h_{t,l-1} + U_u \cdot h_{t-1,l} + B_u) \\ r_t &= \sigma(W_r \cdot h_{t,l-1} + U_r \cdot h_{t-1,l} + B_r) \\ o_t &= \tanh(W_o \cdot h_{t,l-1} + U_o \cdot (r_t * h_{t-1,l}) + B_o) \\ h_t &= u_t * h_{t-1,l} + (1 - u_t) * o_t \end{aligned}$$

where  $W_*$  are in `weights_layer`,  $U_*$  are in `weights_iter`, and  $B_*$  are stored in `bias`.

---

**Note:** If you need to replace  $u_t$  by  $(1 - u_t)$  when computing  $h_t$ , you can achieve this by multiplying  $W_u$ ,  $U_u$  and  $B_u$  by  $-1$ . This is possible as  $u_t = \sigma(W_u \cdot h_{t,l-1} + U_u \cdot h_{t-1,l} + B_u)$ , and  $1^\circ \sigma(a) = \sigma(-a)$ .

---

#### 7.5.16.1.4 Linear-Before-Reset GRU

A three-gate gated recurrent unit cell with linear layer applied before the reset gate, initialized with `dnnl::lbr_gru_forward::desc` or `dnnl::lbr_gru_backward::desc` as in the following example.

```
auto lbr_gru_desc = dnnl::lbr_gru_forward::desc(
    aprop, direction, src_layer_desc, src_iter_desc,
    weights_layer_desc, weights_iter_desc, bias_desc,
    dst_layer_desc, dst_iter_desc);
```

The following equation describes the mathematical behavior of the Linear-Before-Reset GRU cell.

$$\begin{aligned} u_t &= \sigma(W_u \cdot h_{t,l-1} + U_u \cdot h_{t-1,l} + B_u) \\ r_t &= \sigma(W_r \cdot h_{t,l-1} + U_r \cdot h_{t-1,l} + B_r) \\ o_t &= \tanh(W_o \cdot h_{t,l-1} + r_t * (U_o \cdot h_{t-1,l} + B_{o'}) + B_o) \\ h_t &= u_t * h_{t-1,l} + (1 - u_t) * o_t \end{aligned}$$

Note that for all tensors with a dimension depending on the gates number, except the bias, we implicitly require the order of these gates to be  $u$ ,  $r$ , and  $o$ . For the bias tensor, we implicitly require the order of the gates to be  $u$ ,  $r$ ,  $o$ , and  $u'$ .

---

**Note:** If you need to replace  $u_t$  by  $(1 - u_t)$  when computing  $h_t$ , you can achieve this by multiplying  $W_u$ ,  $U_u$  and  $B_u$  by  $-1$ . This is possible as  $u_t = \sigma(W_u \cdot h_{t,l-1} + U_u \cdot h_{t-1,l} + B_u)$ , and  $\neg\sigma(a) = \sigma(-a)$ .

---

#### 7.5.16.2 Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src_layer	<i>DNNL_ARG_SRC_LAYER</i>
src_iter	<i>DNNL_ARG_SRC_ITER</i>
src_iter_c	<i>DNNL_ARG_SRC_ITER_C</i>
weights_layer	<i>DNNL_ARG_WEIGHTS_LAYER</i>
weights_iter	<i>DNNL_ARG_WEIGHTS_ITER</i>
weights_peephole	<i>DNNL_ARG_WEIGHTS_PEEPHOLE</i>
weights_projection	<i>DNNL_ARG_WEIGHTS_PROJECTION</i>
bias	<i>DNNL_ARG_BIAS</i>
dst_layer	<i>DNNL_ARG_DST_LAYER</i>
dst_iter	<i>DNNL_ARG_DST_ITER</i>
dst_iter_c	<i>DNNL_ARG_DST_ITER_C</i>
workspace	<i>DNNL_ARG_WORKSPACE</i>
diff_src_layer	<i>DNNL_ARG_DIFF_SRC_LAYER</i>
diff_src_iter	<i>DNNL_ARG_DIFF_SRC_ITER</i>
diff_src_iter_c	<i>DNNL_ARG_DIFF_SRC_ITER_C</i>
diff_weights_layer	<i>DNNL_ARG_DIFF_WEIGHTS_LAYER</i>
diff_weights_iter	<i>DNNL_ARG_DIFF_WEIGHTS_ITER</i>
diff_weights_peephole	<i>DNNL_ARG_DIFF_WEIGHTS_PEEPHOLE</i>
diff_weights_projection	<i>DNNL_ARG_DIFF_WEIGHTS_PROJECTION</i>
diff_bias	<i>DNNL_ARG_DIFF_BIAS</i>
diff_dst_layer	<i>DNNL_ARG_DIFF_DST_LAYER</i>
diff_dst_iter	<i>DNNL_ARG_DIFF_DST_ITER</i>
diff_dst_iter_c	<i>DNNL_ARG_DIFF_DST_ITER_C</i>

### 7.5.16.3 Operation Details

N/A

### 7.5.16.4 Data Types Support

The following table lists the combination of data types that should be supported by the RNN primitive for each input and output memory object.

---

**Note:** Here we abbreviate data types names for readability. For example, *dnnl::memory::data\_type::f32* is abbreviated to *f32*.

---

Propagation	Cell Function	Input Data	Recurrent Data (1)	Weights	Bias	Output Data
Forward / Backward	All	<i>f32</i>	<i>f32</i>	<i>f32</i>	<i>f32</i>	<i>f32</i>
Forward / Backward (2)	All (3)	<i>bf16</i>	<i>bf16</i>	<i>bf16</i>	<i>f32</i>	<i>bf16</i>
Forward	All (3)	<i>f16</i>	<i>f16</i>	<i>f16</i>	<i>f16</i>	<i>f16</i>
Forward inference	Vanilla LSTM	<i>u8</i>	<i>u8</i>	<i>s8</i>	<i>f32</i>	<i>u8, f32</i>

(1) With LSTM and Peephole LSTM cells, the cell state data type is always f32.

(2) In backward propagation, all *diff\_\** tensors are in f32.

(3) Projection LSTM is not defined yet.

#### 7.5.16.4.1 Data Representation

In the oneDNN programming model, the RNN primitive is one of a few that support the placeholder memory format `#dnnl::memory::format_tag::any` (shortened to `any` from now on) and can define data and weight memory objects format based on the primitive parameters.

The following table summarizes the data layouts supported by the RNN primitive.

Input/Output Data	Recurrent Data	Layer and Iteration Weights	Peephole Weights and Bias	Projection LSTM Weights
<code>any</code>	<code>any</code>	<code>any</code>	<code>ldgo</code>	<code>any, ldio</code> (Forward propagation)
<code>ntc, tnc</code>	<code>ldnc</code>	<code>ldigo, ldgoi</code>	<code>ldgo</code>	<code>any, ldio</code> (Forward propagation)

While an RNN primitive can be created with memory formats specified explicitly, the performance is likely to be sub-optimal. When using `any` it is necessary to first create an RNN primitive descriptor and then query it for the actual data and weight memory objects formats.

---

**Note:** The RNN primitive should support padded tensors and views. So even if two memory descriptors share the same data layout, they might still be different.

---

#### 7.5.16.4.2 Post-ops and Attributes

Currently post-ops and attributes are only used by the int8 variant of LSTM.

#### 7.5.16.5 API

**enum dnnl::rnn\_flags**

RNN cell flags.

*Values:*

**enumerator undef**

Undefined RNN flags.

**enum dnnl::rnn\_direction**

A direction of RNN primitive execution.

*Values:*

**enumerator unidirectional\_left2right**

Unidirectional execution of RNN primitive from left to right.

**enumerator unidirectional\_right2left**

Unidirectional execution of RNN primitive from right to left.

**enumerator bidirectional\_concat**

Bidirectional execution of RNN primitive with concatenation of the results.

**enumerator bidirectional\_sum**

Bidirectional execution of RNN primitive with summation of the results.

**enumerator unidirectional = unidirectional\_left2right**

Alias for `dnnl::rnn_direction::unidirectional_left2right`.

---

```
struct dnnl::vanilla_rnn_forward : public dnnl::primitive
    Vanilla RNN forward propagation primitive.
```

## Public Functions

**vanilla\_rnn\_forward()**

Default constructor. Produces an empty object.

**vanilla\_rnn\_forward(const primitive\_desc &pd)**

Constructs a vanilla RNN forward propagation primitive.

### Parameters

- pd: Primitive descriptor for a vanilla RNN forward propagation primitive.

**struct desc**

Descriptor for a vanilla RNN forward propagation primitive.

## Public Functions

```
desc(prop_kind aprop_kind, algorithm activation, rnn_direction direction, const memory::desc &src_layer_desc, const memory::desc &src_iter_desc, const memory::desc &weights_layer_desc, const memory::desc &weights_iter_desc, const memory::desc &bias_desc, const memory::desc &dst_layer_desc, const memory::desc &dst_iter_desc, rnn_flags flags = rnn_flags::undef, float alpha = 0.0f, float beta = 0.0f)
```

Constructs a descriptor for a vanilla RNN forward propagation primitive.

The following arguments may point to a zero memory descriptor:

- src\_iter\_desc,
- bias\_desc,
- dst\_iter\_desc.

This would then indicate that the RNN forward propagation primitive should not use them and should default to zero values instead.

**Note** All memory descriptors except src\_iter\_desc can be initialized with an *dnnl::memory::format\_tag::any* value of format\_tag.

### Parameters

- aprop\_kind: Propagation kind. Possible values are *dnnl::prop\_kind::forward\_training*, and *dnnl::prop\_kind::forward\_inference*.
- activation: Activation kind. Possible values are *dnnl::algorithm::eltwise\_relu*, *dnnl::algorithm::eltwise\_tanh*, or *dnnl::algorithm::eltwise\_logistic*.
- direction: RNN direction. See *dnnl::rnn\_direction* for more info.
- src\_layer\_desc: Memory descriptor for the input vector.
- src\_iter\_desc: Memory descriptor for the input recurrent hidden state vector.
- weights\_layer\_desc: Memory descriptor for the weights applied to the layer input.
- weights\_iter\_desc: Memory descriptor for the weights applied to the recurrent input.
- bias\_desc: Bias memory descriptor.
- dst\_layer\_desc: Memory descriptor for the output vector.
- dst\_iter\_desc: Memory descriptor for the output recurrent hidden state vector.
- flags: Unused.
- alpha: Negative slope if activation is *dnnl::algorithm::eltwise\_relu*.
- beta: Unused.

**struct primitive\_desc : public dnnl::rnn\_primitive\_desc\_base**

Primitive descriptor for a vanilla RNN forward propagation primitive.

## Public Functions

**primitive\_desc()**

Default constructor. Produces an empty object.

**primitive\_desc(const desc &adesc, const engine &aengine, bool allow\_empty = false)**

Constructs a primitive descriptor for a vanilla RNN forward propagation primitive.

### Parameters

- adesc: Descriptor for a vanilla RNN forward propagation primitive.
- aengine: Engine to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**primitive\_desc(const desc &adesc, const primitive\_attr &attr, const engine &aengine, bool allow\_empty = false)**

Constructs a primitive descriptor for a vanilla RNN forward propagation primitive.

### Parameters

- adesc: Descriptor for a vanilla RNN forward propagation primitive.
- attr: Primitive attributes to use.
- aengine: Engine to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**memory::desc src\_layer\_desc() const**

Returns source layer memory descriptor.

**Return** Source layer memory descriptor.

**memory::desc src\_iter\_desc() const**

Returns source iteration memory descriptor.

**Return** Source iteration memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source iteration parameter.

**memory::desc weights\_layer\_desc() const**

Returns weights layer memory descriptor.

**Return** Weights layer memory descriptor.

**memory::desc weights\_iter\_desc() const**

Returns weights iteration memory descriptor.

**Return** Weights iteration memory descriptor.

**memory::desc bias\_desc() const**

Returns bias memory descriptor.

**Return** Bias memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a bias parameter.

**memory::desc dst\_layer\_desc() const**

Returns destination layer memory descriptor.

**Return** Destination layer memory descriptor.

**memory::desc dst\_iter\_desc() const**

Returns destination iteration memory descriptor.

**Return** Destination iteration memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination iteration parameter.

**memory::desc workspace\_desc() const**

Returns the workspace memory descriptor.

**Return** Workspace memory descriptor.

**Return** A zero memory descriptor if the primitive does not require workspace parameter.

```
struct dnnl::vanilla_rnn_backward : public dnnl::primitive
```

Vanilla RNN backward propagation primitive.

## Public Functions

```
vanilla_rnn_backward()
```

Default constructor. Produces an empty object.

```
vanilla_rnn_backward(const primitive_desc &pd)
```

Constructs a vanilla RNN backward propagation primitive.

### Parameters

- pd: Primitive descriptor for a vanilla RNN backward propagation primitive.

```
struct desc
```

Descriptor for a vanilla RNN backward propagation primitive.

## Public Functions

```
desc(prop_kind aprop_kind, algorithm activation, rnn_direction direction, const memory::desc &src_layer_desc, const memory::desc &src_iter_desc, const memory::desc &weights_layer_desc, const memory::desc &weights_iter_desc, const memory::desc &bias_desc, const memory::desc &dst_layer_desc, const memory::desc &dst_iter_desc, const memory::desc &diff_src_layer_desc, const memory::desc &diff_src_iter_desc, const memory::desc &diff_weights_layer_desc, const memory::desc &diff_weights_iter_desc, const memory::desc &diff_bias_desc, const memory::desc &diff_dst_layer_desc, const memory::desc &diff_dst_iter_desc, rnn_flags flags = rnn_flags::undef, float alpha = 0.0f, float beta = 0.0f)
```

Constructs a descriptor for a vanilla RNN backward propagation primitive.

The following arguments may point to a zero memory descriptor:

- src\_iter\_desc together with diff\_src\_iter\_desc,
- bias\_desc together with diff\_bias\_desc,
- dst\_iter\_desc together with diff\_dst\_iter\_desc.

This would then indicate that the RNN backward propagation primitive should not use the respective data and should use zero values instead.

**Note** All the memory descriptors may be initialized with the `dnnl::memory::format_tag::any` value of `format_tag`.

### Parameters

- aprop\_kind: Propagation kind. Must be `dnnl::prop_kind::backward`.
- activation: Activation kind. Possible values are `dnnl::algorithm::eltwise_relu`, `dnnl::algorithm::eltwise_tanh`, or `dnnl::algorithm::eltwise_logistic`.
- direction: RNN direction. See `dnnl::rnn_direction` for more info.
- src\_layer\_desc: Memory descriptor for the input vector.
- src\_iter\_desc: Memory descriptor for the input recurrent hidden state vector.
- weights\_layer\_desc: Memory descriptor for the weights applied to the layer input.
- weights\_iter\_desc: Memory descriptor for the weights applied to the recurrent input.
- bias\_desc: Bias memory descriptor.
- dst\_layer\_desc: Memory descriptor for the output vector.
- dst\_iter\_desc: Memory descriptor for the output recurrent hidden state vector.
- diff\_src\_layer\_desc: Memory descriptor for the diff of input vector.

- `diff_src_iter_desc`: Memory descriptor for the diff of input recurrent hidden state vector.
- `diff_weights_layer_desc`: Memory descriptor for the diff of weights applied to the layer input.
- `diff_weights_iter_desc`: Memory descriptor for the diff of weights applied to the recurrent input.
- `diff_bias_desc`: Diff bias memory descriptor.
- `diff_dst_layer_desc`: Memory descriptor for the diff of output vector.
- `diff_dst_iter_desc`: Memory descriptor for the diff of output recurrent hidden state vector.
- `flags`: Unused.
- `alpha`: Negative slope if activation is `dnnl::algorithm::eltwise_relu`.
- `beta`: Unused.

```
struct primitive_desc : public dnnl::rnn_primitive_desc_base
```

Primitive descriptor for an RNN backward propagation primitive.

## Public Functions

**primitive\_desc()**

Default constructor. Produces an empty object.

**primitive\_desc(const desc &adesc, const engine &aengine, const vanilla\_rnn\_forward::primitive\_desc &hint\_fwd\_pd, bool allow\_empty = false)**

Constructs a primitive descriptor for a vanilla RNN backward propagation primitive.

### Parameters

- `adesc`: Descriptor for a vanilla RNN backward propagation primitive.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for a vanilla RNN forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**primitive\_desc(const desc &adesc, const primitive\_attr &attr, const engine &aengine, const vanilla\_rnn\_forward::primitive\_desc &hint\_fwd\_pd, bool allow\_empty = false)**

Constructs a primitive descriptor for a vanilla RNN backward propagation primitive.

### Parameters

- `adesc`: Descriptor for a vanilla RNN backward propagation primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for a vanilla RNN forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**memory::desc src\_layer\_desc() const**

Returns source layer memory descriptor.

**Return** Source layer memory descriptor.

**memory::desc src\_iter\_desc() const**

Returns source iteration memory descriptor.

**Return** Source iteration memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source iteration parameter.

*memory::desc weights\_layer\_desc() const*

Returns weights layer memory descriptor.

**Return** Weights layer memory descriptor.

*memory::desc weights\_iter\_desc() const*

Returns weights iteration memory descriptor.

**Return** Weights iteration memory descriptor.

*memory::desc bias\_desc() const*

Returns bias memory descriptor.

**Return** Bias memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a bias parameter.

*memory::desc dst\_layer\_desc() const*

Returns destination layer memory descriptor.

**Return** Destination layer memory descriptor.

*memory::desc dst\_iter\_desc() const*

Returns destination iteration memory descriptor.

**Return** Destination iteration memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination iteration parameter.

*memory::desc workspace\_desc() const*

Returns the workspace memory descriptor.

**Return** Workspace memory descriptor.

**Return** A zero memory descriptor if the primitive does not require workspace parameter.

*memory::desc diff\_src\_layer\_desc() const*

Returns diff source layer memory descriptor.

**Return** Diff source layer memory descriptor.

*memory::desc diff\_src\_iter\_desc() const*

Returns diff source iteration memory descriptor.

**Return** Diff source iteration memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff source iteration parameter.

*memory::desc diff\_weights\_layer\_desc() const*

Returns diff weights layer memory descriptor.

**Return** Diff weights layer memory descriptor.

*memory::desc diff\_weights\_iter\_desc() const*

Returns diff weights iteration memory descriptor.

**Return** Diff weights iteration memory descriptor.

*memory::desc diff\_bias\_desc() const*

Returns diff bias memory descriptor.

**Return** Diff bias memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff bias parameter.

*memory::desc diff\_dst\_layer\_desc() const*

Returns diff destination layer memory descriptor.

**Return** Diff destination layer memory descriptor.

*memory::desc diff\_dst\_iter\_desc() const*

Returns diff destination iteration memory descriptor.

**Return** Diff destination iteration memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff destination iteration parameter.

---

```
struct dnnl::lstm_forward:public dnnl::primitive
    LSTM forward propagation primitive.
```

## Public Functions

```
lstm_forward()
    Default constructor. Produces an empty object.

lstm_forward(const primitive_desc &pd)
    Constructs an LSTM forward propagation primitive.
```

### Parameters

- pd: Primitive descriptor for an LSTM forward propagation primitive.

```
struct desc
    Descriptor for an LSTM forward propagation primitive.
```

## Public Functions

```
desc(prop_kind aprop_kind, rnn_direction direction, const memory::desc &src_layer_desc,
     const memory::desc &src_iter_desc, const memory::desc &src_iter_c_desc, const
     memory::desc &weights_layer_desc, const memory::desc &weights_iter_desc, const
     memory::desc &weights_peephole_desc, const memory::desc &weights_projection_desc,
     const memory::desc &bias_desc, const memory::desc &dst_layer_desc, const mem-
     ory::desc &dst_iter_desc, const memory::desc &dst_iter_c_desc, rnn_flags flags =
     rnn_flags::undef)
Constructs a descriptor for an LSTM (with or without peephole and with or without projection) for-
ward propagation primitive.
```

The following arguments may point to a zero memory descriptor:

- src\_iter\_desc together with src\_iter\_c\_desc,
- weights\_peephole\_desc,
- bias\_desc,
- dst\_iter\_desc together with dst\_iter\_c\_desc.

This would then indicate that the LSTM forward propagation primitive should not use them and should default to zero values instead.

The weights\_projection\_desc may point to a zero memory descriptor. This would then indicate that the LSTM doesn't have recurrent projection layer.

**Note** All memory descriptors can be initialized with an *dnnl::memory::format\_tag::any* value of format\_tag.

### Parameters

- aprop\_kind: Propagation kind. Possible values are *dnnl::prop\_kind::forward\_training*, and *dnnl::prop\_kind::forward\_inference*.
- direction: RNN direction. See *dnnl::rnn\_direction* for more info.
- src\_layer\_desc: Memory descriptor for the input vector.
- src\_iter\_desc: Memory descriptor for the input recurrent hidden state vector.
- src\_iter\_c\_desc: Memory descriptor for the input recurrent cell state vector.
- weights\_layer\_desc: Memory descriptor for the weights applied to the layer input.
- weights\_iter\_desc: Memory descriptor for the weights applied to the recurrent input.
- weights\_peephole\_desc: Memory descriptor for the weights applied to the cell states (according to the Peephole LSTM formula).
- weights\_projection\_desc: Memory descriptor for the weights applied to the hidden states to get the recurrent projection (according to the Projection LSTM formula).

- bias\_desc: Bias memory descriptor.
- dst\_layer\_desc: Memory descriptor for the output vector.
- dst\_iter\_desc: Memory descriptor for the output recurrent hidden state vector.
- dst\_iter\_c\_desc: Memory descriptor for the output recurrent cell state vector.
- flags: Unused.

```
desc (prop_kind aprop_kind, rnn_direction direction, const memory::desc &src_layer_desc,
      const memory::desc &src_iter_desc, const memory::desc &src_iter_c_desc, const
      memory::desc &weights_layer_desc, const memory::desc &weights_iter_desc, const
      memory::desc &weights_peephole_desc, const memory::desc &bias_desc, const mem-
      ory::desc &dst_layer_desc, const memory::desc &dst_iter_desc, const memory::desc
      &dst_iter_c_desc, rnn_flags flags = rnn_flags::undef)
```

Constructs a descriptor for an LSTM (with or without peephole) forward propagation primitive.

The following arguments may point to a zero memory descriptor:

- src\_iter\_desc together with src\_iter\_c\_desc,
- weights\_peephole\_desc,
- bias\_desc,
- dst\_iter\_desc together with dst\_iter\_c\_desc.

This would then indicate that the LSTM forward propagation primitive should not use them and should default to zero values instead.

**Note** All memory descriptors can be initialized with an *dnnl::memory::format\_tag::any* value of *format\_tag*.

#### Parameters

- aprop\_kind: Propagation kind. Possible values are *dnnl::prop\_kind::forward\_training*, and *dnnl::prop\_kind::forward\_inference*.
- direction: RNN direction. See *dnnl::rnn\_direction* for more info.
- src\_layer\_desc: Memory descriptor for the input vector.
- src\_iter\_desc: Memory descriptor for the input recurrent hidden state vector.
- src\_iter\_c\_desc: Memory descriptor for the input recurrent cell state vector.
- weights\_layer\_desc: Memory descriptor for the weights applied to the layer input.
- weights\_iter\_desc: Memory descriptor for the weights applied to the recurrent input.
- weights\_peephole\_desc: Memory descriptor for the weights applied to the cell states (according to the Peephole LSTM formula).
- bias\_desc: Bias memory descriptor.
- dst\_layer\_desc: Memory descriptor for the output vector.
- dst\_iter\_desc: Memory descriptor for the output recurrent hidden state vector.
- dst\_iter\_c\_desc: Memory descriptor for the output recurrent cell state vector.
- flags: Unused.

```
desc (prop_kind aprop_kind, rnn_direction direction, const memory::desc &src_layer_desc,
      const memory::desc &src_iter_desc, const memory::desc &src_iter_c_desc, const
      memory::desc &weights_layer_desc, const memory::desc &weights_iter_desc, const
      memory::desc &bias_desc, const memory::desc &dst_layer_desc, const memory::desc
      &dst_iter_desc, const memory::desc &dst_iter_c_desc, rnn_flags flags = rnn_flags::undef)
```

Constructs a descriptor for an LSTM forward propagation primitive.

The following arguments may point to a zero memory descriptor:

- src\_iter\_desc together with src\_iter\_c\_desc,
- bias\_desc,
- dst\_iter\_desc together with dst\_iter\_c\_desc.

This would then indicate that the LSTM forward propagation primitive should not use them and should default to zero values instead.

**Note** All memory descriptors can be initialized with an *dnnl::memory::format\_tag::any* value of *format\_tag*.

## Parameters

- `aprop_kind`: Propagation kind. Possible values are `dnnl::prop_kind::forward_training`, and `dnnl::prop_kind::forward_inference`.
- `direction`: RNN direction. See `dnnl::rnn_direction` for more info.
- `src_layer_desc`: Memory descriptor for the input vector.
- `src_iter_desc`: Memory descriptor for the input recurrent hidden state vector.
- `src_iter_c_desc`: Memory descriptor for the input recurrent cell state vector.
- `weights_layer_desc`: Memory descriptor for the weights applied to the layer input.
- `weights_iter_desc`: Memory descriptor for the weights applied to the recurrent input.
- `bias_desc`: Bias memory descriptor.
- `dst_layer_desc`: Memory descriptor for the output vector.
- `dst_iter_desc`: Memory descriptor for the output recurrent hidden state vector.
- `dst_iter_c_desc`: Memory descriptor for the output recurrent cell state vector.
- `flags`: Unused.

`struct primitive_desc : public dnnl::rnn_primitive_desc_base`

Primitive descriptor for an LSTM forward propagation primitive.

## Public Functions

`primitive_desc()`

Default constructor. Produces an empty object.

`primitive_desc(const desc &adesc, const engine &aengine, bool allow_empty = false)`

Constructs a primitive descriptor for an LSTM forward propagation primitive.

### Parameters

- `adesc`: Descriptor for an LSTM forward propagation primitive.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

`primitive_desc(const desc &adesc, const primitive_attr &attr, const engine &aengine, bool allow_empty = false)`

Constructs a primitive descriptor for an LSTM forward propagation primitive.

### Parameters

- `adesc`: Descriptor for an LSTM forward propagation primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

`memory::desc src_layer_desc() const`

Returns source layer memory descriptor.

**Return** Source layer memory descriptor.

`memory::desc src_iter_desc() const`

Returns source iteration memory descriptor.

**Return** Source iteration memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source iteration parameter.

`memory::desc src_iter_c_desc() const`

Returns source iteration memory descriptor.

**Return** Source iteration memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source iteration parameter.

*memory::desc weights\_layer\_desc() const*

Returns weights layer memory descriptor.

**Return** Weights layer memory descriptor.

*memory::desc weights\_iter\_desc() const*

Returns weights iteration memory descriptor.

**Return** Weights iteration memory descriptor.

*memory::desc weights\_peephole\_desc() const*

Returns weights peephole memory descriptor.

**Return** Weights peephole memory descriptor.

*memory::desc weights\_projection\_desc() const*

Returns weights projection memory descriptor.

**Return** Weights projection memory descriptor.

*memory::desc bias\_desc() const*

Returns bias memory descriptor.

**Return** Bias memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a bias parameter.

*memory::desc dst\_layer\_desc() const*

Returns destination layer memory descriptor.

**Return** Destination layer memory descriptor.

*memory::desc dst\_iter\_desc() const*

Returns destination iteration memory descriptor.

**Return** Destination iteration memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination iteration parameter.

*memory::desc dst\_iter\_c\_desc() const*

Returns source iteration memory descriptor.

**Return** Source iteration memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source iteration parameter.

*memory::desc workspace\_desc() const*

Returns the workspace memory descriptor.

**Return** Workspace memory descriptor.

**Return** A zero memory descriptor if the primitive does not require workspace parameter.

**struct** dnnl::lstm\_backward : **public** dnnl::primitive

LSTM backward propagation primitive.

## Public Functions

**lstm\_backward()**

Default constructor. Produces an empty object.

**lstm\_backward(const primitive\_desc &pd)**

Constructs an LSTM backward propagation primitive.

### Parameters

- pd: Primitive descriptor for an LSTM backward propagation primitive.

**struct desc**

Descriptor for an LSTM backward propagation primitive.

## Public Functions

```
desc(prop_kind aprop_kind, rnn_direction direction, const memory::desc &src_layer_desc,
      const memory::desc &src_iter_desc, const memory::desc &src_iter_c_desc, const
      memory::desc &weights_layer_desc, const memory::desc &weights_iter_desc, const
      memory::desc &weights_peephole_desc, const memory::desc &weights_projection_desc,
      const memory::desc &bias_desc, const memory::desc &dst_layer_desc, const mem-
      ory::desc &dst_iter_desc, const memory::desc &dst_iter_c_desc, const memory::desc
      &diff_src_layer_desc, const memory::desc &diff_src_iter_desc, const memory::desc
      &diff_src_iter_c_desc, const memory::desc &diff_weights_layer_desc, const mem-
      ory::desc &diff_weights_iter_desc, const memory::desc &diff_weights_peephole_desc,
      const memory::desc &diff_weights_projection_desc, const memory::desc
      &diff_bias_desc, const memory::desc &diff_dst_layer_desc, const memory::desc
      &diff_dst_iter_desc, const memory::desc &diff_dst_iter_c_desc, rnn_flags flags =
      rnn_flags::undef)
```

projection) descriptor for backward propagation using *prop\_kind*, *direction*, and memory descriptors.

The following arguments may point to a zero memory descriptor:

- *src\_iter\_desc* together with *src\_iter\_c\_desc*, *diff\_src\_iter\_desc*, and *diff\_src\_iter\_c\_desc*,
- *weights\_peephole\_desc* together with *diff\_weights\_peephole\_desc*
- *bias\_desc* together with *diff\_bias\_desc*,
- *dst\_iter\_desc* together with *dst\_iter\_c\_desc*, *diff\_dst\_iter\_desc*, and *diff\_dst\_iter\_c\_desc*.

This would then indicate that the LSTM backward propagation primitive should not use them and should default to zero values instead.

The *weights\_projection\_desc* together with *diff\_weights\_projection\_desc* may point to a zero memory descriptor. This would then indicate that the LSTM doesn't have recurrent projection layer.

**Note** All memory descriptors can be initialized with *dnnl::memory::format\_tag::any* value of *format\_tag*.

### Parameters

- *aprop\_kind*: Propagation kind. Must be *dnnl::prop\_kind::backward*.
- *direction*: RNN direction. See *dnnl::rnn\_direction* for more info.
- *src\_layer\_desc*: Memory descriptor for the input vector.
- *src\_iter\_desc*: Memory descriptor for the input recurrent hidden state vector.
- *src\_iter\_c\_desc*: Memory descriptor for the input recurrent cell state vector.
- *weights\_layer\_desc*: Memory descriptor for the weights applied to the layer input.
- *weights\_iter\_desc*: Memory descriptor for the weights applied to the recurrent input.
- *weights\_peephole\_desc*: Memory descriptor for the weights applied to the cell states (according to the Peephole LSTM formula).
- *weights\_projection\_desc*: Memory descriptor for the weights applied to the hidden states to get the recurrent projection (according to the Projection LSTM formula).
- *bias\_desc*: Bias memory descriptor.
- *dst\_layer\_desc*: Memory descriptor for the output vector.
- *dst\_iter\_desc*: Memory descriptor for the output recurrent hidden state vector.
- *dst\_iter\_c\_desc*: Memory descriptor for the output recurrent cell state vector.
- *diff\_src\_layer\_desc*: Memory descriptor for the diff of input vector.
- *diff\_src\_iter\_desc*: Memory descriptor for the diff of input recurrent hidden state vector.
- *diff\_src\_iter\_c\_desc*: Memory descriptor for the diff of input recurrent cell state vector.

- `diff_weights_layer_desc`: Memory descriptor for the diff of weights applied to the layer input.
- `diff_weights_iter_desc`: Memory descriptor for the diff of weights applied to the recurrent input.
- `diff_weights_peephole_desc`: Memory descriptor for the diff of weights applied to the cell states (according to the Peephole LSTM formula).
- `diff_weights_projection_desc`: Memory descriptor for the diff of weights applied to the hidden states to get the recurrent projection (according to the Projection LSTM formula).
- `diff_bias_desc`: Diff bias memory descriptor.
- `diff_dst_layer_desc`: Memory descriptor for the diff of output vector.
- `diff_dst_iter_desc`: Memory descriptor for the diff of output recurrent hidden state vector.
- `diff_dst_iter_c_desc`: Memory descriptor for the diff of output recurrent cell state vector.
- `flags`: Unused.

```
desc(prop_kind aprop_kind, rnn_direction direction, const memory::desc &src_layer_desc,
     const memory::desc &src_iter_desc, const memory::desc &src_iter_c_desc, const
     memory::desc &weights_layer_desc, const memory::desc &weights_iter_desc, const
     memory::desc &weights_peephole_desc, const memory::desc &bias_desc, const mem-
     ory::desc &dst_layer_desc, const memory::desc &dst_iter_desc, const memory::desc
     &dst_iter_c_desc, const memory::desc &diff_src_layer_desc, const memory::desc
     &diff_src_iter_desc, const memory::desc &diff_src_iter_c_desc, const memory::desc
     &diff_weights_layer_desc, const memory::desc &diff_weights_iter_desc, const mem-
     ory::desc &diff_weights_peephole_desc, const memory::desc &diff_bias_desc, const
     memory::desc &diff_dst_layer_desc, const memory::desc &diff_dst_iter_desc, const
     memory::desc &diff_dst_iter_c_desc, rnn_flags flags = rnn_flags::undef)
```

Constructs an LSTM (with or without peephole) descriptor for backward propagation using `prop_kind`, `direction`, and memory descriptors.

The following arguments may point to a zero memory descriptor:

- `src_iter_desc` together with `src_iter_c_desc`, `diff_src_iter_desc`, and `diff_src_iter_c_desc`,
- `weights_peephole_desc` together with `diff_weights_peephole_desc`
- `bias_desc` together with `diff_bias_desc`,
- `dst_iter_desc` together with `dst_iter_c_desc`, `diff_dst_iter_desc`, and `diff_dst_iter_c_desc`.

This would then indicate that the LSTM backward propagation primitive should not use them and should default to zero values instead.

**Note** All memory descriptors may be initialized with `dnnl::memory::format_tag::any` value of `format_tag`.

#### Parameters

- `aprop_kind`: Propagation kind. Must be `dnnl::prop_kind::backward`.
- `direction`: RNN direction. See `dnnl::rnn_direction` for more info.
- `src_layer_desc`: Memory descriptor for the input vector.
- `src_iter_desc`: Memory descriptor for the input recurrent hidden state vector.
- `src_iter_c_desc`: Memory descriptor for the input recurrent cell state vector.
- `weights_layer_desc`: Memory descriptor for the weights applied to the layer input.
- `weights_iter_desc`: Memory descriptor for the weights applied to the recurrent input.
- `weights_peephole_desc`: Memory descriptor for the weights applied to the cell states (according to the Peephole LSTM formula).
- `bias_desc`: Bias memory descriptor.
- `dst_layer_desc`: Memory descriptor for the output vector.
- `dst_iter_desc`: Memory descriptor for the output recurrent hidden state vector.
- `dst_iter_c_desc`: Memory descriptor for the output recurrent cell state vector.

- `diff_src_layer_desc`: Memory descriptor for the diff of input vector.
- `diff_src_iter_desc`: Memory descriptor for the diff of input recurrent hidden state vector.
- `diff_src_iter_c_desc`: Memory descriptor for the diff of input recurrent cell state vector.
- `diff_weights_layer_desc`: Memory descriptor for the diff of weights applied to the layer input.
- `diff_weights_iter_desc`: Memory descriptor for the diff of weights applied to the recurrent input.
- `diff_weights_peephole_desc`: Memory descriptor for the diff of weights applied to the cell states (according to the Peephole LSTM formula).
- `diff_bias_desc`: Diff bias memory descriptor.
- `diff_dst_layer_desc`: Memory descriptor for the diff of output vector.
- `diff_dst_iter_desc`: Memory descriptor for the diff of output recurrent hidden state vector.
- `diff_dst_iter_c_desc`: Memory descriptor for the diff of output recurrent cell state vector.
- `flags`: Unused.

```
desc (prop_kind aprop_kind, rnn_direction direction, const memory::desc &src_layer_desc,
      const memory::desc &src_iter_desc, const memory::desc &src_iter_c_desc, const
      memory::desc &weights_layer_desc, const memory::desc &weights_iter_desc, const
      memory::desc &bias_desc, const memory::desc &dst_layer_desc, const mem-
      ory::desc &dst_iter_desc, const memory::desc &dst_iter_c_desc, const memory::desc
      &diff_src_layer_desc, const memory::desc &diff_src_iter_desc, const memory::desc
      &diff_src_iter_c_desc, const memory::desc &diff_weights_layer_desc, const mem-
      ory::desc &diff_weights_iter_desc, const memory::desc &diff_bias_desc, const
      memory::desc &diff_dst_layer_desc, const memory::desc &diff_dst_iter_desc, const
      memory::desc &diff_dst_iter_c_desc, rnn_flags flags = rnn_flags::undef)
```

Constructs an LSTM descriptor for backward propagation using `prop_kind`, `direction`, and `memory` descriptors.

The following arguments may point to a zero memory descriptor:

- `src_iter_desc` together with `src_iter_c_desc`, `diff_src_iter_desc`, and `diff_src_iter_c_desc`,
- `bias_desc` together with `diff_bias_desc`,
- `dst_iter_desc` together with `dst_iter_c_desc`, `diff_dst_iter_desc`, and `diff_dst_iter_c_desc`.

This would then indicate that the LSTM backward propagation primitive should not use them and should default to zero values instead.

**Note** All memory descriptors may be initialized with `dnnl::memory::format_tag::any` value of `format_tag`.

#### Parameters

- `aprop_kind`: Propagation kind. Must be `dnnl::prop_kind::backward`.
- `direction`: RNN direction. See `dnnl::rnn_direction` for more info.
- `src_layer_desc`: Memory descriptor for the input vector.
- `src_iter_desc`: Memory descriptor for the input recurrent hidden state vector.
- `src_iter_c_desc`: Memory descriptor for the input recurrent cell state vector.
- `weights_layer_desc`: Memory descriptor for the weights applied to the layer input.
- `weights_iter_desc`: Memory descriptor for the weights applied to the recurrent input.
- `bias_desc`: Bias memory descriptor.
- `dst_layer_desc`: Memory descriptor for the output vector.
- `dst_iter_desc`: Memory descriptor for the output recurrent hidden state vector.
- `dst_iter_c_desc`: Memory descriptor for the output recurrent cell state vector.
- `diff_src_layer_desc`: Memory descriptor for the diff of input vector.

- `diff_src_iter_desc`: Memory descriptor for the diff of input recurrent hidden state vector.
- `diff_src_iter_c_desc`: Memory descriptor for the diff of input recurrent cell state vector.
- `diff_weights_layer_desc`: Memory descriptor for the diff of weights applied to the layer input.
- `diff_weights_iter_desc`: Memory descriptor for the diff of weights applied to the recurrent input.
- `diff_bias_desc`: Diff bias memory descriptor.
- `diff_dst_layer_desc`: Memory descriptor for the diff of output vector.
- `diff_dst_iter_desc`: Memory descriptor for the diff of output recurrent hidden state vector.
- `diff_dst_iter_c_desc`: Memory descriptor for the diff of output recurrent cell state vector.
- `flags`: Unused.

```
struct primitive_desc : public dnnl::rnn_primitive_desc_base
```

Primitive descriptor for LSTM backward propagation.

## Public Functions

```
primitive_desc()
```

Default constructor. Produces an empty object.

```
primitive_desc(const desc &adesc, const engine &aengine, const  
lstm_forward::primitive_desc &hint_fwd_pd, bool allow_empty = false)
```

Constructs a primitive descriptor for an LSTM backward propagation primitive.

### Parameters

- `adesc`: Descriptor for LSTM backward propagation primitive.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for an LSTM forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
primitive_desc(const desc &adesc, const primitive_attr &attr, const engine &aengine,  
const lstm_forward::primitive_desc &hint_fwd_pd, bool allow_empty =  
false)
```

Constructs a primitive descriptor for an LSTM backward propagation primitive.

### Parameters

- `adesc`: Descriptor for an LSTM backward propagation primitive.
- `attr`: Primitive attributes to use.
- `aengine`: Engine to use.
- `hint_fwd_pd`: Primitive descriptor for an LSTM forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
memory::desc src_layer_desc() const
```

Returns source layer memory descriptor.

**Return** Source layer memory descriptor.

`memory::desc src_iter_desc() const`  
 Returns source iteration memory descriptor.  
**Return** Source iteration memory descriptor.  
**Return** A zero memory descriptor if the primitive does not have a source iteration parameter.

`memory::desc src_iter_c_desc() const`  
 Returns source iteration memory descriptor.  
**Return** Source iteration memory descriptor.  
**Return** A zero memory descriptor if the primitive does not have a source iteration parameter.

`memory::desc weights_layer_desc() const`  
 Returns weights layer memory descriptor.  
**Return** Weights layer memory descriptor.

`memory::desc weights_iter_desc() const`  
 Returns weights iteration memory descriptor.  
**Return** Weights iteration memory descriptor.

`memory::desc weights_peephole_desc() const`  
 Returns weights peephole memory descriptor.  
**Return** Weights peephole memory descriptor.

`memory::desc weights_projection_desc() const`  
 Returns weights projection memory descriptor.  
**Return** Weights projection memory descriptor.

`memory::desc bias_desc() const`  
 Returns bias memory descriptor.  
**Return** Bias memory descriptor.  
**Return** A zero memory descriptor if the primitive does not have a bias parameter.

`memory::desc dst_layer_desc() const`  
 Returns destination layer memory descriptor.  
**Return** Destination layer memory descriptor.

`memory::desc dst_iter_desc() const`  
 Returns destination iteration memory descriptor.  
**Return** Destination iteration memory descriptor.  
**Return** A zero memory descriptor if the primitive does not have a destination iteration parameter.

`memory::desc dst_iter_c_desc() const`  
 Returns source iteration memory descriptor.  
**Return** Source iteration memory descriptor.  
**Return** A zero memory descriptor if the primitive does not have a source iteration parameter.

`memory::desc workspace_desc() const`  
 Returns the workspace memory descriptor.  
**Return** Workspace memory descriptor.  
**Return** A zero memory descriptor if the primitive does not require workspace parameter.

`memory::desc diff_src_layer_desc() const`  
 Returns diff source layer memory descriptor.  
**Return** Diff source layer memory descriptor.

`memory::desc diff_src_iter_desc() const`  
 Returns diff source iteration memory descriptor.  
**Return** Diff source iteration memory descriptor.  
**Return** A zero memory descriptor if the primitive does not have a diff source iteration parameter.

```
memory::desc diff_src_iter_c_desc() const
    Returns diff source recurrent cell state memory descriptor.
Return Diff source recurrent cell state memory descriptor.

memory::desc diff_weights_layer_desc() const
    Returns diff weights layer memory descriptor.
Return Diff weights layer memory descriptor.

memory::desc diff_weights_iter_desc() const
    Returns diff weights iteration memory descriptor.
Return Diff weights iteration memory descriptor.

memory::desc diff_weights_peephole_desc() const
    Returns diff weights peephole memory descriptor.
Return Diff weights peephole memory descriptor.

memory::desc diff_weights_projection_desc() const
    Returns diff weights projection memory descriptor.
Return Diff weights projection memory descriptor.

memory::desc diff_bias_desc() const
    Returns diff bias memory descriptor.
Return Diff bias memory descriptor.
Return A zero memory descriptor if the primitive does not have a diff bias parameter.

memory::desc diff_dst_layer_desc() const
    Returns diff destination layer memory descriptor.
Return Diff destination layer memory descriptor.

memory::desc diff_dst_iter_desc() const
    Returns diff destination iteration memory descriptor.
Return Diff destination iteration memory descriptor.
Return A zero memory descriptor if the primitive does not have a diff destination iteration parameter.

memory::desc diff_dst_iter_c_desc() const
    Returns diff destination recurrent cell state memory descriptor.
Return Diff destination recurrent cell state memory descriptor.
```

**struct** dnnl::**gru\_forward**:**public** dnnl::*primitive*  
GRU forward propagation primitive.

## Public Functions

**gru\_forward()**  
Default constructor. Produces an empty object.

**gru\_forward(const primitive\_desc &pd)**  
Constructs a GRU forward propagation primitive.

### Parameters

- pd: Primitive descriptor for a GRU forward propagation primitive.

**struct desc**  
Descriptor for a GRU forward propagation primitive.

## Public Functions

```
desc(prop_kind aprop_kind, rnn_direction direction, const memory::desc &src_layer_desc,
      const memory::desc &src_iter_desc, const memory::desc &weights_layer_desc, const
      memory::desc &weights_iter_desc, const memory::desc &bias_desc, const memory::desc
      &dst_layer_desc, const memory::desc &dst_iter_desc, rnn_flags flags = rnn_flags::undef)
```

Constructs a descriptor for a GRU forward propagation primitive.

The following arguments may point to a zero memory descriptor:

- *src\_iter\_desc*,
- *bias\_desc*,
- *dst\_iter\_desc*.

This would then indicate that the GRU forward propagation primitive should not use them and should default to zero values instead.

**Note** All memory descriptors except *src\_iter\_desc* may be initialized with an *dnnl::memory::format\_tag::any* value of *format\_tag*.

### Parameters

- *aprop\_kind*: Propagation kind. Possible values are *dnnl::prop\_kind::forward\_training*, and *dnnl::prop\_kind::forward\_inference*.
- *direction*: RNN direction. See *dnnl::rnn\_direction* for more info.
- *src\_layer\_desc*: Memory descriptor for the input vector.
- *src\_iter\_desc*: Memory descriptor for the input recurrent hidden state vector.
- *weights\_layer\_desc*: Memory descriptor for the weights applied to the layer input.
- *weights\_iter\_desc*: Memory descriptor for the weights applied to the recurrent input.
- *bias\_desc*: Bias memory descriptor.
- *dst\_layer\_desc*: Memory descriptor for the output vector.
- *dst\_iter\_desc*: Memory descriptor for the output recurrent hidden state vector.
- *flags*: Unused.

```
struct primitive_desc : public dnnl::rnn_primitive_desc_base
```

Primitive descriptor GRU forward propagation primitive.

## Public Functions

```
primitive_desc()
```

Default constructor. Produces an empty object.

```
primitive_desc(const desc &adesc, const engine &aengine, bool allow_empty = false)
```

Constructs a primitive descriptor for a GRU forward propagation primitive.

### Parameters

- *adesc*: Descriptor for a GRU forward propagation primitive.
- *aengine*: Engine to use.
- *allow\_empty*: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
primitive_desc(const desc &adesc, const primitive_attr &attr, const engine &aengine,
              bool allow_empty = false)
```

Constructs a primitive descriptor for a GRU forward propagation primitive.

### Parameters

- *adesc*: Descriptor for a GRU forward propagation primitive.
- *attr*: Primitive attributes to use.
- *aengine*: Engine to use.

- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

`memory::desc src_layer_desc() const`

Returns source layer memory descriptor.

**Return** Source layer memory descriptor.

`memory::desc src_iter_desc() const`

Returns source iteration memory descriptor.

**Return** Source iteration memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source iteration parameter.

`memory::desc weights_layer_desc() const`

Returns weights layer memory descriptor.

**Return** Weights layer memory descriptor.

`memory::desc weights_iter_desc() const`

Returns weights iteration memory descriptor.

**Return** Weights iteration memory descriptor.

`memory::desc bias_desc() const`

Returns bias memory descriptor.

**Return** Bias memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a bias parameter.

`memory::desc dst_layer_desc() const`

Returns destination layer memory descriptor.

**Return** Destination layer memory descriptor.

`memory::desc dst_iter_desc() const`

Returns destination iteration memory descriptor.

**Return** Destination iteration memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination iteration parameter.

`memory::desc workspace_desc() const`

Returns the workspace memory descriptor.

**Return** Workspace memory descriptor.

**Return** A zero memory descriptor if the primitive does not require workspace parameter.

`struct dnnl::gru_backward : public dnnl::primitive`

GRU backward propagation primitive.

## Public Functions

`gru_backward()`

Default constructor. Produces an empty object.

`gru_backward(const primitive_desc &pd)`

Constructs a GRU backward propagation primitive.

### Parameters

- `pd`: Primitive descriptor for a GRU backward propagation primitive.

`struct desc`

Descriptor for a GRU backward propagation primitive.

## Public Functions

```
desc(prop_kind aprop_kind, rnn_direction direction, const memory::desc &src_layer_desc,
      const memory::desc &src_iter_desc, const memory::desc &weights_layer_desc, const
      memory::desc &weights_iter_desc, const memory::desc &bias_desc, const mem-
      ory::desc &dst_layer_desc, const memory::desc &dst_iter_desc, const memory::desc
      &diff_src_layer_desc, const memory::desc &diff_src_iter_desc, const memory::desc
      &diff_weights_layer_desc, const memory::desc &diff_weights_iter_desc, const mem-
      ory::desc &diff_bias_desc, const memory::desc &diff_dst_layer_desc, const mem-
      ory::desc &diff_dst_iter_desc, rnn_flags flags = rnn_flags::undef)
```

Constructs a descriptor for a GRU backward propagation primitive.

The following arguments may point to a zero memory descriptor:

- *src\_iter\_desc* together with *diff\_src\_iter\_desc*,
- *bias\_desc* together with *diff\_bias\_desc*,
- *dst\_iter\_desc* together with *diff\_dst\_iter\_desc*.

This would then indicate that the GRU backward propagation primitive should not use them and should default to zero values instead.

**Note** All memory descriptors may be initialized with *dnnl::memory::format\_tag::any* value of *format\_tag*.

### Parameters

- *aprop\_kind*: Propagation kind. Must be *dnnl::prop\_kind::backward*.
- *direction*: RNN direction. See *dnnl::rnn\_direction* for more info.
- *src\_layer\_desc*: Memory descriptor for the input vector.
- *src\_iter\_desc*: Memory descriptor for the input recurrent hidden state vector.
- *weights\_layer\_desc*: Memory descriptor for the weights applied to the layer input.
- *weights\_iter\_desc*: Memory descriptor for the weights applied to the recurrent input.
- *bias\_desc*: Bias memory descriptor.
- *dst\_layer\_desc*: Memory descriptor for the output vector.
- *dst\_iter\_desc*: Memory descriptor for the output recurrent hidden state vector.
- *diff\_src\_layer\_desc*: Memory descriptor for the diff of input vector.
- *diff\_src\_iter\_desc*: Memory descriptor for the diff of input recurrent hidden state vector.
- *diff\_weights\_layer\_desc*: Memory descriptor for the diff of weights applied to the layer input.
- *diff\_weights\_iter\_desc*: Memory descriptor for the diff of weights applied to the recurrent input.
- *diff\_bias\_desc*: Diff bias memory descriptor.
- *diff\_dst\_layer\_desc*: Memory descriptor for the diff of output vector.
- *diff\_dst\_iter\_desc*: Memory descriptor for the diff of output recurrent hidden state vector.
- *flags*: Unused.

```
struct primitive_desc : public dnnl::rnn_primitive_desc_base
```

Primitive descriptor for a GRU backward propagation primitive.

## Public Functions

**primitive\_desc()**

Default constructor. Produces an empty object.

**primitive\_desc(const desc &adesc, const engine &aengine, const gru\_forward::primitive\_desc &hint\_fwd\_pd, bool allow\_empty = false)**

Constructs a primitive descriptor for a GRU backward propagation primitive.

### Parameters

- **adesc:** Descriptor for a GRU backward propagation primitive.
- **aengine:** Engine to use.
- **hint\_fwd\_pd:** Primitive descriptor for a GRU forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **allow\_empty:** A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**primitive\_desc(const desc &adesc, const primitive\_attr &attr, const engine &aengine, const gru\_forward::primitive\_desc &hint\_fwd\_pd, bool allow\_empty = false)**

Constructs a primitive descriptor for a GRU backward propagation primitive.

### Parameters

- **adesc:** Descriptor for a GRU backward propagation primitive.
- **attr:** Primitive attributes to use.
- **aengine:** Engine to use.
- **hint\_fwd\_pd:** Primitive descriptor for a GRU forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **allow\_empty:** A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**memory::desc src\_layer\_desc() const**

Returns source layer memory descriptor.

**Return** Source layer memory descriptor.

**memory::desc src\_iter\_desc() const**

Returns source iteration memory descriptor.

**Return** Source iteration memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source iteration parameter.

**memory::desc weights\_layer\_desc() const**

Returns weights layer memory descriptor.

**Return** Weights layer memory descriptor.

**memory::desc weights\_iter\_desc() const**

Returns weights iteration memory descriptor.

**Return** Weights iteration memory descriptor.

**memory::desc bias\_desc() const**

Returns bias memory descriptor.

**Return** Bias memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a bias parameter.

**memory::desc dst\_layer\_desc() const**

Returns destination layer memory descriptor.

**Return** Destination layer memory descriptor.

`memory::desc dst_iter_desc() const`  
 Returns destination iteration memory descriptor.  
**Return** Destination iteration memory descriptor.  
**Return** A zero memory descriptor if the primitive does not have a destination iteration parameter.

`memory::desc workspace_desc() const`  
 Returns the workspace memory descriptor.  
**Return** Workspace memory descriptor.  
**Return** A zero memory descriptor if the primitive does not require workspace parameter.

`memory::desc diff_src_layer_desc() const`  
 Returns diff source layer memory descriptor.  
**Return** Diff source layer memory descriptor.

`memory::desc diff_src_iter_desc() const`  
 Returns diff source iteration memory descriptor.  
**Return** Diff source iteration memory descriptor.  
**Return** A zero memory descriptor if the primitive does not have a diff source iteration parameter.

`memory::desc diff_weights_layer_desc() const`  
 Returns diff weights layer memory descriptor.  
**Return** Diff weights layer memory descriptor.

`memory::desc diff_weights_iter_desc() const`  
 Returns diff weights iteration memory descriptor.  
**Return** Diff weights iteration memory descriptor.

`memory::desc diff_bias_desc() const`  
 Returns diff bias memory descriptor.  
**Return** Diff bias memory descriptor.  
**Return** A zero memory descriptor if the primitive does not have a diff bias parameter.

`memory::desc diff_dst_layer_desc() const`  
 Returns diff destination layer memory descriptor.  
**Return** Diff destination layer memory descriptor.

`memory::desc diff_dst_iter_desc() const`  
 Returns diff destination iteration memory descriptor.  
**Return** Diff destination iteration memory descriptor.  
**Return** A zero memory descriptor if the primitive does not have a diff destination iteration parameter.

**struct dnnl::lbr\_gru\_forward : public primitive**  
 LBR GRU forward propagation primitive.

## Public Functions

### `lbr_gru_forward()`

Default constructor. Produces an empty object.

### `lbr_gru_forward(const primitive_desc &pd)`

Constructs an LBR GRU forward propagation primitive.

#### Parameters

- pd: Primitive descriptor for an LBR GRU forward propagation primitive.

### `struct desc`

Descriptor for an LBR GRU forward propagation primitive.

## Public Functions

```
desc(prop_kind aprop_kind, rnn_direction direction, const memory::desc &src_layer_desc,
      const memory::desc &src_iter_desc, const memory::desc &weights_layer_desc, const
      memory::desc &weights_iter_desc, const memory::desc &bias_desc, const memory::desc
      &dst_layer_desc, const memory::desc &dst_iter_desc, rnn_flags flags = rnn_flags::undef)
```

Constructs a descriptor for LBR GRU forward propagation primitive.

The following arguments may point to a zero memory descriptor:

- *src\_iter\_desc*,
- *bias\_desc*,
- *dst\_iter\_desc*.

This would then indicate that the LBR GRU forward propagation primitive should not use them and should default to zero values instead.

**Note** All memory descriptors except *src\_iter\_desc* may be initialized with an *dnnl::memory::format\_tag::any* value of *format\_tag*.

### Parameters

- *aprop\_kind*: Propagation kind. Possible values are *dnnl::prop\_kind::forward\_training*, and *dnnl::prop\_kind::forward\_inference*.
- *direction*: RNN direction. See *dnnl::rnn\_direction* for more info.
- *src\_layer\_desc*: Memory descriptor for the input vector.
- *src\_iter\_desc*: Memory descriptor for the input recurrent hidden state vector.
- *weights\_layer\_desc*: Memory descriptor for the weights applied to the layer input.
- *weights\_iter\_desc*: Memory descriptor for the weights applied to the recurrent input.
- *bias\_desc*: Bias memory descriptor.
- *dst\_layer\_desc*: Memory descriptor for the output vector.
- *dst\_iter\_desc*: Memory descriptor for the output recurrent hidden state vector.
- *flags*: Unused.

```
struct primitive_desc : public dnnl::rnn_primitive_desc_base
```

Primitive descriptor for an LBR GRU forward propagation primitive.

## Public Functions

```
primitive_desc()
```

Default constructor. Produces an empty object.

```
primitive_desc(const desc &adesc, const engine &aengine, bool allow_empty = false)
```

Constructs a primitive descriptor for a LBR GRU forward propagation primitive.

### Parameters

- *adesc*: Descriptor for a LBR GRU forward propagation primitive.
- *aengine*: Engine to use.
- *allow\_empty*: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

```
primitive_desc(const desc &adesc, const primitive_attr &attr, const engine &aengine,
              bool allow_empty = false)
```

Constructs a primitive descriptor for a LBR GRU forward propagation primitive.

### Parameters

- *adesc*: Descriptor for a LBR GRU forward propagation primitive.
- *attr*: Primitive attributes to use.
- *aengine*: Engine to use.

- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

`memory::desc src_layer_desc() const`

Returns source layer memory descriptor.

**Return** Source layer memory descriptor.

`memory::desc src_iter_desc() const`

Returns source iteration memory descriptor.

**Return** Source iteration memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source iteration parameter.

`memory::desc weights_layer_desc() const`

Returns weights layer memory descriptor.

**Return** Weights layer memory descriptor.

`memory::desc weights_iter_desc() const`

Returns weights iteration memory descriptor.

**Return** Weights iteration memory descriptor.

`memory::desc bias_desc() const`

Returns bias memory descriptor.

**Return** Bias memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a bias parameter.

`memory::desc dst_layer_desc() const`

Returns destination layer memory descriptor.

**Return** Destination layer memory descriptor.

`memory::desc dst_iter_desc() const`

Returns destination iteration memory descriptor.

**Return** Destination iteration memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination iteration parameter.

`memory::desc workspace_desc() const`

Returns the workspace memory descriptor.

**Return** Workspace memory descriptor.

**Return** A zero memory descriptor if the primitive does not require workspace parameter.

`struct dnnl::lbr_gru_backward : public dnnl::primitive`

LBR GRU backward propagation primitive.

## Public Functions

`lbr_gru_backward()`

Default constructor. Produces an empty object.

`lbr_gru_backward(const primitive_desc &pd)`

Constructs an LBR GRU backward propagation primitive.

### Parameters

- `pd`: Primitive descriptor for an LBR GRU backward propagation primitive.

`struct desc`

Descriptor for a LBR GRU backward propagation primitive.

## Public Functions

```
desc(prop_kind aprop_kind, rnn_direction direction, const memory::desc &src_layer_desc,
      const memory::desc &src_iter_desc, const memory::desc &weights_layer_desc, const
      memory::desc &weights_iter_desc, const memory::desc &bias_desc, const mem-
      ory::desc &dst_layer_desc, const memory::desc &dst_iter_desc, const memory::desc
      &diff_src_layer_desc, const memory::desc &diff_src_iter_desc, const memory::desc
      &diff_weights_layer_desc, const memory::desc &diff_weights_iter_desc, const mem-
      ory::desc &diff_bias_desc, const memory::desc &diff_dst_layer_desc, const mem-
      ory::desc &diff_dst_iter_desc, rnn_flags flags = rnn_flags::undef)
```

Constructs a descriptor for LBR GRU backward propagation primitive.

The following arguments may point to a zero memory descriptor:

- *src\_iter\_desc* together with *diff\_src\_iter\_desc*,
- *bias\_desc* together with *diff\_bias\_desc*,
- *dst\_iter\_desc* together with *diff\_dst\_iter\_desc*.

This would then indicate that the LBR GRU backward propagation primitive should not use them and should default to zero values instead.

**Note** All memory descriptors may be initialized with *dnnl::memory::format\_tag::any* value of *format\_tag*.

### Parameters

- *aprop\_kind*: Propagation kind. Must be *dnnl::prop\_kind::backward*.
- *direction*: RNN direction. See *dnnl::rnn\_direction* for more info.
- *src\_layer\_desc*: Memory descriptor for the input vector.
- *src\_iter\_desc*: Memory descriptor for the input recurrent hidden state vector.
- *weights\_layer\_desc*: Memory descriptor for the weights applied to the layer input.
- *weights\_iter\_desc*: Memory descriptor for the weights applied to the recurrent input.
- *bias\_desc*: Bias memory descriptor.
- *dst\_layer\_desc*: Memory descriptor for the output vector.
- *dst\_iter\_desc*: Memory descriptor for the output recurrent hidden state vector.
- *diff\_src\_layer\_desc*: Memory descriptor for the diff of input vector.
- *diff\_src\_iter\_desc*: Memory descriptor for the diff of input recurrent hidden state vector.
- *diff\_weights\_layer\_desc*: Memory descriptor for the diff of weights applied to the layer input.
- *diff\_weights\_iter\_desc*: Memory descriptor for the diff of weights applied to the recurrent input.
- *diff\_bias\_desc*: Diff bias memory descriptor.
- *diff\_dst\_layer\_desc*: Memory descriptor for the diff of output vector.
- *diff\_dst\_iter\_desc*: Memory descriptor for the diff of output recurrent hidden state vector.
- *flags*: Unused.

```
struct primitive_desc : public dnnl::rnn_primitive_desc_base
```

Primitive descriptor for an LBR GRU backward propagation primitive.

## Public Functions

**primitive\_desc()** = default

Default constructor. Produces an empty object.

**primitive\_desc(const desc &adesc, const engine &aengine, const lbr\_gru\_forward::primitive\_desc &hint\_fwd\_pd, bool allow\_empty = false)**

Constructs a primitive descriptor for an LBR GRU backward propagation primitive.

### Parameters

- **adesc:** Descriptor for an LBR GRU backward propagation primitive.
- **aengine:** Engine to use.
- **hint\_fwd\_pd:** Primitive descriptor for an LBR GRU forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **allow\_empty:** A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**primitive\_desc(const desc &adesc, const primitive\_attr &attr, const engine &aengine, const lbr\_gru\_forward::primitive\_desc &hint\_fwd\_pd, bool allow\_empty = false)**

Constructs a primitive descriptor for an LBR GRU backward propagation primitive.

### Parameters

- **adesc:** Descriptor for an LBR GRU backward propagation primitive.
- **attr:** Primitive attributes to use.
- **aengine:** Engine to use.
- **hint\_fwd\_pd:** Primitive descriptor for an LBR GRU forward propagation primitive. It is used as a hint for deciding which memory format to use.
- **allow\_empty:** A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**memory::desc src\_layer\_desc() const**

Returns source layer memory descriptor.

**Return** Source layer memory descriptor.

**memory::desc src\_iter\_desc() const**

Returns source iteration memory descriptor.

**Return** Source iteration memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source iteration parameter.

**memory::desc weights\_layer\_desc() const**

Returns weights layer memory descriptor.

**Return** Weights layer memory descriptor.

**memory::desc weights\_iter\_desc() const**

Returns weights iteration memory descriptor.

**Return** Weights iteration memory descriptor.

**memory::desc bias\_desc() const**

Returns bias memory descriptor.

**Return** Bias memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a bias parameter.

**memory::desc dst\_layer\_desc() const**

Returns destination layer memory descriptor.

**Return** Destination layer memory descriptor.

*memory::desc* **dst\_iter\_desc()** **const**  
Returns destination iteration memory descriptor.  
**Return** Destination iteration memory descriptor.  
**Return** A zero memory descriptor if the primitive does not have a destination iteration parameter.

*memory::desc* **workspace\_desc()** **const**  
Returns the workspace memory descriptor.  
**Return** Workspace memory descriptor.  
**Return** A zero memory descriptor if the primitive does not require workspace parameter.

*memory::desc* **diff\_src\_layer\_desc()** **const**  
Returns diff source layer memory descriptor.  
**Return** Diff source layer memory descriptor.

*memory::desc* **diff\_src\_iter\_desc()** **const**  
Returns diff source iteration memory descriptor.  
**Return** Diff source iteration memory descriptor.  
**Return** A zero memory descriptor if the primitive does not have a diff source iteration parameter.

*memory::desc* **diff\_weights\_layer\_desc()** **const**  
Returns diff weights layer memory descriptor.  
**Return** Diff weights layer memory descriptor.

*memory::desc* **diff\_weights\_iter\_desc()** **const**  
Returns diff weights iteration memory descriptor.  
**Return** Diff weights iteration memory descriptor.

*memory::desc* **diff\_bias\_desc()** **const**  
Returns diff bias memory descriptor.  
**Return** Diff bias memory descriptor.  
**Return** A zero memory descriptor if the primitive does not have a diff bias parameter.

*memory::desc* **diff\_dst\_layer\_desc()** **const**  
Returns diff destination layer memory descriptor.  
**Return** Diff destination layer memory descriptor.

*memory::desc* **diff\_dst\_iter\_desc()** **const**  
Returns diff destination iteration memory descriptor.  
**Return** Diff destination iteration memory descriptor.  
**Return** A zero memory descriptor if the primitive does not have a diff destination iteration parameter.

## 7.5.17 Shuffle

The shuffle primitive shuffles data along the shuffle axis (here is designated as  $C$ ) with the group parameter  $G$ . Namely, the shuffle axis is thought to be a 2D tensor of size  $(\frac{C}{G} \times G)$  and it is being transposed to  $(G \times \frac{C}{G})$ . Variable names follow the standard *Conventions*.

The formal definition is shown below:

### 7.5.17.1 Forward

$$\text{dst}(\overline{ou}, c, \overline{in}) = \text{src}(\overline{ou}, c', \overline{in})$$

where

- $c$  dimension is called a shuffle axis,
- $G$  is a group\_size,
- $\overline{ou}$  is the outermost indices (to the left from shuffle axis),
- $\overline{in}$  is the innermost indices (to the right from shuffle axis), and
- $c'$  and  $c$  relate to each other as define by the system:

$$\begin{cases} c &= u + v \cdot \frac{C}{G}, \\ c' &= u \cdot G + v, \end{cases}$$

Here,  $0 \leq u < \frac{C}{G}$  and  $0 \leq v < G$ .

#### 7.5.17.1.1 Difference Between Forward Training and Forward Inference

There is no difference between the *forward\_training* and *forward\_inference* propagation kinds.

### 7.5.17.2 Backward

The backward propagation computes  $\text{diff\_src}(ou, c, in)$ , based on  $\text{diff\_dst}(ou, c, in)$ .

Essentially, backward propagation is the same as forward propagation with  $g$  replaced by  $C/g$ .

### 7.5.17.3 Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	<i>DNNL_ARG_SRC</i>
dst	<i>DNNL_ARG_DST</i>
diff_src	<i>DNNL_ARG_DIFF_SRC</i>
diff_dst	<i>DNNL_ARG_DIFF_DST</i>

#### 7.5.17.4 Operation Details

1. The memory format and data type for `src` and `dst` are assumed to be the same, and in the API are typically referred as `data` (e.g., see `data_desc` in `dnnl::shuffle_forward::desc::desc()`). The same holds for `diff_src` and `diff_dst`. The corresponding memory descriptors are referred to as `diff_data_desc`.

#### 7.5.17.5 Data Types Support

The shuffle primitive supports the following combinations of data types:

---

**Note:** Here we abbreviate data types names for readability. For example, `dnnl::memory::data_type::f32` is abbreviated to `f32`.

---

Propagation	Source / Destination
forward / backward	<code>f32, bf16</code>
forward	<code>s32, s8, u8</code>

#### 7.5.17.6 Data Layouts

The shuffle primitive works with arbitrary data tensors. There is no special meaning associated with any logical dimensions. However, the shuffle axis is typically referred to as channels (hence in formulas we use `c`).

Shuffle operation typically appear in CNN topologies. Hence, in the library the shuffle primitive is optimized for the corresponding memory formats:

Spatial	Logical tensor	Shuffle Axis	Implementations optimized for memory formats
2D	NCHW	1 (C)	<code>nchw (abcd), nhwc (acdb), optimized^</code>
3D	NCDHW	1 (C)	<code>ncdhw (abcde), ndhwc (acdeb), optimized^</code>

Here `optimized^` means the format that comes out of any preceding compute-intensive primitive.

#### 7.5.17.7 Post-ops and Attributes

The shuffle primitive does not have to support any post-ops or attributes.

#### 7.5.17.8 API

```
struct dnnl::shuffle_forward : public dnnl::primitive
    Shuffle forward propagation primitive.
```

## Public Functions

### `shuffle_forward()`

Default constructor. Produces an empty object.

### `shuffle_forward(const primitive_desc &pd)`

Constructs a shuffle forward propagation primitive.

#### Parameters

- pd: Primitive descriptor for a shuffle forward propagation primitive.

### `struct desc`

Descriptor for a shuffle forward propagation primitive.

## Public Functions

### `desc(prop_kind aprop_kind, const memory::desc &data_desc, int axis, int group_size)`

Constructs a descriptor for a shuffle forward propagation primitive.

#### Parameters

- aprop\_kind: Propagation kind. Possible values are `dnnl::prop_kind::forward_training`, and `dnnl::prop_kind::forward_inference`.
- data\_desc: Source and destination memory descriptor.
- axis: The axis along which the data is shuffled.
- group\_size: Shuffle group size.

### `struct primitive_desc : public dnnl::primitive_desc`

Primitive descriptor for a shuffle forward propagation primitive.

## Public Functions

### `primitive_desc()`

Default constructor. Produces an empty object.

### `primitive_desc(const desc &adesc, const engine &aengine, const primitive_attr &attr = primitive_attr(), bool allow_empty = false)`

Constructs a primitive descriptor for a shuffle forward propagation primitive.

#### Parameters

- adesc: Descriptor for a shuffle forward propagation primitive.
- aengine: Engine to use.
- attr: Primitive attributes to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

### `memory::desc src_desc() const`

Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter.

### `memory::desc dst_desc() const`

Returns a destination memory descriptor.

**Return** Destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination parameter.

---

```
struct dnnl::shuffle_backward : public dnnl::primitive
    Shuffle backward propagation primitive.
```

## Public Functions

**shuffle\_backward()**

Default constructor. Produces an empty object.

**shuffle\_backward(const primitive\_desc &pd)**

Constructs a shuffle backward propagation primitive.

### Parameters

- pd: Primitive descriptor for a shuffle backward propagation primitive.

**struct desc**

Descriptor for a shuffle primitive backward propagation primitive.

## Public Functions

**desc(const memory::desc &diff\_data\_desc, int axis, int group\_size)**

Constructs a descriptor for a shuffle backward propagation primitive.

### Parameters

- diff\_data\_desc: Diff source and diff destination memory descriptor.
- axis: The axis along which the data is shuffled.
- group\_size: Shuffle group size.

**struct primitive\_desc : public dnnl::primitive\_desc**

Primitive descriptor for a shuffle backward propagation primitive.

## Public Functions

**primitive\_desc()**

Default constructor. Produces an empty object.

**primitive\_desc(const desc &adesc, const engine &aengine, const shuffle\_forward::primitive\_desc &hint\_fwd\_pd, const primitive\_attr &attr = primitive\_attr(), bool allow\_empty = false)**

Constructs a primitive descriptor for a shuffle backward propagation primitive.

### Parameters

- adesc: Descriptor for a shuffle backward propagation primitive.
- aengine: Engine to use.
- attr: Primitive attributes to use.
- hint\_fwd\_pd: Primitive descriptor for a shuffle forward propagation primitive. It is used as a hint for deciding which memory format to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**memory::desc diff\_src\_desc() const**

Returns a diff source memory descriptor.

**Return** Diff source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff source memory with.

`memory::desc diff_dst_desc() const`  
 Returns a diff destination memory descriptor.  
**Return** Diff destination memory descriptor.  
**Return** A zero memory descriptor if the primitive does not have a diff destination parameter.

## 7.5.18 Softmax

The softmax primitive performs softmax along a particular axis on data with arbitrary dimensions. All other axes are treated as independent (batch).

In general form, the operation is defined by the following formulas. The variable names follow the standard *Conventions*.

### 7.5.18.1 Forward

$$\text{dst}(\overline{ou}, c, \overline{in}) = \frac{e^{\text{src}(\overline{ou}, c, \overline{in}) - \nu(\overline{ou}, \overline{in})}}{\sum_{ic} e^{\text{src}(\overline{ou}, ic, \overline{in}) - \nu(\overline{ou}, \overline{in})}},$$

where

- $c$  axis over which the softmax computation is computed on,
- $\overline{ou}$  is the outermost index (to the left of softmax axis),
- $\overline{in}$  is the innermost index (to the right of softmax axis), and
- $\nu$  is used to produce more accurate results and defined as:

$$\nu(\overline{ou}, \overline{in}) = \max_{ic} \text{src}(\overline{ou}, ic, \overline{in})$$

#### 7.5.18.1.1 Difference Between Forward Training and Forward Inference

There is no difference between the `forward_training` and `forward_inference` propagation kinds.

### 7.5.18.2 Backward

The backward propagation computes  $\text{diff\_src}(ou, c, in)$ , based on  $\text{diff\_dst}(ou, c, in)$  and  $\text{dst}(ou, c, in)$ .

### 7.5.18.3 Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

Primitive input/output	Execution argument index
src	<code>DNNL_ARG_SRC</code>
dst	<code>DNNL_ARG_DST</code>
diff_src	<code>DNNL_ARG_DIFF_SRC</code>
diff_dst	<code>DNNL_ARG_DIFF_DST</code>

#### 7.5.18.4 Operation Details

- Both forward and backward propagation support in-place operations, meaning that `src` can be used as input and output for forward propagation, and `diff_dst` can be used as input and output for backward propagation. In case of in-place operation, the original data will be overwritten.

#### 7.5.18.5 Post-ops and Attributes

The softmax primitive does not have to support any post-ops or attributes.

#### 7.5.18.6 Data Types Support

The softmax primitive supports the following combinations of data types.

---

**Note:** Here we abbreviate data types names for readability. For example, `dnnl::memory::data_type::f32` is abbreviated to `f32`.

---

Propagation	Source / Destination
forward / backward	<code>bf16, f32</code>
forward	<code>f16</code>

#### 7.5.18.7 Data Representation

##### 7.5.18.7.1 Source, Destination, and Their Gradients

The softmax primitive works with arbitrary data tensors. There is no special meaning associated with any logical dimensions. However, the softmax axis is typically referred to as channels (hence in formulas we use  $c$ ).

#### 7.5.18.8 API

```
struct dnnl::softmax_forward : public dnnl::primitive
    Softmax forward propagation primitive.
```

##### Public Functions

###### `softmax_forward()`

Default constructor. Produces an empty object.

###### `softmax_forward(const primitive_desc &pd)`

Constructs a softmax forward propagation primitive.

##### Parameters

- `pd`: Primitive descriptor for a softmax forward propagation primitive.

###### `struct desc`

Descriptor for a softmax forward propagation primitive.

## Public Functions

**desc()**

Default constructor. Produces an empty object.

**desc (prop\_kind aprop\_kind, const memory::desc &data\_desc, int softmax\_axis)**

Constructs a descriptor for a softmax forward propagation primitive.

### Parameters

- aprop\_kind: Propagation kind. Possible values are *dnnl::prop\_kind::forward\_training*, and *dnnl::prop\_kind::forward\_inference*.
- data\_desc: Source and destination memory descriptor.
- softmax\_axis: Axis over which softmax is computed.

**struct primitive\_desc : public dnnl::primitive\_desc**

Primitive descriptor for a softmax forward propagation primitive.

## Public Functions

**primitive\_desc()**

Default constructor. Produces an empty object.

**primitive\_desc (const desc &adesc, const engine &aengine, bool allow\_empty = false)**

Constructs a primitive descriptor for a softmax forward propagation primitive.

### Parameters

- adesc: descriptor for a softmax forward propagation primitive.
- aengine: Engine to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**primitive\_desc (const desc &adesc, const primitive\_attr &attr, const engine &aengine, bool allow\_empty = false)**

Constructs a primitive descriptor for a softmax forward propagation primitive.

### Parameters

- adesc: Descriptor for a softmax forward propagation primitive.
- aengine: Engine to use.
- attr: Primitive attributes to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**memory::desc src\_desc () const**

Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter.

**memory::desc dst\_desc () const**

Returns a destination memory descriptor.

**Return** Destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination parameter.

**struct dnnl::softmax\_backward : public dnnl::primitive**

Softmax backward propagation primitive.

## Public Functions

**softmax\_backward()**

Default constructor. Produces an empty object.

**softmax\_backward(const primitive\_desc &pd)**

Constructs a softmax backward propagation primitive.

### Parameters

- pd: Primitive descriptor for a softmax backward propagation primitive.

**struct desc**

Descriptor for a softmax backward propagation primitive.

## Public Functions

**desc()**

Default constructor. Produces an empty object.

**desc(const memory::desc &diff\_data\_desc, const memory::desc &data\_desc, int softmax\_axis)**

Constructs a descriptor for a softmax backward propagation primitive.

### Parameters

- diff\_data\_desc: Diff source and diff destination memory descriptor.
- data\_desc: Destination memory descriptor.
- softmax\_axis: Axis over which softmax is computed.

**struct primitive\_desc : public dnnl::primitive\_desc**

Primitive descriptor for a softmax backward propagation primitive.

## Public Functions

**primitive\_desc()**

Default constructor. Produces an empty object.

**primitive\_desc(const desc &adesc, const engine &aengine, const softmax\_forward::primitive\_desc &hint\_fwd\_pd, bool allow\_empty = false)**

Constructs a primitive descriptor for a softmax backward propagation primitive.

### Parameters

- adesc: Descriptor for a softmax backward propagation primitive.
- aengine: Engine to use.
- hint\_fwd\_pd: Primitive descriptor for a softmax forward propagation primitive. It is used as a hint for deciding which memory format to use.
- allow\_empty: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**primitive\_desc(const desc &adesc, const primitive\_attr &attr, const engine &aengine, const softmax\_forward::primitive\_desc &hint\_fwd\_pd, bool allow\_empty = false)**

Constructs a primitive descriptor for a softmax backward propagation primitive.

### Parameters

- adesc: Descriptor for a softmax backward propagation primitive.
- attr: Primitive attributes to use.
- aengine: Engine to use.

- `hint_fwd_pd`: Primitive descriptor for a softmax forward propagation primitive. It is used as a hint for deciding which memory format to use.
- `allow_empty`: A flag signifying whether construction is allowed to fail without throwing an exception. In this case an empty object will be produced. This flag is optional and defaults to false.

**`memory::desc dst_desc() const`**

Returns a destination memory descriptor.

**Return** Destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination parameter.

**`memory::desc diff_src_desc() const`**

Returns a diff source memory descriptor.

**Return** Diff source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a diff source memory with.

**`memory::desc diff_dst_desc() const`**

Returns a destination memory descriptor.

**Return** Destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination parameter.

## 7.5.19 Sum

The sum primitive sums  $N$  tensors. The variable names follow the standard *Conventions*.

$$\text{dst}(\bar{x}) = \sum_{i=1}^N \text{scales}(i) \cdot \text{src}_i(\bar{x})$$

The sum primitive does not have a notion of forward or backward propagations. The backward propagation for the sum operation is simply an identity operation.

### 7.5.19.1 Execution Arguments

When executed, the inputs and outputs should be mapped to an execution argument index as specified by the following table.

primitive input/output	execution argument index
src	<code>DNNL_ARG_MULTIPLE_SRC</code>
dst	<code>DNNL_ARG_DST</code>

### 7.5.19.2 Operation Details

- The dst memory format can be either specified by a user or derived the most appropriate one by the primitive. The recommended way is to allow the primitive to choose the appropriate format.
- The sum primitive requires all source and destination tensors to have the same shape. Implicit broadcasting is not supported.

### 7.5.19.3 Post-ops and Attributes

The sum primitive does not support any post-ops or attributes.

### 7.5.19.4 Data Types Support

The sum primitive supports arbitrary data types for source and destination tensors.

### 7.5.19.5 Data Representation

#### 7.5.19.5.1 Sources, Destination

The sum primitive works with arbitrary data tensors. There is no special meaning associated with any logical dimensions.

### 7.5.19.6 API

```
struct dnnl::sum : public dnnl::primitive
```

Out-of-place summation (sum) primitive.

#### Public Functions

##### sum()

Default constructor. Produces an empty object.

##### sum(**const** primitive\_desc &pd)

Constructs a sum primitive.

#### Parameters

- pd: Primitive descriptor for sum primitive.

```
struct primitive_desc : public dnnl::primitive_desc_base
```

Primitive descriptor for a sum primitive.

#### Public Functions

##### primitive\_desc()

Default constructor. Produces an empty object.

##### primitive\_desc(**const** memory::desc &dst, **const** std::vector<float> &scales, **const** std::vector<memory::desc> &srcs, **const** engine &aengine, **const** primitive\_attr &attr = primitive\_attr())

Constructs a primitive descriptor for a sum primitive.

#### Parameters

- dst: Destination memory descriptor.
- scales: Vector of scales to multiply data in each source memory by.
- srcs: Vector of source memory descriptors.
- aengine: Engine to perform the operation on.
- attr: Primitive attributes to use (optional).

---

```
primitive_desc (const std::vector<float> &scales, const std::vector<memory::desc> &srcs,
               const engine &aengine, const primitive_attr &attr = primitive_attr())
```

Constructs a primitive descriptor for a sum primitive.

This version derives the destination memory descriptor automatically.

#### Parameters

- scales: Vector of scales by which to multiply data in each source memory object.
- srcs: Vector of source memory descriptors.
- aengine: Engine on which to perform the operation.
- attr: Primitive attributes to use (optional).

```
memory::desc src_desc (int idx = 0) const
```

Returns a source memory descriptor.

**Return** Source memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a source parameter with index pdx.

#### Parameters

- idx: Source index.

```
memory::desc dst_desc () const
```

Returns a destination memory descriptor.

**Return** Destination memory descriptor.

**Return** A zero memory descriptor if the primitive does not have a destination parameter.

## 7.6 Open Source Implementation

Intel has published an [open source implementation](#) with the Apache license.

## 7.7 Implementation Notes

This specification provides high-level descriptions for oneDNN operations and does not cover all the implementation-specific details of the [open source implementation](#). Specifically, it does not cover highly-optimized memory formats and integration with profiling tools, etc. This is done intentionally to improve specification portability. Code that uses API defined in this specification is expected to be portable across open source implementation and any potential other implementations of this specification to a reasonable extent.

In the future this section will be extended with more details on how different implementations of this specification should cooperate and co-exist.

## 7.8 Testing

Intel's binary distribution of oneDNN contains example code that you can be used to test library functionality.

The [open source implementation](#) includes a comprehensive test suite. Consult the [README](#) for directions.

## 8.1 Introduction

The oneAPI Collective Communications Library (oneCCL) provides primitives for the communication patterns that occur in deep learning applications. oneCCL supports both scale-up for platforms with multiple oneAPI devices and scale-out for clusters with multiple compute nodes.

oneCCL supports the following communication patterns used in deep learning (DL) algorithms:

- allgatherv
- allreduce
- alldtoall
- alldtoallv
- broadcast
- reduce
- reduce\_scatter

oneCCL exposes controls over additional optimizations and capabilities such as:

- Prioritization for communication operations
- Persistent communication operations (enables decoupling one-time initialization and repetitive execution)
- Unordered communication operations

## 8.2 Definitions

### 8.2.1 oneCCL Concepts

oneCCL specification defines the following list of concepts:

- *Environment*
- *Key-Value Store*
- *Communicator*
- *Device Communicator*
- *Request*
- *Event*

- *Stream*
- *Operation Attributes*

### 8.2.1.1 Environment

oneCCL specification defines `environment` class that shall provide a singleton object and helper methods to manage other oneCCL objects, such as communicators, attributes, and so on.

Retrieves the environment singleton object and initializes the library.

```
static environment& environment::instance();
```

**return** `environment` an environment object

### 8.2.1.2 Key-Value Store

`kvs_interface` defines the key-value store (KVS) interface to be used to connect the ranks during the creation of oneCCL communicator. The interface shall include blocking `get` and `set` methods.

Getting a record from the key-value store:

```
virtual vector_class<char> kvs_interface::get(
    const string_class& prefix,
    const string_class& key) const = 0;
```

**prefix** the prefix that allows operations with KVS in a separate namespace to avoid key names conflicts with the existing KVS records

**key** the key at which the value should be stored

**return** `vector_class<char>` the value associated with the given key

Saving a record in the key-value store:

```
void kvs_interface::set(
    const string_class& prefix,
    const string_class& key,
    const vector_class<char>& data) const = 0;
```

**prefix** the prefix that allows operations with KVS in a separate namespace to avoid key names conflicts with the existing KVS records

**key** the key at which the value should be stored

**data** the value that should be associated with the given key

oneCCL specification defines `kvs` class as a built-in KVS provided by oneCCL.

```
class kvs : public kvs_interface {

public:

    static constexpr size_t addr_max_size = 256;
    using addr_t = array_class<char, addr_max_size>;

    const addr_t& get_addr() const;
```

(continues on next page)

(continued from previous page)

```

~kvs() override;

vector_class<char> get(
    const string_class& prefix,
    const string_class& key) const override;

void set(
    const string_class& prefix,
    const string_class& key,
    const vector_class<char>& data) const override;

}

```

Retrieving an address of built-in key-value store:

```
const addr_t& kvs::get_addr() const;
```

**return kvs::addr\_t**

the address of the key-value store

should be retrieved from the main built-in KVS and distributed to other processes for the built-in KVS creation

The environment class shall provide the ability to create an instance of kvs class.

Creating a main built-in key-value store. Its address should be distributed using an out-of-band communication mechanism and be used to create key-value stores on other ranks:

```
kvs_t environment::create_main_kvs() const;
```

**return kvs\_t** the main key-value store object

Creating a new key-value store from main kvs address:

```
kvs_t environment::create_kvs(const kvs::addr_t& addr) const;
```

**addr** the address of the main kvs

**return kvs\_t** key-value store object

### 8.2.1.3 Communicator

oneCCL specification defines `communicator` class that describes a group of communicating ranks, where rank is a single process. `communicator` defines collective operations on host memory buffers.

The environment class shall provide the ability to create an instance of `communicator` class.

Creating a new host communicator with user-supplied size, rank, and kvs:

```

using communicator_t = unique_ptr_class<communicator>;

communicator_t environment::create_communicator(
    const size_t size,
    const size_t rank,
    shared_ptr_class<kvs_interface> kvs) const;

```

**size** user-supplied total number of ranks

**rank** user-supplied rank

**kvs** key-value store for ranks wire-up

**return communicator\_t** communicator object

communicator shall provide methods to retrieve the rank that corresponds to the communicator object and the number of ranks in the communicator. It shall also provide collective communication operations on host memory buffers.

Retrieving the rank in a communicator:

```
size_t communicator::rank() const;
```

**return size\_t** rank of the current process

Retrieving the number of ranks in a communicator:

```
size_t communicator::size() const;
```

**return size\_t** number of the ranks

---

**Note:** See also: *Collective Operations*

---

#### 8.2.1.4 Device Communicator

oneCCL specification defines device\_communicator class that describes a group of communicating ranks, where rank is a single device. device\_communicator defines collective operations on device memory buffers.

---

**Note:** Here and below, a device, a device memory, a device context, a queue, and an event are defined in the scope of SYCL device runtime.

---

The environment class shall provide the ability to create an instance of device\_communicator class.

Creating a new device communicator with user-supplied size, rank, and kvs:

```
using device_communicator_t = unique_ptr_class<device_communicator>;
using native_device_type = sycl::device;
using native_context_type = sycl::context;

vector_class<device_communicator_t> environment::create_device_communicators(
    const size_t size,
    vector_class<pair_class<size_t, native_device_type>>& rank_device_map,
    native_context_type& context,
    shared_ptr_class<kvs_interface> kvs) const;
```

**size** user-supplied total number of ranks

**rank\_device\_map** user-supplied mapping of local ranks on devices

**context** device context

**kvs** key-value store for ranks wire-up

**return vector\_class<device\_communicator\_t>** a vector of device communicators

`device_communicator` shall provide methods to retrieve the rank, the device, and the device context that correspond to the communicator object as well as the number of ranks in the communicator. It shall also provide collective communication operations on device memory buffers.

Retrieving the rank in a communicator:

```
size_t device_communicator::rank() const;
```

**return size\_t** the rank that corresponds to the communicator object

Retrieving the number of ranks in a communicator:

```
size_t device_communicator::size() const;
```

**return size\_t** the number of the ranks

Retrieving an underlying device, which was used as communicator construction argument:

```
native_device_type get_device() const;
```

**return native\_device\_type** the device that corresponds to the communicator object

Retrieving an underlying device context, which was used as communicator construction argument:

```
native_context_type get_context() const;
```

**return native\_context\_type** the device context that corresponds to the communicator object

---

**Note:** See also: *Collective Operations*

---

### 8.2.1.5 Request

Each communication operation of oneCCL shall return a request object for tracking the operation's progress.

---

**Note:** See also: *Operation Progress Tracking*

---

### 8.2.1.6 Event

oneCCL specification defines the `event` class that wraps the `sycl::event` object and encapsulates synchronization context for `device_communicator` communication operations.

A vector of events may be passed to the `device_communicator` communication operation to designate input dependencies for the operation. An event may be retrieved from `request` object to be passed further as an input dependency in other device communication operations or in computation operations.

The `environment` class shall provide the ability to create an instance of the `event` class from the `sycl::event` object.

Creating a new event from `sycl::event` object:

```
using native_event_type = sycl::event;
using event_t = unique_ptr_class<event>;
event_t environment::create_event(native_event_type& native_event) const;
```

**native\_event** the existing native handle for an event

**return event\_t** an event object

### 8.2.1.7 Stream

oneCCL specification defines stream class that wraps `sycl::queue` object and encapsulates execution context for `device_communicator` communication operations.

Stream shall be passed to `device_communicator` communication operation.

The environment class shall provide the ability to create an instance of the stream class from the `sycl::queue` object.

Creating a new stream from `sycl::queue` object:

```
using native_stream_type = sycl::queue;
using stream_t = unique_ptr_class<stream>;
stream_t environment::create_stream(native_stream_type& native_stream) const;
```

**native\_stream** the existing native handle for a stream

**return stream\_t** a stream object

### 8.2.1.8 Operation Attributes

Communication operation behavior may be controlled through operation attributes.

*Operation Attributes*

## 8.2.2 Communication Operations

This section covers communication operations defined by oneCCL specification.

### 8.2.2.1 Datatypes

oneCCL specification defines the following datatypes that may be used for communication operations:

```
enum class datatype : int
{
    int8          = /* unspecified */,
    uint8         = /* unspecified */,
    int16         = /* unspecified */,
    uint16        = /* unspecified */,
    int32         = /* unspecified */,
    uint32        = /* unspecified */,
    int64         = /* unspecified */,
    uint64        = /* unspecified */,

    float16       = /* unspecified */,
    float32       = /* unspecified */,
    float64       = /* unspecified */,
    bfloat16      = /* unspecified */,

    last_predefined = /* unspecified, equal to the largest of all the values above */
};
```

**datatype::int8** 8 bits signed integer

**datatype::uint8** 8 bits unsigned integer  
**datatype::int16** 16 bits signed integer  
**datatype::uint16** 16 bits unsigned integer  
**datatype::int32** 32 bits signed integer  
**datatype::uint32** 32 bits unsigned integer  
**datatype::int64** 64 bits signed integer  
**datatype::uint64** 64 bits unsigned integer  
**datatype::float16** 16-bit/half-precision floating point  
**datatype::float32** 32-bit/single-precision floating point  
**datatype::float64** 64-bit/double-precision floating point  
**datatype::bfloat16** non-standard 16-bit floating point with 7-bit mantissa

### 8.2.2.1.1 Custom Datatypes

oneCCL specification defines the way to register and deregister a custom datatype using the `datatype_attr` attribute object.

The list of identifiers that may be used to fill an attribute object:

```
enum class datatype_attr_id {
    size = /* unspecified */
};
```

**datatype\_attr\_id::size** the size of the datatype in bytes

The `environment` class shall provide the ability to create a datatype attribute object, to register the datatype based on an attribute object, and to deregister the datatype.

Creating a datatype attribute object, which may be used to register custom datatype:

```
using datatype_attr_t = unique_ptr<class<datatype_attr>>;
datatype_attr_t environment::create_datatype_attr() const;
```

**return datatype\_attr\_t** an object containing attributes for the custom datatype

Registering a custom datatype to be used in communication operations:

```
datatype environment::register_datatype(const datatype_attr_t& attr);
```

**attr** the datatype's attributes

**return datatype** the handle for the custom datatype

Deregistering a custom datatype:

```
void environment::deregister_datatype(datatype dtype);
```

**dtype** the handle for the custom datatype

Retrieving a datatype size in bytes:

```
size_t environment::get_datatype_size(datatype dtype) const;
```

**dtype** the datatype's handle  
**return size\_t** datatype size in bytes

### 8.2.2.2 Reductions

oneCCL specification defines the following reduction operations for *Allreduce*, *Reduce* and *ReduceScatter* collective operations:

```
enum class reduction
{
    sum      = /* unspecified */,
    prod     = /* unspecified */,
    min      = /* unspecified */,
    max      = /* unspecified */,
    custom   = /* unspecified */
};
```

**reduction::sum** elementwise summation

**reduction::prod** elementwise multiplication

**reduction::min** elementwise min

**reduction::max** elementwise max

**reduction::custom**

specify user-defined reduction operation

the actual reduction function must be passed through `reduction_fn` operation attribute

*Operation Attributes*

### 8.2.2.3 Collective Operations

oneCCL specification defines the following collective communication operations:

- *Allgatherv*
- *Allreduce*
- *Alltoall*
- *Alltoallv*
- *Barrier*
- *Broadcast*
- *Reduce*
- *ReduceScatter*

These operations are collective, meaning that all participants (ranks) of oneCCL communicator should make a call. The order of collective operation calls should be the same across all ranks, unless the `match_id` attribute is used to match calls done without a strict order.

`communicator` shall provide the ability to perform collective communication operations on host memory buffers.

`device_communicator` shall provide the ability to perform collective communication operations on device memory buffers. Additionally, device collective operations shall accept the device's execution context and a vector of

events that the device collective operation should depend on. The output `request` object shall provide the ability to retrieve the `event` object that is signaled when the collective operation completes on the device.

`BufferType` is used below to define the C++ type of elements in data buffers (`buf`, `send_buf` and `recv_buf`) of collective operations. At least the following types shall be supported: `[u]int{8/16/32/64}_t`, `float`, `double`. The explicit datatype parameter shall be used to enable data types which cannot be inferred from the function arguments.

---

**Note:** See also: *Custom Datatypes*

---

If the arguments provided to a collective operation call do not comply to the requirements of the operation, the behavior is undefined unless it is specified otherwise.

```
using request_t = unique_ptr<class<request>,>;
```

### 8.2.2.3.1 Allgatherv

Allgatherv is a collective communication operation that collects data from all the ranks within a communicator into a single buffer. Different ranks may contribute segments of different sizes. The resulting data in the output buffer must be the same for each rank.

```
template<class BufferType>
request_t communicator::allgatherv(const BufferType* send_buf,
                                     size_t send_count,
                                     BufferType* recv_buf,
                                     const vector<class<size_t>>& recv_counts,
                                     const allgatherv_attr_t& attr = allgatherv_attr_
                                     ↵t());
                                     ↵t());

request_t communicator::allgatherv(const void* send_buf,
                                     size_t send_count,
                                     void* recv_buf,
                                     const vector<class<size_t>>& recv_counts,
                                     datatype dtype,
                                     const allgatherv_attr_t& attr = allgatherv_attr_
                                     ↵t());
```

```
template<class BufferType>
request_t device_communicator::allgatherv(const BufferType* send_buf,
                                            size_t send_count,
                                            BufferType* recv_buf,
                                            const vector<class<size_t>>& recv_counts,
                                            const stream_t& stream,
                                            const vector<class<event_t>>& deps = {},
                                            const allgatherv_attr_t& attr = allgatherv_
                                            ↵attr_t());
                                            ↵attr_t();

request_t device_communicator::allgatherv(const void* send_buf,
                                            size_t send_count,
                                            void* recv_buf,
                                            const vector<class<size_t>>& recv_counts,
                                            datatype dtype,
                                            const stream_t& stream,
```

(continues on next page)

(continued from previous page)

```
const vector<event_t>& deps = {},
const allgatherv_attr_t& attr = allgatherv_
attr_t());
```

**send\_buf** the buffer with `send_count` elements of `BufferType` that stores local data to be gathered

**send\_count** the number of elements of type `BufferType` in `send_buf`

**recv\_buf [out]** the buffer to store the gathered result, must be large enough to hold values from all ranks

#### recv\_counts

an array with the number of elements of type `BufferType` to be received from each rank

the array's size must be equal to the number of ranks

the values in the array are expected to be the same for all ranks

the value at the position of the caller's rank must be equal to `send_count`

#### dtype

the datatype of elements in `send_buf` and `recv_buf`

must be skipped if `BufferType` can be inferred

otherwise must be passed explicitly

#### stream

the stream associated with the operation

relevant for device communicator

#### deps

an optional vector of the events that the operation should depend on

relevant for device communicator

**attr** optional attributes to customize the operation

**return request\_t** an object to track the progress of the operation

### 8.2.2.3.2 Allreduce

Allreduce is a collective communication operation that performs the global reduction operation on values from all ranks of communicator and distributes the result back to all ranks.

```
template <class BufferType>
request_t communicator::allreduce(const BufferType* send_buf,
                                  BufferType* recv_buf,
                                  size_t count,
                                  reduction reduction,
                                  const allreduce_attr_t& attr = allreduce_attr_t());

request_t communicator::allreduce(const void* send_buf,
                                  void* recv_buf,
                                  size_t count,
                                  reduction reduction,
                                  datatype dtype,
                                  const allreduce_attr_t& attr = allreduce_attr_t());
```

```
template <class BufferType>
request_t device_communicator::allreduce(const BufferType* send_buf,
                                         BufferType* recv_buf,
                                         size_t count,
                                         reduction reduction,
                                         const stream_t& stream,
                                         const vector_class<event_t>& deps = {},
                                         const allreduce_attr_t& attr = allreduce_
                                         ↪attr_t());

request_t device_communicator::allreduce(const void* send_buf,
                                         void* recv_buf,
                                         size_t count,
                                         reduction reduction,
                                         datatype dtype,
                                         const stream_t& stream,
                                         const vector_class<event_t>& deps = {},
                                         const allreduce_attr_t& attr = allreduce_
                                         ↪attr_t());
```

**send\_buf** the buffer with **count** elements of **BufferType** that stores local data to be reduced

**recv\_buf [out]** the buffer to store the reduced result, must have the same dimension as **send\_buf**

**count** the number of elements of type **BufferType** in **send\_buf** and **recv\_buf**

**reduction** the type of the reduction operation to be applied

**dtype**

the datatype of elements in **send\_buf** and **recv\_buf**  
 must be skipped if **BufferType** can be inferred  
 otherwise must be passed explicitly

**stream**

the stream associated with the operation  
 relevant for device communicator

**deps**

an optional vector of the events that the operation should depend on  
 relevant for device communicator

**attr** optional attributes to customize the operation

**return request\_t** an object to track the progress of the operation

### 8.2.2.3 Alltoall

Alltoall is a collective communication operation in which each rank sends separate equal-sized blocks of data to each rank. The j-th block of send buffer sent from the i-th rank is received by the j-th rank and is placed in the i-th block of receive buffer.

```
template <class BufferType>
request_t communicator::alltoall(const BufferType* send_buf,
                                BufferType* recv_buf,
                                size_t count,
                                const alltoall_attr_t& attr = alltoall_attr_t());
```

(continues on next page)

(continued from previous page)

```
request_t communicator::alltoall(const void* send_buf,
                                void* recv_buf,
                                size_t count,
                                datatype dtype,
                                const alltoall_attr_t& attr = alltoall_attr_t());
```

```
template <class BufferType>
request_t device_communicator::alltoall(const BufferType* send_buf,
                                         BufferType* recv_buf,
                                         size_t count,
                                         const stream_t& stream,
                                         const vector_class<event_t>& deps = {},
                                         const alltoall_attr_t& attr = alltoall_attr_
                                         ↵t());
request_t device_communicator::alltoall(const void* send_buf,
                                         void* recv_buf,
                                         size_t count,
                                         datatype dtype,
                                         const stream_t& stream,
                                         const vector_class<event_t>& deps = {},
                                         const alltoall_attr_t& attr = alltoall_attr_
                                         ↵t());
```

**send\_buf** the buffer with **count** elements of **BufferType** that stores local data to be sent

#### **recv\_buf [out]**

the buffer to store the received result, must be large enough  
to hold values from all ranks, i.e. at least **comm\_size \* count**

**count** the number of elements to be send to or to received from each rank

#### **dtype**

the datatype of elements in **send\_buf** and **recv\_buf**  
must be skipped if **BufferType** can be inferred  
otherwise must be passed explicitly

#### **stream**

the stream associated with the operation  
relevant for device communicator

#### **deps**

an optional vector of the events that the operation should depend on  
relevant for device communicator

**attr** optional attributes to customize the operation

**return request\_t** an object to track the progress of the operation

### 8.2.2.3.4 Alltoallv

Alltoallv is a collective communication operation in which each rank sends separate blocks of data to each rank. Block sizes may differ. The j-th block of send buffer sent from the i-th rank is received by the j-th rank and is placed in the i-th block of receive buffer.

```
template <class BufferType>
request_t communicator::alltoallv(const BufferType* send_buf,
                                    const vector<size_t>& send_counts,
                                    BufferType* recv_buf,
                                    const vector<size_t>& recv_counts,
                                    const alltoallv_attr_t& attr = alltoallv_attr_t());

request_t communicator::alltoallv(const void* send_buf,
                                    const vector<size_t>& send_counts,
                                    void* recv_buf,
                                    const vector<size_t>& recv_counts,
                                    datatype dtype,
                                    const alltoallv_attr_t& attr = alltoallv_attr_t());
```

```
template <class BufferType>
request_t device_communicator::alltoallv(const BufferType* send_buf,
                                          const vector<size_t>& send_counts,
                                          BufferType* recv_buf,
                                          const vector<size_t>& recv_counts,
                                          const stream_t& stream,
                                          const vector<event_t>& deps = {},
                                          const alltoallv_attr_t& attr = alltoallv_
                                          ↪attr_t());

request_t device_communicator::alltoallv(const void* send_buf,
                                          const vector<size_t>& send_counts,
                                          void* recv_buf,
                                          const vector<size_t>& recv_counts,
                                          datatype dtype,
                                          const stream_t& stream,
                                          const vector<event_t>& deps = {},
                                          const alltoallv_attr_t& attr = alltoallv_
                                          ↪attr_t());
```

**send\_buf** the buffer with elements of `BufferType` that stores local blocks to be sent to each rank

**send\_counts**

an array with number of elements of type `BufferType` in the blocks sent for each rank

the array's size must be equal to the number of ranks

the values at the position of the caller's rank in `send_counts` and `recv_counts` must be equal

**recv\_buf [out]** the buffer to store the received result, must be large enough to hold blocks from all ranks

**recv\_counts**

an array with number of elements of type `BufferType` in the blocks received from each rank

the array's size must be equal to the number of ranks

the values at the position of the caller's rank in `send_counts` and `recv_counts` must be equal

**dtype**

the datatype of elements in `send_buf` and `recv_buf`

must be skipped if `BufferType` can be inferred  
 otherwise must be passed explicitly

**stream**

the stream associated with the operation  
 relevant for device communicator

**deps**

an optional vector of the events that the operation should depend on  
 relevant for device communicator

**attr** optional attributes to customize the operation

**return** `request_t` an object to track the progress of the operation

### 8.2.2.3.5 Barrier

Barrier synchronization is performed across all ranks of the communicator and it is completed only after all the ranks in the communicator have called it.

```
request_t communicator::barrier(const barrier_attr_t& attr = barrier_attr_t());
```

```
request_t device_communicator::barrier(const stream_t& stream,
                                       const vector_class<event_t>& deps = {},
                                       const barrier_attr_t& attr = barrier_attr_t());
```

**stream**

the stream associated with the operation  
 relevant for device communicator

**deps**

an optional vector of the events that the operation should depend on  
 relevant for device communicator

**attr** optional attributes to customize the operation

**return** `request_t` an object to track the progress of the operation

### 8.2.2.3.6 Broadcast

Broadcast is a collective communication operation that broadcasts data from one rank of communicator (denoted as root) to all other ranks.

```
template <class BufferType>
request_t communicator::bcast(BufferType* buf,
                             size_t count,
                             size_t root,
                             const bcast_attr_t& attr = bcast_attr_t());

request_t communicator::bcast(void* buf,
                            size_t count,
                            datatype dtype,
                            size_t root,
                            const bcast_attr_t& attr = bcast_attr_t());
```

```
template <class BufferType>
request_t device_communicator::bcast(BufferType* buf,
                                       size_t count,
                                       size_t root,
                                       const stream_t& stream,
                                       const vector_class<event_t>& deps = {},
                                       const bcast_attr_t& attr = bcast_attr_t());

request_t device_communicator::bcast(void* buf,
                                       size_t count,
                                       datatype dtype,
                                       size_t root,
                                       const stream_t& stream,
                                       const vector_class<event_t>& deps = {},
                                       const bcast_attr_t& attr = bcast_attr_t());
```

**buf [in,out]**

the buffer with **count** elements of **BufferType**  
 serves as **send\_buf** for **root** and as **recv\_buf** for other ranks

**count** the number of elements of type **BufferType** in **buf**

**root** the rank that broadcasts **buf**

**dtype**

the datatype of elements in **buf**  
 must be skipped if **BufferType** can be inferred  
 otherwise must be passed explicitly

**stream**

the stream associated with the operation  
 relevant for device communicator

**deps**

an optional vector of the events that the operation should depend on  
 relevant for device communicator

**attr** optional attributes to customize the operation

**return request\_t** an object to track the progress of the operation

### 8.2.2.3.7 Reduce

Reduce is a collective communication operation that performs the global reduction operation on values from all ranks of the communicator and returns the result to the root rank

```
template <class BufferType>
request_t communicator::reduce(const BufferType* send_buf,
                               BufferType* recv_buf,
                               size_t count,
                               reduction reduction,
                               size_t root,
                               const reduce_attr_t& attr = reduce_attr_t());
```

(continues on next page)

(continued from previous page)

```
request_t communicator::reduce(const void* send_buf,
                             void* recv_buf,
                             size_t count,
                             datatype dtype,
                             reduction reduction,
                             size_t root,
                             const reduce_attr_t& attr = reduce_attr_t());
```

```
template <class BufferType>
request_t device_communicator::reduce(const BufferType* send_buf,
                                      BufferType* recv_buf,
                                      size_t count,
                                      reduction reduction,
                                      size_t root,
                                      const stream_t& stream,
                                      const vector_class<event_t>& deps = {},
                                      const reduce_attr_t& attr = reduce_attr_t());

request_t device_communicator::reduce(const void* send_buf,
                                      void* recv_buf,
                                      size_t count,
                                      datatype dtype,
                                      reduction reduction,
                                      size_t root,
                                      const stream_t& stream,
                                      const vector_class<event_t>& deps = {},
                                      const reduce_attr_t& attr = reduce_attr_t());
```

**send\_buf** the buffer with **count** elements of **BufferType** that stores local data to be reduced

#### **recv\_buf [out]**

the buffer to store the reduced result, must have the same dimension as **send\_buf**.

Used by the **root** rank only, ignored by other ranks.

**count** the number of elements of type **BufferType** in **send\_buf** and **recv\_buf**

**reduction** the type of the reduction operation to be applied

**root** the rank that gets the result of the reduction

#### **dtype**

the datatype of elements in **send\_buf** and **recv\_buf**

must be skipped if **BufferType** can be inferred

otherwise must be passed explicitly

#### **stream**

the stream associated with the operation

relevant for device communicator

#### **deps**

an optional vector of the events that the operation should depend on

relevant for device communicator

**attr** optional attributes to customize the operation

**return request\_t** an object to track the progress of the operation

### 8.2.2.3.8 ReduceScatter

Reduce-scatter is a collective communication operation that performs the global reduction operation on values from all ranks of the communicator and scatters the result in blocks back to all ranks.

```
template <class BufferType>
request_t communicator::reduce_scatter(const BufferType* send_buf,
                                         BufferType* recv_buf,
                                         size_t recv_count,
                                         reduction reduction,
                                         const reduce_scatter_attr_t& attr = reduce_
                                         ↪scatter_attr_t());

request_t communicator::reduce_scatter(const void* send_buf,
                                         void* recv_buf,
                                         size_t recv_count,
                                         datatype dtype,
                                         reduction reduction,
                                         const reduce_scatter_attr_t& attr = reduce_
                                         ↪scatter_attr_t());
```

```
template <class BufferType>
request_t device_communicator::reduce_scatter(const BufferType* send_buf,
                                               BufferType* recv_buf,
                                               size_t recv_count,
                                               reduction reduction,
                                               const stream_t& stream,
                                               const vector<class<event_t>>& deps = {},
                                               const reduce_scatter_attr_t& attr =_
                                               ↪reduce_scatter_attr_t()));

request_t device_communicator::reduce_scatter(const void* send_buf,
                                              void* recv_buf,
                                              size_t recv_count,
                                              datatype dtype,
                                              reduction reduction,
                                              const stream_t& stream,
                                              const vector<class<event_t>>& deps = {},
                                              const reduce_scatter_attr_t& attr =_
                                              ↪reduce_scatter_attr_t());
```

**send\_buf** the buffer with `comm_size * count` elements of `BufferType` that stores local data to be reduced

**recv\_buf [out]** the buffer to store the result block containing `recv_count` elements of type `BufferType`

**recv\_count** the number of elements of type `BufferType` in the received block

**reduction** the type of the reduction operation to be applied

**dtype**

the datatype of elements in `send_buf` and `recv_buf`

must be skipped if `BufferType` can be inferred

otherwise must be passed explicitly

**stream**

the stream associated with the operation

relevant for device communicator

**deps**

an optional vector of the events that the operation should depend on  
relevant for device communicator

**attr** optional attributes to customize the operation

**return request\_t** an object to track the progress of the operation

**Note:** See also:

- *Communicator*
- *Device Communicator*
- *Request*
- *Stream*
- *Event*
- *Operation Progress Tracking*

#### 8.2.2.4 Operation Attributes

oneCCL specification defines collective attributes that serve as modifiers of a collective operation's behavior. Optionally, they may be passed to the corresponding collective operations.

- allgatherv\_attr
- allreduce\_attr
- alltoall\_attr
- alltoallv\_attr
- barrier\_attr
- bcast\_attr
- reduce\_attr
- reduce\_scatter\_attr

oneCCL specification defines attribute identifiers that may be used to fill collective attribute objects.

The list of attribute identifiers that may be used for any communication operation:

```
enum class operation_attr_id {
    priority      = /* unspecified */,
    to_cache      = /* unspecified */,
    synchronous   = /* unspecified */,
    match_id      = /* unspecified */
};
```

**operation\_attr\_id::priority** the priority of the communication operation

**operation\_attr\_id::to\_cache**

persistent/non-persistent communication operation  
should be used in conjunction with `match_id`

**operation\_attr\_id::synchronous** synchronous/asynchronous communication operation

**operation\_attr\_id::match\_id**

the unique identifier of the operation

enables correct matching and execution of the operations started in different order on different ranks  
in conjunction with `to_cache`, it also enables the caching of the communication operation

The list of attribute identifiers that may be used for `Allreduce`, `Reduce` and `ReduceScatter` collective operations:

```
enum class allreduce_attr_id {
    reduction_fn = /* unspecified */
};

enum class reduce_attr_id {
    reduction_fn = /* unspecified */
};

enum class reduce_scatter_attr_id {
    reduction_fn = /* unspecified */
};
```

`allreduce_attr_id::reduction_fn / reduce_attr_id::reduction_fn / reduce_scatter_attr_id::reduction_fn` a function pointer for the custom reduction operation that follows the signature:

```
typedef void (*ccl_reduction_fn_t)
(
    const void*,      // in_buf
    size_t,           // in_count
    void*,            // inout_buf
    size_t*,          // out_count
    datatype,         // datatype
    const fn_context* // context
);

typedef struct {
    const char* match_id;
    const size_t offset;
} fn_context;
```

The environment class shall provide the ability to create an attribute object for a collective communication operation.

Creating an operation attribute object, which may be used in a corresponding collective communication operation:

```
using allgatherv_attr_t = unique_ptr_class<allgatherv_attr>;
using allreduce_attr_t = unique_ptr_class<allreduce_attr>;
using alltoall_attr_t = unique_ptr_class<alltoall_attr>;
using alltoallv_attr_t = unique_ptr_class<alltoallv_attr>;
using barrier_attr_t = unique_ptr_class<barrier_attr>;
using bcast_attr_t = unique_ptr_class<bcast_attr>;
using reduce_attr_t = unique_ptr_class<reduce_attr>;
using reduce_scatter_attr_t = unique_ptr_class<reduce_scatter_attr>;

allgatherv_attr_t environment::create_allgatherv_attr() const;
allreduce_attr_t environment::create_allreduce_attr() const;
alltoall_attr_t environment::create_alltoall_attr() const;
alltoallv_attr_t environment::create_alltoallv_attr() const;
barrier_attr_t environment::create_barrier_attr() const;
bcast_attr_t environment::create_bcast_attr() const;
```

(continues on next page)

(continued from previous page)

```
reduce_attr_t environment::create_reduce_attr() const;
reduce_scatter_attr_t environment::create_reduce_scatter_attr() const;
```

**return <coll\_name>\_attr\_t** an object containing attributes for a specific collective communication operation

### 8.2.2.5 Operation Progress Tracking

oneCCL communication operation shall return a request object to be used for tracking the operation's progress.

The `request` class shall provide the ability to wait for completion of an operation in a blocking manner, the ability to check the completion status in a non-blocking manner, and the ability to retrieve the underlying `event` object that is signaled when the device communication operation completes on the device.

#### 8.2.2.5.1 Request

```
using request_t = unique_ptr<class<request>>;
using event_t = unique_ptr<class<event>>;
```

Waiting for the completion of an operation in a blocking manner:

```
void request::wait();
```

Checking for the completion of an operation in a non-blocking manner:

```
bool request::test();
```

**return bool** true if the operation has been completed false if the operation has not been completed

Retrieving an event object that is signaled when the device communication operation completes on the device:

```
event_t request::get_event() const;
```

**return event\_t** an event object that is signaled when the communication operation completes on the device

### 8.2.3 Error Handling

oneCCL error handling relies on the mechanism of C++ exceptions. If an error occurs, it shall be propagated at the point of a function call where it is caught using standard C++ error handling mechanism.

oneCCL specification defines `error` as a type for exceptions that may be thrown by oneCCL API.

```
class error : public std::runtime_error {
    ...
}
```

## 8.3 Programming Model

### 8.3.1 Generic Workflow

Below is a generic workflow with oneCCL API

1. Initialize the library:

```
auto env = environment::instance();
```

2. Create a main built-in key-value store. Its address should be distributed using an out-of-band communication mechanism and be used to create key-value stores on other processes:

```
/* for example use MPI as an out-of-band communication mechanism */
int mpi_rank, mpi_size;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);

kvs_interface_t kvs;
kvs::addr_t kvs_addr;

if (mpi_rank == 0) {
    kvs = env.create_main_kvs();
    kvs_addr = kvs->get_addr();
    MPI_Bcast((void*)kvs_addr.data(), kvs::addr_max_size, MPI_BYTE, 0, MPI_COMM_
    ↵WORLD);
}
else {
    MPI_Bcast((void*)kvs_addr.data(), kvs::addr_max_size, MPI_BYTE, 0, MPI_COMM_
    ↵WORLD);
    kvs = env.create_kvs(kvs_addr);
}
```

3. Create a host communicator or device communicator(s):

```
/* for host communications */
auto comm = env.create_communicator(mpi_size, kvs);
```

```
/* for device communications, for example with multiple devices per process */

/* devices -> vector<sycl::device> */
/* queues -> vector<sycl::queue> */
/* ctx -> sycl::context */
auto comms = env.create_device_communicators(mpi_size * devices.size(), devices, ctx, ↵
    ↵kvs);

/* rank assignment will happen automatically, the actual rank can be retrieved using ↵
    ↵comm->rank() */

/* create ccl::stream objects from sycl::queue objects */
std::vector<request_t> streams;
for (auto& comm : comms) {
    streams.push_back(env.create_stream(sycl_queues[comm->rank()]));
}
```

4. Execute a collective operation of choice on the communicator(s):

```
/* for host communications */
auto request = comm->allreduce(...);
request->wait();
```

```
/* for device communications */
std::vector<request_t> reqs;
for (auto& comm : comms) {
    reqs.push_back(comm->allreduce(..., streams[comm->rank()]));
}

for (auto& req : reqs) {
    req->wait();
}
```

## 8.3.2 Host Communication Support

Host memory communications between processes are provided by *Communicator*.

The example below demonstrates the main concepts of communication on host memory buffers.

### 8.3.2.1 Example

Consider a simple oneCCL allreduce example for CPU.

1. Create a communicator object with user-supplied size, rank, and kvs:

```
auto env = environment::instance();
auto comm = env.create_communicator(size, rank, kvs);
```

2. Initialize send\_buf (in real scenario it is supplied by the user):

```
const size_t elem_count = <N>;

/* initialize send_buf */
for (idx = 0; idx < elem_count; idx++) {
    send_buf[idx] = rank + 1;
}
```

3. allreduce invocation performs the reduction of values from all the processes and then distributes the result to all the processes. In this case, the result is an array with elem\_count elements, where all elements are equal to the sum of arithmetical progression:

$$p \cdot (p + 1)/2$$

```
comm->allreduce(send_buf,
                  recv_buf,
                  elem_count,
                  reduction::sum)->wait();
```

4. Check the correctness of allreduce operation:

```

auto comm_size = comm->size();
auto expected = comm_size * (comm_size + 1) / 2;

for (idx = 0; idx < elem_count; idx++) {
    if (recv_buf[idx] != expected) {
        std::cout << "unexpected value at index " << idx << std::endl;
        break;
}
}

```

### 8.3.3 Device Communication Support

Device memory communications between processes are provided by *Device Communicator*.

The example below demonstrates the main concepts of communication on device memory buffers.

#### 8.3.3.1 Example

Consider a simple oneCCL `allreduce` example for GPU:

1. Create oneCCL device communicator objects with user-supplied size, rank  $\leftrightarrow$  SYCL device mapping, SYCL context and kvs:

```

auto env = environment::instance();
auto comms = env.create_device_communicators(size, rank_sycl_dev_map, sycl_ctx, kvs);

```

2. Create oneCCL stream object from user-supplied `sycl::queue` object:

```

auto stream = env.create_stream(sycl_queue);

```

3. Initialize `send_buf` (in real scenario it is supplied by the user):

```

const size_t elem_count = <N>;

/* using SYCL buffer and accessor */
auto send_buf_host_acc = send_buf.get_access<mode::write>();
for (idx = 0; idx < elem_count; idx++) {
    send_buf_host_acc[idx] = rank;
}

```

```

/* or using SYCL USM */
for (idx = 0; idx < elem_count; idx++) {
    send_buf[idx] = rank;
}

```

4. For demonstration purposes, modify the `send_buf` on the GPU side:

```

/* using SYCL buffer and accessor */
sycl_queue.submit([&](cl::sycl::handler& cgh) {
    auto send_buf_dev_acc = send_buf.get_access<mode::write>(cgh);
    cgh.parallel_for<class allreduce_send_buf_modify>(range<1>{elem_count}, [=](item
    <1> idx) {
        send_buf_dev_acc[idx] += 1;
    });
});

```

```
/* or using SYCL USM */
for (idx = 0; idx < elem_count; idx++) {
    send_buf[idx] += 1;
}
```

5. allreduce invocation performs reduction of values from all processes and then distributes the result to all processes. In this case, the result is an array with elem\_count elements, where all elements are equal to the sum of arithmetical progression:

$$p \cdot (p + 1)/2$$

```
std::vector<request_t> reqs;
for (auto& comm : comms) {
    reqs.push_back(comm->allreduce(send_buf,
                                    recv_buf,
                                    elem_count,
                                    reduction::sum,
                                    streams[comm->rank()]));
}

for (auto& req : reqs) {
    req->wait();
}
```

6. Check the correctness of allreduce operation on the GPU:

```
/* using SYCL buffer and accessor */

auto comm_size = comm->size();
auto expected = comm_size * (comm_size + 1) / 2;

sycl_queue.submit([&](handler& cgh) {
    auto recv_buf_dev_acc = recv_buf.get_access<mode::write>(cgh);
    cgh.parallel_for<class allreduce_recv_buf_check>(range<1>{elem_count}, [=](item<1>
    ~ idx) {
        if (recv_buf_dev_acc[idx] != expected) {
            recv_buf_dev_acc[idx] = -1;
        }
    });
});

...
auto recv_buf_host_acc = recv_buf.get_access<mode::read>();
for (idx = 0; idx < elem_count; idx++) {
    if (recv_buf_host_acc[idx] == -1) {
        std::cout << "unexpected value at index " << idx << std::endl;
        break;
    }
}
```

```
/* or using SYCL USM */

auto comm_size = comm->size();
```

(continues on next page)

(continued from previous page)

```
auto expected = comm_size * (comm_size + 1) / 2;

for (idx = 0; idx < elem_count; idx++) {
    if (recv_buf[idx] != expected) {
        std::cout << "unexpected value at index " << idx << std::endl;
        break;
}
```

## LEVEL ZERO

The oneAPI Level Zero (Level Zero) provides low-level direct-to-metal interfaces that are tailored to the devices in a oneAPI platform. Level Zero supports broader language features such as function pointers, virtual functions, unified memory, and I/O capabilities while also providing fine-grain explicit controls needed by higher-level runtime APIs including:

- Device discovery
- Memory allocation
- Peer-to-peer communication
- Inter-process sharing
- Kernel submission
- Asynchronous execution and scheduling
- Synchronization primitives
- Metrics reporting

The API architecture exposes both physical and logical abstractions of the underlying oneAPI platform devices and their capabilities. The device, sub-device, and memory are exposed at a physical level while command queues, events, and synchronization methods are defined as logical entities. All logical entities are bound to device-level physical capabilities. The API provides a scheduling model that is tailored to multiple uses including a low-latency submission model to the devices as well as one that is tailored to the construction and submission of work across simultaneous host threads. While heavily influenced by other low-level APIs, such as OpenCL, Level Zero is designed to evolve independently. While heavily influenced by GPU architecture, Level Zero is supportable across different oneAPI compute device architectures, such as FPGAs.

### 9.1 Detailed API Descriptions

The detailed specification can be found online in the [Level Zero Specification](#).

## ONEDAL

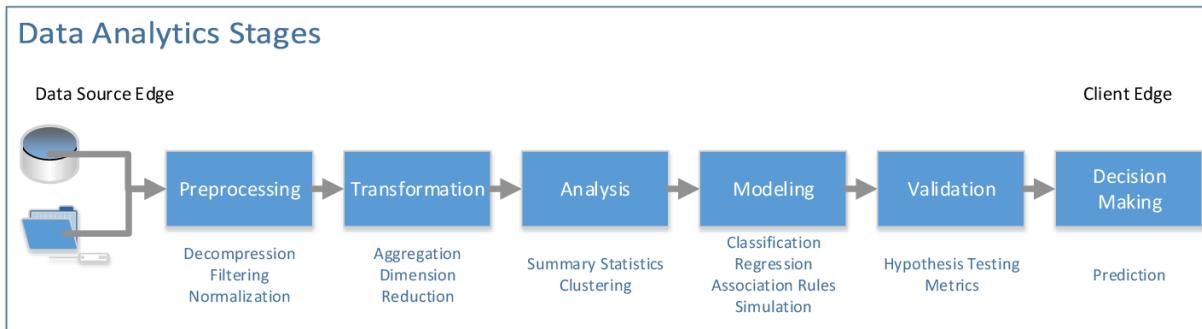
This document specifies requirements for implementations of oneAPI Data Analytics Library (oneDAL).

oneDAL is a library that helps speed up big data analysis by providing highly optimized algorithmic building blocks for all stages of data analytics (preprocessing, transformation, analysis, modeling, validation, and decision making) in batch, online, and distributed processing modes of computation. The current version of oneDAL provides Data Parallel C++ (DPC++) API extensions to the traditional C++ interface.

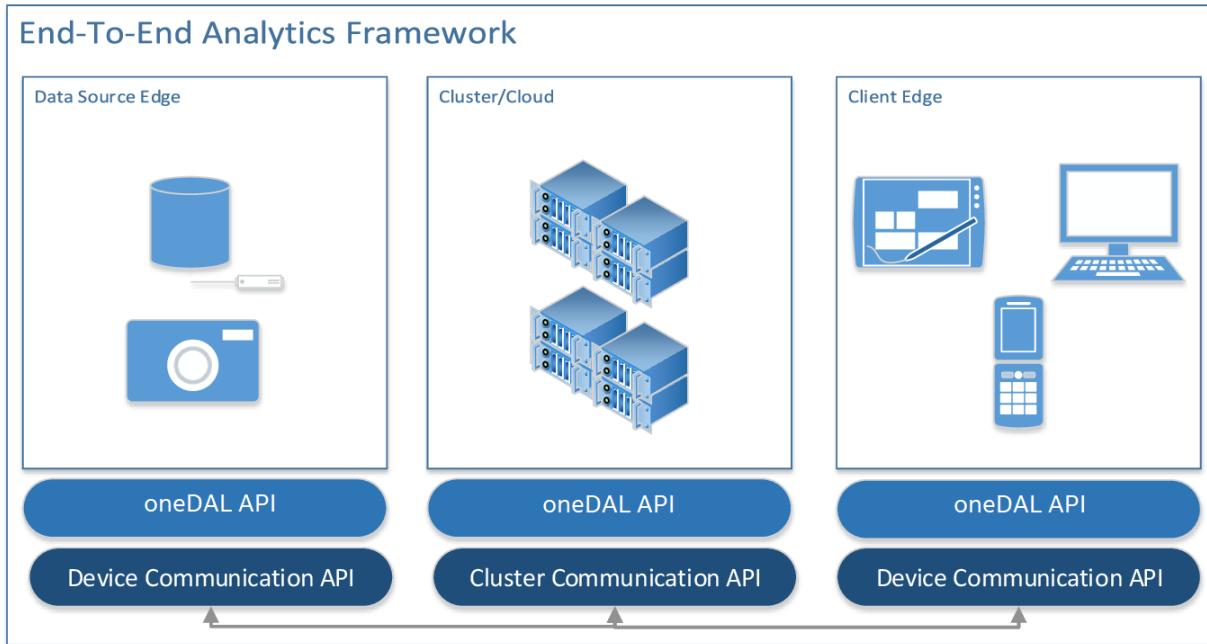
For general information, visit [oneDAL GitHub\\*](#) page.

### 10.1 Introduction

oneAPI Data Analytics Library (oneDAL) is a library that provides building blocks covering all stages of data analytics: data acquisition from a data source, preprocessing, transformation, data mining, modeling, validation, and decision making.



oneDAL supports the concept of the end-to-end analytics when some of data analytics stages are performed on the edge devices (close to where the data is generated and where it is finally consumed). Specifically, oneDAL Application Programming Interfaces (APIs) are agnostic about a particular cross-device communication technology and, therefore, can be used within different end-to-end analytics frameworks.



oneDAL consists of the following major components:

- The *Data Management* component includes classes and utilities for data acquisition, initial preprocessing and normalization, for data conversion into numeric formats (performed by one of supported Data Sources), and for model representation.
- The *Algorithms* component consists of classes that implement algorithms for data analysis (data mining) and data modeling (training and prediction). These algorithms include clustering, classification, regression, and recommendation algorithms. Algorithms support the following computation modes:
  - *Batch processing*: algorithms work with the entire data set to produce the final result
  - *Online processing*: algorithms process a data set in blocks streamed into the device's memory
  - *Distributed processing*: algorithms operate on a data set distributed across several devices (compute nodes)

Distributed algorithms in oneDAL are abstracted from underlying cross-device communication technology, which enables use of the library in a variety of multi-device computing and data transfer scenarios.

Depending on the usage, algorithms operate both on actual data (data set) and data models:

- Analysis algorithms typically operate on data sets.
- Training algorithms typically operate on a data set to train an appropriate data model.
- Prediction algorithms typically work with the trained data model and with a working data set.
- The **Utilities** component includes auxiliary functionality intended to be used for design of classes and implementation of methods such as memory allocators or type traits.
- The **Miscellaneous** component includes functionality intended to be used by oneDAL algorithms and applications for algorithm customization and optimization on various stages of the analytical pipeline. Examples of such algorithms include solvers and random number generators.

Classes in Data Management, Algorithms, Utilities, and Miscellaneous components cover the most important usage scenarios and allow seamless implementation of complex data analytics workflows through direct API calls. At the same time, the library is an object-oriented framework that helps customize the API by redefining particular classes and methods of the library.

## 10.2 Glossary

### 10.2.1 Machine learning terms

**Categorical feature** A *feature* with a discrete domain. Can be *nominal* or *ordinal*.

**Synonyms:** discrete feature, qualitative feature

**Classification** A *supervised machine learning problem* of assigning *labels* to *feature vectors*.

**Examples:** predict what type of object is on the picture (a dog or a cat?), predict whether or not an email is spam

**Clustering** An *unsupervised machine learning problem* of grouping *feature vectors* into bunches, which are usually encoded as *nominal* values.

**Example:** find big star clusters in the space images

**Continuous feature** A *feature* with values in a domain of real numbers. Can be *interval* or *ratio*

**Synonyms:** quantitative feature, numerical feature

**Examples:** a person's height, the price of the house

**Dataset** A collection of *observations*.

**Feature** A particular property or quality of a real object or an event. Has a defined type and domain. In machine learning problems, features are considered as input variable that are independent from each other.

**Synonyms:** attribute, variable, input variable

**Feature vector** A vector that encodes information about real object, an event or a group of objects or events. Contains at least one *feature*.

**Example:** A rectangle can be described by two features: its width and height

**Inference** A process of applying a *trained model* to the *dataset* in order to predict *response* values based on input *feature vectors*.

**Synonym:** prediction

**Inference set** A *dataset* used at the *inference* stage. Usually without *responses*.

**Interval feature** A *continuous feature* with values that can be compared, added or subtracted, but cannot be multiplied or divided.

**Examples:** a timeframe scale, a temperature in Celcius or Fahrenheit

**Label** A *response* with *categorical* or *ordinal* values. This is an output in *classification* and *clustering* problems.

**Example:** the spam-detection problem has a binary label indicating whether the email is spam or not

**Model** An entity that stores information necessary to run *inference* on a new *dataset*. Typically a result of a *training* process.

**Example:** in linear regression algorithm, the model contains weight values for each input feature and a single bias value

**Nominal feature** A *categorical feature* without ordering between values. Only equality operation is defined for nominal features.

**Examples:** a person's gender, color of a car

**Observation** A *feature vector* and zero or more *responses*.

**Synonyms:** instance, sample

**Ordinal feature** A *categorical feature* with defined operations of equality and ordering between values.

**Example:** student's grade

**Outlier** *Observation* which is significantly different from the other observations.

**Ratio feature** A *continuous feature* with defined operations of equality, comparison, addition, subtraction, multiplication, and division. Zero value element means the absence of any value.

**Example:** the height of a tower

**Regression** A *supervised machine learning problem* of assigning *continuous responses* for *feature vectors*.

**Example:** predict temperature based on weather conditions

**Response** A property of some real object or event which dependency from *feature vector* need to be defined in *supervised learning* problem. While a *feature* is an input in the machine learning problem, the response is one of the outputs can be made by the *model* on the *inference* stage.

**Synonym:** dependent variable

**Supervised learning** *Training* process that uses a *dataset* with information about dependencies between *features* and *responses*. The goal is to get a *model* of dependencies between input *feature vector* and *responses*.

**Training** A process of creating a *model* based on information extracted from a *training set*. Resulting *model* is selected in accordance with some quality criteria.

**Training set** A *dataset* used at the *training* stage to create a *model*.

**Unsupervised learning** *Training* process that uses a *training set* with no *responses*. The goal is to find hidden patters inside *feature vectors* and dependencies between them.

## 10.2.2 oneDAL terms

**Accessor** A oneDAL concept for an object that provides access to the data of another object in the special *data format*. It abstracts data access from interface of an object and provides uniform access to the data stored in objects of different types.

**Batch mode** The computation mode for an algorithm in oneDAL, where all the data needed for computation is available at the start and fits the memory of the device on which the computations are performed.

**Builder** A oneDAL concept for an object that encapsulates the creation process of another object and enables its iterative creation.

**Contiguous data** Data that are stored as one contiguous memory block. One of the characteristics of a *data format*.

**Data format** Representation of the internal structure of the data.

**Examples:** data can be stored in array-of-structures or compressed-sparse-row format

**Data layout** A characteristic of *data format* which describes the order of elements in a *contiguous data* block.

**Example:** row-major format, where elements are stored row by row

**Data type** An attribute of data used by a compiler to store and access them. Includes size in bytes, encoding principles, and available operations (in terms of a programming language).

**Examples:** `int32_t, float, double`

**Flat data** A block of *contiguous homogeneous* data.

**Getter** A method that returns the value of the private member variable.

**Example:**

```
std::int64_t get_row_count() const;
```

**Heterogeneous data** Data which contain values either of different *data types* or different sets of operations defined on them. One of the characteristics of a *data format*.

**Example:** A *dataset* with 100 *observations* of three *interval features*. The first two features are of float32 *data type*, while the third one is of float64 data type.

**Homogeneous data** Data with values of single *data type* and the same set of available operations defined on them. One of the characteristics of a *data format*.

**Example:** A *dataset* with 100 *observations* of three *interval features*, each of type float32

**Immutability** The object is immutable if it is not possible to change its state after creation.

**Metadata** Information about logical and physical structure of an object. All possible combinations of metadata values present the full set of possible objects of a given type. Metadata do not expose information that is not a part of a type definition, e.g. implementation details.

**Example:** *table* object can contain three *nominal features* with 100 *observations* (logical part of metadata). This object can store data as sparse csr array and provides direct access to them (physical part)

**Online mode** The computation mode for an algorithm in oneDAL, where the data needed for computation becomes available in parts over time.

**Reference-counted object** A copy-constructible and copy-assignable oneDAL object which stores the number of references to the unique implementation. Both copy operations defined for this object are lightweight, which means that each time a new object is created, only the number of references is increased. An implementation is automatically freed when the number of references becomes equal to zero.

**Setter** A method that accepts the only parameter and assigns its value to the private member variable.

**Example:**

```
void set_row_count(std::int64_t row_count);
```

**Table** A oneDAL concept for a *dataset* that contains only numerical data, *categorical* or *continuous*. Serves as a transfer of data between user's application and computations inside oneDAL. Hides details of *data format* and generalizes access to the data.

**Workload** A problem of applying a oneDAL algorithm to a *dataset*.

### 10.2.3 Common oneAPI terms

**API** Application Programming Interface

**DPC++** Data Parallel C++ (DPC++) is a high-level language designed for data parallel programming productivity. DPC++ is based on *SYCL*\* from the Khronos\* Group to support data parallelism and heterogeneous programming.

**Host/Device** OpenCL [OpenCLSpec] refers to CPU that controls the connected GPU executing kernels.

**JIT** Just in Time Compilation — compilation during execution of a program.

**Kernel** Code written in OpenCL [OpenCLSpec] or *SYCL* and executed on a GPU device.

**SPIR-V** Standard Portable Intermediate Representation - V is a language for intermediate representation of compute kernels.

**SYCL** SYCL(TM) [SYCLSpec] — high-level programming model for OpenCL(TM) that enables code for heterogeneous processors to be written in a “single-source” style using completely standard C++.

## 10.3 Mathematical Notations

Notation	Definition
$n$ or $m$	The number of <i>observations</i> in a <i>dataset</i> . Typically $n$ is used, but sometimes $m$ is required to distinguish two datasets, e.g., the <i>training set</i> and the <i>inference set</i> .
$p$ or $r$	The number of features in a dataset. Typically $p$ is used, but sometimes $r$ is required to distinguish two datasets.
$a \times b$	The dimensionality of a matrix (dataset) has $a$ rows (observations) and $b$ columns (features).
$ A $	Depending on the context may be interpreted as follows: <ul style="list-style-type: none"> <li>If <math>A</math> is a set, this denotes its cardinality, i.e., the number of elements in the set <math>A</math>.</li> <li>If <math>A</math> is a real number, this denotes an absolute value of <math>A</math>.</li> </ul>
$\ x\ $	The $L_2$ -norm of a vector $x \in \mathbb{R}^d$ ,
	$\ x\  = \sqrt{x_1^2 + x_2^2 + \cdots + x_d^2}.$
$\text{sgn}(x)$	Sign function for $x \in \mathbb{R}$ ,
	$\text{sgn}(x) = \begin{cases} -1, & x < 0, \\ 0, & x = 0, \\ 1, & x > 0. \end{cases}$
$x_i$	In the description of an algorithm, this typically denotes the $i$ -th <i>feature vector</i> in the training set.
$x'_i$	In the description of an algorithm, this typically denotes the $i$ -th feature vector in the inference set.
$y_i$	In the description of an algorithm, this typically denotes the $i$ -th <i>response</i> in the training set.
$y'_i$	In the description of an algorithm, this typically denotes the $i$ -th response that needs to be predicted by the inference algorithm given the feature vector $x'_i$ from the inference set.

## 10.4 Programming model

### 10.4.1 Basic usage scenario

Below you can find a typical workflow of using oneDAL algorithm on GPU. The example is provided for Principal Component Analysis algorithm (PCA).

The following steps depict how to:

- Pass the data
- Initialize the algorithm
- Request statistics to be calculated (means, variances, eigenvalues)
- Compute results
- Get calculated eigenvalues and eigenvectors

1. Include the following header file to enable the DPC++ interface for oneDAL:

```
#include "/daal_in_code/_sycl.h"
```

2. Create a DPC++ queue with the desired device selector. In this case, GPU selector is used:

```
cl::sycl::queue queue { cl::sycl::gpu_selector() };
```

3. Create an execution context from the DPC++ queue and set up as the default for all algorithms. The execution context is the oneDAL concept that is intended for delivering queue and device information to the algorithm kernel:

```
daal::services::Environment::getInstance() -> setDefaultExecutionContext (
    daal::services::SyclExecutionContext(queue) );
```

4. Create a DPC++ buffer from the data allocated on host:

```
constexpr size_t nRows = 10;
constexpr size_t nCols = 5;
const float dataHost[] = {
    0.42, -0.88, 0.46, 0.04, -0.86,
    -0.74, -0.59, 0.42, -1.44, -0.40,
    -1.45, 1.07, -1.00, -0.29, 0.35,
    -0.67, 0.20, 0.47, -1.07, 0.71,
    -1.19, 0.20, 0.84, -0.26, 1.47,
    -1.87, -0.94, -1.16, -0.64, -2.10,
    -0.65, -0.40, -1.88, -0.48, 0.70,
    -0.52, -0.34, -1.48, -0.63, -0.87,
    -0.74, -0.46, 1.07, 0.65, -1.68,
    0.94, 1.88, -0.73, -1.16, 0.10
};
auto dataBuffer = cl::sycl::buffer<float, 1>(dataHost, nCols * nRows);
```

5. Create a DPC++ numeric table from a DPC++ buffer. DPC++ numeric table is a new concept introduced as a part of DPC++ interfaces to work with data stored in DPC++ buffer. It implements an interface of a classical numeric table acting as an adapter between DPC++ and oneDAL APIs for data representation.

```
auto data = daal::data_management::SyclHomogenNumericTable<float>::create(
    dataBuffer, nCols, nRows);
```

6. Create an algorithm object, configure its parameters, set up input data, and run the computations.

```
daal::algorithms::pca::Batch<float> pca;

pca.parameter.nComponents = 3;
pca.parameter.resultsToCompute = daal::algorithms::pca::mean |
    daal::algorithms::pca::variance |
    daal::algorithms::pca::eigenvalue;
pca.input.set(daal::algorithms::pca::data, data);

pca.compute();
```

7. Get the algorithm result:

```
auto result = pca.getResult();
NumericTablePtr eigenvalues = result->get(daal::algorithms::pca::eigenvalues);
    ↵
NumericTablePtr eigenvectors = result->get(daal::algorithms::pca::eigenvectors);
    ↵
```

8. Get the raw data as DPC++ buffer from the resulting numeric tables:

```

const size_t startRowIndex = 0;
const size_t numberOfRows = eigenvectors->getNumberOfRows();

BlockDescriptor<float> block;
eigenvectors->getBlockOfRows(startRowIndex, numberOfRows, readOnly, block);

cl::sycl::buffer<float, 1> buffer = block.getBuffer().toSycl();

eigenvectors->releaseBlockOfRows(block);

```

At the end of the stage, the resulting numeric tables can be used as an input for another algorithm, or the buffer can be passed to the user-defined kernel.

## 10.4.2 Memory objects

## 10.4.3 Algorithm Anatomy

oneDAL primarily targets algorithms that are extensively used in data analytics. These algorithms typically have many parameters, i.e. knobs to control its internal behavior and produced result. In machine learning, those parameters are often referred as *meta-parameters* to distinguish them from the model parameters learnt during the training. Some algorithms define a dozen meta-parameters, while others depend on another algorithm as, for example, the logistic regression training procedure depends on optimization algorithm.

Besides meta-parameters, machine learning algorithms may have different *stages*, such as *training* and *inference*. Moreover, the stages of an algorithm may be implemented in a variety of *computational methods*. For instance, a linear regression model could be trained by solving a system of linear equations [Friedman17] or by applying an iterative optimization solver directly to the empirical risk function [Zhang04].

From computational perspective, algorithm implementation may rely on different *floating-point types*, such as `float`, `double` or `bfloat16`. Having a capability to specify what type is needed is important for the end user as their precision requirements vary depending on a workload.

To best tackle the mentioned challenges, each algorithm is decomposed into *descriptors* and *operations*.

### 10.4.3.1 Descriptors

A **descriptor** is an object that represents an algorithm including all its meta-parameters, dependencies on other algorithms, floating-point types, and computational methods. A descriptor serves as:

- A dispatching mechanism for *operations*. Based on a descriptor type, an operation executes a particular algorithm implementation.
- An aggregator of meta-parameters. It provides an interface for setting up meta-parameters at either compile-time or run-time.
- An object that stores the state of the algorithm. In the general case, a descriptor is a stateful object whose state changes after an operation is applied.

Each oneDAL algorithm has its own dedicated namespace, where the corresponding descriptor is defined (for more details, see *Namespace*s). Descriptor, in its turn, defines the following:

- **Template parameters.** A descriptor is allowed to have any number of template parameters, but shall support at least two:
  - `Float` is a *floating-point type* that the algorithm uses for computations. This parameter is defined first and has the `onedeal::default_float_t` default value.

- Method is a tag-type that specifies the *computational method*. This parameter is defined second and has the `method::by_default` default value.
- **Properties.** A property is a run-time parameter that can be accessed by means of the corresponding *getter* and *setter* methods.

*The following code sample* shows the common structure of a descriptor's definition for an abstract algorithm. To define a particular algorithm, the following strings shall be substituted:

- `%ALGORITHM%` is the name of an algorithm and its namespace. All classes and structures related to that algorithm are defined within the namespace.
- `%PROPERTY_NAME%` and `%PROPERTY_TYPE%` are the name and the type of one of the algorithm's properties.

```
namespace onedal::%ALGORITHM% {

template <typename Float = default_float_t,
          typename Method = method::by_default,
          /* more template parameters */
class descriptor {
public:
    /* Getter & Setter for the property called `%PROPERTY_NAME%` */
    descriptor& set_%PROPERTY_NAME%(%PROPERTY_TYPE% value);
    %PROPERTY_TYPE% get_%PROPERTY_NAME%() const;

    /* more properties */
};

} // namespace onedal::%ALGORITHM%
```

Each meta-parameter of an algorithm is mapped to a property that shall satisfy the following requirements:

- Properties are defined with getter and setter methods. The underlying class member variable that stores the property's value is never exposed in the descriptor interface.
- The getter returns the value of the underlying class member variable.
- The setter accepts only one parameter of the property's type and assigns it to the underlying class member variable.
- Most of the properties are preset with default values, others are initialized by passing the required parameters to the constructor.
- The setter returns a reference to the descriptor object to allow chaining calls as shown in the example below.

```
auto desc = descriptor{}
    .set_property_name_1(value_1)
    .set_property_name_2(value_2)
    .set_property_name_3(value_3);
```

### 10.4.3.1.1 Floating-point Types

It is required for each algorithm to support at least one implementation-defined floating-point type. Other floating-point types are optional, for example `float`, `double`, `float16`, and `bfloat16`. It is up to a specific oneDAL implementation whether or not to support these types.

The floating-point type used as a default in descriptors is implementation-defined and shall be declared within the top-level namespace.

```
namespace onedal {
    using default_float_t = /* implementation defined */;
} // namespace onedal
```

### 10.4.3.1.2 Computational Methods

The supported computational methods are declared within the `%ALGORITHM%::method` namespace using tag-types. Algorithm shall support at least one computational method and declare the `by_default` type alias that refers to one of the computational methods as shown in the example below.

```
namespace onedal::%ALGORITHM% {
    namespace method {
        struct x {};
        struct y {};
        using by_default = x;
    } // namespace method
} // namespace onedal::%ALGORITHM%
```

## 10.4.3.2 Operations

### 10.4.3.2.1 Input

### 10.4.3.2.2 Result

## 10.4.4 Managing execution context

## 10.4.5 Computational modes

### 10.4.5.1 Batch

In the batch processing mode, the algorithm works with the entire data set to produce the final result. A more complex scenario occurs when the entire data set is not available at the moment or the data set does not fit into the device memory.

### 10.4.5.2 Online

In the online processing mode, the algorithm processes a data set in blocks streamed into the device's memory. Partial results are updated incrementally and finalized when the last data block is processed.

### 10.4.5.3 Distributed

In the distributed processing mode, the algorithm operates on a data set distributed across several devices (compute nodes). On each node, the algorithm produces partial results that are later merged into the final result on the master node.

## 10.5 Common Interface

### 10.5.1 Header files

oneDAL public identifiers are represented in the following header files:

Header file	Description
onedal/ onedal.hpp	The main header file of oneDAL library.
onedal/ algo/ %ALGO%/ %ALGO%.hpp	A header file for a particular algorithm. The string %ALGO% should be substituted with the name of the algorithm, for example, kmeans or knn.
onedal/ algo/misc/ %FUNC%/ %FUNC%.hpp	A header file for miscellaneous data types and functionality that is intended to be used by oneDAL algorithms and applications of the analytical pipeline. The string %FUNC% should be substituted with the functionality name, for example, mt19937 or cross_entropy_loss.
onedal/ util/ %UTIL%. hpp`	A header file for auxiliary functionality, such as memory allocators or type traits, that is intended to be used for the design of classes and implementation of various methods. The string %UTIL% should be substituted with the auxiliary functionality name, for example, usm_allocator or type_traits.

## 10.5.2 Namespaces

oneDAL functionality is represented with a system of C++ namespaces described below:

Namespace	oneDAL content
onedal	The namespace of the library that contains externally exposable data types, processing and service functionality of oneDAL.
onedal::%ALGORITHM%	The namespace of the algorithm. All classes and structures related to that algorithm shall be defined within a particular namespace. To define a specific algorithm, the string %ALGORITHM% should be substituted with its name, for example, onedal::kmeans or onedal::knn.
onedal::misc	The namespace that contains miscellaneous data types and functionality intended to be used by oneDAL algorithms and applications for algorithm customization and optimization on various stages of the analytical pipeline.
%PARENT%::detail	The namespace that contains implementation details of the data types and functionality for the parent namespace. The namespace can be on any level in the namespace hierarchy. To define a specific namespace, the string %PARENT% should be substituted with the namespace for which the details are provided, for example, onedal::detail or onedal::kmeans::detail. The application shall not use any data types nor call any functionality located in the detail namespaces.

## 10.5.3 Managing object lifetimes

### 10.5.4 Error handling

oneDAL error handling relies on the mechanism of C++ exceptions. If an error occurs, it shall be propagated at the point of a function call where it is caught using standard C++ error handling mechanism.

#### 10.5.4.1 Exception classification

Exception classification in onDAL is aligned with C++ Standard Library classification. oneDAL shall introduce abstract classes that define the base class in the hierarchy of exception classes. Concrete exception classes are derived from respective C++ Standard Library exception classes. oneDAL library shall throw exceptions represented with concrete classes.

In the hierarchy of onDAL exceptions, `onedal::exception` is the base abstract class that all other exception classes are derived from.

```
class onedal::exception;
```

Exception	Description	Abstract
onedal::exception	Base class of oneDAL exception hierarchy.	Yes

All oneDAL exceptions shall be divided into three groups:

- logic errors
- runtime errors
- errors with allocation

```
class onedal::logic_error : public onedal::exception;
class onedal::runtime_error : public onedal::exception;
class onedal::bad_alloc : public onedal::exception, public std::bad_alloc;
```

Exception	Description	Abstract
onedal::logic_error	Reports violations of preconditions and invariants.	Yes
onedal::runtime_error	Reports violations of postconditions and other errors happened during the execution of oneDAL functionality.	Yes
onedal::bad_alloc	Reports failure to allocate storage.	No

All precondition and invariant errors represented by `onedal::logic_error` shall be divided into the following groups:

- invalid argument errors
- domain errors
- out of range errors
- errors with an unimplemented method or algorithm
- unavailable device or data

```
class onedal::invalid_argument : public onedal::logic_error, public std::invalid_
→argument;
class onedal::domain_error : public onedal::logic_error, public std::domain_error;
class onedal::out_of_range : public onedal::logic_error, public std::out_of_range;
class onedal::unimplemented_error : public onedal::logic_error, public std::logic_
→error;
class onedal::unavailable_error : public onedal::logic_error, public std::logic_
→error;
```

Exception	Description	Abstract
onedal::invalid_argument	Reports situations when the argument was not been accepted.	No
onedal::domain_error	Reports situations when the argument is outside of the domain on which the operation is defined. Higher priority than <code>onedal::invalid_argument</code> .	No
onedal::out_of_range	Reports situations when the index is out of range. Higher priority than <code>onedal::invalid_argument</code> .	No
onedal::unimplemented_error	Reports errors that arise because an algorithm or a method is not implemented.	No
onedal::unavailable_error	Reports situations when a device or data is not available.	No

If an error occurs during function execution after preconditions and invariants were checked, it is reported via `onedal::runtime_error` inheritors. oneDAL distinguishes errors happened during interaction with OS facilities and errors of destination type's range in internal computations, while other errors are reported via `onedal::internal_error`.

```
class onedal::range_error : public onedal::runtime_error, public std::range_error;
class onedal::system_error : public onedal::runtime_error, public std::system_error;
class onedal::internal_error : public onedal::runtime_error, public std::runtime_
→error;
```

Exception	Description	Abstract
<code>onedal::range_error</code>	Reports situations where a result of a computation cannot be represented by the destination type.	No
<code>onedal::system_error</code>	Reports errors occurred during interaction with OS facilities.	No
<code>onedal::internal_error</code>	Reports all runtime errors that couldn't be assigned to other inheritors.	No

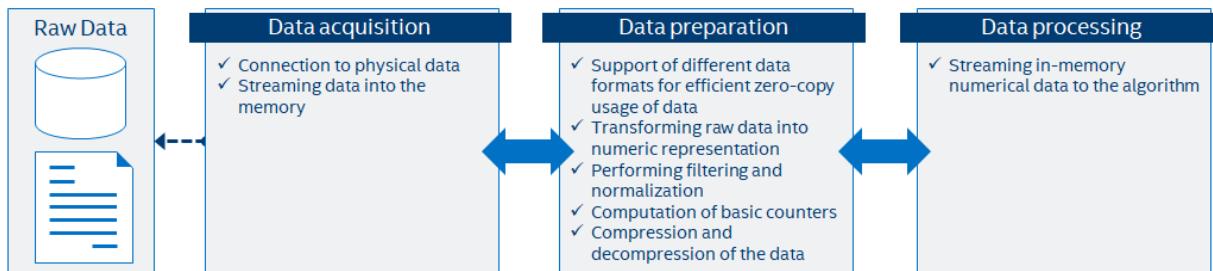
## 10.6 Data management

This section includes descriptions of concepts and objects that operate on data. For oneDAL, such set of operations, or **data management**, is distributed between different stages of the *data analytics pipeline*. From a perspective of data management, this pipeline contains three main steps of data acquisition, preparation, and computation (see *the picture below*):

1. Raw data acquisition
  - Transfer out-of-memory data from various sources (databases, files, remote storages) into an in-memory representation.
2. Data preparation
  - Support different in-memory *data formats*.
  - Compress and decompress the data.
  - Convert the data into numeric representation.
  - Recover missing values.
  - Filter the data and perform data normalization.
  - Compute various statistical metrics for numerical data, such as mean, variance, and covariance.
3. Algorithm computation
  - Stream in-memory numerical data to the algorithm.

In complex usage scenarios, data flow goes through these three stages back and forth. For example, when the data are not fully available at the start of the computation, it can be done step-by-step using blocks of data. After the computation on the current block is completed, the next block should be obtained and prepared.

### Typical data management flow within oneDAL

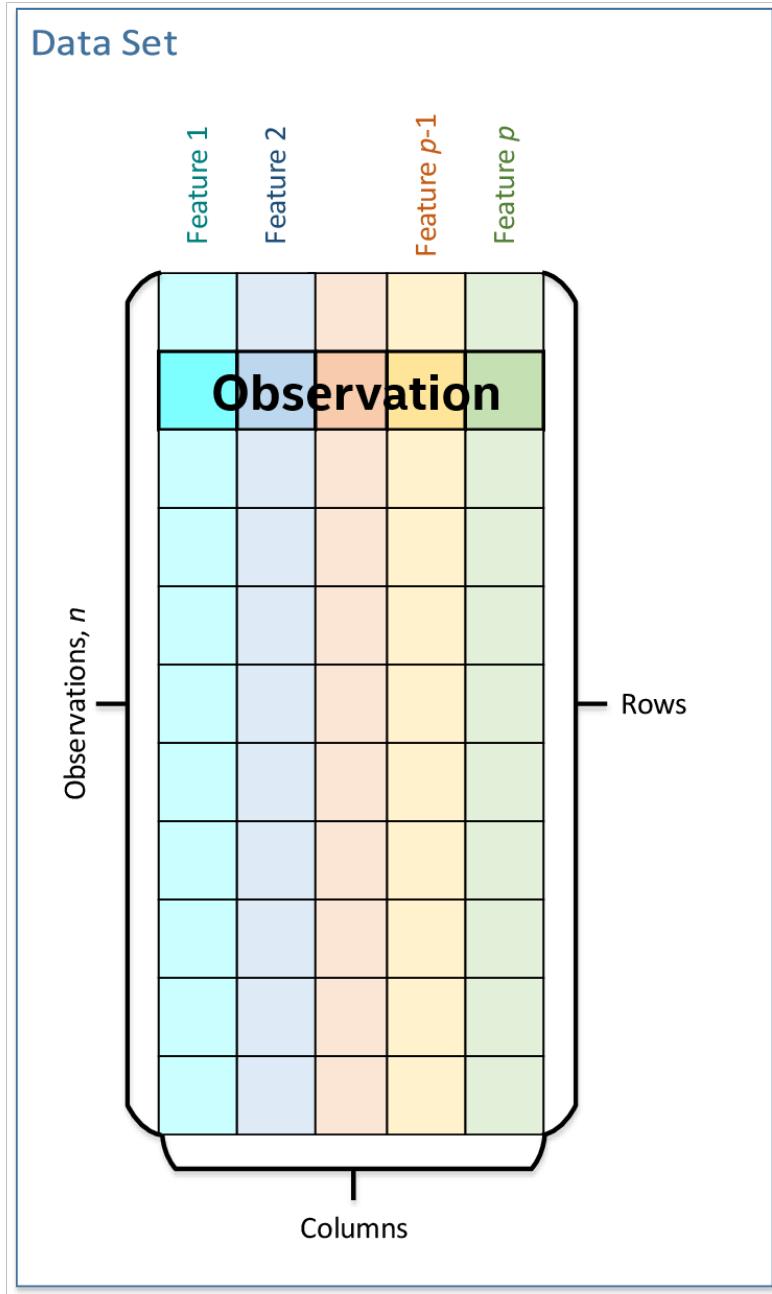


## 10.6.1 Key concepts

oneDAL provides a set of concepts to operate on out-of-memory and in-memory data during different stages of the *data analytics pipeline*.

### 10.6.1.1 Dataset

The main data-related concept that oneDAL works with is a *dataset*. It is an in-memory or out-of-memory tabular view of data, where table rows represent the *observations* and columns represent the *features*.



The dataset is used across all stages of the data analytics pipeline. For example:

1. At the acquisition stage, it is downloaded into the local memory.
2. At the preparation stage, it is converted into a numerical representation.
3. At the computation stage, it is used as one of the *inputs* or *results* of an algorithm or a *descriptor* properties.

### 10.6.1.2 Data source

Data source is a concept of an out-of-memory storage for a *dataset*. It is used at the data acquisition and data preparation stages for the following:

- To extract datasets from external sources such as databases, files, remote storages.
- To load datasets into the device's local memory. Data do not always fit the local memory, especially when processing with accelerators. A data source provides the ability to load data by batches and extracts it directly into the device's local memory. Therefore, a data source enables complex data analytics scenarios, such as *online computations*.
- To filter and normalize *feature* values that are being extracted.
- To recover missing *feature* values.
- To detect *outliers* and recover the abnormal data.
- To transform datasets into numerical representation. Data source shall automatically transform non-numeric *categorical* and *continuous* data values into one of the numeric *data formats*.

For details, see *data sources* section.

### 10.6.1.3 Table

Table is a concept of a *dataset* with in-memory numerical data. It is used at the data preparation and data processing stages for the following:

- To store heterogeneous in-memory data in various *data formats*, such as dense, sparse, chunked, contiguous.
- To avoid unnecessary data copies during conversion from external data representations.
- To transfer memory ownership of the data from user application to the table, or share it between them.
- To connect with the *data source* to convert from an out-of-memory into an in-memory dataset representation.
- To support streaming of the data to the algorithm.
- To access the underlying data on a device in a required *data format*, e.g. by blocks with the defined *data layout*.

For thread-safety reasons and better integration with external entities, a table provides a read-only access to the data within it, thus, table concept implementations shall be *immutable*.

This concept has different logical organization and physical *format of the data*:

- Logically, a table is a *dataset* with  $n$  rows and  $p$  columns. Each row represents an *observation* and each column is a *feature* of a dataset. Physical amount of bytes needed to store the data differ from the number of elements  $n \times p$  within a table.
- Physically, a table can be organized in different ways: as a *homogeneous*, *contiguous* array of bytes, as a *heterogeneous* list of arrays of different *data types*, in a compressed-sparse-row format.

For details, see *tables* section.

#### 10.6.1.4 Metadata

Metadata concept is associated with a *dataset* and holds information about its structure and type. This information shall be enough to determine the particular type of a dataset, and it helps to understand how to interact with a dataset in oneDAL (for example, how to use it at a particular stage of *data analytics pipeline* or how to access its data).

For each dataset, its metadata shall contain:

- The number of rows  $n$  and columns  $p$  in a dataset.
- The type of each *feature* (e.g. *nominal*, *interval*).
- The *data type* of each feature (e.g. *float* or *double*).

---

**Note:** Metadata can contain both compile-time and run-time information. For example, basic compile-time metadata is the type of a dataset - whether it is a particular *data source* or a *table*. Run-time information can contain the *feature* types and *data types* of a dataset.

---

#### 10.6.1.5 Table builder

A table *builder* is a concept that is associated with a particular *table* type and is used at the data preparation and data processing stages for:

- Iterative construction of a *table* from another *tables* or a different in-memory *dataset* representations.
- Construction of a *table* from different entities that hold blocks of data, such as arrays, pointers to the memory, external entities.
- Changing dataset values. Since *table* is an *immutable* dataset, a builder provides the ability to change the values in a dataset under construction.
- Encapsulating construction process of a *table*. This is used to hide the implementation details as they are irrelevant for users. This also allows to select the most appropriate table implementation for each particular case.
- Providing additional information on how to create a *table* inside an algorithm for *results*. This information includes metadata, memory allocators that shall be used, or even a particular table implementation.

For details, see *table builders* section.

#### 10.6.1.6 Accessor

Accessor is a concept that defines a single way to get the data from an in-memory numerical *dataset*. It allows:

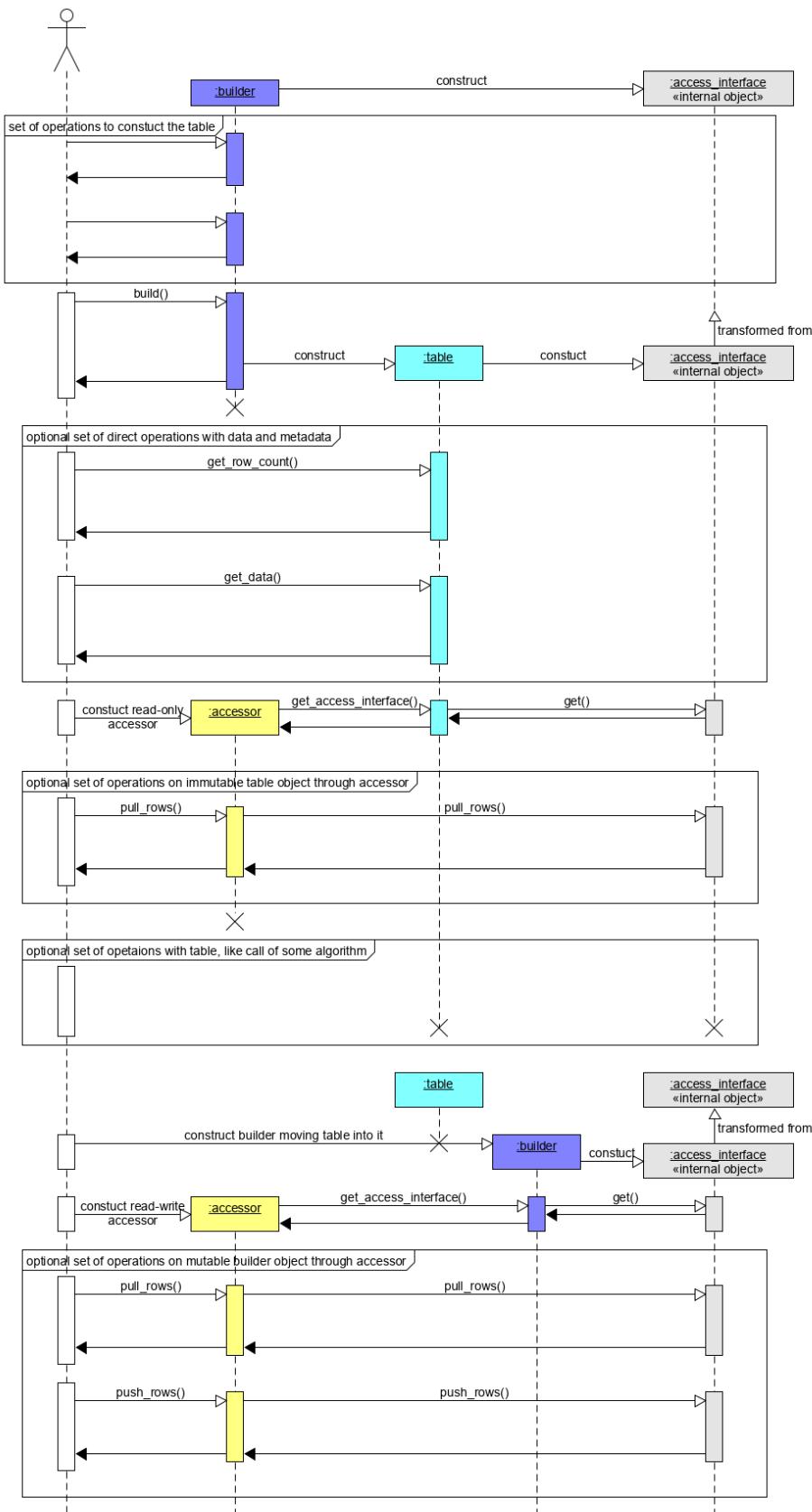
- To have unified access to the data from various sets of different objects, such as *tables* or *table builders*, without exposing their implementation details.
- To convert a variety of numeric *data formats* into a smaller set of formats.
- To provide a *flat* view on the data blocks of a *dataset* for better data locality. For example, some accessor implementation returns *feature* values as a contiguous array, while the original dataset stored row-by-row (there are strides between values of a single feature).
- To acquire data in a desired *data format* for which a specific set of operations is defined.
- To have read-only, read-write and write-only access to the data. Accessor implementations are not required to have read-write and write-only access modes for *immutable* entities like *tables*.

For details, see *accessors* section.

#### 10.6.1.7 Use-case example for table, accessor and table builder

This section provides a basic usage scenario of the *table*, *table builder*, and *accessor* concepts and demonstrates the relations between them. *The following diagram* shows objects of these concepts, which are highlighted by colors:

- *Table builder* objects are blue.
- *Table* objects are cyan.
- *Accessors* are yellow.
- Grey objects are not a part of oneDAL specification and they are provided just for illustration purposes.



To perform computations on a dataset, one shall create a *table* object first. It can be done using a *data source* or a *table builder* object depending on the situation. The diagram briefly shows the situation when *table* is interatively created from a various external entities (not shown on a diagram) using a *table builder*.

Once a table object is created, the data inside it can be accessed by its own interface or with a help of a read-only accessor as shown on the diagram. The table can be used as an input in computations or as a parameter of some algorithm.

Algorithms' results also contain table objects. If one needs to change the data within some table, a builder object can be constructed for this. Data inside a table builder can be retrieved by read-only, write-only or read-write accessors.

Accessors shown on the diagram allow to get data from tables and table builders as *flat* blocks of rows.

## 10.6.2 Details

### 10.6.2.1 Data Sources

TBD

### 10.6.2.2 Tables

This section describes the types related to the *table* and *metadata* concepts. oneDAL defines the following types that implement these concepts:

- The *table* is a base class that implements the table concept and provides capability to get a metadata. Each implementation of the *table* concept shall be derived from the *table* class (for more details, see *table API* section).
- The *table\_meta* class implements the metadata concept for the table. Each derived table type may provide its own implementation of the *table\_meta* that extends the metadata concept (for more details, see *metadata API* section).

#### 10.6.2.2.1 Requirements

Each implementation of *table* concept shall:

1. Follow definition of the table concept.
2. Be derived from the *table* class. The behavior of this class can be extended, but cannot be weaken.
3. Provide an implementation of the *metadata* concept derived from the *table\_meta* class.
4. Be *reference-counted*. An assignment operator or copy constructor shall be used to create another reference to the same data.

```
onedeal::table table2 = table1;
// table1 and table2 share the same data (no data copy is performed)

onedeal::table table3 = table2;
// table1, table2 and table3 share the same data
```

### 10.6.2.2.2 Table Types

oneDAL defines a set of classes. Each class implements the *table* concept and represents a specific data format.

Table type	Description
<i>table</i>	A common implementation of the table concept. Base class for other table types.
<i>homogen_table</i>	Dense table that contains <i>contiguous homogeneous</i> data.
<i>soa_table</i>	Dense heterogeneous table which data are stored column-by-column in a list of contiguous arrays (structure-of-arrays format).
<iaos_table< i=""></iaos_table<>	Dense heterogeneous table which data are stored as one contiguous block of memory (array-of-structures format).
<i>csr_table</i>	Sparse homogeneous table which data are stored in compressed sparse row (CSR) format.

### 10.6.2.2.3 Table API

```
class table {
public:
    table() = default;

    template <typename TableImpl,
              typename = std::enable_if_t<is_table_impl_v<TableImpl>>>
    table(TableImpl&&);

    table(const table&);
    table(table&&);

    table& operator=(const table&);

    std::int64_t get_feature_count() const noexcept;
    std::int64_t get_observation_count() const noexcept;
    bool is_empty() const noexcept;
    const dal::table_meta& get_metadata() const noexcept;
};
```

**class table**

**table()**  
Creates an empty table with no data and `table_meta` constructed by default

**table(TableImpl&&)**  
Creates a table object using the entity passed as a parameter

**Template Parameters TableImpl** – The class that contains the table's implementation

#### Invariants

contract `is_table_impl` is satisfied

**table(const table&)**  
Creates new reference object on the table data

**table(table&&)**  
Moves one table object into another

*table* &**operator=**(**const** *table*&)  
 Sets the current object reference to point to another one

std::int64\_t **feature\_count** = 0  
 The number of *features* *p* in the table.

**Getter**

```
std::int64_t get_feature_count() const noexcept
```

**Invariants**

feature\_count >= 0

std::int64\_t **observation\_count** = 0  
 The number of *observations* *N* in the table.

**Getter**

```
std::int64_t get_observation_count() const noexcept
```

**Invariants**

observation\_count >= 0

bool **is\_empty** = true  
 If feature\_count or observation\_count are zero, the table is empty.

**Getter**

```
bool is_empty() const noexcept
```

*table\_meta* **metadata** = *table\_meta*()  
 The object that represents data structure inside the table

**Getter**

```
const dal::table_meta& get_metadata() const noexcept
```

**Invariants**

is\_empty = false

### 10.6.2.2.3.1 Homogeneous table

Class homogen\_table is an implementation of a table type for which the following is true:

- Its data is dense and it is stored as one contiguous memory block
- All features have the same *data type* (but *feature types* may differ)

```
class homogen_table : public table {
public:
  // TODO:
  // Consider constructors with user-provided allocators & deleters

  homogen_table(const homogen_table&);
  homogen_table(homogen_table&&);

  homogen_table(std::int64_t N, std::int64_t p, data_layout layout);

  template <typename T>
  homogen_table(const T* const data_pointer, std::int64_t N, std::int64_t p, data_
  ↵ layout layout);
```

(continues on next page)

(continued from previous page)

```
homogen_table& operator=(const homogen_table&);

data_type get_data_type() const noexcept;
bool has_equal_feature_types() const noexcept;

template <typename T>
const T* get_data_pointer() const noexcept;
};
```

**class homogen\_table****homogen\_table(const homogen\_table&)**

Creates new reference object on the table data

**homogen\_table(homogen\_table&&)**

Moves current reference object into another one

**homogen\_table(std::int64\_t N, std::int64\_t p, data\_layout layout)**Creates a homogeneous table of shape  $N \times p$  with default oneDAL allocator**homogen\_table(const T \*const data\_pointer, std::int64\_t N, std::int64\_t p, data\_layout layout)****Template Parameters** **T** – The type of pointer to the dataCreates a homogeneous table of shape  $N \times p$  with the user-defined data. Uses the provided pointer to access data (no copy is performed).**homogen\_table &operator=(const homogen\_table&)**

Sets the current object reference to point to another

**onedal::data\_type data\_type**

The type of underlying data

**Getter**

data\_type get\_data\_type() const noexcept

**bool feature\_types\_equal**Flag that indicates whether or not the *feature\_type* fields of *metadata* are all equal**Getter**

bool has\_equal\_feature\_types() const noexcept

**const T \*data\_pointer****Template Parameters** **T** – The type of pointer to the data

The pointer to underlying data

**Getter**

const T\* get\_data\_pointer() const noexcept

### 10.6.2.2.3.2 Structure-of-arrays table

TBD

### 10.6.2.2.3.3 Arrays-of-structure table

TBD

### 10.6.2.2.3.4 Compressed-sparse-row table

TBD

## 10.6.2.2.4 Metadata API

Table metadata contains structures describing how the data are stored inside the table and how efficiently access them.

```
class table_meta {
public:
    table_meta();

    std::int64_t get_feature_count() const noexcept;
    table_meta& set_feature_count(std::int64_t);

    const feature_info& get_feature(std::int64_t index) const;
    table_meta& add_feature(const feature_info&);

    data_layout get_layout() const noexcept;
    table_meta& set_layout(data_layout);

    bool is_contiguous() const noexcept;
    table_meta& set_contiguous(bool);

    bool is_homogeneous() const noexcept;

    data_format get_format() const noexcept;
    table_meta& set_format(data_format);
};
```

**class table\_meta**

std::int64\_t **feature\_count** = 0

The number of *features* *p* in the table.

#### Getter & Setter

```
std::int64_t get_feature_count() const noexcept
table_meta& set_feature_count(std::int64_t)
```

#### Invariants

feature\_count >= 0

*feature\_info* **feature**

Information about a particular *feature* in the table

**Getter & Setter**

```
const feature_info& get_feature(std::int64_t index) const
table_meta& add_feature(const feature_info&)
```

*data\_layout* **layout** = *data\_layout*::row\_major

Flag that indicates whether the data is in a row-major or column-major format.

**Getter & Setter**

```
data_layout get_layout() const noexcept
table_meta& set_layout(data_layout)
```

**bool is\_contiguous = true**

Flag that indicates whether the data is stored in contiguous blocks of memory by the axis of *layout*. For example, if *is\_contiguous* == *true* and *data\_layout* is *row\_major*, the data is stored contiguously in each row.

**Getter & Setter**

```
bool is_contiguous() const noexcept
table_meta& set_contiguous(bool)
```

**bool is\_homogeneous () const noexcept**

Returns true if all features have the same *data\_type*

*data\_format* **format** = *data\_format*::dense

Description of the format used for data representation inside the table

**Getter & Setter**

```
data_format get_format() const noexcept
table_meta& set_format(data_format)
```

**10.6.2.2.4.1 Data layout**

```
enum class data_layout : std::int64_t {
    row_major,
    column_major
};
```

**class data\_layout**

Structure that represents underlying data layout

**10.6.2.2.4.2 Data format**

```
enum class data_format : std::int64_t {
    dense,
    csr
};
```

**class data\_format**

Structure that represents underlying format of the data

#### 10.6.2.2.4.3 Feature info

```
class feature_info {
public:
    feature(data_type, feature_type);

    data_type get_data_type() const noexcept;
    feature_type get_type() const noexcept;
};
```

##### **class feature\_info**

Structure that represents information about particular *feature*

###### **Invariants:**

- feature\_type::nominal or feature\_type::ordinal are available only with integer data\_type
- feature\_type::contiguous available only with floating-point data\_type

#### 10.6.2.2.4.4 Data type

```
enum class data_type : std::int64_t {
    u32, u64
    i32, i64,
    f32, f64
};
```

##### **class data\_type**

Structure that represents runtime information about feature data type.

oneDAL supports next data types:

- std::uint32\_t
- std::uint64\_t
- std::int32\_t
- std::int64\_t
- float
- double

#### 10.6.2.2.4.5 Feature type

```
enum class feature_type : std::int64_t {
    nominal,
    ordinal,
    contiguous
};
```

##### **class feature\_type**

Structure that represents runtime information about feature logical type.

**feature\_type::nominal** Discrete feature type, non-ordered

**feature\_type::ordinal** Discrete feature type, ordered

**feature\_type::contiguous** Contiguous feature type

### 10.6.2.3 Table Builders

This section contains definitions of classes that implement *table builder* concept.

#### 10.6.2.3.1 Requirements

Each implementation of *table builder* concept shall:

1. Provide the ability to create a single *table* concept implementation. Each builder shall be associated with a single table type.
2. Be a stateful object which state is used to access data inside builder via *accessors* or to create a table object.
3. Provide *build()* member function that creates a new table object based on the current snapshot of a builder state.

#### 10.6.2.3.2 Table Builder Types

oneDAL defines a set of accessor classes each associated with a single *table* implementation.

Table builder type	Description
<i>simple_homogen_table_builder</i>	Allows to create <i>homogen_table</i> from raw pointers and standard C++ smart pointers.
<i>simple_soa_table_builder</i>	Allows to create <i>soa_table</i> from raw pointers and standard C++ smart pointers.

#### 10.6.2.3.3 Simple Homogeneous Table Builder

TBD

#### 10.6.2.3.4 Simple SOA Table Builder

TBD

### 10.6.2.4 Accessors

This section defines *requirements* to an *accessor* implementation and introduces several *accessor types*.

#### 10.6.2.4.1 Requirements

Each accessor implementation shall:

1. Define a single *format of the data* for the accessor. Every single accessor type shall return and use only one data format.
2. Provide an access to at least one in-memory *dataset* implementation (such as *table*, its sub-types, or *table builders*).

3. Provide read-only, write-only, or read-write access to the data. If an accessor supports several *dataset* implementations to be passed in, it is not necessary for an accessor to support all access modes for every input object. For example, tables shall support a single read-only mode according to their *table concept* definition.
4. Provide the names for read and write operations following the pattern:
  - `pull_*` () for reading
  - `push_*` () for writing
5. Be lightweight. Its constructors from *dataset* implementations shall not have heavy operations such as copy of data, reading, writing, any sort of conversions. These operations shall be performed by heavy operations `pull_*` () and `push_*` (). It is not necessary to have copy- or move- constructors for accessor implementations since it shall be designed for use in a local scope.

#### 10.6.2.4.2 Accessor Types

oneDAL defines a set of accessor classes. Each class is associated with a single specific way of interacting with data within a *dataset*. The following table briefly explains these classes and shows which *dataset* implementations are supported by each accessor type.

Accessor type	Description	List of supported types
<code>row_accessor</code>	Provides access to the range of dataset rows as one <i>contiguous homogeneous</i> block of memory.	<code>homogen_table</code> , <code>soa_table</code> , <code>aos_table</code> , <code>csr_table</code> , and their builders.
<code>column_accessor</code>	Provides access to the range of values within a single column as one <i>contiguous homogeneous</i> block of memory.	<code>homogen_table</code> , <code>soa_table</code> , <code>aos_table</code> , <code>csr_table</code> , and their builders.

##### 10.6.2.4.2.1 Row accessor

TBD

##### 10.6.2.4.2.2 Column accessor

TBD

## 10.7 Algorithms

The Algorithms component consists of classes that implement algorithms for data analysis (data mining) and data modeling (training and prediction). These algorithms include matrix decompositions, clustering, classification, and regression algorithms, as well as association rules discovery.

## 10.7.1 Clustering

### 10.7.1.1 K-Means

The K-Means algorithm partitions  $n$  feature vectors into  $k$  clusters minimizing some criterion. Each cluster is characterized by a representative point, called a *centroid*.

Given the training set  $X = \{x_1, \dots, x_n\}$  of  $p$ -dimensional feature vectors and a positive integer  $k$ , the problem is to find a set  $C = \{c_1, \dots, c_k\}$  of  $p$ -dimensional centroids that minimize the objective function

$$\Phi_X(C) = \sum_{i=1}^n d^2(x_i, C),$$

where  $d^2(x_i, C)$  is the squared Euclidean distance from  $x_i$  to the closest centroid in  $C$ ,

$$d^2(x_i, C) = \min_{1 \leq j \leq k} \|x_i - c_j\|^2, \quad 1 \leq i \leq n.$$

Expression  $\|\cdot\|$  denotes  $L_2$  norm.

---

**Note:** In the general case,  $d$  may be an arbitrary distance function. Current version of the oneDAL spec defines only Euclidean distance case.

---

#### 10.7.1.1.1 Lloyd's method

The Lloyd's method [Lloyd82] consists in iterative updates of centroids by applying the alternating *Assignment* and *Update* steps, where  $t$  denotes a index of the current iteration, e.g.,  $C^{(t)} = \{c_1^{(t)}, \dots, c_k^{(t)}\}$  is the set of centroids at the  $t$ -th iteration. The method requires the initial centroids  $C^{(1)}$  to be specified at the beginning of the algorithm ( $t = 1$ ).

**(1) Assignment step:** Assign each feature vector  $x_i$  to the nearest centroid.  $y_i^{(t)}$  denotes the assigned label (cluster index) to the feature vector  $x_i$ .

$$y_i^{(t)} = \arg \min_{1 \leq j \leq k} \|x_i - c_j^{(t)}\|^2, \quad 1 \leq i \leq n.$$

Each feature vector from the training set  $X$  is assigned to exactly one centroid so that  $X$  is partitioned to  $k$  disjoint sets (clusters)

$$S_j^{(t)} = \{x_i \in X : y_i^{(t)} = j\}, \quad 1 \leq j \leq k.$$

**(2) Update step:** Recalculate centroids by averaging feature vectors assigned to each cluster.

$$c_j^{(t+1)} = \frac{1}{|S_j^{(t)}|} \sum_{x \in S_j^{(t)}} x, \quad 1 \leq j \leq k.$$

The steps (1) and (2) are performed until the following **stop condition**,

$$\sum_{j=1}^k \|c_j^{(t)} - c_j^{(t+1)}\|^2 < \varepsilon,$$

is satisfied or number of iterations exceeds the maximal value  $T$  defined by the user.

### 10.7.1.1.2 Usage example

```
onedal::kmeans::model run_training(const onedal::table& data,
                                    const onedal::table& initial_centroids) {

    const auto kmeans_desc = onedal::kmeans::desc<float>{ }
        .set_cluster_count(10)
        .set_max_iteration_count(50)
        .set_accuracy_threshold(1e-4);

    const auto result = onedal::train(kmeans_desc, data, initial_centroids);

    print_table("labels", result.get_labels());
    print_table("centroids", result.get_model().get_centroids());
    print_value("objective", result.get_objective_function_value());

    return result.get_model();
}
```

```
onedal::table run_inference(const onedal::kmeans::model& model,
                           const onedal::table& new_data) {

    const auto kmeans_desc = onedal::kmeans::desc<float>{ }
        .set_cluster_count(model.get_cluster_count());

    const auto result = onedal::infer(kmeans_desc, model, new_data);

    print_table("labels", result.get_labels());
}
```

### 10.7.1.1.3 API

#### 10.7.1.1.3.1 Methods

```
namespace method {
    struct lloyd {};
    using by_default = lloyd;
} // namespace method
```

**struct lloyd**  
Tag-type that denotes *Lloyd's method*.

**using by\_default = lloyd**  
Alias tag-type for the *Lloyd's method*.

### 10.7.1.1.3.2 Descriptor

```
template <typename Float = float,
          typename Method = method::by_default>
class desc {
public:
    desc();

    int64_t get_cluster_count() const;
    int64_t get_max_iteration_count() const;
    double get_accuracy_threshold() const;

    desc& set_cluster_count(int64_t);
    desc& set_max_iteration_count(int64_t);
    desc& set_accuracy_threshold(double);
};
```

template<typename **Float** = float, typename **Method** = method::*by\_default*>  
**class desc**

#### Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be float or double.
- **Method** – Tag-type that specifies an implementation of K-Means algorithm. Can be method::*lloyd* or method::*by\_default*.

#### **desc()**

Creates new instance of descriptor with the default attribute values.

#### std::int64\_t **cluster\_count** = 2

The number of clusters  $k$ .

#### Getter & Setter

```
std::int64_t get_cluster_count() const
desc& set_cluster_count(std::int64_t)
```

#### Invariants

*cluster\_count* > 0

#### std::int64\_t **max\_iteration\_count** = 100

The maximum number of iterations  $T$ .

#### Getter & Setter

```
std::int64_t get_max_iteration_count() const
desc& set_max_iteration_count(std::int64_t)
```

#### Invariants

*max\_iteration\_count* >= 0

#### double **accuracy\_threshold** = 0.0

The threshold  $\varepsilon$  for the stop condition.

#### Getter & Setter

```
double get_accuracy_threshold() const
desc& set_accuracy_threshold(double)
```

**Invariants**

*accuracy\_threshold* >= 0.0

**10.7.1.1.3.3 Model**

```
class model {
public:
    model();

    const table& get_centroids() const;
    int64_t get_cluster_count() const;
};
```

**class model**

**model()**

Creates a model with the default attribute values.

**table centroids = table()**

$k \times p$  table with the cluster centroids. Each row of the table stores one centroid.

**Getter**

const table& get\_centroids() const

std::int64\_t **cluster\_count** = 0

Number of clusters  $k$  in the trained model.

**Getter**

std::int64\_t get\_cluster\_count() const

**Invariants**

*cluster\_count* == *centroids.row\_count*

**10.7.1.1.3.4 Training onedal::train...****10.7.1.1.3.5 Input**

```
class train_input {
public:
    train_input();
    train_input(const table& data);
    train_input(const table& data, const table& initial_centroids);

    const table& get_data() const;
    const table& get_initial_centroids() const;

    train_input& set_data(const table&);
    train_input& set_initial_centroids(const table&);
};
```

**class train\_input**

**train\_input()**  
Creates input for the training operation with the default attribute values.

**train\_input(const *table* &*data*)**  
Creates input for the training operation with the given *data*, the other attributes get default values.

**train\_input(const *table* &*data*, const *table* &*initial\_centroids*)**  
Creates input for the training operation with the given data and *initial\_centroids*.

**table *data* = *table*()**  
*n × p* table with the data to be clustered, where each row stores one feature vector.

**Getter & Setter**

```
const table& get_data() const
train_input& set_data(const table&)
```

**table *initial\_centroids* = *table*()**  
*k × p* table with the initial centroids, where each row stores one centroid.

**Getter & Setter**

```
const table& get_initial_centroids() const
train_input& set_initial_centroids(const table&)
```

#### 10.7.1.1.3.6 Result

```
class train_result {
public:
    train_result();

    const model& get_model() const;
    const table& get_labels() const;
    int64_t get_iteration_count() const;
    double get_objective_function_value() const;
};
```

**class train\_result**

**train\_result()**  
Creates result of the training operation with the default attribute values.

**kmeans::*model* *model* = kmeans::*model*()**  
The trained K-means model.

**Getter**

```
const model& get_model() const
```

**table *labels* = *table*()**  
*n × 1* table with the labels  $y_i$  assigned to the samples  $x_i$  in the input data,  $1 \leq 1 \leq n$ .

**Getter**

```
const table& get_labels() const
```

**std::int64\_t *iteration\_count* = 0**  
The number of iterations performed by the algorithm.

**Invariants**

```

iteration_count >= 0
double objective_function_value = 0.0
The value of the objective function  $\Phi_X(C)$ , where  $C$  is model.centroids (see
kmeans::model::centroids).
```

### Invariants

```
objective_function_value >= 0.0
```

#### 10.7.1.1.3.7 Operation semantics

```
template<typename Descriptor>
kmeans::train_result train(const Descriptor &desc, const kmeans::train_input &input)
```

**Template Parameters** `Descriptor` – K-Means algorithm descriptor `kmeans::desc`.

### Preconditions

```

.data.is_empty == false
.initial_centroids.is_empty == false
.initial_centroids.row_count == desc.cluster_count
.initial_centroids.column_count == input.data.column_count
```

### Postconditions

```

result.labels.is_empty == false
result.labels.row_count == input.data.row_count
result.model.centroids.is_empty == false
result.model.clusters.row_count == desc.cluster_count
result.model.clusters.column_count == input.data.column_count
result.iteration_count <= desc.max_iteration_count
```

#### 10.7.1.1.3.8 Inference `onedal::infer...`

#### 10.7.1.1.3.9 Input

```

class infer_input {
public:
    infer_input();
    infer_input(const model& m);
    infer_input(const model& m, const table& data);

    const model& get_model() const;
    const table& get_data() const;

    infer_input& set_model(const model&);
    infer_input& set_data(const table&);
};
```

```
class infer_input
```

```
infer_input()
```

Creates input for the inference operation with the default attribute values.

```
infer_input (const kmeans::model &model)
```

Creates input for the inference operation with the given *model*, the other attributes get default values.

```
infer_input (const kmeans::model &model, const table &data)
```

Creates input for the inference operation with the given *model* and *data*.

```
table data = table()
```

$n \times p$  table with the data to be assigned to the clusters, where each row stores one feature vector.

#### Getter & Setter

```
const table& get_data() const
infer_input& set_data(const table&)
```

```
kmeans::model model = kmeans::model()
```

The trained K-Means model (see kmeans ::*model*).

#### Getter & Setter

```
const kmeans::model& get_model() const
infer_input& set_model(const kmeans::model&)
```

### 10.7.1.1.3.10 Result

```
class infer_result {
public:
    infer_result();

    const table& get_labels() const;
    double get_objective_function_value() const;
};
```

```
class infer_result
```

```
infer_result()
```

Creates result of the inference operation with the default attribute values.

```
table labels = table()
```

$n \times 1$  table with assignments labels to feature vectors in the input data.

#### Getter

```
const table& get_labels() const
```

```
double objective_function_value = 0.0
```

The value of the objective function  $\Phi_X(C)$ , where  $C$  is defined by the corresponding *infer\_input*::*model*::centroids.

#### Invariants

```
objective_function_value >= 0.0
```

### 10.7.1.1.3.11 Operation semantics

```
template<typename Descriptor>
kmeans::infer_result infer(const Descriptor &desc, const kmeans::infer_input &input)
```

**Template Parameters** **Descriptor** – K-Means algorithm descriptor `kmeans::desc`.

#### Preconditions

```
input.data.is_empty == false
input.model.centroids.is_empty == false
input.model.centroids.row_count == desc.cluster_count
input.model.centroids.column_count == input.data.column_count
```

#### Postconditions

```
result.labels.is_empty == false
result.labels.row_count == input.data.row_count
```

## 10.7.2 Nearest Neighbors (kNN)

### 10.7.2.1 k-Nearest Neighbors Classification (k-NN)

*k*-NN classification algorithm infers the class for the new feature vector by computing majority vote of the *k* nearest observations from the training set.

Operation	Computational methods		Programming Interface		
<i>Training</i>	<i>Brute-force</i>	<i>k-d tree</i>	<i>train(...)</i>	<i>train_input</i>	<i>train_result</i>
<i>Inference</i>	<i>Brute-force</i>	<i>k-d tree</i>	<i>infer(...)</i>	<i>infer_input</i>	<i>infer_result</i>

#### 10.7.2.1.1 Mathematical formulation

##### 10.7.2.1.1.1 Training

Let  $X = \{x_1, \dots, x_n\}$  be the training set of  $p$ -dimensional feature vectors, let  $Y = \{y_1, \dots, y_n\}$  be the set of class labels, where  $y_i \in \{0, \dots, c - 1\}$ ,  $1 \leq i \leq n$ . Given  $X$ ,  $Y$  and the number of nearest neighbors  $k$ , the problem is to build a model that allows distance computation between the feature vectors in training and inference sets at the inference stage.

##### 10.7.2.1.1.2 Training method: *brute-force*

The training operation produces the model that stores all the feature vectors from the initial training set  $X$ .

### 10.7.2.1.1.3 Training method: *k-d tree*

The training operation builds a *k-d tree* that partitions the training set  $X$  (for more details, see [k-d Tree](#)).

### 10.7.2.1.1.4 Inference

Let  $X' = \{x'_1, \dots, x'_m\}$  be the inference set of  $p$ -dimensional feature vectors. Given  $X'$ , the model produced at the training stage and the number of nearest neighbors  $k$ , the problem is to predict the label  $y'_j$  for each  $x'_j$ ,  $1 \leq j \leq m$ , by performing the following steps:

1. Identify the set  $N(x'_j) \subseteq X$  of the  $k$  feature vectors in the training set that are nearest to  $x'_j$  with respect to the Euclidean distance.
2. Estimate the conditional probability for the  $l$ -th class as the fraction of vectors in  $N(x'_j)$  whose labels  $y_j$  are equal to  $l$ :

$$P_{jl} = \frac{1}{|N(x'_j)|} \left| \{x_r \in N(x'_j) : y_r = l\} \right|, \quad 1 \leq j \leq m, \quad 0 \leq l < c. \quad (10.1)$$

3. Predict the class that has the highest probability for the feature vector  $x'_j$ :

$$y'_j = \arg \max_{0 \leq l < c} P_{jl}, \quad 1 \leq j \leq m. \quad (10.2)$$

### 10.7.2.1.1.5 Inference method: *brute-force*

The inference operation determines the set  $N(x'_j)$  of the nearest feature vectors by iterating over all the pairs  $(x'_j, x_i)$  in the implementation defined order,  $1 \leq i \leq n$ ,  $1 \leq j \leq m$ . The final prediction is computed according to the equations (10.1) and (10.2).

### 10.7.2.1.1.6 Inference method: *k-d tree*

The inference operation traverses the *k-d tree* to find feature vectors associated with a leaf node that are closest to  $x'_j$ ,  $1 \leq j \leq m$ . The set  $\tilde{n}(x'_j)$  of the currently-known nearest  $k$ -th neighbors is progressively updated during tree traversal. The search algorithm limits exploration of the nodes for which the distance between the  $x'_j$  and respective part of the feature space is not less than the distance between  $x'_j$  and the most distant feature vector from  $\tilde{n}(x'_j)$ . Once tree traversal is finished,  $\tilde{n}(x'_j) \equiv N(x'_j)$ . The final prediction is computed according to the equations (10.1) and (10.2).

## 10.7.2.1.2 Programming Interface

### 10.7.2.1.2.1 Descriptor

```
template <typename Float = float,
          typename Method = method::by_default>
class descriptor {
public:
    explicit descriptor(std::int64_t class_count,
                        std::int64_t neighbor_count);
```

(continues on next page)

(continued from previous page)

```
std::int64_t get_class_count() const;
descriptor& set_class_count(std::int64_t);

std::int64_t get_neighbor_count() const;
descriptor& set_neighbor_count(std::int64_t);
};
```

template<typename **Float** = float, typename **Method** = method:*by\_default*>  
**class descriptor**

#### Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
- **Method** – Tag-type that specifies an implementation of algorithm. Can be `method::bruteforce` or `method::kd_tree`.

#### Constructors

**descriptor**(std::int64\_t *class\_count*, std::int64\_t *neighbor\_count*)

Creates a new instance of the class with the given `class_count` and `neighbor_count` property values.

#### Properties

std::int64\_t **class\_count**

The number of classes *c*.

#### Getter & Setter

```
std::int64_t get_class_count() const
descriptor & set_class_count(std::int64_t)
```

#### Invariants

*class\_count* > 1

std::int64\_t **neighbor\_count**

The number of neighbors *k*.

#### Getter & Setter

```
std::int64_t get_neighbor_count() const
descriptor & set_neighbor_count(std::int64_t)
```

#### Invariants

*neighbor\_count* > 0

### 10.7.2.1.2.2 Computational methods

```
namespace method {
    struct bruteforce {};
    struct kd_tree {};
    using by_default = bruteforce;
} // namespace method
```

#### struct **bruteforce**

Tag-type that denotes `brute-force` computational method.

```
struct kd_tree
    Tag-type that denotes k-d tree computational method.
```

```
using by_default = bruteforce
    Alias tag-type for brute-force computational method.
```

#### 10.7.2.1.2.3 Model

```
class model {
public:
    model();
};
```

**class model**  
Constructors

```
model()
    Creates a new instance of the class with the default property values.
```

#### 10.7.2.1.2.4 Training train...

#### 10.7.2.1.2.5 Input

```
class train_input {
public:
    train_input(const table& data = table{},
                const table& labels = table {});

    const table& get_data() const;
    train_input& set_data(const table&);

    const table& get_labels() const;
    train_input& set_labels(const table&);
};
```

**class train\_input**  
Constructors

```
train_input(const table &data = table{}, const table &labels = table{})
```

Properties

```
const table &data = table{}
```

The training set  $X$ .

Getter & Setter

```
const table & get_data() const
    train_input & set_data(const table &)
```

```
const table &labels = table{}
```

Vector of labels  $y$  for the training set  $X$ .

Getter & Setter

```
const table & get_labels() const
    train_input & set_labels(const table &)
```

### 10.7.2.1.2.6 Result

```
class train_result {
public:
    train_result();
    const model& get_model() const;
};
```

**class train\_result**

**Constructors**

**train\_result()**

**Properties**

**const model &model = model{}**

The trained *k*-NN model.

**Getter & Setter**

const model & get\_model() const

### 10.7.2.1.2.7 Operation

```
template<typename Float, typename Method>
train_result train(const descriptor<Float, Method> &desc, const train_input &input)
```

Runs the training operation for *k*-NN classifier. For more details see `onedal::train`.

**Template Parameters**

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
- **Method** – Tag-type that specifies an implementation of algorithm. Can be `method::bruteforce` or `method::kd_tree`.

**Parameters**

- **desc** – Descriptor of the algorithm.
- **input** – Input data for the training operation.

**Preconditions**

```
input.data.has_data == true
input.labels.has_data == true
input.data.rows == input.labels.rows
input.data.columns == 1
input.labels[i] >= 0
input.labels[i] < desc.class_count
```

### 10.7.2.1.2.8 Inference infer...

#### 10.7.2.1.2.9 Input

```
class infer_input {
public:
    infer_input(const model& m = model{},
                const table& data = table{});

    const model& get_model() const;
    infer_input& set_model(const model&);

    const table& get_data() const;
    infer_input& set_data(const table&);
};
```

##### class infer\_input

###### Constructors

`infer_input(const model &m = model{}, const table &data = table{})`

###### Properties

`const model &model = model{}`

The trained  $k$ -NN model.

###### Getter & Setter

`const model & get_model() const`  
`infer_input & set_model(const model &)`

`const table &data = table{}`

The dataset for inference  $X'$ .

###### Getter & Setter

`const table & get_data() const`  
`infer_input & set_data(const table &)`

### 10.7.2.1.2.10 Result

```
class infer_result {
public:
    infer_result();

    const table& get_labels() const;
};
```

##### class infer\_result

###### Constructors

`infer_result()`

###### Properties

`const table &labels = model{}`

The predicted labels.

###### Getter & Setter

---

```
const table & get_labels() const
```

### 10.7.2.1.2.11 Operation

template<typename **Float**, typename **Method**>  
**infer\_result** **infer**(**const descriptor**<**Float**, **Method**> &**desc**, **const infer\_input** &**input**)  
Runs the inference operation for  $k$ -NN classifier. For more details see `onedal::infer`.

#### Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
- **Method** – Tag-type that specifies an implementation of algorithm. Can be `method::bruteforce` or `method::kd_tree`.

#### Parameters

- **desc** – Descriptor of the algorithm.
- **input** – Input data for the inference operation.

#### Preconditions

```
input.data.has_data == true
```

#### Preconditions

```
result.labels.rows == input.data.rows
result.labels.columns == 1
result.labels[i] >= 0
result.labels[i] < desc.class_count
```

## 10.7.3 Decomposition

### 10.7.3.1 Principal Components Analysis (PCA)

Principal Component Analysis (PCA) is an algorithm for exploratory data analysis and dimensionality reduction. PCA transforms a set of feature vectors of possibly correlated features to a new set of uncorrelated features, called principal components. Principal components are the directions of the largest variance, that is, the directions where the data is mostly spread out.

Given the training set  $X = \{x_1, \dots, x_n\}$  of  $p$ -dimensional feature vectors and the number of principal components  $r$ , the problem is to compute  $r$  principal directions ( $p$ -dimensional eigenvectors) for the training set. The eigenvectors can be grouped into the  $r \times p$  matrix  $T$  that contains one eigenvector in each row.

oneDAL specifies two methods for PCA computation:

1. *Covariance-based method*
2. *SVD-based method*

#### 10.7.3.1.1 Covariance-based method

[TBD]

#### 10.7.3.1.2 SVD-based method

[TBD]

#### 10.7.3.1.3 Sign-flip technique

Eigenvectors computed by some eigenvalue solvers are not uniquely defined due to sign ambiguity. To get the deterministic result, a sign-flip technique should be applied. One of the sign-flip techniques proposed in [Bro07] requires the following modification of matrix  $T$ :

$$\hat{T}_i = T_i \cdot \operatorname{sgn}(\max_{1 \leq j \leq p} |T_{ij}|), \quad 1 \leq i \leq r,$$

where  $T_i$  is  $i$ -th row,  $T_{ij}$  is the element in the  $i$ -th row and  $j$ -th column,  $\operatorname{sgn}(\cdot)$  is the signum function,

$$\operatorname{sgn}(x) = \begin{cases} -1, & x < 0, \\ 0, & x = 0, \\ 1, & x > 0. \end{cases}$$

---

**Note:** The sign-flip technique described above is an example. oneDAL spec does not require implementation of this sign-flip technique. Implementer can choose an arbitrary technique that modifies the eigenvectors' signs.

---

#### 10.7.3.1.4 Usage example

```
onedal::pca::model run_training(const onedal::table& data) {

    const auto pca_desc = onedal::pca::desc<float>{}
        .set_component_count(5)
        .set_deterministic(true);

    const auto result = onedal::train(pca_desc, data);

    print_table("means", result.get_means());
    print_table("variances", result.get_variances());
    print_table("eigenvalues", result.get_eigenvalues());
    print_table("eigenvectors", result.get_model().get_eigenvectors());

    return result.get_model();
}
```

```
onedal::table run_inference(const onedal::pca::model& model,
                           const onedal::table& new_data) {

    const auto pca_desc = onedal::pca::desc<float>{}
        .set_component_count(model.get_component_count());
```

(continues on next page)

(continued from previous page)

```
const auto result = onedal::infer(pca_desc, model, new_data);

print_table("labels", result.get_transformed_data());
}
```

### 10.7.3.1.5 API

#### 10.7.3.1.5.1 Methods

```
namespace method {
    struct cov {};
    struct svd {};
    using by_default = cov;
} // namespace method
```

##### struct cov

Tag-type that denotes *Covariance-based method*.

##### struct svd

Tag-type that denotes *SVD-based method*.

##### using by\_default = cov

Alias tag-type for the *Covariance-based method*.

#### 10.7.3.1.5.2 Descriptor

```
template <typename Float = float,
          typename Method = method::by_default>
class desc {
public:
    desc();

    int64_t get_component_count() const;
    bool get_deterministic() const;

    desc& set_component_count(int64_t);
    desc& set_deterministic(bool);
};
```

```
template<typename Float = float, typename Method = method::by_default>
class desc
```

##### Template Parameters

- **Float** – The floating-point type that the algorithm uses for intermediate computations. Can be `float` or `double`.
- **Method** – Tag-type that specifies an implementation of PCA algorithm. Can be `method::cov`, `method::svd` or `method::by_default`.

##### desc()

Creates a new instance of the descriptor with the default attribute values.

```
std::int64_t component_count = 0
```

The number of principal components  $r$ . If it is zero, the algorithm computes the eigenvectors for all features,  $r = p$ .

#### Getter & Setter

```
std::int64_t get_component_count() const  
desc& set_component_count(std::int64_t)
```

#### Invariants

*component\_count*  $\geq 0$

bool **set\_deterministic** = true

Specifies whether the algorithm applies the *Sign-flip technique* or uses a deterministic eigenvalues solver. If it is *true*, directions of the eigenvectors must be deterministic.

#### Getter & Setter

```
bool get_deterministic() const  
desc& set_deterministic(bool)
```

### 10.7.3.1.5.3 Model

```
class model {  
public:  
    model();  
  
    const table& get_eigenvectors() const;  
    int64_t get_component_count() const;  
};
```

**class model**

**model()**

Creates a model with the default attribute values.

*table* **eigenvectors** = *table*()

$r \times p$  table with the eigenvectors. Each row contains one eigenvector.

#### Getter

```
const table& get_eigenvectors() const
```

**std::int64\_t component\_count = 0**

The number of components  $r$  in the trained model.

#### Getter

```
std::int64_t get_component_count() const
```

#### Invariants

*component\_count* == *eigenvectors.row\_count*

#### 10.7.3.1.5.4 Training `onedal::train...`

##### 10.7.3.1.5.5 Input

```
class train_input {
public:
    train_input();
    train_input(const table& data);

    const table& get_data() const;

    train_input& set_data(const table&);
};
```

**class train\_input**

**train\_input()**

Creates an input for the training operation with the default attribute values.

**train\_input(const table &data)**

Creates an input for the training operation with the given *data*.

**table data = table()**

$n \times p$  table with the training data, where each row stores one feature vector.

##### Getter & Setter

```
const table& get_data() const
train_input& set_data(const table&)
```

#### 10.7.3.1.5.6 Result

```
class train_result {
public:
    train_result();

    const model& get_model() const;
    const table& get_means() const;
    const table& get_variances() const;
    const table& get_eigenvalues() const;
};
```

**class train\_result**

**train\_result()**

Creates a result of the training operation with the default attribute values.

**pca::model model = pca::model()**

The trained PCA model.

##### Getter

```
const model& get_model() const
```

**table means = table()**

$1 \times r$  table that contains mean value for the first  $r$  features.

**Getter**

```
const table& get_means() const





```

$1 \times r$  table that contains variance for the first  $r$  features.

**Getter**

```
const table& get_variances() const





```

$1 \times r$  table that contains eigenvalue for for the first  $r$  features.

**Getter**

```
const table& get_eigenvalues() const
```

**10.7.3.1.5.7 Operation semantics**

```
template<typename Descriptor>
pca::train_result train(const Descriptor &desc, const pca::train_input &input)
```

**Template Parameters** **Descriptor** – PCA algorithm descriptor `pca::desc`.

**Preconditions**

```
input.data.is_empty == false
input.data.column_count >= desc.component_count
```

**Postconditions**

```
result.means.row_count == 1
result.means.column_count == desc.component_count
result.variances.row_count == 1
result.variances.column_count == desc.component_count
result.variances >= 0.0
result.eigenvalues.row_count == 1
result.eigenvalues.column_count == desc.component_count
result.model.eigenvectors.row_count == 1
result.model.eigenvectors.column_count == desc.component_count
```

**10.7.3.1.5.8 Inference onedal::infer...****10.7.3.1.5.9 Input**

```
class infer_input {
public:
    infer_input();
    infer_input(const model& m);
    infer_input(const model& m, const table& data);

    const model& get_model() const;
    const table& get_data() const;
```

(continues on next page)

(continued from previous page)

```
infer_input& set_model(const model&);
infer_input& set_data(const table&);
};
```

**class infer\_input**

**infer\_input()**

Creates an input for the inference operation with the default attribute values.

**infer\_input(**const** pca::model &model)**

Creates an input for the inference operation with the given *model*, the other attributes get default values.

**infer\_input(**const** pca::model &model, **const** table &data)**

Creates an input for the inference operation with the given *model* and *data*.

**table data = table()**

*n × p* table with the data to be projected to the *r* principal components previously extracted from a training set.

#### Getter & Setter

```
const table& get_data() const
infer_input& set_data(const table&)
```

**pca::model model = pca::model()**

The trained PCA model (see `pca::model`).

#### Getter & Setter

```
const pca::model& get_model() const
infer_input& set_model(const pca::model&)
```

### 10.7.3.1.5.10 Result

```
class infer_result {
public:
    infer_result();
    const table& get_transformed_data() const;
};
```

**class infer\_result**

**infer\_result()**

Creates a result of the inference operation with the default attribute values.

**table transformed\_data = table()**

*n × r* table that contains data projected to the *r* principal components.

#### Getter

```
const table& get_transformed_data() const
```

### 10.7.3.1.5.11 Operation semantics

```
template<typename Descriptor>
pca::infer_result infer(const Descriptor &desc, const pca::infer_input &input)
```

**Template Parameters** **Descriptor** – PCA algorithm descriptor `pca::desc`.

#### Preconditions

```
input.data.is_empty == false
input.model.eigenvectors.row_count == desc.component_count
input.model.eigenvectors.column_count = input.data.column_count
```

#### Postconditions

```
result.transformed_data.row_count == input.data.row_count
result.transformed_data.column_count == desc.component_count
```

## 10.8 Appendix

### 10.8.1 k-d Tree

*k-d* tree is a space-partitioning binary tree [Bentley80], where

- Each non-leaf node induces the hyperplane that splits the feature space into two parts. To define the splitting hyperplane explicitly, a non-leaf node stores the identifier of the feature (that defines axis in the feature space) and *a cut-point*
- Each leaf node of the tree has an associated subset (*a bucket*) of elements of the training data set. Feature vectors from a bucket belong to the region of the space defined by tree nodes on the path from the root node to the respective leaf.

#### 10.8.1.1 Related terms

**A cut-point** A feature value that corresponds to a non-leaf node of a *k-d* tree and defines the splitting hyperplane orthogonal to the axis specified by the given feature.

## 10.9 Bibliography

For more information about algorithms implemented in oneAPI Data Analytics Library (oneDAL), refer to the following publications:

## 11.1 General Information

### 11.1.1 Introduction

[intro]

This document specifies requirements for implementations of oneAPI Threading Building Blocks (oneTBB).

oneAPI Threading Building Blocks is a programming model for scalable parallel programming using standard ISO C++ code. A program uses oneTBB to specify logical parallelism in its algorithms, while a oneTBB implementation maps that parallelism onto execution threads.

oneTBB employs generic programming via C++ templates, with most of its interfaces defined by requirements on types and not specific types. Generic programming makes oneTBB flexible yet efficient, through customizing APIs to specific needs of an application.

C++11 is the minimal version of the C++ standard required by oneTBB. An implementation of oneTBB shall not require newer versions of the standard except where explicitly specified; also it shall not require any non-standard language extensions.

An implementation may use platform-specific APIs if those are compatible with the C++ execution and memory models. For example, a platform-specific implementation of threads may be used if that implementation provides the same execution guarantees as C++ threads.

An implementation of oneTBB shall support execution on single core and multi-core CPUs, including those that provide simultaneous multithreading capabilities.

On CPU, an implementation shall support nested parallelism, so that one can build larger parallel components from smaller parallel components.

### 11.1.2 Notational Conventions

[notational\_conventions]

The following conventions may be used in this document.

Convention	Explanation	Example
<i>Italic</i>	Used for introducing new terms, denotation of terms, placeholders, or titles of manuals.	The filename consists of the <i>basename</i> and the <i>extension</i> . For more information, refer to the <i>TBB Developer Guide</i> .
Monospaced	Indicates directory paths and filenames, commands and command line options, function names, methods, classes, data structures in body text, source code.	tbb.h \alt\include Use the okCreateObjs() function to... printf("hello, world\n");
Monospaced italic	Indicates source code placeholders.	blocked_range<Type>
Monospaced bold	Emphasizes parts of source code.	x = ( h > 0 ? sizeof(m) : 0xF ) + min;
[ ]	Items enclosed in brackets are optional.	Fa[c] Indicates Fa or F ac.
{   }	Braces and vertical bars indicate the choice of one item from a selection of two or more items.	X{K   W   P} Indicates XK, XW, or XP.
“[” “]” “{” “ }” “ ”	Writing a metacharacter in quotation marks negates the syntactical meaning stated above; the character is taken as a literal.	“[” X “]” [ Y ] Denotes the letter X enclosed in brackets, optionally followed by the letter Y.
...	The ellipsis indicates that the previous item can be repeated several times.	<b>filename</b> ... Indicates that one or more filenames can be specified.
....	The ellipsis preceded by a comma indicates that the previous item can be repeated several times, separated by commas.	<b>word</b> ,... Indicates that one or more words can be specified. If more than one word is specified, the words are comma-separated.

Class members are summarized by informal class declarations that describe the class as it seems to clients, not how it is actually implemented. For example, here is an informal declaration of class Foo:

```
class Foo {
public:
    int x();
    int y;
    ~Foo();
};
```

The actual implementation might look like:

```
namespace internal {
    class FooBase {
        protected:
            int x();
    };

    class Foo_v3: protected FooBase {
        private:
            int internal_stuff;
        public:
            using FooBase::x;
            int y;
    };
};
```

(continues on next page)

(continued from previous page)

```

}

typedef internal::Foo_v3 Foo;

```

The example shows two cases where the actual implementation departs from the informal declaration:

- `Foo` is actually a `typedef` to `Foo_v3`.
- Method `x()` is inherited from a protected base class.
- The destructor is an implicit method generated by the compiler.

The informal declarations are intended to show you what you need to know to use the class without the distraction of irrelevant clutter particular to the implementation.

### 11.1.3 Identifiers

#### [identifiers]

This section describes the identifier conventions used by oneAPI Threading Building Blocks.

#### 11.1.3.1 Case

The identifier convention in the library follows the style in the ISO C++ standard library. Identifiers are written in `underscore_style`, and concepts in `PascalCase`.

#### 11.1.3.2 Reserved Identifier Prefixes

The library reserves the prefix `__TBB` for internal identifiers and macros that should never be directly referenced by your code.

### 11.1.4 Named Requirements

#### [named\_requirements]

This section describes named requirements used in the oneAPI Threading Building Blocks Specification.

A *named requirement* is a set of requirements on a type. The requirements may be syntactic or semantic. The *named\_requirement* term is similar to “Requirements on types and expressions” term which is defined by the ISO C++ Standard (chapter “Library Introduction”) or “Named Requirements” section on the [cppreference.com](http://cppreference.com) site.

For example, the named requirement of *sortable* could be defined as a set of requirements that enable an array to be sorted. A type `T` would be *sortable* if:

- `x < y` returns a boolean value, and represents a total order on items of type `T`.
- `swap(x, y)` swaps items `x` and `y`

You can write a sorting template function in C++ that sorts an array of any type that is *sortable*.

Two approaches for defining named requirements are *valid expressions* and *pseudo-signatures*. The ISO C++ standard follows the *valid expressions* approach, which shows what the usage pattern looks like for a requirement. It has the drawback of relegating important details to notational conventions. This document uses *pseudo-signatures*, because they are concise, and can be cut-and-pasted for an initial implementation.

For example, the table below shows *pseudo-signatures* for a *sortable* type `T`:

## Sortable Requirements : Pseudo-Signature, Semantics

`bool operator< (const T &x, const T &y)`  
Compare x and y.

`void swap (T &x, T &y)`  
Swap x and y.

A real signature may differ from the pseudo-signature that it implements in ways where implicit conversions would deal with the difference. For an example type `U`, the real signature that implements `operator<` in the table above can be expressed as `int operator< ( U x, U y )`, because C++ permits implicit conversion from `int` to `bool`, and implicit conversion from `U` to `(const U&)`. Similarly, the real signature `bool operator< ( U& x, U& y )` is acceptable because C++ permits implicit addition of a `const` qualifier to a reference type.

### 11.1.4.1 Algorithms

#### 11.1.4.1.1 Range

##### [req.range]

A *Range* can be recursively subdivided into two parts. Subdivision is done by calling *splitting constructor* of *Range*. There are two types of splitting constructors:

- Basic splitting constructor. It is recommended that the division be into nearly equal parts in this constructor, but it is not required. Splitting as evenly as possible typically yields the best parallelism.
- Proportional splitting constructor. This constructor is optional and can be omitted. When using this type of constructor, for the best results, follow the given proportion with rounding to the nearest integer if necessary.

Ideally, a range is recursively splittable until the parts represent portions of work that are more efficient to execute serially rather than split further. The amount of work represented by a Range typically depends upon higher level context, hence a typical type that models a Range should provide a way to control the degree of splitting. For example, the template class `blocked_range` has a `grainsize` parameter that specifies the biggest range considered indivisible.

If the set of values has a sense of direction, then by convention the splitting constructor should construct the second part of the range, and update its argument to be the first part of the range. This enables `parallel_for`, `parallel_reduce` and `parallel_scan` algorithms, when running sequentially, to work across a range in the increasing order, typical of an ordinary sequential loop.

Since a Range declares a splitting and copy constructors, the default constructor for it will not be automatically generated. You will need to explicitly define the default constructor or add any other constructor to create an instance of Range type in the program.

A type `R` meets the *Range* if it satisfies the following requirements:

#### Range Requirements: Pseudo-Signature, Semantics

`R::R (const R&)`  
Copy constructor.

`R::~R ()`  
Destructor.

`bool R::empty () const`  
True if range is empty.

`bool R::is_divisible() const`

True if range can be partitioned into two subranges.

`R::R(R &r, split)`

Basic splitting constructor. Splits `r` into two subranges.

`R::R(R &r, proportional_split proportion)`

**Optional.** Proportional splitting constructor. Splits `r` into two subranges in accordance with `proportion`.

See also:

- [blocked\\_range class](#)
- [blocked\\_range2d class](#)
- [blocked\\_range3d class](#)
- [parallel\\_reduce algorithm](#)
- [parallel\\_for algorithm](#)
- [split class](#)

#### 11.1.4.1.2 Splittable

[req.splittable]

A type is splittable if it has a *splitting constructor* that allows an instance to be split into two pieces. The splitting constructor takes as arguments a reference to the original object, and a dummy argument of type `split`, which is defined by the library. The dummy argument distinguishes the splitting constructor from a copy constructor. After the constructor runs, `x` and the newly constructed object should represent the two pieces of the original `x`. The library uses splitting constructors in two contexts:

- *Partitioning* a range into two subranges that can be processed concurrently.
- *Forking* a body (function object) into two bodies that can run concurrently.

Types that meets [Range requirements](#) may additionally defines an optional *proportional splitting constructor*, distinguished by an argument of type [proportional\\_split Class](#).

A type `X` satisfies the *Splittable* if it meets the following requirements:

#### Splittable Requirements: Pseudo-Signature, Semantics

`X::X(X &x, split)`

Split `x` into `x` and newly constructed object.

See also:

- [Range requirements](#)

### 11.1.4.1.3 ParallelForBody

#### [req.parallel\_for\_body]

A type *Body* satisfies the *ParallelForBody* if it meets the following requirements:

#### ParallelForBody Requirements: Pseudo-Signature, Semantics

*Body* :: **Body** (**const Body&**)

Copy constructor.

*Body* :: ~**Body** ()

Destructor.

void *Body* :: **operator()** (*Range &range*) **const**

Apply body to range. Range type shall meet the *Range requirements*.

See also:

- *parallel\_for algorithm*

### 11.1.4.1.4 ParallelReduceBody

#### [req.parallel\_reduce\_body]

A type *Body* satisfies the *ParallelReduceBody* if it meets the following requirements:

#### ParallelReduceBody Requirements: Pseudo-Signature, Semantics

*Body* :: **Body** (*Body&*, *split*)

Splitting constructor. Must be able to run concurrently with *operator()* and method *join*.

*Body* :: ~**Body** ()

Destructor.

void *Body* :: **operator()** (**const Range &range**)

Accumulate result for subrange. Range type shall meet the *Range requirements*.

void *Body* :: **join** (*Body &rhs*)

Join results. The result in rhs should be merged into the result of *this*.

See also:

- *parallel\_reduce algorithm*
- *parallel\_deterministic\_reduce algorithm*

### 11.1.4.1.5 ParallelReduceFunc

#### [req.parallel\_reduce\_body]

A type *Func* satisfies the *ParallelReduceFunc* if it meets the following requirements:

#### ParallelReduceFunc Requirements: Pseudo-Signature, Semantics

Value Func`::operator()` (`const Range &range, const Value &x`) `const`

Accumulate result for subrange, starting with initial value x. Range type shall meet the *Range requirements*.

Value type must be the same as a corresponding template parameter for *parallel\_reduce algorithm* algorithm.

See also:

- *parallel\_reduce algorithm*
- *parallel\_deterministic\_reduce algorithm*

#### 11.1.4.1.6 ParallelReduceReduction

**[req.parallel\_reduce\_reduction]**

A type *Reduction* satisfies the *ParallelReduceReduction* if it meets the following requirements:

##### ParallelReduceReduction Requirements: Pseudo-Signature, Semantics

Value Reduction`::operator()` (`const Value &x, const Value &y`) `const`

Combine results x and y. Value type must be the same as a corresponding template parameter for *parallel\_reduce algorithm* algorithm.

See also:

- *parallel\_reduce algorithm*
- *parallel\_deterministic\_reduce algorithm*

#### 11.1.4.1.7 ParallelForEachBody

**[req.parallel\_for\_each\_body]**

A type *Body* satisfies the *ParallelForBody* if it meet the *Function Objects* requirements from [function.objects] ISO C++ Standard section. Also it should meet one of the following requirements:

##### ParallelForEachBody Requirements: Pseudo-Signature, Semantics

Body`::operator()` (`ItemType item`) `const`

Process the received item.

Body`::operator()` (`ItemType item, tbb::feeder<ItemType> &feeder`) `const`

Process the received item. May invoke `feeder.add(T)` function to spawn the additional items.

**Note:** ItemType may be optionally passed to Body`::operator()` by reference. const and volatile type qualifiers are also applicable.

### 11.1.4.1.7.1 ItemType

The argument type `ItemType` should either satisfy the *CopyConstructible*, *MoveConstructible* or both requirements from ISO C++ [utility.arg.requirements] section. If the type is not *CopyConstructible*, there are additional usage restrictions:

- If `Body::operator()` accepts its argument by value, or if the `InputIterator` type from *parallel\_for\_each algorithm* does not also satisfy the *Forward Iterator* requirements from [forward.iterators] ISO C++ Standard section, then dereferencing an `InputIterator` must produce an rvalue reference.
- Additional work items should be passed to the feeder as rvalues, for example via the `std::move` function.

See also:

- *parallel\_for\_each algorithm*
- *feeder class*

### 11.1.4.1.8 ContainerBasedSequence

#### [req.container\_based\_sequence]

A type *T* satisfies the *ContainerBasedSequence* if it is an array type or a type that meets the following requirements:

---

#### ContainerBasedSequence Requirements: Pseudo-Signature, Semantics

##### `T::begin()`

Returns an iterator to the first element of the contained sequence.

##### `T::end()`

Returns an iterator to the element behind the last element of the contained sequence.

### 11.1.4.1.9 ParallelScanBody

#### [req.parallel\_scan]

A type *Body* satisfies the *ParallelScanBody* if it meets the following requirements:

---

#### ParallelScanBody Requirements: Pseudo-Signature, Semantics

##### `void Body::operator() (const Range &r, pre_scan_tag)`

Accumulate summary for range *r*. For example, if computing a running sum of an array, the summary for a range *r* is the sum of the array elements corresponding to *r*.

##### `void Body::operator() (const Range &r, final_scan_tag)`

Compute scan result and summary for range *r*.

##### `Body::Body (Body &b, split)`

Split *b* so that *this* and *b* can accumulate summaries separately.

##### `void Body::reverse_join (Body &b)`

Merge summary accumulated by *b* into summary accumulated by *this*, where *this* was created earlier from *b* by splitting constructor.

##### `void Body::assign (Body &b)`

Assign summary of *b* to *this*.

#### 11.1.4.1.10 ParallelScanCombine

##### [req.parallel\_scan\_combine]

A type *Combine* satisfies the *ParallelScanCombine* if it meets the following requirements:

---

##### ParallelScanCombine Requirements: Pseudo-Signature, Semantics

**Value Combine::operator() (const Value &left, const Value &right) const**  
 Combine summaries *left* and *right*, and return the result *Value* type must be the same as a corresponding template parameter for *parallel\_scan* algorithm.

#### 11.1.4.1.11 ParallelScanFunc

##### [req.parallel\_scan\_func]

A type *Scan* satisfies the *ParallelScanFunc* if it meets the following requirements:

---

##### ParallelScanFunc Requirements: Pseudo-Signature, Semantics

**Value Scan::operator() (const Range &r, const Value &sum, bool is\_final) const**  
 Starting with *sum*, compute the summary and, for *is\_final* == true, the scan result for range *r*. Return the computed summary. *Value* type must be the same as a corresponding template parameter for *parallel\_scan* algorithm.

#### 11.1.4.1.12 BlockedRangeValue

##### [req.blocked\_range\_value]

A type *Value* satisfies the *BlockedRangeValue* if it meets the following requirements:

---

##### BlockedRangeValue Requirements: Pseudo-Signature, Semantics

**Value::Value(const Value&)**

Copy constructor.

**Value::~Value()**

Destructor.

**void operator=(const Value&)**

---

**Note:** The return type **void** in the pseudo-signature denotes that **operator=** is not required to return a value. The actual **operator=** can return a value, which will be ignored by **blocked\_range**.

Assignment.

**bool operator<(const Value &i, const Value &j)**

Value *i* precedes value *j*.

**D operator-(const Value &i, const Value &j)**

Number of values in range [i, j].

Value **operator+** (**const** Value &*i*, D *k*)  
*k*-th value after *i*.

D is the type of the expression *j* – *i*. It can be any integral type that is convertible to `size_t`. Examples that model the Value requirements are integral types, pointers, and STL random-access iterators whose difference can be implicitly converted to a `size_t`.

#### 11.1.4.1.13 FilterBody

##### [req.filter\_body]

A type *Body* should meet one of the following requirements depending on the filter type:

---

##### MiddleFilterBody Requirements: Pseudo-Signature, Semantics

**OutputType Body::operator() (InputType item) const**  
 Process the received item and then returns it.

---

##### FirstFilterBody Requirements: Pseudo-Signature, Semantics

**OutputType Body::operator() (tbb::flow\_control fc) const**  
 Returns the next item from an input stream. Calls `fc.stop()` at the end of an input stream.

---

##### LastFilterBody Requirements: Pseudo-Signature, Semantics

**void Body::operator() (InputType item) const**  
 Process the received item.

---

##### SingleFilterBody Requirements: Pseudo-Signature, Semantics

**void Body::operator() (tbb::flow\_control fc) const**  
 Process element from an input stream. Calls `fc.stop()` at the end of an input stream.

#### 11.1.4.2 Mutexes

##### 11.1.4.2.1 Mutex

##### [req.mutex]

The mutexes and locks here have relatively spartan interfaces that are designed for high performance. The interfaces enforce the *scoped locking pattern*, which is widely used in C++ libraries because:

- Does not require the programmer to remember to release the lock
- Releases the lock if an exception is thrown out of the mutual exclusion region protected by the lock

There are two parts to the pattern: a *mutex* object, for which construction of a *lock* object acquires a lock on the mutex and destruction of the *lock* object releases the lock. Here's an example:

```
{
    // Construction of myLock acquires lock on myMutex
    M::scoped_lock myLock( myMutex );
    // ... actions to be performed while holding the lock ...
    // Destruction of myLock releases lock on myMutex
}
```

If the actions throw an exception, the lock is automatically released as the block is exited.

```
class M {
    // Implementation specifics
    // ...

    // Represents acquisition of a mutex
    class scoped_lock {
        public:
            constexpr scoped_lock() noexcept;
            scoped_lock(M& m);
            ~scoped_lock();

            scoped_lock(const scoped_lock&) = delete;
            scoped_lock& operator=(const scoped_lock&) = delete;

            void acquire(M& m);
            bool try_acquire(M& m);
            void release();
    };
};
```

A type *M* satisfies the *Mutex* if it meets the following requirements:

**type M::scoped\_lock**

Corresponding scoped lock type.

**M::scoped\_lock()**

Construct scoped\_lock without acquiring mutex.

**M::scoped\_lock(M&)**

Construct scoped\_lock and acquire the lock on a provided mutex.

**M::~scoped\_lock()**

Releases a lock (if acquired).

**void M::scoped\_lock::acquire(M&)**

Acquire a lock on a provided mutex.

**bool M::scoped\_lock::try\_acquire(M&)**

Attempts to acquire a lock on a provided mutex. Returns true if the lock is acquired, false otherwise.

**void M::scoped\_lock::release()**

Releases an acquired lock.

Also, the Mutex type requires a set of traits to be defined:

**static constexpr bool M::is\_rw\_mutex**

True if mutex is reader-writer mutex; false otherwise.

**static constexpr bool M::is\_recursive\_mutex**

True if mutex is recursive mutex; false otherwise.

```
static constexpr bool M::is_fair_mutex
    True if mutex is fair; false otherwise.
```

A mutex type and an M::scoped\_lock type are neither copyable nor movable.

The following table summarizes the library classes that model the Mutex requirement and provided guarantees.

Table 1: Provided guarantees for Mutexes that model the Mutex requirement

.	Fair	Reentrant
spin_mutex	No	No
speculative_spin_mutex	No	No
queuing_mutex	Yes	No
null_mutex	Yes	Yes

---

**Note:** Implementation is allowed to have an opposite guarantees (positive) in case of negative statements from the table above.

---

See the oneAPI Threading Building Blocks Developer Guide for a discussion of the mutex properties and the rationale for null mutexes.

See also:

- [spin\\_mutex](#)
- [speculative\\_spin\\_mutex](#)
- [queuing\\_mutex](#)
- [null\\_mutex](#)

#### 11.1.4.2.2 ReaderWriterMutex

##### [req.rw\_mutex]

The *ReaderWriterMutex* requirement extends the [Mutex Requirement](#) to include the notion of reader-writer locks. It introduces a boolean parameter `write` that specifies whether a writer lock (`write = true`) or reader lock (`write = false`) is being requested. Multiple reader locks can be held simultaneously on a *ReaderWriterMutex* if it does not have a writer lock on it. A writer lock on a *ReaderWriterMutex* excludes all other threads from holding a lock on the mutex at the same time.

```
class RWM {
    // Implementation specifics
    // ...

    // Represents acquisition of a mutex.
    class scoped_lock {
        public:
            constexpr scoped_lock() noexcept;
            scoped_lock(RWM& m, bool write = true);
            ~scoped_lock();

            scoped_lock(const scoped_lock&) = delete;
            scoped_lock& operator=(const scoped_lock&) = delete;
    };
}
```

(continues on next page)

(continued from previous page)

```

void acquire(RWM& m, bool write = true);
bool try_acquire(RWM& m, bool write = true);
void release();

bool upgrade_to_writer();
bool downgrade_to_reader();
};

};

```

A type *RWM* satisfies the *ReaerWriterMutex* if it meets the following requirements. They form a superset of the *Mutex requirements*.

**type** RWM::**scoped\_lock**

Corresponding scoped-lock type.

RWM::**scoped\_lock**()

Constructs scoped\_lock without acquiring any mutex.

RWM::**scoped\_lock**(RWM&, **bool** write = **true**)

Constructs scoped\_lock and acquire a lock on a given mutex. The lock is a writer lock if write is true; a reader lock otherwise.

RWM::**~scoped\_lock**()

Releases a lock (if acquired).

**void** RWM::**scoped\_lock**::**acquire**(RWM&, **bool** write = **true**)

Acquires lock on a given mutex. The lock is a writer lock if write is true; a reader lock otherwise.

**bool** RWM::**scoped\_lock**::**try\_acquire**(RWM&, **bool** write = **true**)

Attempts to acquire a lock on a given mutex. The lock is a writer lock if write is true; a reader lock otherwise. Returns **true** if the lock is acquired, **false** otherwise.

RWM::**scoped\_lock**::**release**()

Releases a lock. The effect is undefined if no lock is held.

**bool** RWM::**scoped\_lock**::**upgrade\_to\_writer**()

Changes a reader lock to a writer lock. Return **false** if lock was released and reacquired. **true** otherwise, including if the lock was already a writer lock.

**bool** RWM::**scoped\_lock**::**downgrade\_to\_reader**()

Changes a writer lock to a reader lock. Return **false** if lock was released and reacquired. **true** otherwise, including if the lock was already a reader lock.

Like the *Mutex* requirement, *ReaderWriterMutex* also requires a set of traits to be defined.

**static constexpr** **bool** M::**is\_rw\_mutex**

True if mutex is reader-writer mutex; false otherwise.

**static constexpr** **bool** M::**is\_recursive\_mutex**

True if mutex is recursive mutex; false otherwise.

**static constexpr** **bool** M::**is\_fair\_mutex**

True if mutex is fair; false otherwise.

The following table summarizes the library classes that model the *ReaderWriterMutex* requirement and provided guarantees.

Table 2: Provided guarantees for Mutexes that model the ReaderWriter-Mutex requirement

.	Fair	Reentrant
<i>spin_rw_mutex</i>	No	No
<i>speculative_spin_rw_mutex</i>	No	No
<i>queuing_rw_mutex</i>	Yes	No
<i>null_rw_mutex</i>	Yes	Yes

---

**Note:** Implementation is allowed to have an opposite guarantees (positive) in case of negative statements from the table above.

---

**Note:** For all currently provided reader-writer mutexes,

- `is_recursive_mutex` is `false`;
- `scoped_lock:: downgrade_to_reader` always returns `true`.

However, other implementations of the ReaderWriterMutex requirement are not required to do the same.

---

See also:

- *spin\_rw\_mutex*
- *speculative\_spin\_rw\_mutex*
- *queuing\_rw\_mutex*
- *null\_rw\_mutex*

#### 11.1.4.3 Containers

##### 11.1.4.3.1 HashCompare

###### [req.hash\_compare]

HashCompare is an object which is used to calculate hash code for an object and compare two objects for equality.

The type `H` satisfies `HashCompare` if it meets the following requirements:

---

###### HashCompare Requirements: Pseudo-Signature, Semantics

`H::H(const H&)`

Copy constructor

`H::~H()`

Destructor

`std::size_t H::hash(const KeyType &k)`

Calculates the hash for provided key.

`ReturnType H::equal(const KeyType &k1, const KeyType &k2)`

Requirements:

- The type `ReturnType` should be implicitly convertible to `bool`

Compares k1 and k2 for equality.

If this function returns `true` then `H::hash(k1)` should be equal to `H::hash(k2)`.

#### 11.1.4.3.2 ContainerRange

##### [req.container\_range]

`ContainerRange` is a range that represents a concurrent container or a part of the container.

`ContainerRange` object can be used to traverse the container in parallel algorithms like `parallel_for`.

The type `CR` satisfies the `ContainerRange` requirements if:

- The type `CR` meets the requirements of *Range requirements*.
- The type `CR` provides the following member types and functions:

**type CR::value\_type**  
The type of the item in the range.

**type CR::reference**  
Reference type to the item in the range.

**type CR::const\_reference**  
Constant reference type to the item in the range.

**type CR::iterator**  
Iterator type for range traversal.

**type CR::size\_type**  
Unsigned integer type for obtaining grainsize.

**type CR::difference\_type**  
The type of the difference between two iterators

**iterator CR::begin()**  
Returns iterator to the beginning of the range.

**iterator CR::end()**  
Returns iterator follows the last element in the range.

**size\_type CR::grainsize() const**  
Retuns the range grainsize.

#### 11.1.4.4 Task scheduler

##### 11.1.4.4.1 SuspendFunc

##### [req.suspend\_func]

A type `Func` satisfies the `SuspendFunc` if it meets the following requirements:

---

##### SuspendFunc Requirements: Pseudo-Signature, Semantics

`Func::Func(const Func&)`

Copy constructor.

`void Func::operator()(tbb::task::suspend_point)`

Body that accepts the current task execution point to resume later.

See also:

- *resumable tasks*

#### 11.1.4.5 Flow Graph

##### 11.1.4.5.1 AsyncNodeBody

###### [req.async\_node\_body]

A type *Body* satisfies the *AsyncNodeBody* if it meets the following requirements:

###### **AsyncNodeBody Requirements: Pseudo-Signature, Semantics**

*Body* : :**Body** (**const** *Body*&)

Copy constructor.

*Body* : :~**Body** ()

Destructor.

void **operator=** (**const** *B&*)

Assignment.

void *Body* : :**operator()** (**const** *Input* &*v*, *GatewayType* &*gateway*)

**Requirements:**

- The *Input* type must be the same as the *Input* template type argument of the *async\_node* instance in which *Body* object is passed during construction.
- The *GatewayType* type must be the same as the *gateway\_type* member type of the *async\_node* instance in which *Body* object is passed during construction.

The input value *v* is submitted by the flow graph to an external activity. The *gateway interface* allows the external activity to communicate with the enclosing flow graph.

#### 11.1.4.5.2 ContinueNodeBody

###### [req.continue\_node\_body]

A type *Body* satisfies the *ContinueNodeBody* if it meets the following requirements:

###### **ContinueNodeBody Requirements: Pseudo-Signature, Semantics**

*Body* : :**Body** (**const** *Body*&)

Copy constructor.

*Body* : :~**Body** ()

Destructor.

Output *Body* : :**operator()** (**const** *continue\_msg* &*v*) **const**

**Requirements:** The type *Output* must be the same as template type argument *Output* of the *continue\_node* instance in which *Body* object is passed during construction.

Perform operation and return value of type *Output*.

See also:

- *continue\_node class*

- *continue\_msg class*

#### 11.1.4.5.3 GatewayType

##### [req.gateway\_type]

A type *T* satisfies the *GatewayType* if it meets the following requirements:

---

##### GatewayType Requirements: Pseudo-Signature, Semantics

`bool T::try_put (const Output &v)`

**Requirements:** The type `Output` must be the same as template type argument `Output` of the corresponding `async_node` instance.

Broadcasts *v* to all successors of the corresponding `async_node` instance.

`void T::reserve_wait ()`

Notifies a flow graph that work has been submitted to an external activity.

`void T::release_wait ()`

Notifies a flow graph that work submitted to an external activity has completed.

#### 11.1.4.5.4 FunctionNodeBody

##### [req.function\_node\_body]

A type *Body* satisfies the *FunctionNodeBody* if it meets the following requirements:

---

##### FunctionNodeBody Requirements: Pseudo-Signature, Semantics

`Body::Body (const Body&)`

Copy constructor.

`Body::~Body ()`

Destructor.

`void operator= (const B&)`

Assignment.

`Output Body::operator () (const Input &v)`

**Requirements:** The `Input` and `Output` types must be the same as the `Input` and `Output` template type arguments of the `function_node` instance in which `Body` object is passed during construction.

Perform operation on *v* and return value of type `Output`.

#### 11.1.4.5.5 JoinNodeFunctionObject

##### [req.join\_node\_function\_object]

A type *Func* satisfies the *JoinNodeFunctionObject* if it meets the following requirements:

---

##### JoinNodeFunctionObject Requirements: Pseudo-Signature, Semantics

`Func`: **`:Func (const Func&)`**

Copy constructor.

`Func`: **`:~Func ()`**

Destructor.

**Key** `Func::operator () (const Input &v)`

**Requirements:** The `Key` and `Input` types must be the same as the `K` and the corresponding element of the `OutputTuple` template arguments of the `join_node` instance to which `Func` object is passed during construction.

Returns key to be used for hashing input messages.

#### 11.1.4.5.6 InputNodeBody

**[req.input\_node\_body]**

A type `Body` satisfies the `InputNodeBody` if it meets the following requirements:

---

##### InputNodeBody Requirements: Pseudo-Signature, Semantics

`Body`: **`:Body (const Body&)`**

Copy constructor.

`Body`: **`:~Body ()`**

Destructor.

**Output** `Body::operator () (tbb::flow_control &fc)`

**Requirements:** The type `Output` must be the same as template type argument `Output` of the `input_node` instance in which `Body` object is passed during construction.

Apply body to generate next item. Call `fc.stop()` when new element can not be generated. Since `Output` needs to be returned, `Body` may return any valid value of `Output`, to be immediately discarded.

#### 11.1.4.5.7 MultifunctionNodeBody

**[req.multipfunction\_node\_body]**

A type `Body` satisfies the `MultifunctionNodeBody` if it meets the following requirements:

---

##### MultifunctionNodeBody Requirements: Pseudo-Signature, Semantics

`Body`: **`:Body (const Body&)`**

Copy constructor.

`Body`: **`:~Body ()`**

Destructor.

**void** `operator= (const Body&)`

Assignment.

**void** `Body::operator () (const Input &v, OutputPortsType &p)`

**Requirements:**

- The `Input` type must be the same as the `Input` template type argument of the `multifunction_node` instance in which `Body` object is passed during construction.

- The `OutputPortsType` type must be the same as the `output_ports_type` member type of the `multifunction_node` instance in which `Body` object is passed during construction.

Perform operation on `v`. May call `try_put()` on zero or more of the output ports. May call `try_put()` on any output port multiple times.

#### 11.1.4.5.8 Sequencer

##### [req.sequencer]

A type `S` satisfies the *Sequencer* if it meets the following requirements:

---

##### Sequencer Requirements: Pseudo-Signature, Semantics

`S::S(const S&)`

Copy constructor.

`S::~S()`

Destructor.

`void operator=(const S&)`

Assignment. The return type `void` in the pseudo-signature denotes that `operator=` is not required to return a value. The actual `operator=` can return a value, which will be ignored.

`size_t S::operator()(const T &v)`

**Requirements:** The type `T` must be the same as template type argument `T` of the `sequencer_node` instance in which `S` object is passed during construction.

Returns the sequence number for the provided message `v`.

See also:

- *sequencer\_node class*

#### 11.1.5 Thread Safety

##### [thread\_safety]

Unless otherwise stated, the thread safety rules for the library are as follows:

- Two threads can invoke a method or function concurrently on different objects, but not the same object.
- It is unsafe for two threads to invoke concurrently methods or functions on the same object.

Descriptions of the classes note departures from this convention. For example, the concurrent containers are more liberal. By their nature, they do permit some concurrent operations on the same container object.

## 11.2 oneTBB Interfaces

### 11.2.1 Configuration

##### [configuration]

This section describes the most general features of oneAPI Threading Building Blocks such as namespaces, versioning, macros etc.

### 11.2.1.1 Namespaces

#### [configuration.namespaces]

This section describes the oneAPI Threading Building Blocks namespace conventions.

##### 11.2.1.1.1 tbb Namespace

Namespace `tbb` contains public identifiers defined by the library that you can reference in your program.

##### 11.2.1.1.2 tbb::flow Namespace

Namespace `tbb::flow` contains public identifiers defined by the library that you can reference in your program related to the flow graph feature. See *Flow Graph* for more information.

### 11.2.1.2 Version Information

#### [configuration.version\_information]

oneAPI Threading Building Blocks has macros, an environment variable, and a function that reveal version and run-time information.

```
// Defined in header <tbb/version.h>

#define TBB_VERSION_MAJOR /*implementation-defined*/
#define TBB_VERSION_MINOR /*implementation-defined*/
#define TBB_VERSION_STRING /*implementation-defined*/

#define TBB_INTERFACE_VERSION_MAJOR /*implementation-defined*/
#define TBB_INTERFACE_VERSION_MINOR /*implementation-defined*/
#define TBB_INTERFACE_VERSION /*implementation-defined*/

const char* TBB_runtime_version();
int TBB_runtime_interface_version();
```

#### Version Macros

oneTBB defines macros related to versioning, as described below.

- `TBB_VERSION_MAJOR` macro defined to integral value that represents major library version.
- `TBB_VERSION_MINOR` macro defined to integral value that represents minor library version.
- `TBB_VERSION_STRING` macro defined to the string representation of the full library version.
- `TBB_INTERFACE_VERSION` macro defined to current interface version. The value is a decimal numeral of the form `xyz` where `x` is the major interface version number and `y` is the minor interface version number. This macro is increased in each release.
- `TBB_INTERFACE_VERSION_MAJOR` macro defined to `TBB_INTERFACE_VERSION/1000` that is, the major interface version number.
- `TBB_INTERFACE_VERSION_MINOR` macro defined to `TBB_INTERFACE_VERSION%1000/10` that is, the minor interface version number.

### 11.2.1.2.1 TBB\_runtime\_interface\_version Function

Function that returns the interface version of the oneTBB library that was loaded at runtime.

The value returned by `TBB_runtime_interface_version()` may differ from the value of `TBB_INTERFACE_VERSION` obtained at compile time. This can be used to identify whether an application was compiled against a compatible version of the TBB headers.

In general, the run-time value `TBB_runtime_interface_version()` must be greater than or equal to the compile-time value of `TBB_INTERFACE_VERSION`. Otherwise the application may fail to resolve all symbols at run time.

### 11.2.1.2.2 TBB\_runtime\_version Function

Function that returns the version string of the oneTBB library that was loaded at runtime.

The value returned by `TBB_runtime_version()` may differ from the value of `TBB_VERSION_STRING` obtained at compile time.

### 11.2.1.2.3 TBB\_VERSION Environment Variable

Set the environment variable `TBB_VERSION` to 1 to cause the library to print information on `stderr`. Each line is of the form "TBB: tag value", where *tag* and *value* provides additional library information below.

**Caution:** This output is implementation specific and may change at any time.

### 11.2.1.3 Enabling Debugging Features

#### [configuration.debug\_features]

The following macros control certain debugging features. In general, it is useful to compile with these features on for development code, and off for production code, because the features may decrease performance. The table below summarizes the macros and their default values. A value of 1 enables the corresponding feature; a value of 0 disables the feature.

Table 3: Debugging Macros

Macro	Default Value	Feature
<code>TBB_USE_DEBUG</code>	<ul style="list-style-type: none"> <li>Windows* OS: 1 if <code>_DEBUG</code> is defined, 0 otherwise.</li> <li>All other systems: 0.</li> </ul>	Default value for all other macros in this table.
<code>TBB_USE_ASSERT</code>	<code>TBB_USE_DEBUG</code>	Enable internal assertion checking. Can significantly slow performance.
<code>TBB_USE_PROFILING_TOOLS</code>	<code>TBB_USE_DEBUG</code>	Enable full support for analysis tools.

### 11.2.1.3.1 TBB\_USE\_ASSERT Macro

The macro `TBB_USE_ASSERT` controls whether error checking is enabled in the header files. Define `TBB_USE_ASSERT` as 1 to enable error checking.

If an error is detected, the library prints an error message on `stderr` and calls the standard C routine `abort`. To stop a program when internal error checking detects a failure, place a breakpoint on `tbb::assertion_failure`.

### 11.2.1.3.2 TBB\_USE\_PROFILING\_TOOLS Macro

The macro `TBB_USE_PROFILING_TOOLS` controls support for Intel® Inspector XE, Intel® VTune™ Amplifier XE and Intel® Advisor.

Define `TBB_USE_PROFILING_TOOLS` as 1 to enable full support for these tools. Leave `TBB_USE_PROFILING_TOOLS` undefined or zero to enable top performance in release builds, at the expense of turning off some support for tools.

## 11.2.1.4 Feature Macros

### [configuration.feature\_macros]

Macros in this section control optional features in the library.

#### 11.2.1.4.1 TBB\_USE\_EXCEPTIONS macro

The macro `TBB_USE_EXCEPTIONS` controls whether the library headers use exception-handling constructs such as `try`, `catch`, and `throw`. The headers do not use these constructs when `TBB_USE_EXCEPTIONS=0`.

For the Microsoft Windows\*, Linux\*, and macOS\* operating systems, the default value is 1 if exception handling constructs are enabled in the compiler, and 0 otherwise.

**Caution:** The runtime library may still throw an exception when `TBB_USE_EXCEPTIONS=0`.

#### 11.2.1.4.2 TBB\_USE\_GLIBCXX\_VERSION macro

The macro `TBB_USE_GLIBCXX_VERSION` can be used to specify the proper version of GNU libstdc++ if the detection fails. Define the value of the macro equal to `Major*10000 + Minor*100 + Patch`, where `Major.Minor.Patch` is the actual GCC/libstdc++ version (if unknown, it can be obtained with '`gcc -dumpversion`' command). For example, if you use libstdc++ from GCC 4.9.2, define `TBB_USE_GLIBCXX_VERSION=40902`.

## 11.2.2 Algorithms

### [algorithms]

oneAPI Threading Building Blocks provides a set of generic parallel algorithms.

### 11.2.2.1 Parallel Functions

#### 11.2.2.1.1 parallel\_for

[**algorithms.parallel\_for**]

Function template that performs parallel iteration over a range of values.

```
// Defined in header <tbb/parallel_for.h>

namespace tbb {

    template<typename Index, typename Func>
    void parallel_for(Index first, Index last, const Func& f, /* see-below */_
        partitioner, task_group_context& group);
    template<typename Index, typename Func>
    void parallel_for(Index first, Index last, const Func& f, task_group_context&_
        group);
    template<typename Index, typename Func>
    void parallel_for(Index first, Index last, const Func& f, /* see-below */_
        partitioner);
    template<typename Index, typename Func>
    void parallel_for(Index first, Index last, const Func& f);

    template<typename Index, typename Func>
    void parallel_for(Index first, Index last, Index step, const Func& f, /* see-
        below */ partitioner, task_group_context& group);
    template<typename Index, typename Func>
    void parallel_for(Index first, Index last, Index step, const Func& f, task_group_-
        context& group);
    template<typename Index, typename Func>
    void parallel_for(Index first, Index last, Index step, const Func& f, /* see-
        below */ partitioner);
    template<typename Index, typename Func>
    void parallel_for(Index first, Index last, Index step, const Func& f);

    template<typename Range, typename Body>
    void parallel_for(const Range& range, const Body& body, /* see-below */_
        partitioner, task_group_context& group);
    template<typename Range, typename Body>
    void parallel_for(const Range& range, const Body& body, task_group_context&_
        group);
    template<typename Range, typename Body>
    void parallel_for(const Range& range, const Body& body, /* see-below */_
        partitioner);
    template<typename Range, typename Body>
    void parallel_for(const Range& range, const Body& body);

} // namespace tbb
```

A `partitioner` type may be one of the following entities:

- `const auto_partitioner&`
- `const simple_partitioner&`
- `const static_partitioner&`
- `affinity_partitioner&`

Requirements:

- The Range type shall meet the *Range requirements*.
- The Body type shall meet the *ParallelForBody requirements*.

A `tbb::parallel_for(first, last, step, f)` overload represents parallel execution of the loop:

```
for (auto i = first; i < last; i += step) f(i);
```

The index type must be an integral type. The loop must not wrap around. The step value must be positive. If omitted, it is implicitly 1. There is no guarantee that the iterations run in parallel. Deadlock may occur if a lesser iteration waits for a greater iteration. The partitioning strategy is `auto_partitioner` when the parameter is not specified.

A `parallel_for(range, body, partitioner)` overload provides a more general form of parallel iteration. It represents parallel execution of `body` over each value in `range`. The optional `partitioner` parameter specifies a partitioning strategy.

`parallel_for` recursively splits the range into subranges to the point such that `is_divisible()` is false for each subrange, and makes copies of the body for each of these subranges. For each such body/subrange pair, it invokes `Body::operator()`.

Some of the copies of the range and body may be destroyed after `parallel_for` returns. This late destruction is not an issue in typical usage, but is something to be aware of when looking at execution traces or writing range or body objects with complex side effects.

`parallel_for` may execute iterations in non-deterministic order. Do not rely upon any particular execution order for correctness. However, for efficiency, do expect `parallel_for` to tend towards operating on consecutive runs of values.

In case of serial execution `parallel_for` performs iterations from left to right in the following sense.

All overloads can accept a `task_group_context` object so that the algorithm's tasks are executed in this group. By default the algorithm is executed in a bound group of its own.

## Complexity

If the range and body take  $O(1)$  space, and the range splits into nearly equal pieces, then the space complexity is  $O(P \log(N))$ , where  $N$  is the size of the range and  $P$  is the number of threads.

See also:

- *Partitioners*

### 11.2.2.1.2 parallel\_reduce

#### [algorithms.parallel\_reduce]

Function template that computes reduction over a range.

```
// Defined in header <tbb/parallel_reduce.h>

namespace tbb {

    template<typename Range, typename Value, typename Func, typename Reduction>
    Value parallel_reduce(const Range& range, const Value& identity, const Func& func,
    ↵ const Reduction& reduction, /* see-below */ partitioner, task_group_context&_
    ↵ group);
    template<typename Range, typename Value, typename Func, typename Reduction>
    Value parallel_reduce(const Range& range, const Value& identity, const Func& func,
    ↵ const Reduction& reduction, /* see-below */ partitioner);
}
```

(continues on next page)

(continued from previous page)

```

template<typename Range, typename Value, typename Func, typename Reduction>
Value parallel_reduce(const Range& range, const Value& identity, const Func& func,
→ const Reduction& reduction, task_group_context& group);
template<typename Range, typename Value, typename Func, typename Reduction>
Value parallel_reduce(const Range& range, const Value& identity, const Func& func,
→ const Reduction& reduction);

template<typename Range, typename Body>
void parallel_reduce(const Range& range, Body& body, /* see-below */ partitioner,
→ task_group_context& group);
template<typename Range, typename Body>
void parallel_reduce(const Range& range, Body& body, /* see-below */ partitioner);
template<typename Range, typename Body>
void parallel_reduce(const Range& range, Body& body, task_group_context& group);
template<typename Range, typename Body>
void parallel_reduce(const Range& range, Body& body);

} // namespace tbb

```

A partitioner type may be one of the following entities:

- **const auto\_partitioner&**
- **const simple\_partitioner&**
- **const static\_partitioner&**
- **affinity\_partitioner&**

Requirements:

- The Range type shall meet the *Range requirements*.
- The Body type shall meet the *ParallelReduceBody requirements*.
- The Func type shall meet the *ParallelReduceFunc requirements*.
- The Reduction types shall meet :*ParallelReduceReduction requirements*.

The function template `parallel_reduce` has two forms. The functional form is designed to be easy to use in conjunction with lambda expressions. The imperative form is designed to minimize copying of data.

The functional form `parallel_reduce(range, identity, func, reduction)` performs a parallel reduction by applying `func` to subranges in `range` and reducing the results using binary operator `reduction`. It returns the result of the reduction. Parameter `identity` specifies the left identity element for `func`'s operator(). Parameter `func` and `reduction` can be lambda expressions.

The imperative form `parallel_reduce(range, body)` performs parallel reduction of `body` over each value in `range`.

A `parallel_reduce` recursively splits the range into subranges to the point such that `is_divisible()` is false for each subrange. A `parallel_reduce` uses the splitting constructor to make one or more copies of the body for each thread. It may copy a body while the body's `operator()` or method `join` runs concurrently. You are responsible for ensuring the safety of such concurrency. In typical usage, the safety requires no extra effort.

`parallel_reduce` may invoke the splitting constructor for the body. For each such split of the body, it invokes method `join` in order to merge the results from the bodies. Define `join` to update this to represent the accumulated result for this and `rhs`. The reduction operation should be associative, but does not have to be commutative. For a noncommutative operation `op`, `left.join(right)` should update `left` to be the result of `left op right`.

A body is split only if the range is split, but the converse is not necessarily so. You must neither rely on a particular choice of body splitting nor on the subranges processed by a given body object being consecutive. `parallel_reduce` makes the choice of body splitting nondeterministically.

When executed serially `parallel_reduce` runs sequentially from left to right in the same sense as for `parallel_for`. Sequential execution never invokes the splitting constructor or method `join`.

All overloads can accept a `task_group_context` object so that the algorithm's tasks are executed in this group. By default the algorithm is executed in a bound group of its own.

### Complexity

If the range and body take  $O(1)$  space, and the range splits into nearly equal pieces, then the space complexity is  $O(P \times \log(N))$ , where  $N$  is the size of the range and  $P$  is the number of threads.

#### 11.2.2.1.2.1 Example (Imperative Form)

The following code sums the values in an array.

```
#include "tbb/parallel_reduce.h"
#include "tbb/blocked_range.h"

using namespace tbb;

struct Sum {
    float value;
    Sum() : value(0) {}
    Sum( Sum& s, split ) {value = 0;}
    void operator()( const blocked_range<float*>& r ) {
        float temp = value;
        for( float* a=r.begin(); a!=r.end(); ++a ) {
            temp += *a;
        }
        value = temp;
    }
    void join( Sum& rhs ) {value += rhs.value;}
};

float ParallelSum( float array[], size_t n ) {
    Sum total;
    parallel_reduce( blocked_range<float*>( array, array+n ), total );
    return total.value;
}
```

The example generalizes to reduction for any associative operation  $op$  as follows:

- Replace occurrences of 0 with the identity element for  $op$
- Replace occurrences of  $+=$  with  $op=$  or its logical equivalent.
- Change the name `Sum` to something more appropriate for  $op$ .

The operation may be noncommutative. For example,  $op$  could be matrix multiplication.

### 11.2.2.1.2.2 Example with Lambda Expressions

The following is analogous to the previous example, but written using lambda expressions and the functional form of parallel\_reduce.

```
#include "tbb/parallel_reduce.h"
#include "tbb/blocked_range.h"

using namespace tbb;

float ParallelSum( float array[], size_t n ) {
    return parallel_reduce(
        blocked_range<float*>( array, array+n ),
        0.f,
        [] (const blocked_range<float*>& r, float init) ->float {
            for( float* a=r.begin(); a!=r.end(); ++a )
                init += *a;
            return init;
        },
        [] ( float x, float y ) ->float {
            return x+y;
        }
    );
}
```

See also:

- *Partitioners*

### 11.2.2.1.3 parallel\_deterministic\_reduce

#### [algorithms.parallel\_deterministic\_reduce]

Function template that computes reduction over a range, with deterministic split/join behavior.

```
// Defined in header <tbb/parallel_reduce.h>

namespace tbb {

    template<typename Range, typename Value, typename Func, typename Reduction>
    Value parallel_deterministic_reduce( const Range& range, const Value& identity, ↵
    ↵const Func& func, const Reduction& reduction, /* see-below */ partitioner, task_ ↵
    ↵group_context& group);
    template<typename Range, typename Value, typename Func, typename Reduction>
    Value parallel_deterministic_reduce( const Range& range, const Value& identity, ↵
    ↵const Func& func, const Reduction& reduction, /* see-below */ partitioner);
    template<typename Range, typename Value, typename Func, typename Reduction>
    Value parallel_deterministic_reduce( const Range& range, const Value& identity, ↵
    ↵const Func& func, const Reduction& reduction, task_group_context& group);
    template<typename Range, typename Value, typename Func, typename Reduction>
    Value parallel_deterministic_reduce( const Range& range, const Value& identity, ↵
    ↵const Func& func, const Reduction& reduction);

    template<typename Range, typename Body>
    void parallel_deterministic_reduce( const Range& range, Body& body, /* see-below */ ↵
    ↵/* partitioner, task_group_context& group);
    template<typename Range, typename Body>
```

(continues on next page)

(continued from previous page)

```

void parallel_deterministic_reduce( const Range& range, Body& body, /* see-below */
→ */ partitioner);
template<typename Range, typename Body>
void parallel_deterministic_reduce( const Range& range, Body& body, task_group_
→ context& group);
template<typename Range, typename Body>
void parallel_deterministic_reduce( const Range& range, Body& body);

} // namespace tbb

```

A partitioner type may be one of the following entities:

- **const simple\_partitioner&**
- **const static\_partitioner&**

The function template `parallel_deterministic_reduce` is very similar to the `parallel_reduce` template. It also has the functional and imperative forms and has *similar requirements*.

Unlike `parallel_reduce`, `parallel_deterministic_reduce` has deterministic behavior with regard to splits of both `Body` and `Range` and joins of the bodies. For the functional form, `Func` is applied to a deterministic set of `Ranges`, and `Reduction` merges partial results in a deterministic order. To achieve that, `parallel_deterministic_reduce` uses a `simple_partitioner` or a `static_partitioner` only because other partitioners react to random work stealing behavior.

**Caution:** Since `simple_partitioner` does not automatically coarsen ranges, make sure to specify an appropriate grain size. See *Partitioners section* for more information.

`parallel_deterministic_reduce` always invokes the `Body` splitting constructor for each range split.

As a result, `parallel_deterministic_reduce` recursively splits a range until it is no longer divisible, and creates a new body (by calling `Body` splitting constructor) for each new subrange. Like `parallel_reduce`, for each body split the method `join` is invoked in order to merge the results from the bodies.

Therefore for given arguments, `parallel_deterministic_reduce` executes the same set of split and join operations no matter how many threads participate in execution and how tasks are mapped to the threads. If the user-provided functions are also deterministic (i.e. different runs with the same input result in the same output), then multiple calls to `parallel_deterministic_reduce` will produce the same result. Note however that the result might differ from that obtained with an equivalent sequential (linear) algorithm.

### Complexity

If the range and body take  $O(1)$  space, and the range splits into nearly equal pieces, then the space complexity is  $O(P \log(N))$ , where  $N$  is the size of the range and  $P$  is the number of threads.

See also:

- *parallel\_reduce*
- *Partitioners*

### 11.2.2.1.4 parallel\_scan

#### [algorithms.parallel\_scan]

Function template that computes parallel prefix.

```
// Defined in header <tbb/parallel_scan.h>

template<typename Range, typename Body>
void parallel_scan( const Range& range, Body& body );
template<typename Range, typename Body>
void parallel_scan( const Range& range, Body& body, /* see-below */ partitioner );

template<typename Range, typename Value, typename Scan, typename Combine>
Value parallel_scan( const Range& range, const Value& identity, const Scan& scan, const Combine& combine );
template<typename Range, typename Value, typename Scan, typename Combine>
Value parallel_scan( const Range& range, const Value& identity, const Scan& scan, const Combine& combine, /* see-below */ partitioner );
```

A partitioner type may be one of the following entities:

- const auto\_partitioner&
- const simple\_partitioner&

Requirements:

- The Range type shall meet the *Range requirement*.
- The Body type shall meet the *ParallelScanBody requirements*.
- The Scan type shall meet the *ParallelScanFunc requirements*.
- The Combine type shall meet the *ParallelScanCombine requirements*.

The function template `parallel_scan` computes a parallel prefix, also known as parallel scan. This computation is an advanced concept in parallel computing that is sometimes useful in scenarios that appear to have inherently serial dependences.

A mathematical definition of the parallel prefix is as follows. Let  $\times$  be an associative operation with left-identity element  $\text{id}_\times$ . The parallel prefix of  $\times$  over a sequence  $z_0, z_1, \dots * z_{n-1}$  is a sequence  $y_0, y_1, y_2, \dots * y_{n-1}$  where:

- $y_0 = \text{id}_\times \times z_0$
- $y_i = y_{i-1} \times z_i$

For example, if  $\times$  is addition, the parallel prefix corresponds a running sum. A serial implementation of parallel prefix is:

```
T temp = id;
for( int i=1; i<=n; ++i ) {
    temp = temp + z[i];
    y[i] = temp;
}
```

Parallel prefix performs this in parallel by reassociating the application of  $\times$  (+ in example) and using two passes. It may invoke  $\times$  up to twice as many times as the serial prefix algorithm. Even though it does more work, given the right grain size the parallel algorithm can outperform the serial one because it distributes the work across multiple hardware threads.

The function template `parallel_scan` has two forms. The imperative form `parallel_scan(range, body)` implements parallel prefix generically.

A summary (look at [ParallelScanBody requirements](#)) contains enough information such that for two consecutive subranges  $r$  and  $s$ :

- If  $r$  has no preceding subrange, the scan result for  $s$  can be computed from knowing  $s$  and the summary for  $r$ .
- A summary of  $r$  concatenated with  $s$  can be computed from the summaries of  $r$  and  $s$ .

The functional form `parallel_scan(range, identity, scan, combine)` is designed to use with functors and lambda expressions, hiding some complexities of the imperative form. It uses the same `scan` functor in both passes, differentiating them via a Boolean parameter, combines summaries with `combine` functor, and returns the summary computed over the whole `range`. The `identity` argument is the left identity element for `Scan::operator()`.

#### 11.2.2.1.4.1 `pre_scan` and `final_scan` Classes

#### 11.2.2.1.4.2 `pre_scan_tag` and `final_scan_tag`

##### [algorithms.parallel\_scan.scan\_tags]

Types that distinguish the phases of `parallel_scan`.

Types `pre_scan_tag` and `final_scan_tag` are dummy types used in conjunction with `parallel_scan`. See the example in the [parallel\\_scan](#) section for how they are used in the signature of `operator()`.

```
// Defined in header <tbb/parallel_scan.h>

namespace tbb {

    struct pre_scan_tag {
        static bool is_final_scan();
        operator bool();
    };

    struct final_scan_tag {
        static bool is_final_scan();
        operator bool();
    };

}
```

#### 11.2.2.1.4.3 Member functions

`bool is_final_scan()`  
true for a `final_scan_tag`, otherwise false.

`operator bool()`  
true for a `final_scan_tag`, otherwise false.

The `parallel_scan` template makes an effort to avoid prescanning where possible. When executed serially `parallel_scan` processes the subranges without any pre-scans, by processing the subranges from left to right using final scans. That's why final scans must compute a summary as well as the final scan result. The summary might be needed to process the next subrange if no other thread has pre-scanned it yet.

#### 11.2.2.1.4.4 Example (Imperative Form)

The following code demonstrates how Body could be implemented for parallel\_scan to compute the same result as the earlier sequential example.

```

class Body {
    T sum;
    T* const y;
    const T* const z;
public:
    Body( T y_[], const T z_[] ) : sum(id), z(z_), y(y_) {}
    T get_sum() const { return sum; }

    template<typename Tag>
    void operator()( const tbb::blocked_range<int>& r, Tag ) {
        T temp = sum;
        for( int i=r.begin(); i<r.end(); ++i ) {
            temp = temp + z[i];
            if( Tag::is_final_scan() )
                y[i] = temp;
        }
        sum = temp;
    }
    Body( Body& b, tbb::split ) : z(b.z), y(b.y), sum(id) {}
    void reverse_join( Body& a ) { sum = a.sum + sum; }
    void assign( Body& b ) { sum = b.sum; }
};

T DoParallelScan( T y[], const T z[], int n ) {
    Body body(y,z);
    tbb::parallel_scan( tbb::blocked_range<int>(0,n), body );
    return body.get_sum();
}

```

The definition of operator() demonstrates typical patterns when using parallel\_scan.

- A single template defines both versions. Doing so is not required, but usually saves coding effort, because the two versions are usually similar. The library defines static method `is_final_scan()` to enable differentiation between the versions.
- The prescan variant computes the `x` reduction, but does not update `y`. The prescan is used by `parallel_scan` to generate look-ahead partial reductions.
- The final scan variant computes the `x` reduction and updates `y`.

The operation `reverse_join` is similar to the operation `join` used by `parallel_reduce`, except that the arguments are reversed. That is, `this` is the *right* argument of `x`. Template function `parallel_scan` decides if and when to generate parallel work. It is thus crucial that `x` is associative and that the methods of `Body` faithfully represent it. Operations such as floating-point addition that are somewhat associative can be used, with the understanding that the results may be rounded differently depending upon the association used by `parallel_scan`. The reassociation may differ between runs even on the same machine. However, when executed serially, `parallel_scan` associates identically to the serial form shown at the beginning of this section.

If you change the example to use a `simple_partitioner`, be sure to provide a grain size. The code below shows the how to do this for the grain size of 1000:

```
parallel_scan( blocked_range<int>(0,n,1000), total, simple_partitioner() );
```

### 11.2.2.1.4.5 Example with Lambda Expressions

The following is analogous to the previous example, but written using lambda expressions and the functional form of `parallel_scan`:

```
T DoParallelScan( T y[], const T z[], int n ) {
    return tbb::parallel_scan(
        tbb::blocked_range<int>(0,n),
        id,
        [](const tbb::blocked_range<int>& r, T sum, bool is_final_scan)->T {
            T temp = sum;
            for( int i=r.begin(); i<r.end(); ++i ) {
                temp = temp + z[i];
                if( is_final_scan )
                    y[i] = temp;
            }
            return temp;
        },
        []( T left, T right ) {
            return left + right;
        }
    );
}
```

See also:

- *blocked\_range class*
- *parallel\_reduce algorithm*

### 11.2.2.1.5 parallel\_for\_each

#### [algorithms.parallel\_for\_each]

Function template that processes work items in parallel.

```
// Defined in header <tbb/parallel_for_each.h>

namespace tbb {

    template<typename InputIterator, typename Body>
    void parallel_for_each( InputIterator first, InputIterator last, Body body );
    template<typename InputIterator, typename Body>
    void parallel_for_each( InputIterator first, InputIterator last, Body body, task_
        ↪group_context& group );

    template<typename Container, typename Body>
    void parallel_for_each( Container c, Body body );
    template<typename Container, typename Body>
    void parallel_for_each( Container c, Body body, task_group_context& group );

} // namespace tbb
```

Requirements:

- The `Body` type shall meet the *ParallelForEachBody requirements*.

- The `InputIterator` type shall meet the *Input Iterator* requirements from [input.iterators] ISO C++ Standard section.
- The `Container` type shall meet the *ContainerBasedSequence requirements*

The `parallel_for_each` template has two forms.

The sequence form `parallel_for_each(first, last, body)` applies a function object `body` over a sequence `[first, last)`. Items may be processed in parallel.

The container form `parallel_for_each(c, body)` is equivalent to `parallel_for_each(std::begin(c), std::end(c), body)`.

All overloads can accept a `task_group_context` object so that the algorithm's tasks are executed in this group. By default the algorithm is executed in a bound group of its own.

#### 11.2.2.1.5.1 feeder Class

Additional work items can be added by `body` if it has a second argument of type `feeder`. The function terminates when `body(x)` returns for all items `x` that were in the input sequence or added by method `feeder::add`.

#### 11.2.2.1.5.2 feeder

##### [algorithms.parallel\_for\_each.feeder]

Inlet into which additional work items for a `parallel_for_each` can be fed.

```
// Defined in header <tbb/parallel_for_each.h>

namespace tbb {

    template<typename Item>
    class feeder {
    public:
        void add( const Item& item );
        void add( Item&& item );
    };

} // namespace tbb
```

#### 11.2.2.1.5.3 Member functions

##### `void add(const Item &item)`

Adds item to collection of work items to be processed.

##### `void add(Item &&item)`

Same as the above but uses the move constructor of `Item` if available.

**Caution:** Must be called from a `Body::operator()` created by `parallel_for_each` function. Otherwise, the termination semantics of method `operator()` are undefined.

#### 11.2.2.1.5.4 Example

The following code sketches a body with the two-argument form of operator().

```
struct MyBody {
    void operator()(item_t item, parallel_do_feeder<item_t>& feeder) {
        for each new piece of work implied by item do {
            item_t new_item = initializer;
            feeder.add(new_item);
        }
    }
};
```

#### 11.2.2.1.6 parallel\_invoke

##### [algorithms.parallel\_invoke]

Function template that evaluates several functions in parallel.

```
// Defined in header <tbb/parallel_invoke.h>

namespace tbb {

    template<typename... Functions>
    void parallel_invoke(Functions&&... fs);

} // namespace tbb
```

Requirements:

- All members of Functions parameter pack shall meet Function Objects requirements from [function.objects] ISO C++ Standard section or be a pointer to a function.
- Last member of Functions parameter pack may be a task\_group\_context& type.

Evaluates each member passed to parallel\_invoke possibly in parallel. Return values are ignored.

The algorithm can accept a *task\_group\_context* object so that the algorithm's tasks are executed in this group. By default the algorithm is executed in a bound group of its own.

#### 11.2.2.1.6.1 Example

The following example evaluates f(), g(), h(), and bar(1) in parallel.

```
#include "tbb/parallel_invoke.h"

extern void f();
extern void bar(int);

class MyFunctor {
    int arg;
public:
    MyFunctor(int a) : arg(a) {}
    void operator()() const {bar(arg);}
};
```

(continues on next page)

(continued from previous page)

```
void RunFunctionsInParallel() {
    MyFunctor g(2);
    MyFunctor h(3);

    tbb::parallel_invoke(f, g, h, []{bar(1);});
}
```

### 11.2.2.1.7 parallel\_pipeline

#### [algorithms.parallel\_pipeline]

Strongly-typed interface for pipelined execution.

```
// Defined in header <tbb/parallel_pipeline.h>

namespace tbb {

    void parallel_pipeline( size_t max_number_of_live_tokens, const filter<void,void>&
    filter_chain );
    void parallel_pipeline( size_t max_number_of_live_tokens, const filter<void,void>&
    filter_chain, task_group_context& group );

} // namespace tbb
```

A `parallel_pipeline` algorithm represents pipelined application of a series of filters to a stream of items. Each filter operates in a particular mode: parallel, serial in-order, or serial out-of-order.

To build and run a pipeline from functors  $g_0, g_1, g_2, \dots, g_n$ , write:

```
parallel_pipeline( max_number_of_live_tokens,
                    make_filter<void, I1>(mode0, g0) &
                    make_filter<I1, I2>(mode1, g1) &
                    make_filter<I2, I3>(mode2, g2) &
                    ...
                    make_filter<In, void>(moden, gn) );
```

In general, functor  $g_i$  should define its `operator()` to map objects of type  $I_i$  to objects of type  $I_{i+1}$ . Functor  $g_0$  is a special case, because it notifies the pipeline when the end of an input stream is reached. Functor  $g_0$  must be defined such that for a `flow_control` object  $fc$ , the expression  $g_0(fc)$  either returns the next value in an input stream, or if at the end of an input stream, invokes `fc.stop()` and returns a dummy value.

Each `filter` should be specified by two template arguments. These arguments define filters input and output types. The first and last filters are special cases. Input type of the first filter must be `void`, output type of the last filter must be `void` too.

Before passing to `parallel_pipeline` all filters must be concatenated to one(`filter<void, void>`) by `filter::operator&()`. Operator `&` requires that the second template argument of its left operand matches the first template argument of its second operand.

The number of items processed in parallel depends upon the structure of the pipeline and number of available threads. `max_number_of_live_tokens` sets the threshold for concurrently processed items.

If the `group` argument is specified, pipeline's tasks are executed in this group. By default the algorithm is executed in a bound group of its own.

### 11.2.2.1.7.1 Example

The following example uses parallel\_pipeline compute the root-mean-square of a sequence defined by [first, last ).

```
float RootMeanSquare( float* first, float* last ) {
    float sum=0;
    parallel_pipeline( /*max_number_of_live_token=*/16,
        make_filter<void,float>(
            filter::serial,
            [&](flow_control& fc) -> float*{
                if( first<last ) {
                    return first++;
                } else {
                    fc.stop();
                    return nullptr;
                }
            }
        ) &
        make_filter<float*,float>(
            filter::parallel,
            [](float* p){return (*p)*(*p); }
        ) &
        make_filter<float,void>(
            filter::serial,
            [&](float x) {sum+=x; }
        )
    );
    return sqrt(sum);
}
```

### 11.2.2.1.7.2 filter Class Template

#### 11.2.2.1.7.3 filter

##### [algorithms.parallel\_pipeline.filter]

A filter class template represents a filter in a parallel\_pipeline algorithm.

A filter is a strongly-typed filter that specifies its input and output types. A filter can be constructed from a functor or by composing of two filter objects with operator&(). The same filter object can be shared by multiple & expressions.

Class filter should only be used in conjunction with parallel\_pipeline functions.

```
// Defined in header <tbb/parallel_pipeline.h>

namespace tbb {

    template<typename InputType, typename OutputType>
    class filter {
    public:
        filter() = default;
        filter( const filter& rhs ) = default;
        filter( filter&& rhs ) = default;
        void operator=(const filter& rhs) = default;
    };
}
```

(continues on next page)

(continued from previous page)

```

void operator=( filter&& rhs ) = default;  

template<typename Body>  

filter( filter_mode mode, const Body& body );  

filter& operator&= ( const filter<OutputType,OutputType>& right );  

void clear();  

}  

template<typename T, typename U, typename Func>  

filter<T,U> make_filter( filter::mode mode, const Func& f );  

template<typename T, typename V, typename U>  

filter<T,U> operator&( const filter<T,V>& left, const filter<V,U>& right );  

}

```

Requirements:

- If *InputType* is `void` then Body type shall meet the *StartFilterBody requirements*.
- If *OutputType* is `void` then Body type shall meet the *OutputFilterBody requirements*.
- If *InputType* and *OutputType* are not `void` then Body type shall meet the *MiddleFilterBody requirements*.
- If *InputType* and *OutputType* are `void` then Body type shall meet the *SingleFilterBody requirements*.

#### 11.2.2.1.7.4 `filter_mode` Enumeration

#### 11.2.2.1.7.5 `filter_mode`

##### [algorithms.parallel\_pipeline.filter\_mode]

A `filter_mode` enumeration represents an execution mode of a `filter` in a `parallel_pipeline` algorithm. Its enumerated values and their meanings are as follows:

- A parallel filter can process multiple items in parallel and in no particular order.
- A serial\_out\_of\_order filter processes items one at a time, and in no particular order.
- A serial\_in\_order filter processes items one at a time. The order in which items are processed is implicitly set by the first *serial\_in\_order* filter and respected by all other such filters in the pipeline.

```

// Defined in header <tbb/parallel_pipeline.h>

namespace tbb {  

enum class filter_mode {  

    parallel = /*implementation-defined*/,  

    serial_in_order = /*implementation-defined*/,  

    serial_out_of_order = /*implementation-defined*/  

};  

}

```

### 11.2.2.1.7.6 Member functions

#### `filter()`

Construct an undefined filter.

**Caution:** The effect of using an undefined filter by `operator&()` or `parallel_pipeline` is undefined.

#### `template<typename Body>`

#### `filter(filter_mode mode, const Body &body)`

Construct a `filter` that uses a copy of a provided `body` to map an input value of type `InputType` to an output value of type `OutputType`. Each filter must be specified by `filter_mode` value.

#### `void clear()`

Set `*this` to an undefined filter.

### 11.2.2.1.7.7 Non-member functions

#### `template<typename T, typename U, typename Func>`

#### `filter<T, U> make_filter(filter::mode mode, const Func &f)`

Returns `filter<T, U>(mode, f)`.

#### `template<typename T, typename V, typename U>`

#### `filter<T, U> operator& (const filter<T, V> &left, const filter<V, U> &right)`

Returns a `filter` representing the composition of filters `left` and `right`. The composition behaves as if the output value of `left` becomes the input value of `right`.

### 11.2.2.1.7.8 Deduction Guides

```
template<typename Body>
filter(filter_mode, Body) -> filter<filter_input<Body>, filter_output<Body>>;
```

Where:

- `filter_input<Body>` is an alias to `Body::operator()` input parameter type. If `Body::operator()` input parameter type is `flow_control` then `filter_input<Body>` is `void`.
- `filter_output<Body>` is an alias to `Body::operator()` return type.

### 11.2.2.1.7.9 `flow_control` Class

#### 11.2.2.1.7.10 `flow_control`

#### [algorithms.parallel\_pipeline.flow\_control]

Enables the first filter in a composite filter to indicate when the end of input stream has been reached.

Template function `parallel_pipeline` passes a `flow_control` object to the functor of the first filter. When the functor reaches the end of its input stream, it should invoke `fc.stop()` and return a dummy value that will not be passed to the next filter.

```
// Defined in header <tbb/parallel_pipeline.h>

namespace tbb {

    class flow_control {
public:
    void stop();
};

}
```

### 11.2.2.1.7.11 Member functions

**void `stop()`**

Indicates that first filter of the pipeline reaches the end of its output.

See also:

- *FilterBody requirements*
- *filter class*

See also:

- *task\_group\_context*

### 11.2.2.1.8 parallel\_sort

**[algorithms.parallel\_sort]**

Function template that sorts a sequence.

```
// Defined in header <tbb/parallel_invoke.h>

namespace tbb {

    template<typename RandomAccessIterator>
    void parallel_sort( RandomAccessIterator begin, RandomAccessIterator end );
    template<typename RandomAccessIterator, typename Compare>
    void parallel_sort( RandomAccessIterator begin, RandomAccessIterator end, const Compare& comp );

    template<typename Container>
    void parallel_sort( Container c );
    template<typename Container>
    void parallel_sort( Container c, const Compare& comp );

} // namespace tbb
```

Requirements:

- The `RandomAccessIterator` type shall meet the *Random Access Iterators* requirements from [random.access.iterators] and *Swappable* requirements from [swappable.requirements] ISO C++ Standard section.
- The `Compare` type shall meet the `Compare` type requirements from [alg.sorting] ISO C++ Standard section.
- The `Container` type shall meet the *ContainerBasedSequence requirements*

Sorts a sequence or a container. The sort is neither stable nor deterministic: relative ordering of elements with equal keys is not preserved and not guaranteed to repeat if the same sequence is sorted again.

A call `parallel_sort( begin, end, comp )` sorts the sequence  $[begin, end)$  using the argument `comp` to determine relative orderings. If `comp( x, y )` returns `true` then `x` appears before `y` in the sorted sequence.

A call `parallel_sort( begin, end )` is equivalent `parallel_sort( begin, end, comp )` where `comp` uses `operator<` to determine relative orderings.

A call `parallel_sort( c, comp )` is equivalent to `parallel_sort( std::begin(c), std::end(c), comp )`.

A call `parallel_sort( c )` is equivalent to `parallel_sort( c, comp )` where `comp` uses `operator<` to determine relative orderings.

## Complexity

`parallel_sort` is comparison sort with an average time complexity of  $O(N \times \log(N))$ , where  $N$  is the number of elements in the sequence. `parallel_sort` may be executed concurrently to improve execution time.

### 11.2.2.2 Blocked Ranges

Types that meet the *Range requirements*.

#### 11.2.2.2.1 blocked\_range

##### [algorithms.blocked\_range]

Class template for a recursively divisible half-open interval.

A `blocked_range` represents a half-open range  $[i, *j*)$  that can be recursively split.

A `blocked_range` meets the *Range requirements*.

A `blocked_range` specifies a *grain size* of type `size_t`.

A `blocked_range` is splittable into two subranges if the size of the range exceeds its grain size. The ideal grain size depends upon the context of the `blocked_range`, which is typically as the range argument to the loop templates `parallel_for`, `parallel_reduce`, or `parallel_scan`.

```
// Defined in header <tbb/blocked_range.h>

namespace tbb {

    template<typename Value>
    class blocked_range {
    public:
        // types
        using size_type = size_t;
        using const_iterator = Value;

        // constructors
        blocked_range( Value begin, Value end, size_type grainsize=1 );
        blocked_range( blocked_range& r, split );
        blocked_range( blocked_range& r, proportional_split& proportion );

        // capacity
        size_type size() const;
        bool empty() const;
    };
}
```

(continues on next page)

(continued from previous page)

```

// access
size_type grainsize() const;
bool is_divisible() const;

// iterators
const_iterator begin() const;
const_iterator end() const;
};

}

```

Requirements:

- The `Value` type shall meet the *BlockedRangeValue requirements*.

### 11.2.2.2.1.1 Member functions

#### `type size_type`

The type for measuring the size of a `blocked_range`. The type is always a `size_t`.

#### `type const_iterator`

The type of a value in the range. Despite its name, the type `const_iterator` is not necessarily an STL iterator; it merely needs to meet the *BlockedRangeValue requirements*. However, it is convenient to call it `const_iterator` so that if it is a `const_iterator`, then the `blocked_range` behaves like a read-only STL container.

#### `blocked_range` (`Value begin, Value end, size_type grainsize = 1`)

**Requirements:** The parameter `grainsize` must be positive. The debug version of the library raises an assertion failure if this requirement is not met.

**Effects:** Constructs a `blocked_range` representing the half-open interval `[begin, end)` with the given `grainsize`.

**Example:** The statement "`blocked_range<int> r(5, 14, 2);`" constructs a range of `int` that contains the values 5 through 13 inclusive, with the grain size of 2. Afterwards, `r.begin() == 5` and `r.end() == 14`.

#### `blocked_range` (`blocked_range &range, split`)

Basic splitting constructor.

**Requirements:** `is_divisible()` is true.

**Effects:** Partitions `range` into two subranges. The newly constructed `blocked_range` is approximately the second half of the original `range`, and `range` is updated to be the remainder. Each subrange has the same `grainsize` as the original `range`.

**Example:** Let `r` be a `blocked_range` that represents a half-open interval `[i, j)` with a grain size `g`. Running the statement `blocked_range<int> s(r, split);` subsequently causes `r` to represent `[i, i+(j-i)/2)` and `s` to represent `[i+(j-i)/2, j)`, both with grain size `g`.

#### `blocked_range` (`blocked_range &range, proportional_split proportion`)

Proportional splitting constructor.

**Requirements:** `is_divisible()` is true.

**Effects:** Partitions `range` into two subranges such that the ratio of their sizes is close to the ratio of `proportion.left()` to `proportion.right()`. The newly constructed `blocked_range` is the subrange at the right, and `range` is updated to be the subrange at the left.

**Example:** Let  $r$  be a `blocked_range` that represents a half-open interval  $[i, j)$  with a grain size  $g$ . Running the statement `blocked_range<int> s(r, proportional_split(2, 3));` subsequently causes  $r$  to represent  $[i, i+2*(j-i)/(2+3))$  and  $s$  to represent  $[i+2*(j-i)/(2+3), j)$ , both with grain size  $g$ .

`size_type size() const`

**Requirements:** `end() < begin()` is false.

**Effects:** Determines size of range.

**Returns:** `end() - begin()`.

`bool empty() const`

**Effects:** Determines if range is empty.

**Returns:** `!(begin() < end())`

`size_type grainsize() const`

**Returns:** Grain size of range.

`bool is_divisible() const`

**Requirements:** `end() < begin()` is false.

**Effects:** Determines if range can be split into subranges.

**Returns:** True if `size() > grainsize()`; false otherwise.

`const_iterator begin() const`

**Returns:** Inclusive lower bound on range.

`const_iterator end() const`

**Returns:** Exclusive upper bound on range.

See also:

- [parallel\\_reduce](#)
- [parallel\\_for](#)
- [parallel\\_scan](#)

### 11.2.2.2 `blocked_range2d`

#### [algorithms.blocked\_range2d]

Class template that represents recursively divisible two-dimensional half-open interval.

A `blocked_range2d` represents a half-open two-dimensional range  $[i_0, j_0) \times [i_1, j_1)$ . Each axis of the range has its own splitting threshold. A `blocked_range2d` is divisible if either axis is divisible.

A `blocked_range2d` meets the *Range requirements*.

```
// Defined in header <tbb/blocked_range2d.h>

namespace tbb {

    template<typename RowValue, typename ColValue=RowValue>
    class blocked_range2d {
public:
    // Types
    using row_range_type = blocked_range<RowValue>;
    using col_range_type = blocked_range<ColValue>;
}
```

(continues on next page)

(continued from previous page)

```

// Constructors
blocked_range2d(
    RowValue row_begin, RowValue row_end,
    typename row_range_type::size_type row_grainsize,
    ColValue col_begin, ColValue col_end,
    typename col_range_type::size_type col_grainsize);
blocked_range2d( RowValue row_begin, RowValue row_end,
                 ColValue col_begin, ColValue col_end );
// Splitting constructors
blocked_range2d( blocked_range2d& r, split );
blocked_range2d( blocked_range2d& r, proportional_split proportion );

// Capacity
bool empty() const;

// Access
bool is_divisible() const;
const row_range_type& rows() const;
const col_range_type& cols() const;
};

} // namespace tbb

```

Requirements:

- The *RowValue* and *ColValue* shall meet the *blocked\_range requirements*

#### 11.2.2.2.1 Member types

```
using row_range_type = blocked_range<RowValue>;
```

The type of the row values.

```
using col_range_type = blocked_range<ColValue>;
```

The type of the column values.

#### 11.2.2.2.2 Member functions

```

blocked_range2d(
    RowValue row_begin, RowValue row_end,
    typename row_range_type::size_type row_grainsize,
    ColValue col_begin, ColValue col_end,
    typename col_range_type::size_type col_grainsize);

```

**Effects:** Constructs a *blocked\_range2d* representing a two-dimensional space of values. The space is the half-open Cartesian product  $[row\_begin, row\_end) \times [col\_begin, col\_end)$ , with the given grain sizes for the rows and columns.

**Example:** The statement `blocked_range2d<char,int> r('a', 'z'+1, 3, 0, 10, 2);` constructs a two-dimensional space that contains all value pairs of the form  $(i, j)$ , where  $i$  ranges from 'a' to 'z' with a grain size of 3, and  $j$  ranges from 0 to 9 with a grain size of 2.

```
blocked_range2d(RowValue row_begin, RowValue row_end,
                ColValue col_begin, ColValue col_end);
```

Same as `blocked_range2d(row_begin, row_end, 1, col_begin, col_end, 1)`.

```
blocked_range2d(blocked_range2d& range, split);
```

Basic splitting constructor.

**Requirements:** `is_divisible()` is true.

**Effects:** Partitions range into two subranges. The newly constructed `blocked_range2d` is approximately the second half of the original `range`, and `range` is updated to be the remainder. Each subrange has the same grain size as the original `range`. The split is either by rows or columns. The choice of which axis to split is intended to cause, after repeated splitting, the subranges to approach the aspect ratio of the respective row and column grain sizes.

```
blocked_range2d(blocked_range2d& range, proportional_split proportion);
```

Proportional splitting constructor.

**Requirements:** `is_divisible()` is true.

**Effects:** Partitions `range` into two subranges in the given `proportion` across one of its axes. The choice of which axis to split is made in the same way as for the basic splitting constructor; then, proportional splitting is done for the chosen axis. The second axis and the grain sizes for each subrange remain the same as in the original `range`.

```
bool empty() const;
```

**Effects:** Determines if `range` is empty.

**Returns:** `rows().empty() || cols().empty()`

```
bool is_divisible() const;
```

**Effects:** Determines if `range` can be split into subranges.

**Returns:** `rows().is_divisible() || cols().is_divisible()`

```
const row_range_type& rows() const;
```

**Returns:** Range containing the rows of the value space.

```
const col_range_type& cols() const;
```

**Returns:** Range containing the columns of the value space.

See also:

- `blocked_range`

### 11.2.2.2.3 blocked\_range3d

#### [algorithms.blocked\_range3d]

Class template that represents recursively divisible three-dimensional half-open interval.

A `blocked_range3d` is the three-dimensional extension of `blocked_range2d`.

```
namespace tbb {
    template<typename PageValue, typename RowValue=PageValue, typename ColValue=RowValue>
    class blocked_range3d {
    public:
        // Types
        using page_range_type = blocked_range<PageValue>;
        using row_range_type = blocked_range<RowValue>;
        using col_range_type = blocked_range<ColValue>;

        // Constructors
        blocked_range3d(
            PageValue page_begin, PageValue page_end,
            typename page_range_type::size_type page_grainsize,
            RowValue row_begin, RowValue row_end,
            typename row_range_type::size_type row_grainsize,
            ColValue col_begin, ColValue col_end,
            typename col_range_type::size_type col_grainsize );
        blocked_range3d( PageValue page_begin, PageValue page_end
                        RowValue row_begin, RowValue row_end,
                        ColValue col_begin, ColValue col_end );
        blocked_range3d( blocked_range3d& r, split );
        blocked_range3d( blocked_range3d& r, proportional_split& proportion );

        // Capacity
        bool empty() const;

        // Access
        bool is_divisible() const;
        const page_range_type& pages() const;
        const row_range_type& rows() const;
        const col_range_type& cols() const;
    };
}
```

Requirements:

- The `PageValue`, `RowValue` and `ColValue` shall meet the *blocked\_range requirements*

#### 11.2.2.2.3.1 Member types

```
using page_range_type = blocked_range<PageValue>;
```

The type of the page values.

```
using row_range_type = blocked_range<RowValue>;
```

The type of the row values.

```
using col_range_type = blocked_range<ColValue>;
```

The type of the column values.

### 11.2.2.3.2 Member functions

```
blocked_range3d(PageValue page_begin, PageValue page_end,
    typename page_range_type::size_type page_grainsize,
    RowValue row_begin, RowValue row_end,
    typename row_range_type::size_type row_grainsize,
    ColValue col_begin, ColValue col_end,
    typename col_range_type::size_type col_grainsize);
```

**Effects:** Constructs a `blocked_range3d` representing a three-dimensional space of values. The space is the half-open Cartesian product  $[page\_begin, page\_end) \times [row\_begin, row\_end) \times [col\_begin, col\_end)$ , with the given grain sizes for the pages, rows and columns.

**Example:** The statement `blocked_range3d<int,char,int> r(0, 6, 2, 'a', 'z'+1, 3, 0, 10, 2);` constructs a three-dimensional space that contains all value pairs of the form  $(i, j, k)$ , where  $i$  ranges from 0 to 6 with a grain size of 2,  $j$  ranges from 'a' to 'z' with a grain size of 3, and  $k$  ranges from 0 to 9 with a grain size of 2.

```
blocked_range3d(PageValue page_begin, PageValue page_end,
    RowValue row_begin, RowValue row_end,
    ColValue col_begin, ColValue col_end);
```

Same as `blocked_range3d(page_begin,page_end,1,row_begin,row_end,1,col_begin,col_end,1)`.

```
blocked_range3d( blocked_range3d& range, split );
```

Basic splitting constructor.

**Requirements:** `is_divisible()` is true.

**Effects:** Partitions `range` into two subranges. The newly constructed `blocked_range3d` is approximately the second half of the original `range`, and `range` is updated to be the remainder. Each subrange has the same grain size as the original `range`. The split is either by pages, rows or columns. The choice of which axis to split is intended to cause, after repeated splitting, the subranges to approach the aspect ratio of the respective page, row and column grain sizes.

```
blocked_range3d( blocked_range3d& range, proportional_split proportion );
```

Proportional splitting constructor.

**Requirements:** `is_divisible()` is true.

**Effects:** Partitions `range` into two subranges in the given `proportion` across one of its axes. The choice of which axis to split is made in the same way as for the basic splitting constructor; then, proportional splitting is done for the chosen axis. The second axis and the grain sizes for each subrange remain the same as in the original `range`.

```
bool empty() const;
```

**Effects:** Determines if `range` is empty.

**Returns:** `pages.empty() || rows().empty() || cols().empty()`

```
bool is_divisible() const;
```

**Effects:** Determines if range can be split into subranges.

**Returns:** `pages().is_divisible() || rows().is_divisible() || cols().is_divisible()`

```
const page_range_type& pages() const;
```

**Returns:** Range containing the pages of the value space.

```
const row_range_type& rows() const;
```

**Returns:** Range containing the rows of the value space.

```
const col_range_type& cols() const;
```

**Returns:** Range containing the columns of the value space.

See also:

- `blocked_range`
- `blocked_range2d`

### 11.2.2.3 Partitioners

A partitioner specifies how a loop template should partition its work among threads.

#### 11.2.2.3.1 auto\_partitioner

##### [algorithms.auto\_partitioner]

Specify that a parallel loop should optimize its range subdivision based on work-stealing events.

A loop template with an `auto_partitioner` attempts to minimize range splitting while providing ample opportunities for work-stealing.

The range subdivision is initially limited to S subranges, where S is proportional to the number of threads specified by the `global_ctrl` or `task_arena`. Each of these subranges is not divided further unless it is stolen by an idle thread. If stolen, it is further subdivided to create additional subranges. Thus a loop template with an `auto_partitioner` creates additional subranges only when necessary to balance a load.

Performs sufficient splitting to balance load, not necessarily splitting as finely as `Range::is_divisible` permits. When used with classes such as `blocked_range`, the selection of an appropriate grain size is less important, and often acceptable performance can be achieved with the default grain size of 1.

The `auto_partitioner` class satisfies the *CopyConstructible* requirement from ISO C++ [utility.arg.requirements] section.

---

**Tip:** When using `auto_partitioner` and a `blocked_range` for a parallel loop, the body may receive a subrange larger than the grain size of the `blocked_range`. Therefore do not assume that the grain size is an upper bound on the size of a subrange. Use `simple_partitioner` if an upper bound is required.

```
// Defined in header <tbb/partitioner.h>

namespace tbb {

    class auto_partitioner {
public:
    auto_partitioner() = default;
    ~auto_partitioner() = default;
};

}
```

### 11.2.2.3.2 affinity\_partitioner

#### [algorithms.affinity\_partitioner]

Hint that loop iterations should be assigned to threads in a way that optimizes for cache affinity.

An `affinity_partitioner` hints that execution of a loop template should use the same task affinity pattern for splitting the work as used by previous execution of the loop (or another loop) with the same `affinity_partitioner` object.

`affinity_partitioner` uses proportional splitting when it is enabled for a `Range` type.

Unlike the other partitioners, it is important that the same `affinity_partitioner` object be passed to the loop templates to be optimized for affinity.

The `affinity_partitioner` class satisfies the *CopyConstructible* requirement from ISO C++ [utility.arg.requirements] section.

```
// Defined in header <tbb/partitioner.h>

namespace tbb {

    class affinity_partitioner {
public:
    affinity_partitioner() = default;
    ~affinity_partitioner() = default;
};

}
```

See also:

- *Range named requirement*

### 11.2.2.3.3 static\_partitioner

#### [algorithms.static\_partitioner]

Specify that a parallel algorithm should distribute the work uniformly across threads and should not do additional load balancing.

An algorithm with a `static_partitioner` distributes the range across threads in subranges of approximately equal size. The number of subranges is equal to the number of threads that can possibly participate in task execution, as specified by `global_control` or `task_arena` classes. These subranges are not further split.

**Caution:** The regularity of subrange sizes is not guaranteed if the range type does not support proportional splitting, or if the grain size is set larger than the size of the range divided by the number of threads participating in task execution.

In addition, `static_partitioner` uses a deterministic task affinity pattern to hint the task scheduler how the subranges should be assigned to threads.

The `static_partitioner` class satisfies the *CopyConstructible* requirement from ISO C++ [utility.arg.requirements] section.

**Tip:** Use of `static_partitioner` is recommended for:

- Parallelizing small well-balanced workloads where enabling additional load balancing opportunities would bring more overhead than performance benefits.
- Porting OpenMP\* parallel loops with `schedule(static)` if deterministic work partitioning across threads is important.

```
// Defined in header <tbb/partitioner.h>

namespace tbb {

    class static_partitioner {
public:
    static_partitioner() = default;
    ~static_partitioner() = default;
};

}
```

See also:

- *Range named requirement*

#### 11.2.2.3.4 simple\_partitioner

##### [algorithms.simple\_partitioner]

Specify that a parallel loop should recursively split its range until it cannot be subdivided further.

A `simple_partitioner` specifies that a loop template should recursively divide its range until for each subrange  $r$ , the condition `!r.is_divisible()` holds. This is the default behavior of the loop templates that take a range argument.

The `simple_partitioner` class satisfies the *CopyConstructible* requirement from ISO C++ [utility.arg.requirements] section.

**Tip:** When using `simple_partitioner` and a `blocked_range` for a parallel loop, be careful to specify an appropriate grain size for the `blocked_range`. The default grain size is 1, which may make the subranges much too small for efficient execution.

```
// Defined in header <tbb/partitioner.h>
```

(continues on next page)

(continued from previous page)

```
namespace tbb {

    class simple_partitioner {
    public:
        simple_partitioner() = default;
        ~simple_partitioner() = default;
    };
}
```

See also:

- *Range named requirement*

#### 11.2.2.4 Split Tags

##### 11.2.2.4.1 proportional split

###### [algorithms.proportional\_split]

Type of an argument for a proportional splitting constructor of *Range*.

An argument of type `proportional_split` may be used by classes that satisfies *Range requirements* to distinguish a proportional splitting constructor from a basic splitting constructor and from a copy constructor, and to suggest a ratio in which a particular instance of the class should be split.

```
// Defined in header <tbb/blocked_range.h>
// Defined in header <tbb/blocked_range2d.h>
// Defined in header <tbb/blocked_range3d.h>
// Defined in header <tbb/partitioner.h>
// Defined in header <tbb/parallel_for.h>
// Defined in header <tbb/parallel_reduce.h>
// Defined in header <tbb/parallel_scan.h>

namespace tbb {
    class proportional_split {
    public:
        proportional_split(std::size_t _left = 1, std::size_t _right = 1);

        std::size_t left() const;
        std::size_t right() const;

        operator split() const;
    };
}
```

### 11.2.2.4.1.1 Member functions

**proportional\_split** (std::size\_t \_left = 1, std::size\_t \_right = 1)

Constructs a proportion with the ratio specified by coefficients *\_left* and *\_right*.

**std::size\_t left () const**

Returns size of the left part of the proportion.

**std::size\_t right () const**

Returns size of the right part of the proportion.

**operator split () const**

Makes `proportional_split` implicitly convertible to `split` type to use with ranges that do not support proportional splitting.

See also:

- *split*
- *Range requirements*

### 11.2.2.4.2 split

#### [algorithms.split]

Type of an argument for a splitting constructor of `Range`. An argument of type `split` is used to distinguish a splitting constructor from a copy constructor.

```
// Defined in header <tbb/blocked_range.h>
// Defined in header <tbb/blocked_range2d.h>
// Defined in header <tbb/blocked_range3d.h>
// Defined in header <tbb/partitioner.h>
// Defined in header <tbb/parallel_for.h>
// Defined in header <tbb/parallel_reduce.h>
// Defined in header <tbb/parallel_scan.h>

class split;
```

See also:

- *Range requirements*

### 11.2.3 Flow Graph

#### [flow\_graph]

In addition to loop parallelism, the oneAPI Threading Building Blocks library also supports graph parallelism. It's possible to create graphs that are highly scalable, but it is also possible to create graphs that are completely sequential.

There are 3 types of components used to implement a graph:

- A graph class instance
- Nodes
- Ports and edges

### 11.2.3.1 Graph Class

The graph class instance is the owner of the tasks created on behalf of the flow graph. Users can wait on the graph if they need to wait for the completion of all of the tasks related to the flow graph execution. One can also register external interactions with the graph and run tasks under the ownership of the flow graph.

#### 11.2.3.1.1 graph

##### [flow\_graph.graph]

Class that serves as a handle to a flow graph of nodes and edges.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    class graph {
    public:
        graph();
        graph(task_group_context& context);
        ~graph();

        void wait_for_all();

        bool is_cancelled();
        bool exception_thrown();
        void reset(reset_flags f = rf_reset_protocol);
    };

} // namespace flow
} // namespace tbb
```

##### 11.2.3.1.1.1 reset\_flags enumeration

##### 11.2.3.1.1.2 reset\_flags Enumeration

##### [flow\_graph.reset\_flags]

A reset\_flags enumeration represents flags that can be passed to graph::reset() function.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    enum reset_flags {
        rf_reset_protocol = /*implementation-defined*/,
        rf_reset_bodies = /*implementation-defined*/,
        rf_clear_edges = /*implementation-defined*/
    };

} // namespace flow
} // namespace tbb
```

Its enumerated values and their meanings are as follows:

- `rf_reset_protocol` - All buffers are emptied, internal state of nodes reinitialized. All calls to `reset()` perform these actions.
- `rf_reset_bodies` - When nodes with bodies are created, the body specified in the constructor is copied and preserved. When `rf_reset_bodies` is specified, the current body of the node is deleted and replaced with a copy of the body saved during construction.

**Caution:** If the body contains state which has an external component (such as a file descriptor) then the node may not behave the same on re-execution of the graph after body replacement. In this case the node should be re-created.

- `rf_clear_edges` - All edges are removed from the graph.

#### 11.2.3.1.1.3 Member functions

##### `graph(task_group_context &group)`

Constructs a graph with no nodes. If `group` is specified the graph tasks are executed in this group. By default the graph is executed in a bound context of its own.

##### `~graph()`

Calls `wait_for_all` on the graph, then destroys the graph.

##### `void wait_for_all()`

Blocks until all tasks associated with the graph have completed.

##### `void reset(reset_flags f = rf_reset_protocol)`

Reset the graph regards to specified flags. Flags to `reset()` can be combined with bitwise-or.

##### `bool is_cancelled()`

Returns: `true` if the graph was cancelled during the last call to `wait_for_all()`, `false` otherwise.

##### `bool exception_thrown()`

Returns: `true` if during the last call to `wait_for_all()` an exception was thrown, `false` otherwise.

#### 11.2.3.2 Nodes

##### 11.2.3.2.1 Abstract Interfaces

In order to be used as a graph node type, a class needs to inherit certain abstract types and implement the corresponding interfaces. `graph_node` is the base class for any other node type; its interfaces always have to be implemented. If a node sends messages to other nodes, it has to implement the `sender` interface, while with `receiver` interface the node may accept messages. For nodes that have multiple inputs and/or outputs, each input port is a `receiver` and each output port is a `sender`.

### 11.2.3.2.1.1 graph\_node

#### [flow\_graph.graph\_node]

A base class for all graph nodes.

```
namespace tbb {
namespace flow {

class graph_node {
public:
    explicit graph_node( graph &g );
    virtual ~graph_node();
};

} // namespace flow
} // namespace tbb
```

The class `graph_node` is a base class for all flow graph nodes. The virtual destructor allows flow graph nodes to be destroyed through pointers to `graph_node`. For example, a `vector< graph_node * >` could be used to hold the addresses of flow graph nodes that will later need to be destroyed.

### 11.2.3.2.1.2 sender

#### [flow\_graph.sender]

A base class for all nodes that may send messages.

```
namespace tbb {
namespace flow {

template< typename T >
class sender { /*unspecified*/ };

} // namespace flow
} // namespace tbb
```

The `T` type is a message type.

### 11.2.3.2.1.3 receiver

#### [flow\_graph.receiver]

A base class for all nodes that may receive messages.

```
namespace tbb {
namespace flow {

template< typename T >
class receiver { /*unspecified*/ };

} // namespace flow
} // namespace tbb
```

The `T` type is a message type.

### 11.2.3.2.2 Properties

Every node in flow graph has its own properties.

#### 11.2.3.2.2.1 Forwarding and Buffering

[flow\_graph.forwarding\_and\_buffering]

#### 11.2.3.2.2.2 Forwarding

In an oneAPI Threading Building Blocks `flow::graph`, nodes which forward messages to successors have one of two possible forwarding policies, which are a property of the node:

- **broadcast-push** - the message will be pushed to as many successors as will accept the message. If no successor accepts the message, the fate of the message depends on the output buffering policy of the node.
- **single-push** - if the message is accepted by a successor, no further push of that message will occur. If a successor rejects the message the next successor in the set is tried. This continues until a successor accepts the message, or all successors have been attempted. If no successor accepts the message, it will be retained for a possible future resend. Message that is successfully transferred to a successor is removed from the node.

#### 11.2.3.2.2.3 Buffering

There are two policies for handling a message which cannot be pushed to any successor:

- **buffering** - if no successor accepts a message, it is stored so subsequent node processing can use it. Nodes that buffer outputs have “yes” in the column “try\_get()?” below.
- **discarding** - if no successor accepts a message, it is discarded and has no further effect on graph execution. Nodes that discard outputs have “no” in the column “try\_get()?” below.

The following table lists the policies of each node:

Table 4: Buffering and Forwarding properties summary

Node	try_get()?	Forwarding
<b>Functional Nodes</b>		
input_node	yes	broadcast-push
function_node<rejecting>	no	broadcast-push
function_node<queueing>	no	broadcast-push
continue_node	no	broadcast-push
multifunction_node<rejecting>	no	broadcast-push
multifunction_node<queueing>	no	broadcast-push
<b>Buffering Nodes</b>		
buffer_node	yes	single-push
priority_queue_node	yes	single-push
queue_node	yes	single-push
sequencer_node	yes	single-push
overwrite_node	yes	broadcast-push
write_once_node	yes	broadcast-push
<b>Split/Join Nodes</b>		
join_node<queueing>	yes	broadcast-push
join_node<reserving>	yes	broadcast-push
join_node<tag_matching>	yes	broadcast-push
split_node	no	broadcast-push
indexer_node	no	broadcast-push
<b>Other Nodes</b>		
broadcast_node	no	broadcast-push
limiter_node	no	broadcast-push

### 11.2.3.2.3 Functional Nodes

Functional nodes do computations in response to input messages (if any), and send the result or a signal to their successors.

#### 11.2.3.2.3.1 continue\_node

##### [flow\_graph.continue\_node]

A node that executes a specified body object when triggered.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template< typename Output, typename Policy = /*implementation-defined*/ >
    class continue_node : public graph_node, public receiver<continue_msg>, public_
    ~sender<Output> {
        public:
            template<typename Body>
            continue_node( graph &g, Body body, node_priority_t priority = no_priority );
            template<typename Body>
            continue_node( graph &g, Body body, Policy /*unspecified*/ = Policy(),
                           node_priority_t priority = no_priority );

```

(continues on next page)

(continued from previous page)

```

template<typename Body>
continue_node( graph &g, int number_of_predecessors, Body body,
              node_priority_t priority = no_priority );
template<typename Body>
continue_node( graph &g, int number_of_predecessors, Body body,
              Policy /*unspecified*/ = Policy(), node_priority_t priority =_
              ↵no_priority );

    continue_node( const continue_node &src );
    ~continue_node();

    bool try_put( const input_type &v );
    bool try_get( output_type &v );
};

} // namespace flow
} // namespace tbb

```

#### Requirements:

- The type `Output` shall meet the *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.
- The type `Policy` may be specified as *lightweight policy* or defaulted.
- The type `Body` shall meet the *ContinueNodeBody requirements*.

A `continue_node` is a `graph_node`, `receiver<continue_msg>` and `sender<Output>`.

This node is used for nodes that wait for their predecessors to complete before executing, but no explicit data is passed across the incoming edges.

A `continue_node` maintains an internal threshold that defines the number of predecessors. This value may be provided at construction. Call of the *make\_edge function* with `continue_node` as a receiver increases its threshold. Call of the *remove\_edge function* with `continue_node` as a receiver decreases it.

Each time the number of `try_put()` calls reaches the defined threshold, node's `body` is called and the node starts counting the number of `try_put()` calls from the beginning.

`continue_node` has a *discarding* and *broadcast-push properties*.

The body object passed to a `continue_node` is copied. Therefore updates to member variables will not affect the original object used to construct the node. If the state held within a body object must be inspected from outside of the node, the *copy\_body function* can be used to obtain an updated copy.

#### 11.2.3.2.3.2 Member functions

```

template<typename Body>
continue_node( graph &g, Body body, node_priority_t priority = no_priority );

```

Constructs a `continue_node` that invokes `body`. The internal threshold is set to 0.

Allows to specify *node priority*.

```
template<typename Body>
continue_node( graph &g, Body body, Policy /*unspecified*/ = Policy(),
               node_priority_t priority = no_priority );
```

Constructs a `continue_node` that invokes `body`. The internal threshold is set to 0.

Allows to specify *lightweight policy* and *node priority*.

```
template<typename Body>
continue_node( graph &g, int number_of_predecessors, Body body,
               node_priority_t priority = no_priority );
```

Constructs a `continue_node` that invokes `body`. The internal threshold is set to `number_of_predecessors`.

Allows to specify *node priority*.

```
template<typename Body>
continue_node( graph &g, int number_of_predecessors, Body body,
               Policy /*unspecified*/ = Policy(), node_priority_t priority = no_
               ↪priority );
```

Constructs a `continue_node` that invokes `body`. The internal threshold is set to `number_of_predecessors`.

Allows to specify *lightweight policy* and *node priority*.

```
template<typename Body>
continue_node( graph &g, int number_of_predecessors, Body body );
```

Constructs a `continue_node` that invokes `body`. The internal threshold is set to `number_of_predecessors`.

```
continue_node( const continue_node &src )
```

Constructs a `continue_node` that has the same initial state that `src` had after its construction. It does not copy the current count of `try_puts` received, or the current known number of predecessors. The `continue_node` that is constructed will have a reference to the same `graph` object as `src`, have a copy of the initial `body` used by `src`, and only have a non-zero threshold if `src` was constructed with a non-zero threshold.

The new `body` object is copy-constructed from a copy of the original `body` provided to `src` at its construction.

```
bool try_put( const Input &v )
```

Increments the count of `try_put()` calls received. If the incremented count is equal to the number of known predecessors, performs the `body` function object execution. It does not wait for the execution of the `body` to complete.

**Returns:** `true`

```
bool try_get( Output &v )
```

**Returns:** `false`

### 11.2.3.2.3.3 Deduction Guides

```

template <typename Body, typename Policy>
continue_node(graph&, Body, Policy, node_priority_t = no_priority)
    -> continue_node<continue_output_t<std::invoke_result_t<Body, continue_msg>>, 
        ↵Policy>;

template <typename Body, typename Policy>
continue_node(graph&, int, Body, Policy, node_priority_t = no_priority)
    -> continue_node<continue_output_t<std::invoke_result_t<Body, continue_msg>>, 
        ↵Policy>;

template <typename Body>
continue_node(graph&, Body, node_priority_t = no_priority)
    -> continue_node<continue_output_t<std::invoke_result_t<Body, continue_msg>>, / 
        ↵*default-policy*/>;

template <typename Body>
continue_node(graph&, int, Body, node_priority_t = no_priority)
    -> continue_node<continue_output_t<std::invoke_result_t<Body, continue_msg>>, / 
        ↵*default-policy*/>;

```

Where:

- `continue_output_t<Output>` is an alias to `Output` template argument type. If `Output` specified as `void` then `continue_output_t<Output>` is an alias to `continue_msg` type.

### 11.2.3.2.3.4 Example

A set of `continue_nodes` forms a *Dependency Flow Graph*.

### 11.2.3.2.3.5 function\_node

#### [flow\_graph.function\_node]

A node that executes a user-provided body on incoming messages.

```

// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template < typename Input, typename Output = continue_msg, typename Policy = / 
        ↵*implementation-defined*/ >
        class function_node : public graph_node, public receiver<Input>, public sender
        ↵<Output> {
            public:
                template<typename Body>
                function_node( graph &g, size_t concurrency, Body body, Policy /*unspecified*/
                    ↵ = Policy(), 
                    node_priority_t priority = no_priority );
                template<typename Body>
                function_node( graph &g, size_t concurrency, Body body,
                    node_priority_t priority = no_priority );
                ~function_node();
}

```

(continues on next page)

(continued from previous page)

```

        function_node( const function_node &src );

        bool try_put( const Input &v );
        bool try_get( Output &v );
    };

} // namespace flow
} // namespace tbb

```

#### Requirements:

- The `Input` and `Output` types shall meet the *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.
- The type `Policy` may be specified as *lightweight, queueing and rejecting policies* or defaulted.
- The type `Body` shall meet the *FunctionNodeBody requirements*.

`function_node` has a user-settable concurrency limit. It can be set to one of *predefined values*. The user can also provide a value of type `std::size_t` to limit concurrency to a value between 1 and `tbb::flow::unlimited`.

Messages that cannot be immediately processed due to concurrency limits are handled according to the `Policy` template argument.

`function_node` is a `graph_node`, `receiver<Input>` and `sender<Output>`.

`function_node` has a *discarding* and *broadcast-push properties*.

The body object passed to a `function_node` is copied. Therefore updates to member variables will not affect the original object used to construct the node. If the state held within a body object must be inspected from outside of the node, the `copy_body` function can be used to obtain an updated copy.

#### 11.2.3.2.3.6 Member functions

```

template<typename Body>
function_node( graph &g, size_t concurrency, Body body,
               node_priority_t priority = no_priority );

```

Constructs a `function_node` that will invoke a copy of `body`. At most `concurrency` calls to `body` may be made concurrently.

Allows to specify *node priority*.

```

template<typename Body>
function_node( graph &g, size_t concurrency, Body body, Policy /*unspecified*/ = _
               Policy(),
               node_priority_t priority = no_priority );

```

Constructs a `function_node` that will invoke a copy of `body`. At most `concurrency` calls to `body` may be made concurrently.

Allows to specify a *policy* and *node priority*.

```
function_node( const function_node &src )
```

Constructs a `function_node` that has the same initial state that `src` had when it was constructed. The `function_node` that is constructed will have a reference to the same `graph` object as `src`, will have a copy of the initial body used by `src`, and have the same concurrency threshold as `src`. The predecessors and successors of `src` will not be copied.

The new body object is copy-constructed from a copy of the original body provided to `src` at its construction. Therefore changes made to member variables in `src`'s body after the construction of `src` will not affect the body of the new `function_node`.

```
bool try_put( const Input &v )
```

**Returns:** `true` if the input was accepted; and `false` otherwise.

```
bool try_get( Output &v )
```

**Returns:** `false`

#### 11.2.3.2.3.7 Deduction Guides

```
template <typename Body, typename Policy>
function_node(graph&, size_t, Body, Policy, node_priority_t = no_priority)
    ->function_node<std::decay_t<input_t<Body>>, output_t<Body>, Policy>;
template <typename Body>
function_node(graph&, size_t, Body, node_priority_t = no_priority)
    ->function_node<std::decay_t<input_t<Body>>, output_t<Body>, /*default-policy*/>;
```

Where:

- `input_t` is an alias to `Body` input argument type.
- `output_t` is an alias to `Body` return type.

#### 11.2.3.2.3.8 Example

*Data Flow Graph example* illustrates how `function_node` could do computation on input data and pass the result to successors.

#### 11.2.3.2.3.9 `input_node`

##### [`flow_graph.input_node`]

A node that generates messages by invoking the user-provided functor and broadcasts the result to all of its successors.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {
```

(continues on next page)

(continued from previous page)

```

template < typename Output >
class input_node : public graph_node, public sender<Output> {
public:
    template< typename Body >
    input_node( graph &g, Body body );
    input_node( const input_node &src );
    ~input_node();

    void activate();
    bool try_get( Output &v );
};

} // namespace flow
} // namespace tbb

```

#### Requirements:

- The `Output` type shall meet the *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.
- The type `Body` shall meet the *InputNodeBody requirements*.

This node can have no predecessors. It executes a user-provided `body` function object to generate messages that are broadcast to all successors. It is a serial node and will never call its `body` concurrently. It is able to buffer a single item. If no successor accepts an item that it has generated, the message is buffered and will be provided to successors before a new item is generated.

`input_node` is a `graph_node` and `sender<Output>`.

`input_node` has a *buffering* and *broadcast-push properties*.

An `input_node` will continue to invoke `body` and broadcast messages until the `body` toggles `fc.stop()` or it has no valid successors. A message may be generated and then rejected by all successors. In that case, the message is buffered and will be the next message sent once a successor is added to the node or `try_get` is called. Calls to `try_get` will return a buffer message if available or will invoke `body` to attempt to generate a new message. A call to `body` is made only when the internal buffer is empty.

The `body` object passed to a `input_node` is copied. Therefore updates to member variables will not affect the original object used to construct the node. If the state held within a `body` object must be inspected from outside of the node, the *copy\_body function* can be used to obtain an updated copy.

#### 11.2.3.2.3.10 Member functions

`template<typename Body>`

`input_node(graph &g, Body body)`

Constructs an `input_node` that will invoke `body`. By default the node is created in an inactive state, that is, messages will not be generated until a call to `activate` is made.

`input_node(const input_node &src)`

Constructs an `input_node` that has the same initial state that `src` had when it was constructed. The `input_node` that is constructed will have a reference to the same `graph` object as `src`, will have a copy of the initial `body` used by `src`, and have the same initial active state as `src`. The predecessors and successors of `src` will not be copied.

The new `body` object is copy-constructed from a copy of the original `body` provided to `src` at its construction. Therefore changes made to member variables in `src`'s `body` after the construction of `src` will not affect the

body of the new `input_node`.

`void activate()`  
Sets the `input_node` to the active state, allowing it to begin generating messages.

`bool try_get (Output &v)`  
Will copy the buffered message into `v` if available or will invoke `body` to attempt to generate a new message that will be copied into `v`.

**Returns:** `true` if a message is copied to `v`. `false` otherwise.

### 11.2.3.2.3.11 Deduction Guides

```
template <typename Body>
input_node(graph&, Body) -> input_node<std::decay_t<input_t<Body>>>;
```

Where:

- `input_t` is an alias to `Body` input argument type.

### 11.2.3.2.3.12 multifunction\_node

#### [flow\_graph.multifunction\_node]

A node that used for nodes that receive messages at a single input port and may generate one or more messages that are broadcast to successors.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template < typename Input, typename Output, typename Policy = /*implementation-
→defined*/ >
    class multifunction_node : public graph_node, public receiver<Input> {
    public:
        template<typename Body>
        multifunction_node( graph &g, size_t concurrency, Body body, Policy /
→*unspecified*/ = Policy(),
                            node_priority_t priority = no_priority );
        template<typename Body>
        multifunction_node( graph &g, size_t concurrency, Body body,
                            node_priority_t priority = no_priority );

        multifunction_node( const multifunction_node& other );
        ~multifunction_node();

        bool try_put( const Input &v );

        using output_ports_type = /*implementation-defined*/;
        output_ports_type& output_ports();
    };

} // namespace flow
} // namespace tbb
```

Requirements:

- The `Input` and `Output` types shall meet the *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.
- The type `Policy` may be specified as *lightweight, queueing and rejecting policies* or defaulted.
- The type `Body` shall meet the *MultifunctionNodeBody requirements*.

`multifunction_node` has a user-settable concurrency limit. It can be set to one of *predefined values*. The user can also provide a value of type `std::size_t` to limit concurrency to a value between 1 and `tbb::flow::unlimited`.

When the concurrency limit allows, it executes the user-provided body on incoming messages. The body may create one or more output messages and broadcast them to successors.

`multifunction_node` is a `graph_node`, `receiver<InputType>` and has a tuple of `sender<Output>` outputs.

`multifunction_node` has a *discarding* and *broadcast-push properties*.

The body object passed to a `multifunction_node` is copied. Therefore updates to member variables will not affect the original object used to construct the node. If the state held within a body object must be inspected from outside of the node, the `copy_body function` can be used to obtain an updated copy.

#### 11.2.3.2.3.13 Member types

`output_ports_type` is an alias to a tuple of output ports.

#### 11.2.3.2.3.14 Member functions

```
template<typename Body>
multifunction_node( graph &g, size_t concurrency, Body body,
                    node_priority_t priority = no_priority );
```

Constructs a `multifunction_node` that will invoke a copy of `body`. At most `concurrency` calls to `body` may be made concurrently.

Allows to specify *node priority*.

```
template<typename Body>
multifunction_node( graph &g, size_t concurrency, Body body, Policy /*unspecified*/ = _Policy(),
                    node_priority_t priority = no_priority );
```

Constructs a `multifunction_node` that will invoke a copy of `body`. At most `concurrency` calls to `body` may be made concurrently.

Allows to specify a *policy* and *node priority*.

```
multifunction_node( const multifunction_node &src )
```

Constructs a `multifunction_node` that has the same initial state that `other` had when it was constructed. The `multifunction_node` that is constructed will have a reference to the same `graph` object as `other`, will have a copy of the initial `body` used by `other`, and have the same concurrency threshold as `other`. The predecessors and successors of `other` will not be copied.

The new body object is copy-constructed from a copy of the original body provided to other at its construction. Therefore changes made to member variables in other's body after the construction of other will not affect the body of the new multifunction\_node.

```
bool try_put( const input_type &v )
```

If concurrency limit allows, executes the user-provided body on incoming messages and returns true. Otherwise executes nothing and return false.

```
output_ports_type& output_ports();
```

**Returns:** a tuple of output ports.

#### 11.2.3.2.3.15 async\_node

##### [flow\_graph.async\_node]

A node that allows a flow graph to communicate with an external activity managed by the user or another runtime.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template < typename Input, typename Output, typename Policy = /*implementation-
→defined*/ >
    class async_node : public graph_node, public receiver<Input>, public sender
→<Output> {
    public:
        template<typename Body>
        async_node( graph &g, size_t concurrency, Body body, Policy /*unspecified*/ =_
→Policy(),
            node_priority_t priority = no_priority );
        template<typename Body>
        async_node( graph &g, size_t concurrency, Body body, node_priority_t priority_
→= no_priority );

        async_node( const async_node& src );
        ~async_node();

        using gateway_type = /*implementation-defined*/;
        gateway_type& gateway();

        bool try_put( const input_type& v );
        bool try_get( output_type& v );
    };

} // namespace flow
} // namespace tbb
```

Requirements:

- The Input and Output types shall meet the *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.

- The type `Policy` may be specified as *lightweight, queueing and rejecting policies* or defaulted.
- The type `Body` shall meet the *AsyncNodeBody requirements*.

`async_node` executes a user-provided body on incoming messages. The body submits input messages to an external activity for processing outside of the task scheduler. This node also provides `gateway_type` interface that allows the external activity to communicate with the flow graph.

`async_node` is a `graph_node`, `receiver<Input>` and a `sender<Output>`.

`async_node` has a *discarding* and *broadcast-push properties*.

`async_node` has a user-settable concurrency limit. It can be set to one of *predefined values*. The user can also provide a value of type `std::size_t` to limit concurrency to a value between 1 and `tbb::flow::unlimited`.

The body object passed to a `async_node` is copied. Therefore updates to member variables will not affect the original object used to construct the node. If the state held within a body object must be inspected from outside of the node, the `copy_body` function can be used to obtain an updated copy.

#### 11.2.3.2.3.16 Member functions

`gateway_type` meets the *GatewayType requirements*.

#### 11.2.3.2.3.17 Member functions

```
template<typename Body>
async_node( graph &g, size_t concurrency, Body body,
           node_priority_t priority = no_priority );
```

Constructs a `async_node` that will invoke a copy of `body`. At most `concurrency` calls to `body` may be made concurrently.

Allows to specify *node priority*.

```
template<typename Body>
async_node( graph &g, size_t concurrency, Body body, Policy /*unspecified*/ = _  
          Policy(),
           node_priority_t priority = no_priority );
```

Constructs a `async_node` that will invoke a copy of `body`. At most `concurrency` calls to `body` may be made concurrently.

Allows to specify a *policy* and *node priority*.

```
async_node( const async_node &src )
```

Constructs an `async_node` that has the same initial state that `src` had when it was constructed. The `async_node` that is constructed will have a reference to the same `graph` object as `src`, will have a copy of the initial body used by `src`, and have the same concurrency threshold as `src`. The predecessors and successors of `src` will not be copied.

The new body object is copy-constructed from a copy of the original body provided to `src` at its construction. Therefore changes made to member variables in `src`'s body after the construction of `src` will not affect the body of the new `async_node`.

```
gateway_type& gateway()
```

Returns reference to `gateway_type` interface.

```
bool try_put( const input_type& v )
```

A task is spawned that executes the `body(v)`.

**Returns:** always returns `true`, it is responsibility of `body` to be able to pass `v` to an external activity. If a message is not properly processed by the `body` it will be lost.

```
bool try_get( output_type& v )
```

**Returns:** `false`

#### 11.2.3.2.3.18 Example

The example below shows an `async_node` that submits some work to `AsyncActivity` for processing by a user thread.

```
#include "tbb/flow_graph.h"
#include "tbb/concurrent_queue.h"
#include <thread>

using namespace tbb::flow;
typedef int input_type;
typedef int output_type;
typedef tbb::flow::async_node<input_type, output_type> async_node_type;

class AsyncActivity {
public:
    typedef async_node_type::gateway_type gateway_type;

    struct work_type {
        input_type input;
        gateway_type* gateway;
    };

    AsyncActivity() : service_thread( [this]() {
        while( !end_of_work() ) {
            work_type w;
            while( my_work_queue.try_pop(w) ) {
                output_type result = do_work( w.input );
                //send the result back to the graph
                w.gateway->try_put( result );
                // signal that work is done
                w.gateway->release_wait();
            }
        }
    } ) {} }

    void submit( input_type i, gateway_type* gateway ) {
```

(continues on next page)

(continued from previous page)

```

        work_type w = {i, gateway};
        gateway->reserve_wait();
        my_work_queue.push(w);
    }

private:
    bool end_of_work() {
        // indicates that the thread should exit
    }

    output_type do_work( input_type& v ) {
        // performs the work on input converting it to output
    }

    tbb::concurrent_queue<work_type> my_work_queue;
    std::thread service_thread;
};

int main() {
    AsyncActivity async_activity;
    tbb::flow::graph g;

    async_node_type async_consumer( g, unlimited,
        // user functor to initiate async processing
        [&] ( input_type input, async_node_type::gateway_type& gateway ) {
            async_activity.submit( input, &gateway );
        } );

    tbb::flow::input_node<input_type> s( g, [](input_type& v) ->bool { /* produce data
        →for async work */ } );
    tbb::flow::async_node<output_type> f( g, unlimited, [](const output_type& v) { /*
        →consume data from async work */ } );

    tbb::flow::make_edge( s, async_consumer );
    tbb::flow::make_edge( async_consumer, f );

    s.activate();
    g.wait_for_all();
}

```

## Auxiliary

### 11.2.3.2.3.19 Function Nodes Policies

#### [flow\_graph.function\_node\_policies]

`function_node`, `multifunction_node`, `async_node` and `continue_node` may be specified by `Policy` parameter which represented as a set of tag classes. This parameter affects node's execution behavior.

```

// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    class queueing { /*unspecified*/ };
    class rejecting { /*unspecified*/ };
}

```

(continues on next page)

(continued from previous page)

```

class lightweight { /*unspecified*/ };
class queueing_lightweight { /*unspecified*/ };
class rejecting_lightweight { /*unspecified*/ };

} // namespace flow
} // namespace tbb

```

Each policy class satisfies the the *CopyConstructible* requirements from [copyconstructible] ISO C++ Standard sections.

#### 11.2.3.2.3.20 Queueing

This policy defines behavior for input messages acceptance. `queueing` policy means that input messages that cannot be processed right away are stored in the internal buffer of the node to be processed when possible.

#### 11.2.3.2.3.21 Rejecting

This policy defines behavior for input messages acceptance. `rejecting` policy means that input messages that cannot be processed right away are not accepted by the node and it is responsibility of a predecessor to handle this.

#### 11.2.3.2.3.22 Lightweight

This policy helps to reduce the overhead associated with the execution scheduling of the node.

For functional nodes that have a default value for the `Policy` template parameter, specifying the `lightweight` policy results in extending the behavior of the default value of `Policy` with the behavior defined by the `lightweight` policy. For example, if the default value of `Policy` is `queueing`, specifying `lightweight` as the `Policy` value is equivalent to specifying `queueing_lightweight`.

#### 11.2.3.2.3.23 Example

The example below shows the application of the `lightweight` policy to a graph with a pipeline topology. It is reasonable to apply the `lightweight` policy to the second and third nodes because the bodies of these nodes are small. This allows the second and third nodes to execute without task scheduling overhead. The `lightweight` policy is not specified for the first node in order to permit concurrent invocations of the graph.

```

#include "tbb/flow_graph.h"

int main() {
    using namespace tbb::flow;

    graph g;

    function_node< int, int > add( g, unlimited, [](const int &v) {
        return v+1;
    } );
    function_node< int, int, lightweight > multiply( g, unlimited, [](const int &v) {
        return v*2;
    } );
    function_node< int, int, lightweight > cube( g, unlimited, [](const int &v) {
        return v*v*v;
    } );
}

```

(continues on next page)

(continued from previous page)

```

    return v*v*v;
} );

make_edge(add, multiply);
make_edge(multiply, cube);

for(int i = 1; i <= 10; ++i)
    add.try_put(i);
g.wait_for_all();

return 0;
}

```

#### 11.2.3.2.3.24 Nodes Priorities

##### [flow\_graph.node\_priorities]

Flow graph provides interface for setting relative priorities at construction of flow graph functional nodes, guiding threads that execute the graph to prefer nodes with higher priority.

```

namespace tbb {
namespace flow {

    typedef unsigned int node_priority_t;

    const node_priority_t no_priority = node_priority_t(0);

} // namespace flow
} // namespace tbb

```

`function_node`, `multipfunction_node`, `async_node` and `continue_node` has a constructor with parameter of `node_priority_t` type, which sets the node priority in the graph: the larger the specified value for the parameter, the higher the priority. The special constant value `no_priority`, also the default value of the parameter, switches priority off for a particular node.

For a particular graph, tasks to execute the nodes whose priority is specified have precedence over tasks for the nodes with lower or no priority value set. When looking for a task to execute, a thread will choose the one with the highest priority from those in the graph which are available for execution.

#### 11.2.3.2.3.25 Example

The following basic example demonstrates prioritization of one path in the graph over the other, which may help to improve overall performance of the graph.

Consider executing the graph from the picture above using two threads. Let the nodes `f1` and `f3` take equal time to execute, while the node `f2` takes longer. That makes the nodes `b5`, `f2` and `f6` constitute the critical path in this graph. Due to the non-deterministic behavior in selection of the tasks, oneTBB might execute nodes `f1` and `f3` in parallel first, which would make the whole graph execution time last longer than the case when one of the threads chooses the node `f2` just after the broadcast node. By setting a higher priority on node `f2`, threads are guided to take the critical path task earlier, thus reducing overall execution time.

```

#include <iostream>
#include <cmath>

```

(continues on next page)

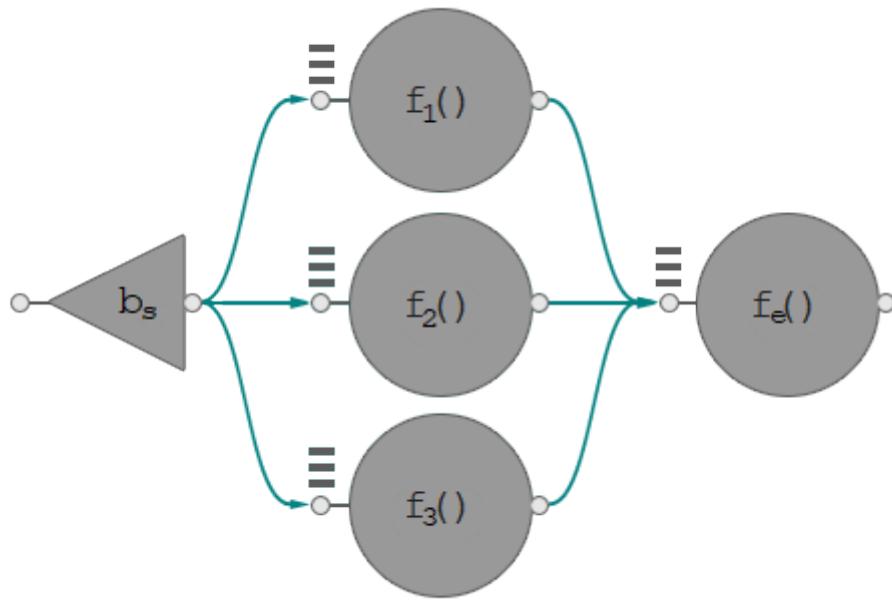


Fig. 1: Data flow graph with critical path.

(continued from previous page)

```
#include "tbb/tick_count.h"
#include "tbb/global_control.h"

#include "tbb/flow_graph.h"

void spin_for( double delta_seconds ) {
    tbb::tick_count start = tbb::tick_count::now();
    while( (tbb::tick_count::now() - start).seconds() < delta_seconds ) ;
}

static const double unit_of_time = 0.1;

struct Body {
    unsigned factor;
    Body( unsigned times ) : factor( times ) {}
    void operator()( const tbb::flow::continue_msg& ) {
        // body execution takes 'factor' units of time
        spin_for( factor * unit_of_time );
    }
};

int main() {
    using namespace tbb::flow;

    const int max_threads = 2;
    tbb::global_control control(tbb::global_control::max_allowed_parallelism, max_
    threads);

    graph g;
```

(continues on next page)

(continued from previous page)

```

broadcast_node<continue_msg> bs(g);

continue_node<continue_msg> f1(g, Body(5));

// f2 is a heavy one and takes the most execution time as compared to the other
// nodes in the
// graph. Therefore, let the graph start this node as soon as possible by
// prioritizing it over
// the other nodes.
continue_node<continue_msg> f2(g, Body(10), node_priority_t(1));

continue_node<continue_msg> f3(g, Body(5));

continue_node<continue_msg> fe(g, Body(7));

make_edge( bs, f1 );
make_edge( bs, f2 );
make_edge( bs, f3 );

make_edge( f1, fe );
make_edge( f2, fe );
make_edge( f3, fe );

tbb::tick_count start = tbb::tick_count::now();

bs.try_put( continue_msg() );
g.wait_for_all();

double elapsed = std::floor((tbb::tick_count::now() - start).seconds() / unit_of_
➥time);

std::cout << "Elapsed approximately " << elapsed << " units of time" << std::endl;

return 0;
}

```

### 11.2.3.2.3.26 Predefined Concurrency Limits

#### [flow\_graph.concurrency\_limits]

Predefined constants that may be used as `function_node`, `multifunction_node` and `async_node` constructors arguments to define concurrency limit.

```

// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    std::size_t unlimited = /*implementation-defined*/;
    std::size_t serial = /*implementation-defined*/;

} // namespace flow
} // namespace tbb

```

unlimited concurrency allows an unlimited number of invocations of the body to execute concurrently.

serial concurrency allows only a single call of body to execute concurrently.

### 11.2.3.2.3.27 copy\_body

#### [flow\_graph.copy\_body]

copy\_body is a function template that returns a copy of the body function object from the following nodes:

- *continue\_node*
- *function\_node*
- *multipfunction\_node*
- *input\_node*

```
namespace tbb {
namespace flow {

    // Defined in header <tbb/flow_graph.h>

    template< typename Body, typename Node >
    Body copy_body( Node &n );

} // namespace flow
} // namespace tbb
```

### 11.2.3.2.4 Buffering Nodes

Buffering nodes are designed to accumulate input messages and pass them to successors in a predefined order, depending on the node type.

#### 11.2.3.2.4.1 overwrite\_node

##### [flow\_graph.overwrite\_node]

A node that is a buffer of a single item that can be over-written.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template<typename T>
    class overwrite_node : public graph_node, public receiver<T>, public sender<T> {
public:
    explicit overwrite_node( graph &g );
    overwrite_node( const overwrite_node &src );
    ~overwrite_node();

    bool try_put( const T &v );
    bool try_get( T &v );

    bool is_valid();
    void clear();
}
```

(continues on next page)

(continued from previous page)

```

};

} // namespace flow
} // namespace tbb

```

**Requirements:**

- The type `T` shall meet the *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.

This type of node buffers a single item of type `T`. The value is initially invalid. Gets from the node are non-destructive.

`overwrite_node` is a `graph_node`, `receiver<T>` and `sender<T>`.

`overwrite_node` has a *buffering* and *broadcast-push properties*.

`overwrite_node` allows overwriting its single item buffer.

**11.2.3.2.4.2 Member functions****explicit overwrite\_node (graph &g)**

Constructs an object of type `overwrite_node` that belongs to the graph `g` with an invalid internal buffer item.

**overwrite\_node (const overwrite\_node &src)**

Constructs an object of type `overwrite_node` that belongs to the graph `g` with an invalid internal buffer item. The buffered value and list of successors are not copied from `src`.

**~overwrite\_node ()**

Destroys the `overwrite_node`.

**bool try\_put (const T &v)**

Stores `v` in the internal single item buffer and calls `try_put (v)` on all successors.

**Returns:** `true`

**bool try\_get (T &v)**

If the internal buffer is valid, assigns the value to `v`.

**Returns:** `true` if `v` is assigned to. `false` if `v` is not assigned to.

**bool is\_valid()**

**Returns:** `true` if the buffer holds a valid value, otherwise returns `false`.

**void clear ()**

Invalidates the value held in the buffer.

**11.2.3.2.4.3 Examples**

The example demonstrates `overwrite_node` as a single-value storage that might be updated. Data can be accessed with direct `try_get ()` call.

```

#include "tbb/flow_graph.h"

int main() {
    const int data_limit = 20;
    int count = 0;

```

(continues on next page)

(continued from previous page)

```

tbb::flow::graph g;

tbb::flow::function_node< int, int > data_set_preparation(g,
    tbb::flow::unlimited, []( int data ) {
        printf("Prepare large data set and keep it inside node storage\n");
        return data;
    });
}

tbb::flow::overwrite_node< int > overwrite_storage(g);

tbb::flow::source_node<int> data_generator(g,
    [&]( int& v ) -> bool {
        if ( count < data_limit ) {
            ++count;
            v = count;
            return true;
        } else {
            return false;
        }
    });
}

tbb::flow::function_node< int > process(g, tbb::flow::unlimited,
    [&]( const int& data) {
        int data_from_storage = 0;
        overwrite_storage.try_get(data_from_storage);
        printf("Data from a storage: %d\n", data_from_storage);
        printf("Data to process: %d\n", data);
    });
}

tbb::flow::make_edge(data_set_preparation, overwrite_storage);
tbb::flow::make_edge(data_generator, process);

data_set_preparation.try_put(1);
data_generator.activate();

g.wait_for_all();

return 0;
}

```

overwrite\_node supports reserving join\_node as its successor. See example in *the example section of write\_once\_node*.

#### 11.2.3.2.4.4 write\_once\_node

##### [flow\_graph.write\_once\_node]

A node that is a buffer of a single item that cannot be over-written.

```

// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

```

(continues on next page)

(continued from previous page)

```

template< typename T >
class write_once_node : public graph_node, public receiver<T>, public sender<T> {
public:
    explicit write_once_node( graph &g );
    write_once_node( const write_once_node &src );
    ~write_once_node();

    bool try_put( const T &v );
    bool try_get( T &v );

    bool is_valid();
    void clear();
};

} // namespace flow
} // namespace tbb

```

#### Requirements:

- The `T` type shall meet the *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.

This type of node buffers a single item of type `T`. The value is initially invalid. Gets from the node are non-destructive.

`write_once_node` is a `graph_node`, `receiver<T>` and `sender<T>`.

`write_once_node` has a *buffering* and *broadcast-push properties*.

`write_once_node` does not allow overwriting its single item buffer.

#### 11.2.3.2.4.5 Member functions

**explicit write\_once\_node(graph &g)**

Constructs an object of type `write_once_node` that belongs to the graph `g`, with an invalid internal buffer item.

**write\_once\_node(const write\_once\_node &src)**

Constructs an object of type `write_once_node` with an invalid internal buffer item. The buffered value and list of successors is not copied from `src`.

**~write\_once\_node()**

Destroys the `write_once_node`.

**bool try\_put(const T &v)**

Stores `v` in the internal single item buffer if it does not contain a valid value already. If a new value is set, the node broadcast it to all successors.

**Returns:** `true` for the first time after construction or a call to `clear()`, `false` otherwise.

**bool try\_get(T &v)**

If the internal buffer is valid, assigns the value to `v`.

**Returns:** `true` if `v` is assigned to. `false` if `v` is not assigned to.

**bool is\_valid()**

**Returns:** `true` if the buffer holds a valid value, otherwise returns `false`.

**void clear()**

Invalidates the value held in the buffer.

### 11.2.3.2.4.6 Example

Usage scenario is similar to `overwrite_node` but an internal buffer can be updated only after `clear()` call. The following example shows the possibility to connect the node to a reserving `join_node`, avoiding direct calls to the `try_get()` method from the body of the successor node.

```
#include "tbb/flow_graph.h"

typedef int data_type;

int main() {
    using namespace tbb::flow;

    graph g;

    function_node<data_type, data_type> static_result_computer_n(
        g, serial,
        [&](const data_type& msg) {
            // compute the result using incoming message and pass it further, e.g.:
            data_type result = data_type((msg << 2 + 3) / 4);
            return result;
        });
    write_once_node<data_type> write_once_n(g); // for buffering once computed value

    buffer_node<data_type> buffer_n(g);
    join_node<tuple<data_type, data_type>, reserving> join_n(g);

    function_node<tuple<data_type, data_type>> consumer_n(
        g, unlimited,
        [&](const tuple<data_type, data_type>& arg) {
            // use the precomputed static result along with dynamic data
            data_type precomputed_result = get<0>(arg);
            data_type dynamic_data = get<1>(arg);
        });
    make_edge(static_result_computer_n, write_once_n);
    make_edge(write_once_n, input_port<0>(join_n));
    make_edge(buffer_n, input_port<1>(join_n));
    make_edge(join_n, consumer_n);

    // do one-time calculation that will be reused many times further in the graph
    static_result_computer_n.try_put(1);

    for (int i = 0; i < 100; i++) {
        buffer_n.try_put(1);
    }

    g.wait_for_all();

    return 0;
}
```

### 11.2.3.2.4.7 buffer\_node

#### [flow\_graph.buffer\_node]

A node that is an unbounded buffer of messages. Messages are forwarded in arbitrary order.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

template< typename T>
class buffer_node : public graph_node, public receiver<T>, public sender<T> {
public:
    explicit buffer_node( graph &g );
    buffer_node( const buffer_node &src );
    ~buffer_node();

    bool try_put( const T &v );
    bool try_get( T &v );
};

} // namespace flow
} // namespace tbb
```

Requirements:

- The type *T* shall meet the *CopyConstructible* requirements from [copyconstructible] ISO C++ Standard sections.

*buffer\_node* is a *graph\_node*, *receiver*<*T*> and *sender*<*T*>.

*buffer\_node* has a *buffering* and *single-push properties*.

*buffer\_node* forwards messages in arbitrary order to a single successor in its successor set.

### 11.2.3.2.4.8 Member functions

#### explicit buffer\_node(*graph* &*g*)

Constructs an empty *buffer\_node* that belongs to the *graph* *g*.

#### explicit buffer\_node(*const buffer\_node* &*src*)

Constructs an empty *buffer\_node*. The buffered value and list of successors is not copied from *src*.

#### bool try\_put(*const T* &*v*)

Adds *v* to the buffer. If *v* is the only item in the buffer, a task is also spawned to forward the item to a successor.

**Returns:** true

#### bool try\_get(*T* &*v*)

**Returns:** true if an item can be removed from the buffer and assigned to *v*. Returns false if there is no non-reserved item currently in the buffer.

### 11.2.3.2.4.9 queue\_node

#### [flow\_graph.queue\_node]

A node that forwards messages in first-in first-out (FIFO) order by maintaining a buffer of messages.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

template <typename T >
class queue_node : public graph_node, public receiver<T>, public sender<T> {
public:
    explicit queue_node( graph &g );
    queue_node( const queue_node &src );
    ~queue_node();

    bool try_put( const T &v );
    bool try_get( T &v );
};

} // namespace flow
} // namespace tbb
```

Requirements:

- The type `T` shall meet the *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.

`queue_node` forwards messages in first-in first-out (FIFO) order to a single successor in its successor set.

`queue_node` is a `graph_node`, `receiver` and `sender`.

`queue_node` has a *buffering* and *single-push properties*.

### 11.2.3.2.4.10 Member functions

#### `explicit queue_node(graph &g)`

Constructs an empty `queue_node` that belongs to the graph `g`.

#### `queue_node(const queue_node &src)`

Constructs an empty `queue_node` that belongs to the same graph `g` as `src`. The list of predecessors, the list of successors and the messages in the buffer are not copied.

#### `bool try_put(const T &v)`

Add item to internal FIFO buffer. If `v` is the only item in the `queue_node`, a task is spawned to forward the item to a successor.

**Returns:** `true`.

#### `bool try_get(T &v)`

**Returns:** `true` if an item can be removed from the front of the `queue_node` and assigned to `v`. Returns `false` if there is no item currently in the `queue_node` or if the node is reserved.

### 11.2.3.2.4.11 Example

Usage scenario is similar to `buffer_node` except that messages are passed in first-in first-out (FIFO) order.

### 11.2.3.2.4.12 priority\_queue\_node

#### [flow\_graph.priority\_queue\_node]

A class template that forwards messages in priority order by maintaining a buffer of messages.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template< typename T, typename Compare = std::less<T>>
    class priority_queue_node : public graph_node, public receiver<T>, public sender
    <T> {
        public:
            typedef size_t size_type;
            explicit priority_queue_node( graph &g );
            priority_queue_node( const priority_queue_node &src );
            ~priority_queue_node();

            bool try_put( const T &v );
            bool try_get( T &v );
    };

} // namespace flow
} // namespace tbb
```

Requirements:

- The type `T` shall meet the *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.
- The type `Compare` shall meet the *Compare* type requirements from [alg.sorting] ISO C++ Standard section. If `Compare` instance throws an exception, then behavior is undefined.

The next message to be forwarded has the largest priority as determined by `Compare` template argument.

`priority_queue_node` is a `graph_node`, `receiver<T>` and `sender<T>`.

`priority_queue_node` has a *buffering* and *single-push properties*.

### 11.2.3.2.4.13 Member functions

#### `explicit priority_queue_node(graph &g)`

Constructs an empty `priority_queue_node` that belongs to the graph `g`.

#### `priority_queue_node(const priority_queue_node &src)`

Constructs an empty `priority_queue_node` that belongs to the same graph `g` as `src`. The list of predecessors, the list of successors and the messages in the buffer are not copied.

#### `bool try_put(const T &v)`

Adds `v` to the `priority_queue_node`. If `v`'s priority is the largest of all of the currently buffered messages, a task is spawned to forward the item to a successor.

**Returns:** true

bool **try\_put** (T &v)

**Returns:** true if a message is available in the node and the node is not currently reserved. Otherwise returns false. If the node returns true, the message with the largest priority will have been copied to v.

#### 11.2.3.2.4.14 Example

Usage scenario is similar to *sequencer\_node* except that the *priority\_queue\_node* provides local order, passing the message with highest priority of all stored at the moment, while *sequencer\_node* enforces global order and does not allow a “smaller priority” message to pass through before all its preceding messages.

#### 11.2.3.2.4.15 sequencer\_node

##### [flow\_graph.sequencer\_node]

A node that forwards messages in sequence order by maintaining an internal buffer.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

template< typename T >
class sequencer_node : public graph_node, public receiver<T>, public sender<T> {
public:
    template< typename Sequencer >
    sequencer_node( graph &g, const Sequencer &s );
    sequencer_node( const sequencer_node &src );

    bool try_put( const T &v );
    bool try_get( output_type &v );
};

} // namespace flow
} // namespace tbb
```

Requirements:

- The type T shall meet the *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.
- The type Sequencer shall meet the *Sequencer requirements*. If Sequencer instance throws an exception, then behavior is undefined.

*sequencer\_node* forwards messages in sequence order to a single successor in its successor set.

*sequencer\_node* is a *graph\_node*, *receiver*<T> and *sender*<T>.

Each item that passes through a *sequencer\_node* is ordered by its sequencer order number. These sequence order numbers range from 0 to the largest integer representable by the std::size\_t type. An item’s sequencer order number is determined by passing the item to a user-provided Sequencer function object.

---

**Note:** The *sequencer\_node* rejects duplicate sequencer numbers.

---

#### 11.2.3.2.4.16 Member functions

```
template<typename Sequencer>
sequencer_node(graph &g, const Sequencer &s)
    Constructs an empty sequencer_node that belongs to the graph g and uses s to compute sequence numbers for items.

sequencer_node(const sequencer_node &src)
    Constructs an empty sequencer_node that belongs to the same graph g as src and will use a copy of the Sequencer s used to construct src. The list of predecessors, the list of successors and the messages in the buffer are not copied.
```

**Caution:** The new sequencer object is copy-constructed from a copy of the original sequencer object provided to src at its construction. Therefore changes made to member variables in src's object will not affect the sequencer of the new sequencer\_node.

```
bool try_put(const T &v)
    Adds v to the sequencer_node. If v's sequence number is the next item in the sequence, a task is spawned to forward the item to a successor.

Returns: true

bool try_get(T &v)
    Returns: true if the next item in the sequence is available in the sequencer_node. If so, it is removed from the node and assigned to v. Returns false if the next item in sequencer order is not available or if the node is reserved.
```

#### 11.2.3.2.4.17 Deduction Guides

```
template <typename Body>
sequencer_node(graph&, Body) -> input_node<std::decay_t<input_t<Body>>>;
```

Where:

- input\_t is an alias to Body input argument type.

#### 11.2.3.2.4.18 Example

The example demonstrates ordering capabilities of the sequencer\_node. While being processed in parallel, the data are passed to the successor node in the exact same order they were read.

```
#include "tbb/flow_graph.h"

struct Message {
    int id;
    int data;
};

int main() {
    tbb::flow::graph g;

    // Due to parallelism the node can push messages to its successors in any order
    (continues on next page)
```

(continued from previous page)

```

tbb::flow::function_node< Message, Message > process(g, tbb::flow::unlimited, []_r
    ↪(Message msg) -> Message {
        msg.data++;
        return msg;
    });

tbb::flow::sequencer_node< Message > ordering(g, [](const Message& msg) -> int {
    return msg.id;
});

tbb::flow::function_node< Message > writer(g, tbb::flow::serial, [] (const_
    ↪Message& msg) {
    printf("Message received with id: %d\n", msg.id);
});

tbb::flow::make_edge(process, ordering);
tbb::flow::make_edge(ordering, writer);

for (int i = 0; i < 100; ++i) {
    Message msg = { i, 0 };
    process.try_put(msg);
}

g.wait_for_all();
}

```

### 11.2.3.2.5 Service Nodes

These nodes are designed for advanced control of the message flow, such as combining messages from different paths in a graph or limiting the number of simultaneously processed messages, as well as for creating reusable custom nodes.

#### 11.2.3.2.5.1 limiter\_node

##### [flow\_graph.limiter\_node]

A node that counts and limits the number of messages that pass through it.

```

// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template< typename T, typename DecrementType=continue_msg >
    class limiter_node : public graph_node, public receiver<T>, public sender<T> {
        public:
            limiter_node( graph &g, size_t threshold );
            limiter_node( const limiter_node &src );

            InternalReceiverType<DecrementType> decrement;

            bool try_put( const T &v );
            bool try_get( T &v );
    };
}

```

(continues on next page)

(continued from previous page)

```
} // namespace flow
} // namespace tbb
```

**Requirements:**

- T type should meet the *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard section.
- The DecrementType type shall be an integral type or continue\_msg.

limiter\_node is a graph\_node, receiver<T> and sender<T>

limiter\_node has a *discarding* and *broadcast-push properties*.

It does not accept new messages once its user-specified threshold is reached. The internal count of broadcasts is adjusted through use of its embedded decrement object, and its values are truncated to be inside [0, threshold] interval.

The template parameter DecrementType specifies the type of the message that can be sent to the member object decrement. This template parameter defined to continue\_msg the default. If an integral type is specified, positive values sent to decrement port determine the value on which the internal counter of broadcasts will be decreased, while negative values determine the value on which the internal counter of broadcasts will be increased.

The continue\_msg sent to the member object decrement decreases the internal counter of broadcasts by one.

When try\_put call on the member object decrement results in the new value of internal counter of broadcasts to be less than the threshold, the limiter\_node will try to get a message from one of its known predecessors and forward that message to all of its successors. If it cannot obtain a message from a predecessor, it will decrement a counter of broadcasts.

### 11.2.3.2.5.2 Member functions

**limiter\_node(graph &g, size\_t threshold)**

Constructs a limiter\_node that allows up to threshold items to pass through before rejecting try\_put's.

**limiter\_node(const limiter\_node &src)**

Constructs a limiter\_node that has the same initial state that src had at its construction. The new limiter\_node will belong to the same graph g as src, have the same threshold. The list of predecessors, the list of successors, and the current count of broadcasts are not copied from src.

**bool try\_put(const T &v)**

If the broadcast count is below the threshold, v is broadcast to all successors.

**Returns:** true if v is broadcast. false if v is not broadcast because the threshold has been reached.

**bool try\_get(T &v)**

**Returns:** false.

### 11.2.3.2.5.3 broadcast\_node

#### [flow\_graph.broadcast\_node]

A node that broadcasts incoming messages to all of its successors.

```
// Defined in header <tbb/flow_graph.h>
namespace tbb {
namespace flow {

template< typename T >
class broadcast_node :
public graph_node, public receiver<T>, public sender<T> {
public:
    explicit broadcast_node( graph &g );
    broadcast_node( const broadcast_node &src );

    bool try_put( const T &v );
    bool try_get( T &v );
};

} // namespace flow
} // namespace tbb
```

Requirements:

- T type should meet the *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard section.

`broadcast_node` is a `graph_node`, `receiver<T>` and `sender<T>`.

`broadcast_node` has a *discarding* and *broadcast-push properties*.

All messages are forwarded immediately to all successors.

### 11.2.3.2.5.4 Member functions

#### `explicit broadcast_node(graph &g)`

Constructs an object of type `broadcast_node` that belongs to the graph `g`.

#### `broadcast_node(const broadcast_node &src)`

Constructs an object of type `broadcast_node` that belongs to the same graph `g` as `src`. The list of predecessors, the list of successors and the messages in the buffer are not copied.

#### `bool try_put(const input_type &v)`

Adds `v` to all successors.

**Returns:** always returns `true`, even if it was unable to successfully forward the message to any of its successors.

#### `bool try_get(output_type &v)`

If the internal buffer is valid, assigns the value to `v`.

**Returns:** `true` if `v` is assigned to. `false` if `v` is not assigned to.

### 11.2.3.2.5.5 join\_node

#### [flow\_graph.join\_node]

A node that creates a tuple from a set of messages received at its input ports and broadcasts the tuple to all of its successors.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {
    using tag_value = /*implementation-specific*/;

    template<typename OutputTuple, class JoinPolicy = /*implementation-defined*/>
    class join_node : public graph_node, public sender<OutputTuple> {
public:
    using input_ports_type = /*implementation-defined*/;

    explicit join_node( graph &g );
    join_node( const join_node &src );

    input_ports_type &input_ports( );

    bool try_get( OutputTuple &v );
};

    template<typename OutputTuple, typename K, class KHash=tbb_hash_compare<K> >
    class join_node< OutputTuple, key_matching<K,KHash> > : public graph_node, public
    ↵sender< OutputTuple > {
public:
    using input_ports_type = /*implementation-defined*/;

    explicit join_node( graph &g );
    join_node( const join_node &src );

    template<typename B0, typename B1>
    join_node( graph &g, B0 b0, B1 b1 );
    template<typename B0, typename B1, typename B2>
    join_node( graph &g, B0 b0, B1 b1, B2 b2 );
    template<typename B0, typename B1, typename B2, typename B3>
    join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3 );
    template<typename B0, typename B1, typename B2, typename B3, typename B4>
    join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4 );
    template<typename B0, typename B1, typename B2, typename B3, typename B5>
    join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5 );
    template<typename B0, typename B1, typename B2, typename B3, typename B5, ↵
    ↵typename B6>
    join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6 );
    template<typename B0, typename B1, typename B2, typename B3, typename B5, ↵
    ↵typename B6, typename B6>
    join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6, B7 b7 );
    template<typename B0, typename B1, typename B2, typename B3, typename B5, ↵
    ↵typename B6, typename B7>
    join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6, B7 b7 );
    template<typename B0, typename B1, typename B2, typename B3, typename B5, ↵
    ↵typename B6, typename B7, typename B8>
    join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6, B7 b7,
    ↵B8 b8 );
};
```

(continues on next page)

(continued from previous page)

```

    template<typename B0, typename B1, typename B2, typename B3, typename B5,>
    ~typename B6, typename B7, typename B8, typename B9>
    join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6, B7 b7,>
    ~B8 b8, B9 b9 );

    input_ports_type &input_ports( );
    bool try_get( OutputTuple &v );
};

} // namespace flow
} // namespace tbb

```

Requirements:

- The type `OutputTuple` must be an instantiation of `std::tuple`. Each type that the tuple stores shall meet the *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copy assignable] ISO C++ Standard sections.
- The `JoinPolicy` type should be specified as one of *buffering policies* for `join_node`.
- The `KHash` type shall meet the *HashCompare requirements*.
- The `Bi` types shall meet the *JoinNodeFunctionObject requirements*.

A `join_node` is a `graph_node` and a `sender<OutputTuple>`. It contains a tuple of input ports, each of which is a `receiver<Type>` for each `Type` in `OutputTuple`. It supports multiple input receivers with distinct types and broadcasts a tuple of received messages to all of its successors. All input ports of a `join_node` must use the same buffering policy.

The behavior of a `join_node` is based on its buffering policy.

#### 11.2.3.2.5.6 join\_node Policies

##### [flow\_graph.join\_node\_policies]

`join_node` supports three buffering policies at its input ports: `reserving`, `queueing` and `key_matching`.

```

// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    struct reserving;
    struct queueing;
    template<typename K, class KHash=tbb_hash_compare<K> > struct key_matching;
    using tag_matching = key_matching<tag_value>;

} // namespace flow
} // namespace tbb

```

- `queueing` - As each input port is put to, the incoming message is added to an unbounded first-in first-out queue in the port. When there is at least one message at each input port, the `join_node` broadcasts a tuple containing the head of each queue to all successors. If at least one successor accepts the tuple, the head of each input port's queue is removed; otherwise, the messages remain in their respective input port queues.
- `reserving` - As each input port is put to, the `join_node` marks that an input may be available at that port and returns `false`. When all ports have been marked as possibly available, the `join_node` will try to reserve

a message at each port from their known predecessors. If it is unable to reserve a message at a port, it unmarks that port, and releases all previously acquired reservations. If it is able to reserve a message at all ports, it broadcasts a tuple containing these messages to all successors. If at least one successor accepts the tuple, the reservations are consumed; otherwise, they are released.

- `key_matching<typename K, class KHash=tbb_hash_compare<K>>` - As each input port is put to, a user-provided function object is applied to the message to obtain its key. The message is then added to a hash table of the input port. When there is a message at each input port for a given key, the `join_node` removes all matching messages from the input ports, constructs a tuple containing the matching messages and attempts to broadcast it to all successors. If no successor accepts the tuple, it is saved and will be forwarded on a subsequent `try_get`.
- `tag_matching` - A specialization of `key_matching` that accepts keys of type `tag_value`.

The function template `input_port` simplifies the syntax for getting a reference to a specific input port.

`join_node` has a *buffering* and *broadcast-push properties*.

#### 11.2.3.2.5.7 Member types

`input_ports_type` is an alias to a tuple of input ports.

#### 11.2.3.2.5.8 Member functions

```
explicit join_node( graph &g );
```

Constructs an empty `join_node` that belongs to the graph `g`.

```
template<typename B0, typename B1>
join_node( graph &g, B0 b0, B1 b1 );
template<typename B0, typename B1, typename B2>
join_node( graph &g, B0 b0, B1 b1, B2 b2 );
template<typename B0, typename B1, , typename B2, typename B3>
join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3 );
template<typename B0, typename B1, , typename B2, typename B3, typename B4>
join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4 );
template<typename B0, typename B1, , typename B2, typename B3, typename B4, typename B5>
join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5 );
template<typename B0, typename B1, , typename B2, typename B3, typename B4, typename B5, typename B6>
join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6 );
template<typename B0, typename B1, , typename B2, typename B3, typename B4, typename B5, typename B6, typename B7>
join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6, B7 b7 );
template<typename B0, typename B1, , typename B2, typename B3, typename B4, typename B5, typename B6, typename B7, typename B8>
join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6, B7 b7 );
template<typename B0, typename B1, , typename B2, typename B3, typename B4, typename B5, typename B6, typename B7, typename B8, typename B9>
join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6, B7 b7, B8 b8 );
join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6, B7 b7, B8 b8 );
join_node( graph &g, B0 b0, B1 b1, B2 b2, B3 b3, B4 b4, B5 b5, B6 b6, B7 b7, B8 b8, B9 b9 );
```

A constructor only available in the `key_matching` specialization of `join_node`.

Creates a `join_node` that uses the function objects `b0, b1, ..., bN` to determine that tags for the input ports 0 through N.

**Caution:** Function objects passed to the `join_node` constructor must not throw. They are called in parallel, and should be pure, take minimal time and be non-blocking.

```
join_node( const join_node &src )
```

Creates a `join_node` that has the same initial state that `src` had at its construction. The list of predecessors, messages in the input ports, and successors are not copied.

```
input_ports_type &input_ports( )
```

**Returns:** a `std::tuple` of receivers. Each element inherits from `receiver<T>` where T is the type of message expected at that input. Each tuple element can be used like any other `receiver<T>`. The behavior of the ports based on the selected `join_node` policy.

```
bool try_get( output_type &v )
```

Attempts to generate a tuple based on the buffering policy of the `join_node`.

If it can successfully generate a tuple, it copies it to `v` and returns `true`. Otherwise it returns `false`.

### 11.2.3.2.5.9 Non-Member Types

```
using tag_value = /*implementation-specific*/;
```

`tag_value` is an unsigned integral type for defining `tag_matching` policy.

### 11.2.3.2.5.10 Deduction Guides

```
template <typename Body, typename... Bodies>
join_node(graph&, Body, Bodies...)
    ->join_node<std::tuple<std::decay_t<input_t<Body>>, std::decay_t<input_t<Bodies>>,
    ...>, key_matching<output_t<Body>>>;
```

Where:

- `input_t` is an alias to the input argument type of the passed function object.
- `output_t` is an alias to the return type of the passed function object.

### 11.2.3.2.5.11 split\_node

#### [flow\_graph.split\_node]

A split\_node sends each element of the incoming tuple to the output port that matches the element's index in the incoming tuple.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

template < typename TupleType >
class split_node : public graph_node, public receiver<TupleType> {
public:
    explicit split_node( graph &g );
    split_node( const split_node &other );
    ~split_node();

    bool try_put( const TupleType &v );

    using output_ports_type = /*implementation-defined*/ ;
    output_ports_type& output_ports();
};

} // namespace flow
} // namespace tbb
```

Requirements:

- The type TupleType must be an instantiation of `std::tuple`. Each type that the tuple stores shall meet the *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copy assignable] ISO C++ Standard sections.

`split_node` is a `receiver<TupleType>` and has a tuple of `sender` output ports; Each of output ports is specified by corresponding tuple element type. This node receives a tuple at its single input port and generates a message from each element of the tuple, passing each to them corresponding output port.

`split_node` has a *discarding* and *broadcast-push properties*.

`split_node` has unlimited concurrency, and behaves as a `broadcast_node` with multiple output ports.

### 11.2.3.2.5.12 Member functions

#### `explicit split_node(graph &g)`

Constructs a `split_node` registered with `graph g`.

#### `split_node(const split_node &other)`

Constructs a `split_node` that has the same initial state that `other` had when it was constructed. The `split_node` that is constructed will have a reference to the same `graph` object as `other`. The predecessors and successors of `other` will not be copied.

#### `~split_node()`

Destructor

#### `bool try_put(const TupleType &v)`

Broadcasts each element of the incoming tuple to the nodes connected to the `split_node`'s output ports. The element at index `i` of `v` will be broadcast through the `ith` output port.

**Returns:** true

`output_ports_type &output_ports()`  
**Returns:** a tuple of output ports.

### 11.2.3.2.5.13 indexer\_node

[`flow_graph.indexer_node`]

`indexer_node` broadcasts messages received at its input ports to all of its successors. The messages are broadcast individually as they are received at each port. The output is a *tagged message* that contains a tag and a value; the tag identifies the input port on which the message was received.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template<typename T0, typename... TN>
    class indexer_node : public graph_node, public sender</*implementation_defined*/>
{
public:
    indexer_node(graph &g);
    indexer_node(const indexer_node &src);

    using input_ports_type = /*implementation_defined*/;
    input_ports_type &input_ports();

    using output_type = tagged_msg<size_t, T0, TN...>;
    bool try_get( output_type &v );
};

} // namespace flow
} // namespace tbb
```

Requirements:

- The `T` type and all types in `TN` template parameter pack shall meet the *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.

An `indexer_node` is a `graph_node` and `sender<tagged_msg<size_t, T0, TN...>>`. It contains a tuple of input ports, each of which is a receiver specified by corresponding input template parameter pack element. It supports multiple input receivers with distinct types and broadcasts each received message to all of its successors. Unlike a `join_node`, each message is broadcast individually to all successors of the `indexer_node` as it arrives at an input port. Before broadcasting, a message is tagged with the index of the port on which the message arrived.

`indexer_node` has a *discarding* and *broadcast-push properties*.

The function template `input_port` simplifies the syntax for getting a reference to a specific input port.

### 11.2.3.2.5.14 Member types

- `input_ports_type` is an alias to a tuple of input ports.
- `output_type` is an alias to the message of type `tagged_msg`, which is sent to successors.

### 11.2.3.2.5.15 Member functions

`indexer_node(graph &g)`

Constructs an `indexer_node` that belongs to the graph `g`.

`indexer_node(const indexer_node &src)`

Constructs an `indexer_node`. The list of predecessors, messages in the input ports, and successors are not copied.

`input_ports_type &input_ports()`

**Returns:** A `std::tuple` of receivers. Each element inherits from `receiver<T>` where `T` is the type of message expected at that input. Each tuple element can be used like any other `receiver<T>`.

`bool try_get(output_type &v)`

An `indexer_node` contains no buffering and therefore does not support gets.

**Returns:** `false`.

See also:

- `input_port function template`
- `tagged_msg template class`

### 11.2.3.2.5.16 composite\_node

[`flow_graph.composite_node`]

A node that encapsulates a collection of other nodes as a first class graph node.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

template< typename InputTuple, typename OutputTuple > class composite_node;

// composite_node with both input ports and output ports
template< typename... InputTypes, typename... OutputTypes>
class composite_node <std::tuple<InputTypes...>, std::tuple<OutputTypes...> > : public graph_node {
public:
    typedef std::tuple< receiver<InputTypes>&... > input_ports_type;
    typedef std::tuple< sender<OutputTypes>&... > output_ports_type;

    composite_node( graph &g );
    virtual ~composite_node();

    void set_external_ports(input_ports_type&& input_ports_tuple, output_ports_
    type&& output_ports_tuple);
    input_ports_type& input_ports();
}
}}
```

(continues on next page)

(continued from previous page)

```

        output_ports_type& output_ports();
    };

    // composite_node with only input ports
    template< typename... InputTypes>
    class composite_node <std::tuple<InputTypes...>, std::tuple<> > : public graph_
    ↵node{
    public:
        typedef std::tuple< receiver<InputTypes>&... > input_ports_type;

        composite_node( graph &g );
        virtual ~composite_node();

        void set_external_ports(input_ports_type&& input_ports_tuple);
        input_ports_type& input_ports();
    };

    // composite_nodes with only output_ports
    template<typename... OutputTypes>
    class composite_node <std::tuple<>, std::tuple<OutputTypes...> > : public graph_
    ↵node{
    public:
        typedef std::tuple< sender<OutputTypes>&... > output_ports_type;

        composite_node( graph &g );
        virtual ~composite_node();

        void set_external_ports(output_ports_type&& output_ports_tuple);
        output_ports_type& output_ports();
    };

} // namespace flow
} // namespace tbb

```

- The `InputTuple` and `OutputTuple` must be instantiations of `std::tuple`. Each type that these tuples stores shall meet the *CopyConstructible* requirements from [copyconstructible] and *CopyAssignable* requirements from [copyassignable] ISO C++ Standard sections.

`composite_node` is a `graph_node`, `receiver<T>` and `sender<T>`.

The `composite_node` can package any number of other nodes. It maintains input and output port references to nodes in the package that border the `composite_node`. This allows for the references to be used to make edges to other nodes outside of the `composite_node`. The `InputTuple` is a tuple of input types. The `composite_node` has an input port for each type in `InputTuple`. Likewise, the `OutputTuple` is a tuple of output types. The `composite_node` has an output port for each type in `OutputTuple`.

The `composite_node` is a multi-port node with three specializations.

- **A multi-port node with multi-input ports and multi-output ports:** This specialization has a tuple of input ports, each of which is a `receiver` of a type in `InputTuple`. Each input port is a reference to a port of a node that the `composite_node` encapsulates. Similarly, this specialization also has a tuple of output ports, each of which is a `sender` of a type in `OutputTuple`. Each output port is a reference to a port of a node that the `composite_node` encapsulates.
- **A multi-port node with only input ports and no output ports:** This specialization only has a tuple of input ports.
- **A multi-port node with only output ports and no input\_ports:** This specialization only has a tuple of output

ports.

The function template `input_port` can be used to get a reference to a specific input port and the function template `output_port` can be used to get a reference to a specific output port.

Construction of a `composite_node` is done in two stages:

- Defining the `composite_node` with specification of `InputTuple` and `OutputTuple`.
- Making aliases from the encapsulated nodes that border the `composite_node` to the input and output ports of the `composite_node`. This step is mandatory as without it the `composite_node`'s input and output ports would not have been bound to any actual nodes. Making the aliases is achieved by calling the method `set_external_ports`.

The `composite_node` does not meet the *CopyConstructible* requirements from [copyconstructible] ISO C++ Standard section.

#### 11.2.3.2.5.17 Member functions

`composite_node (graph &g)`

Constructs a `composite_node` that belongs to the graph `g`.

`void set_external_ports (input_ports_type &&input_ports_tuple, output_ports_type &&output_ports_tuple)`

Creates input and output ports of the `composite_node` as aliases to the ports referenced by `input_ports_tuple` and `output_ports_tuple` respectively. That is, a port referenced at position `N` in `input_ports_tuple` is mapped as the `N`th input port of the `composite_node`, similarly for output ports.

`input_ports_type &input_ports ()`

**Returns:** A `std::tuple` of receivers. Each element is a reference to the actual node or input port that was aliased to that position in `set_external_ports ()`.

**Caution:** Calling `input_ports ()` without a prior call to `set_external_ports ()` results in undefined behavior.

`output_ports_type &output_ports ()`

**Returns:** A `std::tuple` of senders. Each element is a reference to the actual node or output port that was aliased to that position in `set_external_ports ()`.

**Caution:** Calling `output_ports ()` without a prior call to `set_external_ports ()` results in undefined behavior.

See also:

- *input\_port function template*
- *output\_port function template*

### 11.2.3.3 Ports and Edges

Flow Graph provides an API to manage connections between the nodes. For nodes that have more than one input or output port (ex. `join_node`), making a connection requires to specify a certain port by using special helper functions.

#### 11.2.3.3.1 `input_port`

##### [`flow_graph.input_port`]

A template function that returns a reference to a specific input port of a given `join_node`, `indexer_node` or `composite_node`.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template<size_t N, typename NodeType>
    /*implementation-defined*/& input_port(NodeType &n);

} // namespace flow
} // namespace tbb
```

See also:

- *join\_node template class*
- *indexer\_node template class*
- *composite\_node template class*

#### 11.2.3.3.2 `output_port`

##### [`flow_graph.output_port`]

A template function that returns a reference to a specific output port of a given `join_node`, `indexer_node` or `composite_node`.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template<size_t N, typename NodeType>
    /*implementation-defined*/& output_port(NodeType &n);

} // namespace flow
} // namespace tbb
```

See also:

- *split\_node Template Class*
- *multipfunction\_node Template Class*
- *composite\_node Template Class*

### 11.2.3.3.3 make\_edge

#### [flow\_graph.make\_edge]

A function template for building edges between nodes.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template<typename Message>
    inline void make_edge( sender<Message> &p, receiver<Message> &s );

    template< typename MultiOutputNode, typename MultiInputNode >
    inline void make_edge( MultiOutputNode& output, MultiInputNode& input );

    template<typename MultiOutputNode, typename Message>
    inline void make_edge( MultiOutputNode& output, receiver<Message> input );

    template<typename Message, typename MultiInputNode>
    inline void make_edge( sender<Message> output, MultiInputNode& input );

} // namespace flow
} // namespace tbb
```

Requirements:

- The *MultiOutputNode* type shall have a valid *MultiOutputNode::output\_ports\_type* qualified-id that denotes a type.
- The *MultiInputNode* type shall have a valid *MultiInputNode::input\_ports\_type* qualified-id that denotes a type.

The common form of `make_edge(sender, receiver)` creates an edge between provided `sender` and `receiver` instances.

Overloads that accept a *MultiOutputNode* type instance makes an edge from port 0 of a multi-output predecessor.

Overloads that accept a *MultiInputNode* type instance makes an edge to port 0 of a multi-input successor.

### 11.2.3.3.4 remove\_edge

#### [flow\_graph.remove\_edge]

A function template for building edges between nodes.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template<typename Message>
    inline void remove_edge( sender<Message> &p, receiver<Message> &s );

    template< typename MultiOutputNode, typename MultiInputNode >
    inline void remove_edge( MultiOutputNode& output, MultiInputNode& input );

} // namespace flow
} // namespace tbb
```

(continues on next page)

(continued from previous page)

```

template<typename MultiOutputNode, typename Message>
inline void remove_edge( MultiOutputNode& output, receiver<Message> input );

template<typename Message, typename MultiInputNode>
inline void remove_edge( sender<Message> output, MultiInputNode& input );

} // namespace flow
} // namespace tbb

```

Requirements:

- The *MultiOutputNode* type shall have a valid `MultiOutputNode::output_ports_type` qualified-id that denotes a type.
- The *MultiInputNode* type shall have a valid `MultiInputNode::input_ports_type` qualified-id that denotes a type.

The common form of `remove_edge(sender, receiver)` creates an edge between provided `sender` and `receiver` instances.

Overloads that accept a *MultiOutputNode* type instance removes an edge from port 0 of a multi-output predecessor.

Overloads that accept a *MultiInputNode* type instance removes an edge to port 0 of a multi-input successor.

#### 11.2.3.4 Special Messages Types

Flow Graph supports a set of specific message types.

##### 11.2.3.4.1 continue\_msg

###### [flow\_graph.continue\_msg]

An empty class that represent a continue message. An object of this class is used to indicate that the sender has completed.

```

// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    class continue_msg {};
}

} // namespace flow
} // namespace tbb

```

### 11.2.3.4.2 tagged\_msg

#### [flow\_graph.tagged\_msg]

A class template composed of a tag and a message. The message is a value that can be one of several defined types.

```
// Defined in header <tbb/flow_graph.h>

namespace tbb {
namespace flow {

    template<typename TagType, typename... TN>
    class tagged_msg {
    public:
        template<typename T, typename R>
        tagged_msg(T const &index, R const &val);

        TagType tag() const;

        template<typename V>
        const V& cast_to() const;

        template<typename V>
        bool is_a() const;

    };

} // namespace flow
} // namespace tbb
```

Requirements:

- All types in `TN` template parameter pack shall meet the *CopyConstructible* requirements from [copyconstructible] ISO C++ Standard section.
- The type `TagType` shall be an integral unsigned type.

The `tagged_msg` class template is intended for messages whose type is determined at run time. A message of one of the types `TN` is tagged with a tag of type `TagType`. The tag then may serve to identify the message. In the flow graph `tagged_msg` is used as the output of `indexer_node`.

### 11.2.3.4.2.1 Member functions

`template<typename T, typename R>`  
`tagged_msg(T const &index, R const &value)`

Requirements:

- The type `R` shall be the same as one of the `TN` types.
- The type `T` shall be acceptable as a `TagType` constructor parameter.

Constructs a `tagged_msg` with tag `index` and value `val`.

`TagType tag() const`

Returns the current tag.

`template<typename V>`  
`const V &cast_to() const`

Requirements:

- The type V shall be the same as one of the TN types.

Returns the value stored in the tagged\_msg. If the value is not of type V an std::runtime\_error exception is thrown.

```
template<typename V>
bool is_a() const
```

Requirements:

- The type V shall be the same as one of the TN types.

Returns true if V is the type of the value held by the tagged\_msg. Returns false otherwise.

#### 11.2.3.4.2.2 Non-member functions

```
template<typename V, typename T>
const V& cast_to(T const &t) {
    return t.cast_to<V>();
}

template<typename V, typename T>
bool is_a(T const &t);
```

Requirements:

- The type T shall be an instantiated tagged\_msg class template.
- The type V shall be the same as one of the corresponding template arguments for tagged\_msg.

The free-standing template functions cast\_to and is\_a applied to a tagged\_msg object are equivalent to the calls of the corresponding methods of that object.

See also:

- *indexer\_node class template*

#### 11.2.3.5 Examples

##### 11.2.3.5.1 Dependency Flow Graph Example

In the following example, five computations A-E are set up with the partial ordering shown below in “A simple dependency graph.”. For each edge in the flow graph, the node at the tail of the edge must complete its execution before the node at the head may begin.

```
#include <cstdio>
#include "tbb/flow_graph.h"

using namespace tbb::flow;

struct body {
    std::string my_name;
    body(const char *name) : my_name(name) {}
    void operator()(continue_msg) const {
        printf("%s\n", my_name.c_str());
    }
};
```

(continues on next page)

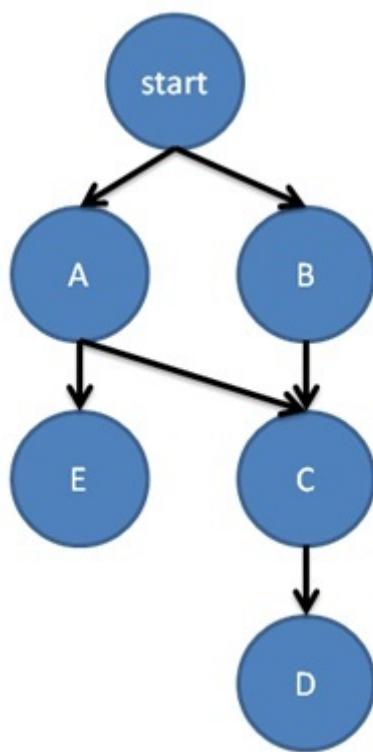


Fig. 2: A simple dependency graph.

(continued from previous page)

```

int main() {
    graph g;

    broadcast_node< continue_msg > start(g);
    continue_node<continue_msg> a(g, body("A"));
    continue_node<continue_msg> b(g, body("B"));
    continue_node<continue_msg> c(g, body("C"));
    continue_node<continue_msg> d(g, body("D"));
    continue_node<continue_msg> e(g, body("E"));

    make_edge(start, a);
    make_edge(start, b);
    make_edge(a, c);
    make_edge(b, c);
    make_edge(c, d);
    make_edge(a, e);

    for (int i = 0; i < 3; ++i) {
        start.try_put(continue_msg());
        g.wait_for_all();
    }

    return 0;
}

```

In this example, nodes A-E print out their names. All of these nodes are therefore able to use `struct body` to construct their body objects.

In function `main`, the flow graph is set up once and then run three times. All of the nodes in this example pass around `continue_msg` objects. This type is used to communicate that a node has completed its execution.

The first line in function `main` instantiates a `graph` object, `g`. On the next line, a `broadcast_node` named `start` is created. Anything passed to this node will be broadcast to all of its successors. The node `start` is used in the `for` loop at the bottom of `main` to launch the execution of the rest of the flow graph.

In the example, five `continue_node` objects are created, named `a` - `e`. Each node is constructed with a reference to `graph g` and the function object to invoke when it runs. The successor / predecessor relationships are set up by the `make_edge` calls that follow the declaration of the nodes.

After the nodes and edges are set up, the `try_put` in each iteration of the `for` loop results in a broadcast of a `continue_msg` to both `a` and `b`. Both `a` and `b` are waiting for a single `continue_msg`, since they both have only a single predecessor, `start`.

When they receive the message from `start`, they execute their body objects. When complete, they each forward a `continue_msg` to their successors, and so on. The graph uses tasks to execute the node bodies as well as to forward messages between the nodes, allowing computation to execute concurrently when possible.

See also:

- [\*continue\\_msg class\*](#)
- [\*continue\\_node class\*](#)

### 11.2.3.5.2 Message Flow Graph Example

This example calculates the sum  $x*x + x*x*x$  for all  $x = 1$  to  $10$ . The layout of this example is shown in the figure below.

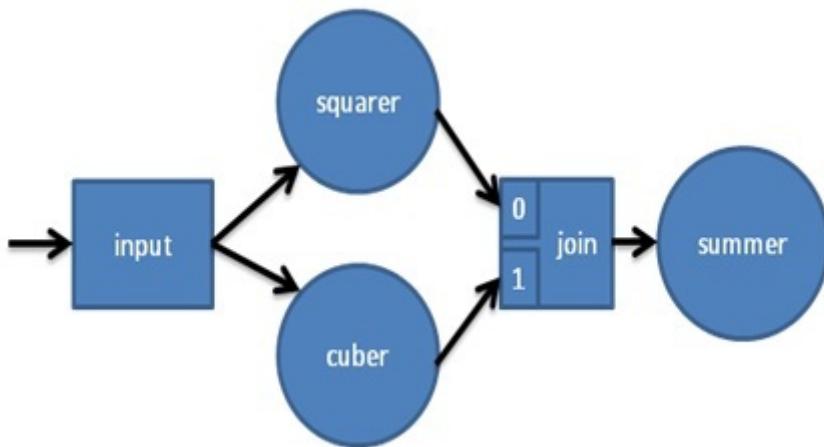


Fig. 3: A simple message flow graph.

Each value enters through the `broadcast_node<int>` `input`. This node broadcasts the value to both `squarer` and `cuber`, which calculate  $x*x$  and  $x*x*x$  respectively. The output of each of these nodes is put to one of `join`'s ports. A tuple containing both values is created by `join_node< tuple<int, int> >` `join` and forwarded to `summer`, which adds both values to the running total. Both `squarer` and `cuber` allow unlimited concurrency, that is they each may process multiple values simultaneously. The final `summer`, which updates a shared total, is only allowed to process a single incoming tuple at a time, eliminating the need for a lock around the shared value.

```

#include <cstdio>
#include "tbb/flow_graph.h"

using namespace tbb::flow;

struct square {
    int operator()(int v) { return v*v; }
};

struct cube {
    int operator()(int v) { return v*v*v; }
};

class sum {
    int &my_sum;
public:
    sum( int &s ) : my_sum(s) {}
    int operator()( tuple< int, int > v ) {
        my_sum += get<0>(v) + get<1>(v);
        return my_sum;
    }
};

int main() {
  
```

(continues on next page)

(continued from previous page)

```

int result = 0;

graph g;
broadcast_node<int> input(g);
function_node<int,int> squarer( g, unlimited, square() );
function_node<int,int> cuber( g, unlimited, cube() );
join_node< tuple<int,int>, queueing > join( g );
function_node<tuple<int,int>,int>
    summer( g, serial, sum(result) );

make_edge( input, squarer );
make_edge( input, cuber );
make_edge( squarer, get<0>( join.input_ports() ) );
make_edge( cuber, get<1>( join.input_ports() ) );
make_edge( join, summer );

for (int i = 1; i <= 10; ++i)
    input.try_put(i);
g.wait_for_all();

printf("Final result is %d\n", result);
return 0;
}

```

In the example code above, the classes `square`, `cube` and `sum` define the three user-defined operations. Each class is used to create a `function_node`.

In function `main`, the flow graph is set up and then the values 1-10 are put into the node `input`. All the nodes in this example pass around values of type `int`. The nodes used in this example are all class templates and therefore can be used with any type that supports copy construction, including pointers and objects.

## 11.2.4 Task Scheduler

### [scheduler]

oneAPI Threading Building Blocks provides a task scheduler, which is the engine that drives the algorithm templates and task groups. The exact tasking API depends on the implementation.

The tasks are quanta of computation. The scheduler implements worker thread pool and maps tasks onto these threads. The mapping is non-preemptive. Once a thread starts running a task, the task is bound to that thread until completion. During that time, the thread services other tasks only when it waits for completion of nested parallel constructs, as described below. While waiting, either user or worker thread may run any available task, including unrelated tasks created by this or other threads.

The task scheduler is intended for parallelizing computationally intensive work. Because task objects are not scheduled preemptively, they should generally avoid making calls that might block a thread for long periods during which the thread cannot service other tasks.

**Caution:** There is no guarantee that *potentially* parallel tasks *actually* execute in parallel, because the scheduler adjusts actual parallelism to fit available worker threads. For example, given a single worker thread, the scheduler creates no actual parallelism. For example, it is generally unsafe to use tasks in a producer consumer relationship, because there is no guarantee that the consumer runs at all while the producer is running.

### 11.2.4.1 Scheduling controls

#### 11.2.4.1.1 task\_group\_context

[scheduler.task\_group\_context]

The `task_group_context` represents a set of properties used by task scheduler for execution of the associated tasks. Each task is associated with the only one `task_group_context` object.

The `task_group_context` objects form a forest of trees. Each tree's root is a `task_group_context` constructed as isolated.

The `task_group_context` is cancelled explicitly by the user request, or implicitly when an exception is thrown out of a associated task. Canceling a `task_group_context` causes the entire subtree rooted at it to be cancelled.

The `task_group_context` carries floating point settings inherited from the parent `task_group_context` object or captured with a dedicated interface.

```
// Defined in header <tbb/task_group.h>

namespace tbb {

    class task_group_context {
    public:
        enum kind_t {
            isolated = /* implementation-defined */,
            bound = /* implementation-defined */
        };
        enum traits_type {
            fp_settings = /* implementation-defined */,
            default_traits = 0
        };

        task_group_context( kind_t relation_with_parent = bound,
                            uintptr_t traits = default_traits );
        ~task_group_context();

        void reset();
        bool cancel_group_execution();
        bool is_group_execution_cancelled() const;
        void capture_fp_settings();
        uintptr_t traits() const;
    };

} // namespace tbb;
```

#### 11.2.4.1.1 Member types and constants

##### `enum kind_t::isolated`

When passed to the specific constructor, created `task_group_context` object has no parent.

##### `enum kind_t::bound`

When passed to the specific constructor, created `task_group_context` object will become a child of the innermost running task's group when the first task associated to the `task_group_context` is passed to the task scheduler. If there is no innermost running task on the current thread, the `task_group_context` will become isolated.

**enum traits\_type::fp\_settings**

When passed to the specific constructor, the flag forces the context to capture floating-point settings from the current thread.

**11.2.4.1.1.2 Member funcitons****task\_group\_context (kind\_t relation\_to\_parent = bound, uintptr\_t traits = default\_traits)**

Constructs an empty task\_group\_context.

**~task\_group\_context ()**

Destroys an empty task\_group\_context. The behavior is undefined if there are still extant tasks associated with this task\_group\_context.

**bool cancel\_group\_execution ()**

Requests that tasks associated with this task\_group\_context.

Returns false if this task\_group\_context is already cancelled; true otherwise. If concurrently called by multiple threads, exactly one call returns true and the rest return false.

**bool is\_group\_execution\_cancelled () const**

Returns true if this task\_group\_context has received the cancellation request.

**void reset ()**

Reinitializes this task\_group\_context to the uncancelled state.

**Caution:** This method is only safe to call once all tasks associated with the group's subordinate groups have completed. This method must not be invoked concurrently by multiple threads.

**void capture\_fp\_settings ()**

Captures floating-point settings from the current thread.

**Caution:** This method is only safe to call once all tasks associated with the group's subordinate groups have completed. This method must not be invoked concurrently by multiple threads.

**uintptr\_t traits () const**

Returns traits of this task\_group\_context.

**11.2.4.1.2 global\_control****[scheduler.global\_control]**

Use this class to control certain settings or behavior of the oneAPI Threading Building Blocks dynamic library.

An object of class global\_control, or a “control variable”, affects one of several behavioral aspects, or parameters, of TBB. Class global\_control is primarily intended for use at the application level, to control the whole application behavior.

The current set of parameters that you can modify is defined by global\_control::parameter enumeration. The parameter and the value it should take are specified as arguments to the constructor of a control variable. The impact of the control variable ends when its lifetime is complete.

Control variables can be created in different threads, and may have nested or overlapping scopes. However, at any point in time each controlled parameter has a single active value that applies to the whole process. This value is selected from all currently existing control variables by applying a parameter-specific selection rule.

```
// Defined in header <tbb/global_control.h>

namespace tbb {
    class global_control {
public:
    enum parameter {
        max_allowed_parallelism,
        thread_stack_size
    };

    global_control(parameter p, size_t value);
    ~global_control();

    static size_t active_value(parameter param);
};

} // namespace tbb
```

#### 11.2.4.1.2.1 Member types and constants

##### **enum parameter::max\_allowed\_parallelism**

**Selection rule:** minimum

Limit total number of worker threads that can be active in the task scheduler to parameter\_value - 1.

---

**Note:** With max\_allowed\_parallelism set to 1, global\_control enforces serial execution of all tasks by the application thread(s), i.e. the task scheduler does not allow worker threads to run. There is one exception: if some work is submitted for execution via task\_arena::enqueue, a single worker thread will still run ignoring the max\_allowed\_parallelism restriction.

---

##### **enum parameter::thread\_stack\_size**

**Selection rule:** maximum

Set stack size for threads created by the library, including working threads in the task scheduler and threads controlled by thread wrapper classes.

#### 11.2.4.1.2.2 Member functions

##### **global\_control (parameter param, size\_t value)**

Constructs a global\_control object with a specified control parameter and it's value.

##### **~global\_control ()**

Destructs a control variable object and ends it's impact.

##### **static size\_t active\_value (parameter param)**

Returns the currently active value of the setting defined by param.

See also:

- *task\_arena*

### 11.2.4.1.3 Resumable tasks

#### [scheduler.resumable\_tasks]

Functions to suspend task execution at a specific point and signal to resume it later.

```
// Defined in header <tbb/task.h>

using tbb::task::suspend_point = /* implementation-defined */;
template < typename Func > void tbb::task::suspend( Func );
void tbb::task::resume( tbb::task::suspend_point );
```

Requirements:

- The `Func` type shall meet the *SuspendFunc requirements*.

The `tbb::task::suspend` function called within a running task suspends execution of the task and switches the thread to participate in other oneTBB parallel work. This function accepts a user callable object with the current execution context `tbb::task::suspend_point` as an argument.

The `tbb::task::suspend_point` context tag shall be passed to the `tbb::task::resume` function to trigger a program execution at the suspended point. The `tbb::task::resume` function can be called at any point of an application, even on a separate thread. In this regard, this function acts as a signal for the task scheduler.

---

**Note:** Note, that there are no guarantees, that the same thread that called `tbb::task::suspend` will continue execution after the suspended point. However, these guarantees are provided for the outermost blocking oneTBB calls (such as `tbb::parallel_for` and `tbb::flow::graph::wait_for_all`) and `tbb::task_arena::execute` calls.

---

### 11.2.4.1.3.1 Example

```
// Parallel computation region
tbb::parallel_for(0, N, [&](int) {
    // Suspend the current task execution and capture the context
    tbb::task::suspend([&] (tbb::task::suspend_point tag) {
        // Dedicated user-managed activity that processes async requests.
        async_activity.submit(tag); // could be OpenCL/IO/Database/Network etc.
    }); // execution will be resumed after this function
});
```

```
// Dedicated user-managed activity:

// Signal to resume execution of the task referenced by the tbb::task::suspend_point
// from a dedicated user-managed activity
tbb::task::resume(tag);
```

## 11.2.4.2 Task Group

### 11.2.4.2.1 task\_group

#### [scheduler.task\_group]

A `task_group` represents concurrent execution of a group of tasks. Tasks may be dynamically added to the group as it is executing.

```
// Defined in header <tbb/task_group.h>

namespace tbb {

    class task_group {
    public:
        task_group();
        ~task_group();

        template<typename Func>
        void run( Func&& f );

        template<typename Func>
        task_group_status run_and_wait( const Func& f );

        task_group_status wait();
        bool is_canceling();
        void cancel();
    };

    bool is_current_task_group_canceling();
}

// namespace tbb
```

#### 11.2.4.2.1.1 Member functions

##### `task_group()`

Constructs an empty `task_group`.

##### `~task_group()`

Destroys the `task_group`.

**Requires:** Method `wait` must be called before destroying a `task_group`, otherwise the destructor throws an exception.

##### template<typename Func> void `run`(`Func` &&`f`)

Adds a task to compute `f()` and returns immediately. The `Func` type shall meet the *Function Objects* requirements from [function.objects] ISO C++ Standard section.

##### template<typename Func> task\_group\_status `run_and_wait`(`const Func` &`f`)

Equivalent to `{run(f); return wait();}`, but guarantees that `f()` runs on the current thread. The `Func` type shall meet the *Function Objects* requirements from [function.objects] ISO C++ Standard section.

**Returns:** The status of `task_group`. See `task_group_status`.

##### `task_group_status wait()`

Waits for all tasks in the group to complete or be cancelled.

**Returns:** The status of `task_group`. See `task_group_status`.

```
bool is_canceling()
```

**Returns:** True if this task group is cancelling its tasks.

```
void cancel()
```

Cancels all tasks in this `task_group`.

#### 11.2.4.2.1.2 Non-member functions

```
bool is_current_task_group_canceling()
```

Returns true if an innermost `task_group` executing on this thread is cancelling its tasks.

#### 11.2.4.2.2 `task_group_status`

[`scheduler.task_group_status`]

A `task_group_status` type represents the status of a `task_group`.

```
namespace tbb {
    enum task_group_status {
        not_complete,
        complete,
        canceled
    };
}
```

#### 11.2.4.2.2.1 Member constants

**not\_complete**

Not cancelled and not all tasks in group have completed.

**complete**

Not cancelled and all tasks in group have completed.

**canceled**

Task group received cancellation request.

#### 11.2.4.3 Task Arena

##### 11.2.4.3.1 `task_arena`

[`scheduler.task_arena`]

A class that represents an explicit, user-managed task scheduler arena.

```
// Defined in header <tbb/task_arena.h>

namespace tbb {

    class task_arena {
    public:
        static const int automatic = /* implementation-defined */;
```

(continues on next page)

(continued from previous page)

```

static const int not_initialized = /* implementation-defined */;
struct attach {};

task_arena(int max_concurrency = automatic, unsigned reserved_for_masters = 1);
task_arena(const task_arena &s);
explicit task_arena(task_arena::attach);
~task_arena();

void initialize();
void initialize(int max_concurrency, unsigned reserved_for_masters = 1);
void initialize(task_arena::attach);
void terminate();

bool is_active() const;
int max_concurrency() const;

template<typename F> auto execute(F&& f) -> decltype(f());
template<typename F> void enqueue(F&& f);
};

} // namespace tbb

```

A `task_arena` class represents a place where threads may share and execute tasks.

The number of threads that may simultaneously execute tasks in a `task_arena` is limited by its concurrency level.

Each user thread that invokes any parallel construction outside an explicit `task_arena` uses an implicit task arena representation object associated with the calling thread.

The tasks spawned or enqueued into one arena cannot be executed in another arena.

---

**Note:** The `task_arena` constructors do not create an internal task arena representation object. It may already exist in case of the “attaching” constructor, otherwise it is created by explicit call to `task_arena::initialize` or lazily on first use.

---

#### 11.2.4.3.1.1 Member types and constants

##### **static const int automatic**

When passed as `max_concurrency` to the specific constructor, arena concurrency will be automatically set based on the hardware configuration.

##### **static const int not\_initialized**

When returned by a method or function, indicates that there is no active `task_arena` or that the `task_arena` object has not yet been initialized.

##### **struct attach**

A tag for constructing a `task_arena` with `attach`.

### 11.2.4.3.1.2 Member functions

**task\_arena** (int *max\_concurrency* = *automatic*, unsigned *reserved\_for\_masters* = 1)

Creates a task\_arena with a certain concurrency limit (*max\_concurrency*). Some portion of the limit can be reserved for application threads with *reserved\_for\_masters*. The amount for reservation cannot exceed the limit.

**Caution:** If *max\_concurrency* and *reserved\_for\_masters* are explicitly set to be equal and greater than 1, oneTBB worker threads will never join the arena. As a result, the execution guarantee for enqueued tasks is not valid in such arena. Do not use `task_arena::enqueue()` with an arena set to have no worker threads.

**task\_arena** (`const task_arena&`)

Copies settings from another task\_arena instance.

**explicit task\_arena** (`task_arena::attach`)

Creates an instance of task\_arena that is connected to the internal task arena representation currently used by the calling thread. If no such arena exists yet, creates a task\_arena with default parameters.

---

**Note:** Unlike other constructors, this one automatically initializes the new task\_arena when connecting to an already existing arena.

---

**~task\_arena** ()

Destroys the task\_arena instance, but the destruction may not be synchronized with any task execution inside this task\_arena. Which means that an internal task arena representation associated with this task\_arena instance can be destroyed later. Not thread safe w.r.t. concurrent invocations of other methods.

**void initialize** ()

Performs actual initialization of internal task arena representation.

---

**Note:** After the call to `initialize`, the arena parameters are fixed and cannot be changed.

---

**void initialize** (int *max\_concurrency*, unsigned *reserved\_for\_masters* = 1)

Same as above, but overrides previous arena parameters.

**void initialize** (`task_arena::attach`)

If an instance of class `task_arena::attach` is specified as the argument, and there exists an internal task arena representation currently used by the calling thread, the method ignores arena parameters and connects task\_arena to that internal task arena representation. The method has no effect when called for an already initialized task\_arena.

**void terminate** ()

Removes the reference to the internal task arena representation without destroying the task\_arena object, which can then be re-used. Not thread safe w.r.t. concurrent invocations of other methods.

**bool is\_active** () `const`

Returns `true` if the task\_arena has been initialized, `false` otherwise.

**int max\_concurrency** () `const`

Returns the concurrency level of the task\_arena. Does not require the task\_arena to be initialized and does not perform initialization.

template<*F*>

```
void enqueue (F &&f)
```

Enqueues a task into the `task_arena` to process the specified functor and immediately returns. The F type shall meet the *Function Objects* requirements from [function.objects] ISO C++ Standard section. The task is scheduled for eventual execution by a worker thread even if no thread ever explicitly waits for the task to complete. If the total number of worker threads is zero, a special additional worker thread is created to execute enqueued tasks.

---

**Note:** The method does not require the calling thread to join the arena; i.e. any number of threads outside of the arena can submit work to it without blocking.

---

**Caution:** There is no guarantee that tasks enqueued into an arena execute concurrently with respect to any other tasks there.

**Caution:** An exception thrown and not caught in the functor results in undefined behavior.

```
template<F>
auto execute (F &&f) -> decltype(f())
```

Executes the specified functor in the `task_arena` and returns the value returned by the functor. The F type shall meet the *Function Objects* requirements from [function.objects] ISO C++ Standard section.

The calling thread joins the `task_arena` if possible, and executes the functor. Upon return it restores the previous task scheduler state and floating-point settings.

If joining the `task_arena` is not possible, the call wraps the functor into a task, enqueues it into the arena, waits using an OS kernel synchronization object for another opportunity to join, and finishes after the task completion.

An exception thrown in the functor will be captured and re-thrown from `execute`.

---

**Note:** Any number of threads outside of the arena can submit work to the arena and be blocked. However, only the maximal number of threads specified for the arena can participate in executing the work.

---

See also:

- *task\_group*
- *task\_scheduler\_observer*

#### 11.2.4.3.2 `this_task_arena`

##### [`scheduler.this_task_arena`]

The namespace for functions applicable to the current `task_arena`.

The namespace `this_task_arena` contains global functions for interaction with the `task_arena` currently used by the calling thread.

```
// Defined in header <tbb/task_arena.h>

namespace tbb {
    namespace this_task_arena {
```

(continues on next page)

(continued from previous page)

```

int current_thread_index();
int max_concurrency();
template<typename F> auto isolate(F&& f) -> decltype(f());
}
}

```

**int current\_thread\_index()**

Returns the thread index in a task\_arena currently used by the calling thread, or task\_arena::not\_initialized if the thread has not yet initialized the task scheduler.

A thread index is an integer number between 0 and the task\_arena concurrency level. Thread indexes are assigned to both application threads and worker threads on joining an arena and are kept until exiting the arena. Indexes of threads that share an arena are unique - i.e. no two threads within the arena may have the same index at the same time - but not necessarily consecutive.

---

**Note:** Since a thread may exit the arena at any time if it does not execute a task, the index of a thread may change between any two tasks, even those belonging to the same task group or algorithm.

---

**Note:** Threads that use different arenas may have the same current index value.

---



---

**Note:** Joining a nested arena in execute () may change current index value while preserving the index in the outer arena which will be restored on return.

---

**int max\_concurrency()**

Returns the concurrency level of the task\_arena currently used by the calling thread. If the thread has not yet initialized the task scheduler, returns the concurrency level determined automatically for the hardware configuration.

**template<F>****auto isolate (F &&f) -> decltype(f())**

Runs the specified functor in isolation by restricting the calling thread to process only tasks scheduled in the scope of the functor (also called the isolation region). The function returns the value returned by the functor. The F type shall meet the *Function Objects* requirements from [function.objects] ISO C++ Standard section.

**Caution:** The object returned by the functor cannot be a reference. std::reference\_wrapper can be used instead.

### 11.2.4.3.3 task\_scheduler\_observer

#### [scheduler.task\_scheduler\_observer]

Class that represents thread's interest in task scheduling services.

```

// Defined in header <tbb/task_scheduler_observer.h>

namespace tbb {
    class task_scheduler_observer {

```

(continues on next page)

(continued from previous page)

```

public:
    task_scheduler_observer();
    explicit task_scheduler_observer( task_arena& a );
    virtual ~task_scheduler_observer();

    void observe( bool state=true );
    bool is_observing() const;

    virtual void on_scheduler_entry( bool is_worker ) {};
    virtual void on_scheduler_exit( bool is_worker ) {};
};

}

```

A `task_scheduler_observer` permits clients to observe when a thread starts and stops processing tasks, either globally or in a certain task scheduler arena. You typically derive your own observer class from `task_scheduler_observer`, and override virtual methods `on_scheduler_entry` or `on_scheduler_exit`. Observation can be enabled and disabled for an observer instance; it is disabled on creation. Remember to call `observe()` to enable observation.

Exceptions thrown and not caught in the overridden methods of `task_scheduler_observer` result in undefined behavior.

#### 11.2.4.3.3.1 Member functions

##### `task_scheduler_observer()`

Constructs a `task_scheduler_observer` object in the inactive state (observation is disabled). For a created observer, entry/exit notifications are invoked whenever a worker thread joins/leaves the arena of the observer's owner thread. If a thread is already in the arena when the observer is activated, the entry notification is called before it executes the first stolen task.

##### `explicit task_scheduler_observer(task_arena&)`

Construct `task_scheduler_observer` object for a given arena in inactive state (observation is disabled). For created observer, entry/exit notifications are invoked whenever a thread joins/leaves arena. If a thread is already in the arena when the observer is activated, the entry notification is called before it executes the first stolen task.

Constructs a `task_scheduler_observer` object in the inactive state (observation is disabled), which receives notifications from threads entering and exiting the specified `task_arena`.

##### `~task_scheduler_observer()`

Disables observing and destroys the observer instance. Waits for extant invocations of `on_scheduler_entry` and `on_scheduler_exit` to complete.

##### `void observe(bool state = true)`

Enables observing if `state` is true; disables observing if `state` is false.

##### `bool is_observing() const`

**Returns:** True if observing is enabled; false otherwise.

##### `virtual void on_scheduler_entry(bool is_worker)`

The task scheduler invokes this method for each thread that starts participating in oneTBB work or enters an arena after the observation is enabled. For threads that already execute tasks, the method is invoked before executing the first task stolen after enabling the observation.

If a thread enables the observation and then spawns a task, it is guaranteed that the task, as well as all the tasks it creates, will be executed by threads which have invoked `on_scheduler_entry`.

The flag `is_worker` is true if the thread was created by oneTBB; false otherwise.

**Effects:** The default behavior does nothing.

**virtual void on\_scheduler\_exit (bool `is_worker`)**

The task scheduler invokes this method when a thread stops participating in task processing or leaves an arena.

**Caution:** A process does not wait for the worker threads to clean up, and can terminate before `on_scheduler_exit` is invoked.

**Effects:** The default behavior does nothing.

#### 11.2.4.3.3.2 Example

The following example sketches the code of an observer that pins oneTBB worker threads to hardware threads.

```
class pinning_observer : public tbb::task_scheduler_observer {
public:
    affinity_mask_t m_mask; // HW affinity mask to be used for threads in an arena
    pinning_observer( tbb::task_arena &a, affinity_mask_t mask )
        : tbb::task_scheduler_observer(a), m_mask(mask) {
            observe(true); // activate the observer
    }
    void on_scheduler_entry( bool worker ) override {
        set_thread_affinity(tbb::this_task_arena::current_thread_index(), m_mask);
    }
    void on_scheduler_exit( bool worker ) override {
        restore_thread_affinity();
    }
};
```

## 11.2.5 Containers

### [containers]

The container classes provided by oneAPI Threading Building Blocks permit multiple threads to simultaneously invoke certain methods on the same container.

#### 11.2.5.1 Sequences

##### 11.2.5.1.1 concurrent\_vector

### [containers.concurrent\_vector]

`concurrent_vector` is a class template for a vector that can be concurrently grown and accessed.

### 11.2.5.1.1.1 Class Template Synopsis

```
// Defined in header <tbb/concurrent_vector.h>

namespace tbb {

    template <typename T,
              typename Allocator = cache_aligned_allocator<T>>
    class concurrent_vector {
        using value_type = T;
        using allocator_type = Allocator;

        using size_type = <implementation-defined unsigned integer type>;
        using difference_type = <implementation-defined signed integer type>;

        using reference = value_type&;
        using const_reference = const value_type&;

        using pointer = typename std::allocator_traits<allocator_type>::pointer;
        using const_pointer = typename std::allocator_traits<allocator_type>::const_
→pointer;

        using iterator = <implementation-defined RandomAccessIterator>;
        using const_iterator = <implementation-defined constant RandomAccessIterator>;

        using reverse_iterator = std::reverse_iterator<iterator>;
        using const_reverse_iterator = std::reverse_iterator<const_iterator>;

        using range_type = <implementation-defined ContainerRange>;
        using const_range_type = <implementation-defined constant ContainerRange>;

        // Construction, destruction, copying
        concurrent_vector();
        explicit concurrent_vector( const allocator_type& alloc ) noexcept;

        explicit concurrent_vector( size_type count, const value_type& value,
                                   const allocator_type& alloc = allocator_type() );

        explicit concurrent_vector( size_type count,
                                   const allocator_type& alloc = allocator_type() );

        template <typename InputIterator>
        concurrent_vector( InputIterator first, InputIterator last,
                           const allocator_type& alloc = allocator_type() );

        concurrent_vector( std::initializer_list<value_type> init,
                           const allocator_type& alloc = allocator_type() );

        concurrent_vector( const concurrent_vector& other );
        concurrent_vector( const concurrent_vector& other, const allocator_type&_
→alloc );

        concurrent_vector( concurrent_vector&& other ) noexcept;
        concurrent_vector( concurrent_vector&& other, const allocator_type& alloc );

        ~concurrent_vector();
    };
}
```

(continues on next page)

(continued from previous page)

```

concurrent_vector& operator=( const concurrent_vector& other );

concurrent_vector& operator=( concurrent_vector&& other ) noexcept( /*See
→details*/);

concurrent_vector& operator=( std::initializer_list<value_type> init );

void assign( size_type count, const value_type& value );

template <typename InputIterator>
void assign( InputIterator first, InputIterator last );

void assign( std::initializer_list<value_type> init );

// Concurrent growth
iterator grow_by( size_type delta );
iterator grow_by( size_type delta, const value_type& value );

template <typename InputIterator>
iterator grow_by( InputIterator first, InputIterator last );

iterator grow_by( std::initializer_list<value_type> init );

iterator grow_to_at_least( size_type n );
iterator grow_to_at_least( size_type n, const value_type& value );

iterator push_back( const value_type& value );
iterator push_back( value_type&& value );

template <typename... Args>
iterator emplace_back( Args&&... args );

// Element access
value_type& operator[]( size_type index );
const value_type& operator[]( size_type index ) const;

value_type& at( size_type index );
const value_type& at( size_type index ) const;

value_type& front();
const value_type& front() const;

value_type& back();
const value_type& back() const;

// Iterators
iterator begin();
const_iterator begin() const;
const_iterator cbegin() const;

iterator end();
const_iterator end() const;
const_iterator cend() const;

reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
const_reverse_iterator crbegin() const;

```

(continues on next page)

(continued from previous page)

```

reverse_iterator rend();
const_reverse_iterator rend() const;
const_reverse_iterator crend() const;

// Size and capacity
size_type size() const noexcept;

bool empty() const noexcept;

size_type max_size() const noexcept;

size_type capacity() const noexcept;

// Concurrently unsafe operations
void reserve( size_type n );

void resize( size_type n );
void resize( size_type n, const value_type& value );

void shrink_to_fit();

void swap( concurrent_vector& other ) noexcept(/*See details*/);

void clear();

allocator_type get_allocator() const;

// Parallel iteration
range_type range( size_type grainsize = 1 );
const_range_type range( size_type grainsize = 1 ) const;
}; // class concurrent_vector

} // namespace tbb

```

### 11.2.5.1.1.2 Requirements

- The type `T` shall meet the following requirements:
  - Requirements of `Erasable` from [container.requirements] ISO C++ Standard section.
  - Its destructor must not throw an exception.
  - If its default constructor can throw an exception, its destructor must be non-virtual and work correctly on zero-filled memory.
  - Member functions can impose stricter requirements depending on the type of the operation.
- The type `Allocator` shall meet the `Allocator` requirements from [allocator.requirements] ISO C++ section.

### 11.2.5.1.1.3 Description

`tbb::concurrent_vector` is a class template which represents a sequence container with the following features:

- Multiple threads can concurrently grow the container and append new elements.
- Random access by index. The index of the first element is zero.
- Growing the container does not invalidate any existing iterators or indices.

### 11.2.5.1.1.4 Exception Safety

Concurrent growing is fundamentally incompatible with ideal exception safety. Nonetheless, `tbb::concurrent_vector` offers a practical level of exception safety.

Growth and vector assignment append a sequence of elements to a vector. If an exception occurs, the impact on the vector depends upon the cause of the exception:

- If the exception is thrown by the constructor of an element, then all subsequent elements in the appended sequence will be zero-filled.
- Otherwise, the exception was thrown by the vector's allocator. The vector becomes broken. Each element in the appended sequence will be in one of three states:
  - constructed
  - zero-filled
  - unallocated in memory

Once a vector becomes broken, care must be taken when accessing it:

- Accessing an unallocated element with the method `at` causes an exception `std::range_error`. Accessing an unallocated element using any other method has undefined behavior.
- The values of `capacity()` and `size()` may be less than expected.
- Access to a broken vector via `back()` has undefined behavior.

However, the following guarantees hold for broken or unbroken vectors:

- Let `k` be an index of an unallocated element. Then `size() <= capacity() <= k`.
- Growth operations never cause `size()` or `capacity()` to decrease.

If a concurrent growth operation successfully completes, the appended sequence remains valid and accessible even if a subsequent growth operations fails.

### 11.2.5.1.1.5 Member functions

#### 11.2.5.1.1.6 Construction, destruction, copying

#### 11.2.5.1.1.7 Empty container constructors

```
concurrent_vector();
explicit concurrent_vector( const allocator_type& alloc );
```

Constructs an empty concurrent\_vector.

If provided, uses the allocator alloc to allocate the memory.

#### 11.2.5.1.1.8 Constructors from the sequence of elements

```
explicit concurrent_vector( size_type count, const value_type& value,
                            const allocator_type& alloc = allocator_type() );
```

Constructs a concurrent\_vector containing count copies of the value using the allocator alloc.

```
explicit concurrent_vector( size_type count,
                            const allocator_type& alloc = allocator_type() );
```

Constructs a concurrent\_vector containing n default constructed in-place elements using the allocator alloc.

```
template <typename InputIterator>
concurrent_vector( InputIterator first, InputIterator last,
                   const allocator_type& alloc = allocator_type() );
```

Constructs a concurrent\_vector contains all elements from the half-open interval [first, last) using the allocator alloc.

**Requirements:** the type InputIterator shall meet the requirements of InputIterator from [input.iterators] ISO C++ Standard section.

```
concurrent_vector( std::initializer_list<value_type> init,
                   const allocator_type& alloc = allocator_type() );
```

Equivalent to concurrent\_vector(init.begin(), init.end(), alloc).

#### 11.2.5.1.1.9 Copying constructors

```
concurrent_vector( const concurrent_vector& other );
concurrent_vector( const concurrent_vector& other,
                   const allocator_type& alloc );
```

Constructs a copy of other.

If the allocator argument is not provided, it is obtained by calling std::allocator\_traits<allocator\_type>::select\_on\_container\_copy\_construction(other.get\_allocator()).

The behavior is undefined in case of concurrent operations with other.

### 11.2.5.1.1.10 Moving constructors

```
concurrent_vector( concurrent_vector&& other );
concurrent_vector( concurrent_vector&& other,
                   const allocator_type& alloc );
```

Constructs a concurrent\_vector with the contents of other using move semantics.

other is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling std::move(other.get\_allocator()).

The behavior is undefined in case of concurrent operations with other.

### 11.2.5.1.1.11 Destructor

```
~concurrent_vector();
```

Destroys the concurrent\_vector. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with \*this.

### 11.2.5.1.1.12 Assignment operators

```
concurrent_vector& operator=( const concurrent_vector& other );
```

Replaces all elements in \*this by the copies of the elements in other.

Copy assigns allocators if std::allocator\_traits<allocator\_type>::propagate\_on\_container\_copy\_assignment is true.

The behavior is undefined in case of concurrent operations with \*this and other.

**Returns:** a reference to \*this.

```
concurrent_vector& operator=( concurrent_vector&& other ) noexcept( /*See below */ );
```

Replaces all elements in \*this by the elements in other using move semantics.

other is left in a valid, but unspecified state.

Move assigns allocators if std::allocator\_traits<allocator\_type>::propagate\_on\_container\_move\_assignment is true.

The behavior is undefined in case of concurrent operations with \*this and other.

**Returns:** a reference to \*this.

**Exceptions:** noexcept specification:

```
noexcept (std::allocator_traits<allocator_type>::propagate_on_
    ~container_move_assignment::value || 
        std::allocator_traits<allocator_type>::is_always_
    ~equal::value)
```

```
concurrent_vector& operator=( std::initializer_list<value_type> init );
```

Replaces all elements in `*this` by the elements in `init`.

The behavior is undefined in case of concurrent operations with `*this`.

**Returns:** a reference to `*this`.

#### 11.2.5.1.1.13 assign

```
void assign( size_type count, const value_type& value );
```

Replaces all elements in `*this` by `count` copies of `value`.

```
template <typename InputIterator>
void assign( InputIterator first, InputIterator last );
```

Replaces all elements in `*this` by the elements from the half-open interval `[first, last)`.

This overload only participates in overload resolution if the type `InputIterator` meets the requirements of `InputIterator` from [input.iterators] ISO C++ Standard section.

```
void assign( std::initializer_list<value_type> init );
```

Equivalent to `assign(init.begin(), init.end())`.

#### 11.2.5.1.1.14 get\_allocator

```
allocator_type get_allocator() const;
```

**Returns:** a copy of the allocator associated with `*this`.

#### 11.2.5.1.1.15 Concurrent growth

All member functions in this section can be performed concurrently with each other, element access methods and while traversing the container.

### 11.2.5.1.1.16 grow\_by

```
iterator grow_by( size_type delta );
```

Appends a sequence comprising `delta` new default-constructed in-place elements to the end of the vector.

**Returns:** iterator to the beginning of the appended sequence.

**Requirements:** the type `value_type` shall meet the `DefaultConstructible` and `EmplaceConstructible` requirements from [defaultconstructible] and [container.requirements] ISO C++ sections.

```
iterator grow_by( size_type delta, const value_type& value );
```

Appends a sequence comprising `delta` copies of `value` to the end of the vector.

**Returns:** iterator to the beginning of the appended sequence.

**Requirements:** the type `value_type` shall meet the `CopyInsertable` requirements from [container.requirements] ISO C++ Standard section.

```
template <typename InputIterator>
iterator grow_by( InputIterator first, InputIterator last );
```

Appends a sequence comprising all elements from the half-open interval `[first, last)` to the end of the vector.

**Returns:** iterator to the beginning of the appended sequence.

This overload only participates in overload resolution if the type `InputIterator` meets the requirements of `InputIterator` from [input.iterators] ISO C++ Standard section.

```
iterator grow_by( std::initializer_list<value_type> init );
```

Equivalent to `grow_by(init.begin(), init.end())`.

### 11.2.5.1.1.17 grow\_to\_at\_least

```
iterator grow_to_at_least( size_type n );
```

Appends minimal sequence of default constructed in-place elements such that `size() >= n`.

**Returns:** iterator to the beginning of the appended sequence.

**Requirements:** the type `value_type` shall meet the `DefaultConstructible` and `EmplaceConstructible` requirements from [defaultconstructible] and [container.requirements] ISO C++ sections.

```
iterator grow_to_at_least( size_type n, const value_type& value );
```

Appends minimal sequence of comprising copies of `value` such that `size() >= n`.

**Returns:** iterator to the beginning of the appended sequence.

**Requirements:** the type `value_type` shall meet the `CopyInsertable` requirements from [container.requirements] ISO C++ Standard section.

#### 11.2.5.1.1.18 `push_back`

```
iterator push_back( const value_type& value );
```

Appends a copy of `value` to the end of the vector.

**Returns:** iterator to the appended element.

**Requirements:** the type `value_type` shall meet the `CopyInsertable` requirements from [container.requirements] ISO C++ Standard section.

```
iterator push_back( value_type&& value );
```

Appends `value` to the end of the vector using move semantics.

`value` is left in a valid, but unspecified state.

**Returns:** iterator to the appended element.

**Requirements:** the type `value_type` shall meet the `MoveInsertable` requirements from [container.requirements] ISO C++ Standard section.

#### 11.2.5.1.1.19 `emplace_back`

```
template <typename... Args>
iterator emplace_back( Args&&... args );
```

Appends an element constructed in-place from `args` to the end of the vector.

**Returns:** iterator to the appended element.

**Requirements:** the type `value_type` shall meet the `EmplaceConstructible` requirements from [container.requirements] ISO C++ section.

#### 11.2.5.1.1.20 Element access

All member functions in this section can be performed concurrently with each other, concurrent growth methods and while traversing the container.

In case of concurrent growth, the element returned by the access method can refer to the element, which is under construction of the other thread.

### 11.2.5.1.1.21 Access by index

```
value_type& operator[]( size_type index );
const value_type& operator[]( size_type index ) const;
```

**Returns:** a reference to the element on the position `index`.

The behavior is undefined if `index() >= size()`.

```
value_type& at( size_type index );
const value_type& at( size_type index ) const;
```

**Returns:** a reference to the element on the position `index`.

**Throws:**

- `std::out_of_range` if `index >= size()`.
- `std::range_error` if the vector is broken and the element on the position `index` unallocated.

### 11.2.5.1.1.22 Access the first and the last element

```
value_type& front();
const value_type& front() const;
```

**Returns:** a reference to the first element in the vector.

```
value_type& back();
const value_type& back() const;
```

**Returns:** a reference to the last element in the vector.

### 11.2.5.1.1.23 Iterators

The types `concurrent_vector::iterator` and `concurrent_vector::const_iterator` meets the requirements of `RandomAccessIterator` from [random.access.iterators] ISO C++ Standard section.

### 11.2.5.1.1.24 begin and cbegin

```
iterator begin();
const_iterator begin() const;
const_iterator cbegin() const;
```

**Returns:** an iterator to the first element in the vector.

### 11.2.5.1.1.25 end and cend

```
iterator end();
const_iterator end() const;
const_iterator cend() const;
```

**Returns:** an iterator to the element which follows the last element in the vector.

### 11.2.5.1.1.26 rbegin and crbegin

```
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
const_reverse_iterator crbegin() const;
```

**Returns:** a reverse iterator to the first element of the reversed vector.

### 11.2.5.1.1.27 rend and crend

```
reverse_iterator rend();
const_reverse_iterator rend() const;
const_reverse_iterator crend() const;
```

**Returns:** a reverse iterator which follows the last element of the reversed vector.

### 11.2.5.1.1.28 Size and capacity

#### 11.2.5.1.1.29 size

```
size_type size() const noexcept;
```

**Returns:** the number of elements in the vector.

#### 11.2.5.1.1.30 empty

```
bool empty() const noexcept;
```

**Returns:** true if the vector is empty, false otherwise.

### 11.2.5.1.1.31 max\_size

```
size_type max_size() const noexcept;
```

**Returns:** the maximum number of elements that the vector can hold.

### 11.2.5.1.1.32 capacity

```
size_type capacity() const noexcept;
```

**Returns:** the maximum number of elements that the vector can hold without allocating more memory.

### 11.2.5.1.1.33 Concurrently unsafe operations

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

### 11.2.5.1.1.34 Reserving

```
void reserve( size_type n );
```

Reserves memory for at least n elements.

**Throws:** std::length\_error if n > max\_size().

### 11.2.5.1.1.35 Resizing

```
void resize( size_type n );
```

If n < size(), the vector is reduced to its first n elements.

Otherwise, appends n - size() new elements default-constructed in-place to the end of the vector.

```
void resize( size_type n, const value_type& value );
```

If n < size(), the vector is reduced to its first n elements.

Otherwise, appends n - size() copies of value to the end of the vector.

### 11.2.5.1.1.36 shrink\_to\_fit

```
void shrink_to_fit();
```

Removes the unused capacity of the vector.

Call for this method can also reorganize the internal vector representation in the memory.

### 11.2.5.1.1.37 clear

```
void clear();
```

Removes all elements from the container.

### 11.2.5.1.1.38 swap

```
void swap( concurrent_vector& other ) noexcept(/*See below*/);
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is true.

Otherwise if `get_allocator() != other.get_allocator()` the behavior is undefined.

**Exceptions:** `noexcept` specification:

```
noexcept(std::allocator_traits<allocator_type>::propagate_on_
  ↵container_swap::value ||
    ↵std::allocator_traits<allocator_type>::is_always_
  ↵equal::value
```

### 11.2.5.1.1.39 Parallel iteration

Member types `concurrent_vector::range_type` and `concurrent_vector::const_range_type` meets the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_vector::const_range_type` are of type `concurrent_vector::const_iterator`, whereas the bounds for a `concurrent_vector::range_type` are of type `concurrent_vector::iterator`.

### 11.2.5.1.1.40 range member function

```
range_type range( size_type grainsize = 1 );
const_range_type range( size_type grainsize = 1 ) const;
```

**Returns:** a range object representing all elements in the container.

### 11.2.5.1.1.41 Non-member functions

These functions provides binary and lexicographical comparison and swap operations on `tbb::concurrent_vector` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `tbb::concurrent_vector` as a type alias for which the non-member functions are reachable only via argument dependent lookup.

```

template <typename T, typename Allocator>
bool operator==( const concurrent_vector<T, Allocator>& lhs,
                   const concurrent_vector<T, Allocator>& rhs );

template <typename T, typename Allocator>
bool operator!=( const concurrent_vector<T, Allocator>& lhs,
                   const concurrent_vector<T, Allocator>& rhs );

template <typename T, typename Allocator>
bool operator<( const concurrent_vector<T, Allocator>& lhs,
                   const concurrent_vector<T, Allocator>& rhs );

template <typename T, typename Allocator>
bool operator<=( const concurrent_vector<T, Allocator>& lhs,
                   const concurrent_vector<T, Allocator>& rhs );

template <typename T, typename Allocator>
bool operator>( const concurrent_vector<T, Allocator>& lhs,
                   const concurrent_vector<T, Allocator>& rhs );

template <typename T, typename Allocator>
bool operator>=( const concurrent_vector<T, Allocator>& lhs,
                   const concurrent_vector<T, Allocator>& rhs );

template <typename T, typename Allocator>
void swap( concurrent_vector<T, Allocator>& lhs,
            concurrent_vector<T, Allocator>& rhs );

```

#### 11.2.5.1.1.42 Non-member binary comparisons

Two objects of `concurrent_vector` are equal if:

- they contains an equal number of elements.
- the elements on the same positions are equal.

```

template <typename T, typename Allocator>
bool operator==( const concurrent_vector<T, Allocator>& lhs,
                   const concurrent_vector<T, Allocator>& rhs );

```

**Returns:** true if `lhs` is equal to `rhs`, false otherwise.

```

template <typename T, typename Allocator>
bool operator!=( const concurrent_vector<T, Allocator>& lhs,
                   const concurrent_vector<T, Allocator>& rhs );

```

**Returns:** true if `lhs` is not equal to `rhs`, false otherwise.

#### 11.2.5.1.1.43 Non-member lexicographical comparisons

```
template <typename T, typename Allocator>
bool operator<(<const concurrent_vector<T, Allocator>& lhs,
                const concurrent_vector<T, Allocator>& rhs );
```

**Returns:** true if lhs is lexicographically *less* than rhs, false otherwise.

```
template <typename T, typename Allocator>
bool operator<=(<const concurrent_vector<T, Allocator>& lhs,
                  const concurrent_vector<T, Allocator>& rhs );
```

**Returns:** true if lhs is lexicographically *less or equal* than rhs, false otherwise.

```
template <typename T, typename Allocator>
bool operator>(<const concurrent_vector<T, Allocator>& lhs,
                 const concurrent_vector<T, Allocator>& rhs );
```

**Returns:** true if lhs is lexicographically *greater* than rhs, false otherwise.

```
template <typename T, typename Allocator>
bool operator>=(<const concurrent_vector<T, Allocator>& lhs,
                   const concurrent_vector<T, Allocator>& rhs );
```

**Returns:** true if lhs is lexicographically *greater or equal* than rhs, false otherwise.

#### 11.2.5.1.1.44 Non-member swap

```
template <typename T, typename Allocator>
void swap(<concurrent_vector<T, Allocator>& lhs,
           concurrent_vector<T, Allocator>& rhs );
```

Equivalent to lhs.swap(rhs).

#### 11.2.5.1.1.45 Other

##### 11.2.5.1.1.46 Deduction guides

Where possible, constructors of concurrent\_vector supports class template argument deduction (since C++17):

```
template <typename InputIterator,
          typename Allocator = cache_aligned_allocator<iterator_value_t<InputIterator>
          >>>
concurrent_vector( InputIterator, InputIterator,
                  const Allocator& = Allocator() )
-> concurrent_vector<iterator_value_t<InputIterator>,
                  Allocator>;
```

Where type alias iterator\_value\_t defines as follows:

```
template <typename InputIterator>
using iterator_value_t = typename std::iterator_traits<InputIterator>::value_type;
```

#### Example

```
#include <tbb/concurrent_vector.h>
#include <array>
#include <memory>

int main() {
    std::array<int, 100> arr;

    // Deduces cv1 as tbb::concurrent_vector<int>
    tbb::concurrent_vector cv1(arr.begin(), arr.end());

    std::allocator<int> alloc;

    // Deduces cv2 as tbb::concurrent_vector<int, std::allocator<int>>
    tbb::concurrent_vector cv2(arr.begin(), arr.end(), alloc);
}
```

## 11.2.5.2 Queues

### 11.2.5.2.1 concurrent\_queue

[**containers.concurrent\_queue**]

`tbb::concurrent_queue` is a class template for an unbounded first-in-first out data structure that permits multiple threads to concurrently push and pop items.

#### 11.2.5.2.1.1 Class Template Synopsis

```
// Defined in header <tbb/concurrent_queue.h>

namespace tbb {

    template <typename T, typename Allocator = cache_aligned_allocator<T>>
    class concurrent_queue {
        public:
            using value_type = T;
            using reference = T&;
            using const_reference = const T&;
            using pointer = typename std::allocator_traits<Allocator>::pointer;
            using const_pointer = typename std::allocator_traits<Allocator>::const_
→pointer;
            using allocator_type = Allocator;

            using size_type = <implementation-defined unsigned integer type>;
            using difference_type = <implementation-defined signed integer type>;

            using iterator = <implementation-defined ForwardIterator>;
            using const_iterator = <implementation-defined constant ForwardIterator>;

            // Construction, destruction, copying
            concurrent_queue();

            explicit concurrent_queue( const allocator_type& alloc );

            template <typename InputIterator>
```

(continues on next page)

(continued from previous page)

```

concurrent_queue( InputIterator first, InputIterator last,
                  const allocator_type& alloc = allocator_type() );

concurrent_queue( const concurrent_queue& other );
concurrent_queue( const concurrent_queue& other, const allocator_type& alloc
    → );

concurrent_queue( concurrent_queue&& other );
concurrent_queue( concurrent_queue&& other, const allocator_type& alloc );

~concurrent_queue();

void push( const value_type& value );
void push( value_type&& value );

template <typename... Args>
void emplace( Args&&... args );

bool try_pop( value_type& result );

allocator_type get_allocator() const;

size_type unsafe_size() const;
bool empty() const;

void clear();

iterator unsafe_begin();
const_iterator unsafe_begin() const;
const_iterator unsafe_cbegin() const;

iterator unsafe_end();
const_iterator unsafe_end() const;
const_iterator unsafe_cend() const;
}; // class concurrent_queue

} // namespace tbb

```

Requirements:

- The type `T` shall meet the `Erasable` requirements from [container.requirements] ISO C++ Standard section. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Allocator` shall meet the `Allocator` requirements from [allocator.requirements] ISO C++ Standard section.

### 11.2.5.2.1.2 Member functions

### 11.2.5.2.1.3 Construction, destruction, copying

### 11.2.5.2.1.4 Empty container constructors

```

concurrent_queue();

explicit concurrent_queue( const allocator_type& alloc );

```

Constructs empty concurrent\_queue. If provided uses the allocator alloc to allocate the memory.

#### 11.2.5.2.1.5 Constructor from the sequence of elements

```
template <typename InputIterator>
concurrent_queue( InputIterator first, InputIterator last,
                  const allocator_type& alloc = allocator_type() );
```

Constructs a concurrent\_queue containing all elements from the half-open interval [first, last) using the allocator alloc to allocate the memory.

**Requirements:** the type InputIterator shall meet the *InputIterator* requirements from [input.iterators] ISO C++ Standard section.

#### 11.2.5.2.1.6 Copying constructors

```
concurrent_queue( const concurrent_queue& other );
concurrent_queue( const concurrent_queue& other,
                  const allocator_type& alloc );
```

Constructs a copy of other.

If the allocator argument is not provided, it is obtained by std::allocator\_traits<allocator\_type>::select\_on\_container\_copy\_construction(other.get\_allocator()).

The behavior is undefined in case of concurrent operations with other.

#### 11.2.5.2.1.7 Moving constructors

```
concurrent_queue( concurrent_queue&& other );
concurrent_queue( concurrent_queue&& other,
                  const allocator_type& alloc );
```

Constructs a concurrent\_queue with the content of other using move semantics.

other is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by std::move(other.get\_allocator()).

The behavior is undefined in case of concurrent operations with other.

### 11.2.5.2.1.8 Destructor

```
~concurrent_queue();
```

Destroys the `concurrent_queue`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

### 11.2.5.2.1.9 Concurrently safe member functions

All member functions in this section can be performed concurrently with each other.

### 11.2.5.2.1.10 Pushing elements

```
void push( const value_type& value );
```

Pushes a copy of `value` into the container.

**Requirements:** the type `T` shall meet the `CopyInsertable` requirements from [container.requirements] ISO C++ Standard section.

```
void push( value_type&& value );
```

Pushes `value` into the container using move semantics.

**Requirements:** the type `T` shall meet the `MoveInsertable` requirements from [container.requirements] ISO C++ Standard section.

`value` is left in a valid, but unspecified state.

```
template <typename... Args>
void emplace( Args&&... args );
```

Pushes a new element constructed from `args` into the container.

**Requirements:** the type `T` shall meet the `EmplaceConstructible` requirements from [container.requirements] ISO C++ Standard section.

### 11.2.5.2.1.11 Popping elements

```
bool try_pop( value_type& value );
```

If the container is empty, does nothing.

Otherwise, copies the last element from the container and assigns it to the `value`. The popped element is destroyed.

**Requirements:** the type `T` shall meet the `MoveAssignable` requirements from [moveassignable] ISO C++ Standard section.

**Returns:** `true` if the element was popped, `false` otherwise.

### 11.2.5.2.1.12 get\_allocator

```
allocator_type get_allocator() const;
```

**Returns:** a copy of the allocator, associated with `*this`.

### 11.2.5.2.1.13 Concurrently unsafe member functions

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these methods with other (either concurrently safe) methods.

### 11.2.5.2.1.14 The number of elements

```
size_type unsafe_size() const;
```

**Returns:** the number of elements in the container.

```
bool empty() const;
```

**Returns:** `true` if the container is empty, `false` otherwise.

### 11.2.5.2.1.15 clear

```
void clear();
```

Removes all elements from the container.

### 11.2.5.2.1.16 Iterators

The types `concurrent_queue::iterator` and `concurrent_queue::const_iterator` meets the requirements of `ForwardIterator` from [forward.iterators] ISO C++ Standard section.

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these methods with other (either concurrently safe) methods.

### 11.2.5.2.1.17 unsafe\_begin and unsafe\_cbegin

```
iterator unsafe_begin();
const_iterator unsafe_begin() const;
const_iterator unsafe_cbegin() const;
```

**Returns:** an iterator to the first element in the container.

### 11.2.5.2.1.18 unsafe\_end and unsafe\_cend

```
iterator unsafe_end();
const_iterator unsafe_end() const;
const_iterator unsafe_cend() const;
```

**Returns:** an iterator to the element which follows the last element in the container.

### 11.2.5.2.1.19 Other

#### 11.2.5.2.1.20 Deduction guides

Where possible, constructors of `tbb::concurrent_queue` supports class template argument deduction (since C++17):

```
template <typename InputIterator,
          typename Allocator = cache_aligned_allocator<iterator_value_t<InputIterator>>
        >
concurrent_queue( InputIterator, InputIterator, const Allocator& = Allocator() )
-> concurrent_queue<iterator_value_t<InputIterator>, Allocator>;
```

Where the type alias `iterator_value_t` defines as follows:

```
template <typename InputIterator>
using iterator_value_t = typename std::iterator_traits<InputIterator>::value_type;
```

#### Example

```
#include <tbb/concurrent_queue.h>
#include <vector>
#include <memory>

int main() {
    std::vector<int> vec;

    // Deduces cq1 as tbb::concurrent_queue<int>
    tbb::concurrent_queue cq1(vec.begin(), vec.end());

    // Deduces cq2 as tbb::concurrent_queue<int, std::allocator<int>>
    tbb::concurrent_queue cq2(vec.begin(), vec.end(), std::allocator<int>{})
}
```

### 11.2.5.2.2 concurrent\_bounded\_queue

#### [containers.concurrent\_bounded\_queue]

`tbb::concurrent_bounded_queue` is a class template for a bounded first-in-first out data structure that permits multiple threads to concurrently push and pop items.

#### 11.2.5.2.2.1 Class Template Synopsis

```
// Defined in header <tbb/concurrent_queue.h>

namespace tbb {

    template <typename T, typename Allocator = cache_aligned_allocator<T>>
    class concurrent_bounded_queue {
    public:
        using value_type = T;
        using reference = T&;
        using const_reference = const T&;
        using pointer = typename std::allocator_traits<Allocator>::pointer;
        using const_pointer = typename std::allocator_traits<Allocator>::const_
→pointer;

        using allocator_type = Allocator;

        using size_type = <implementation-defined signed integer type>;
        using difference_type = <implementation-defined signed integer type>;

        using iterator = <implementation-defined ForwardIterator>;
        using const_iterator = <implementation-defined constant ForwardIterator>;

        concurrent_bounded_queue();

        explicit concurrent_bounded_queue( const allocator_type& alloc );

        template <typename InputIterator>
        concurrent_bounded_queue( InputIterator first, InputIterator last,
                               const allocator_type& alloc = allocator_type() );

        concurrent_bounded_queue( const concurrent_bounded_queue& other );
        concurrent_bounded_queue( const concurrent_bounded_queue& other,
                               const allocator_type& alloc );

        concurrent_bounded_queue( concurrent_bounded_queue&& other );
        concurrent_bounded_queue( concurrent_bounded_queue&& other,
                               const allocator_type& alloc );

        ~concurrent_bounded_queue();

        allocator_type get_allocator() const;

        void push( const value_type& value );
        void push( value_type&& value );

        bool try_push( const value_type& value );
        bool try_push( value_type&& value );
    };
}
```

(continues on next page)

(continued from previous page)

```

template <typename... Args>
void emplace( Args&&... args );

template <typename... Args>
bool try_emplace( Args&&... args );

void pop( value_type& result );

bool try_pop( value_type& result );

void abort();

size_type size() const;

bool empty() const;

size_type capacity() const;
void set_capacity( size_type new_capacity );

void clear();

iterator unsafe_begin();
const_iterator unsafe_begin() const;
const_iterator unsafe_cbegin() const;

iterator unsafe_end();
const_iterator unsafe_end() const;
const_iterator unsafe_cend() const;
};

// class concurrent_bounded_queue

} // namespace tbb

```

Requirements:

- The type `T` shall meet the Erasable requirements from [container.requirements] ISO C++ Standard section. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Allocator` shall meet the `Allocator` requirements from [allocator.requirements] ISO C++ Standard section.

#### 11.2.5.2.2.2 Member functions

#### 11.2.5.2.2.3 Construction, destruction, copying

#### 11.2.5.2.2.4 Empty container constructors

```

concurrent_bounded_queue();

explicit concurrent_bounded_queue( const allocator_type& alloc );

```

Constructs empty `concurrent_bounded_queue` with an unbounded capacity. If provided uses the allocator `alloc` to allocate the memory.

### 11.2.5.2.2.5 Constructor from the sequence of elements

```
template <typename InputIterator>
concurrent_bounded_queue( InputIterator first, InputIterator last,
                          const allocator_type& alloc = allocator_type() );
```

Constructs a concurrent\_bounded\_queue with an unbounded capacity and containing all elements from the half-open interval [first, last) using the allocator alloc to allocate the memory.

**Requirements:** the type InputIterator shall meet the *InputIterator* requirements from [input.iterators] ISO C++ Standard section.

### 11.2.5.2.2.6 Copying constructors

```
concurrent_bounded_queue( const concurrent_bounded_queue& other );
concurrent_bounded_queue( const concurrent_bounded_queue& other,
                          const allocator_type& alloc );
```

Constructs a copy of other.

If the allocator argument is not provided, it is obtained by std::allocator\_traits<allocator\_type>::select\_on\_container\_copy\_construction(other.get\_allocator()).

The behavior is undefined in case of concurrent operations with other.

### 11.2.5.2.2.7 Moving constructors

```
concurrent_bounded_queue( concurrent_bounded_queue&& other );
concurrent_bounded_queue( concurrent_bounded_queue&& other,
                          const allocator_type& alloc );
```

Constructs a concurrent\_bounded\_queue with the content of other using move semantics.

other is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by std::move(other.get\_allocator()).

The behavior is undefined in case of concurrent operations with other.

### 11.2.5.2.2.8 Destructor

```
~concurrent_bounded_queue();
```

Destroys the concurrent\_bounded\_queue. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with \*this.

### 11.2.5.2.2.9 Concurrently safe member functions

All member functions in this section can be performed concurrently with each other.

### 11.2.5.2.2.10 Pushing elements

```
void push( const value_type& value );
```

Waits until the number of items in the queue is less than the capacity and pushes a copy of `value` into the container.

**Requirements:** the type `T` shall meet the `CopyInsertable` requirements from [container.requirements] ISO C++ Standard section.

```
bool try_push( const value_type& value );
```

If the number of items in the queue is less than the capacity, pushes a copy of `value` into the container.

**Requirements:** the type `T` shall meet the `CopyInsertable` requirements from [container.requirements] ISO C++ Standard section.

**Returns:** `true` if the item was pushed, `false` otherwise.

```
void push( value_type&& value );
```

Waits until the number of items in the queue is less than `capacity()` and pushes `value` into the container using move semantics.

**Requirements:** the type `T` shall meet the `MoveInsertable` requirements from [container.requirements] ISO C++ Standard section.

`value` is left in a valid, but unspecified state.

```
bool try_push( value_type&& value );
```

If the number of items in the queue is less than the capacity, pushes `value` into the container using move semantics.

**Requirements:** the type `T` shall meet the `MoveInsertable` requirements from [container.requirements] ISO C++ Standard section.

`value` is left in a valid, but unspecified state.

**Returns:** `true` if the item was pushed, `false` otherwise.

```
template <typename... Args>
void emplace( Args&&... args );
```

Waits until the number of items in the queue is less than `capacity()` and pushes a new element constructed from `args` into the container.

**Requirements:** the type `T` shall meet the `EmplaceConstructible` requirements from [container.requirements] ISO C++ Standard section.

```
template <typename... Args>
bool try_emplace( Args&&... args );
```

If the number of items in the queue is less than the capacity, pushes a new element constructed from `args` into the container.

**Requirements:** the type `T` shall meet the `EmplaceConstructible` requirements from [container.requirements] ISO C++ Standard section.

**Returns:** `true` if the item was pushed, `false` otherwise.

#### 11.2.5.2.2.11 Popping elements

```
void pop( value_type& value );
```

Waits until the item becomes available, copies it from the container and assigns it to the `value`. The popped element is destroyed.

**Requirements:** the type `T` shall meet the `MoveAssignable` requirements from [moveassignable] ISO C++ Standard section.

```
bool try_pop( value_type& value );
```

If the container is empty, does nothing.

Otherwise, copies the last element from the container and assigns it to the `value`. The popped element is destroyed.

**Requirements:** the type `T` shall meet the `MoveAssignable` requirements from [moveassignable] ISO C++ Standard section.

**Returns:** `true` if the element was popped, `false` otherwise.

#### 11.2.5.2.2.12 abort

```
void abort();
```

Wakes up any threads that are waiting on the queue via `push`, `pop` or `emplace` operations and raises `tbb::user_abort` exception on those threads.

#### 11.2.5.2.2.13 Capacity of the queue

```
size_type capacity() const;
```

**Returns:** the maximum number of items that the queue can hold.

```
void set_capacity( size_type new_capacity ) const;
```

Sets the maximum number of items that the queue can hold to `new_capacity`.

#### 11.2.5.2.2.14 get\_allocator

```
allocator_type get_allocator() const;
```

**Returns:** a copy of the allocator, associated with `*this`.

#### 11.2.5.2.2.15 Concurrently unsafe member functions

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these methods with other (either concurrently safe) methods.

#### 11.2.5.2.2.16 The number of elements

```
size_type size() const;
```

**Returns:** the number of elements in the container.

```
bool empty() const;
```

**Returns:** `true` if the container is empty, `false` otherwise.

#### 11.2.5.2.2.17 clear

```
void clear();
```

Removes all elements from the container.

#### 11.2.5.2.2.18 Iterators

The types `concurrent_bounded_queue::iterator` and `concurrent_bounded_queue::const_iterator` meets the requirements of `ForwardIterator` from [forward.iterators] ISO C++ Standard section.

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these methods with other (either concurrently safe) methods.

#### 11.2.5.2.2.19 unsafe\_begin and unsafe\_cbegin

```
iterator unsafe_begin();
const_iterator unsafe_begin() const;
const_iterator unsafe_cbegin() const;
```

**Returns:** an iterator to the first element in the container.

### 11.2.5.2.2.20 unsafe\_end and unsafe\_cend

```
iterator unsafe_end();
const_iterator unsafe_end() const;
const_iterator unsafe_cend() const;
```

**Returns:** an iterator to the element which follows the last element in the container.

### 11.2.5.2.2.21 Other

#### 11.2.5.2.2.22 Deduction guides

Where possible, constructors of `tbb::concurrent_bounded_queue` supports class template argument deduction (since C++17):

```
template <typename InputIterator,
          typename Allocator = cache_aligned_allocator<iterator_value_t<InputIterator>>
        >
concurrent_bounded_queue( InputIterator, InputIterator, const Allocator& = Allocator()
                         )
-> concurrent_bounded_queue<iterator_value_t<InputIterator>, Allocator>;
```

Where the type alias `iterator_value_t` defines as follows:

```
template <typename InputIterator>
using iterator_value_t = typename std::iterator_traits<InputIterator>::value_type;
```

#### Example

```
#include <tbb/concurrent_queue.h>
#include <vector>
#include <memory>

int main() {
    std::vector<int> vec;

    // Deduces cq1 as tbb::concurrent_bounded_queue<int>
    tbb::concurrent_bounded_queue cq1(vec.begin(), vec.end());

    // Deduces cq2 as tbb::concurrent_bounded_queue<int, std::allocator<int>>
    tbb::concurrent_bounded_queue cq2(vec.begin(), vec.end(), std::allocator<int>{})
}
```

### 11.2.5.2.3 concurrent\_priority\_queue

#### [containers.concurrent\_priority\_queue]

`tbb::concurrent_priority_queue` is a class template for an unbounded priority queue that permits multiple threads to concurrently push and pop items. Items are popped in a priority order.

#### 11.2.5.2.3.1 Class Template Synopsis

```
namespace tbb {

    template <typename T, typename Compare = std::less<T>,
              typename Allocator = cache_aligned_allocator<T>>
    class concurrent_priority_queue {
public:
    using value_type = T;
    using reference = T&;
    using const_reference = const T&;
    using size_type = <implementation-defined unsigned integer type>;
    using difference_type = <implementation-defined signed integer type>;
    using allocator_type = Allocator;

    concurrent_priority_queue();
    explicit concurrent_priority_queue( const allocator_type& alloc );

    explicit concurrent_priority_queue( const Compare& compare,
                                       const allocator_type& alloc = allocator_
                                       ↪type() );

    explicit concurrent_priority_queue( size_type init_capacity, const allocator_
                                       ↪type& alloc = allocator_type() );

    explicit concurrent_priority_queue( size_type init_capacity, const Compare&_
                                       ↪compare,
                                       const allocator_type& alloc = allocator_
                                       ↪type() );

    template <typename InputIterator>
    concurrent_priority_queue( InputIterator first, InputIterator last,
                               const allocator_type& alloc = allocator_type() );

    template <typename InputIterator>
    concurrent_priority_queue( InputIterator first, InputIterator last,
                               const Compare& compare, const allocator_type&_
                               ↪alloc = allocator_type() );

    concurrent_priority_queue( std::initializer_list<value_type> init,
                               const allocator_type& alloc = allocator_type() );

    concurrent_priority_queue( std::initializer_list<value_type> init,
                               const Compare& compare, const allocator_type&_
                               ↪alloc = allocator_type() );

    concurrent_priority_queue( const concurrent_priority_queue& other );
    concurrent_priority_queue( const concurrent_priority_queue& other, const_
                               ↪allocator_type& alloc );
};
```

(continues on next page)

(continued from previous page)

```

concurrent_priority_queue( concurrent_priority_queue&& other );
concurrent_priority_queue( concurrent_priority_queue&& other, const allocator_
→type& alloc );

~concurrent_priority_queue();

concurrent_priority_queue& operator=( const concurrent_priority_queue& other_
→);
concurrent_priority_queue& operator=( concurrent_priority_queue&& other );
concurrent_priority_queue& operator=( std::initializer_list<value_type> init_
→);

template <typename InputIterator>
void assign( InputIterator first, InputIterator last );

void assign( std::initializer_list<value_type> init );

void swap( concurrent_priority_queue& other );

allocator_type get_allocator() const;

void clear();

bool empty() const;
size_type size() const;

void push( const value_type& value );
void push( value_type&& value );

template <typename... Args>
void emplace( Args&&... args );

bool try_pop( value_type& value );
}; // class concurrent_priority_queue

}; // namespace tbb

```

#### Requirements:

- The type `T` shall meet the `Erasable` requirements from [container.requirements] ISO C++ Standard section. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Compare` shall meet the `Compare` requirements from [alg.sorting] ISO C++ Standard section.
- The type `Allocator` shall meet the `Allocator` requirements from [allocator.requirements] ISO C++ Standard section.

### 11.2.5.2.3.2 Member functions

#### 11.2.5.2.3.3 Construction, destruction, copying

#### 11.2.5.2.3.4 Empty container constructors

```
concurrent_priority_queue();

explicit concurrent_priority_queue( const allocator_type& alloc );

explicit concurrent_priority_queue( const Compare& compare, const allocator_
    ↵type& alloc );
```

Constructs empty concurrent\_priority\_queue. The initial capacity is unspecified. If provided uses the predicate compare for priority comparisons and the allocator alloc to allocate the memory.

```
concurrent_priority_queue( size_type init_capacity,
                           const allocator_type& alloc = allocator_type() );

concurrent_priority_queue( size_type init_capacity,
                           const Compare& compare,
                           const allocator_type& alloc = allocator_type() );
```

Constructs empty concurrent\_priority\_queue with the initial capacity init\_capacity. If provided uses the predicate compare for priority comparisons and the allocator alloc to allocate the memory.

#### 11.2.5.2.3.5 Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_priority_queue( InputIterator first, InputIterator last,
                           const allocator_type& alloc = allocator_type() );

template <typename InputIterator>
concurrent_priority_queue( InputIterator first, InputIterator last,
                           const Compare& compare,
                           const allocator_type& alloc = allocator_type() );
```

Constructs a concurrent\_priority\_queue containing all elements from the half-open interval [first, last).

If provided uses the predicate compare for priority comparisons and the allocator alloc to allocate the memory.

**Requirements:** the type InputIterator shall meet the *InputIterator* requirements from [input iterators] ISO C++ Standard section.

```
concurrent_priority_queue( std::initializer_list<value_type> init,
                           const allocator_type& alloc = allocator_type() );
```

Equivalent to concurrent\_priority\_queue(init.begin(), init.end(), alloc).

```
concurrent_priority_queue( std::initializer_list<value_type> init,
                           const Compare& compare,
                           const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_priority_queue(init.begin(), init.end(), compare, alloc)`.

#### 11.2.5.2.3.6 Copying constructors

```
concurrent_priority_queue( const concurrent_priority_queue& other );
concurrent_priority_queue( const concurrent_priority_queue& other,
const allocator_type& alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

#### 11.2.5.2.3.7 Moving constructors

```
concurrent_priority_queue( concurrent_priority_queue&& other );
concurrent_priority_queue( concurrent_priority_queue&& other,
const allocator_type& alloc );
```

Constructs a copy of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

#### 11.2.5.2.3.8 Destructor

```
~concurrent_priority_queue();
```

Destroys the `concurrent_priority_queue`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

#### 11.2.5.2.3.9 Assignment operators

```
concurrent_priority_queue& operator=( const concurrent_priority_queue& other _
_ );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment` is true.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

**Returns:** a reference to `*this`.

```
concurrent_priority_queue& operator=( concurrent_priority_queue&& other );
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment` is true.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

**Returns:** a reference to `*this`.

```
concurrent_priority_queue& operator=( std::initializer_list<value_type> init_<br/> );
```

Replaces all elements in `*this` by the elements in `init`.

The behavior is undefined in case of concurrent operations with `*this`.

**Returns:** a reference to `*this`.

### 11.2.5.2.3.10 assign

```
template <typename InputIterator>
void assign( InputIterator first, InputIterator last );
```

Replaces all elements in `*this` by the elements in the half-open interval `[first, last)`.

The behavior is undefined in case of concurrent operations with `*this`.

**Requirements:** the type `InputIterator` shall meet the `InputIterator` requirements from [input iterators] ISO C++ Standard section.

```
void assign( std::initializer_list<value_type> init );
```

Equivalent to `assign(init.begin(), init.end())`.

### 11.2.5.2.3.11 Size and capacity

### 11.2.5.2.3.12 empty

```
bool empty() const;
```

**Returns:** true if the container is empty, false otherwise.

The result may differ with the actual container state in case of pending concurrent push or `try_pop` operations.

### 11.2.5.2.3.13 size

```
size_type size() const;
```

**Returns:** the number of elements in the container.

The result may differ with the actual number of elements in case of pending concurrent push or try\_pop operations.

### 11.2.5.2.3.14 Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other.

### 11.2.5.2.3.15 Pushing elements

```
void push( const value_type& value );
```

Pushes a copy of value into the container.

**Requirements:** the type T shall meet the CopyInsertable requirements from [container.requirements] and the CopyAssignable requirements from [copyassignable] ISO C++ Standard sections.

```
void push( value_type&& value );
```

Pushes value into the container using move semantics.

**Requirements:** the type T shall meet the MoveInsertable requirements from [container.requirements] and the MoveAssignable requirements from [moveassignable] ISO C++ Standard sections.

value is left in a valid, but unspecified state.

```
template <typename... Args>
void emplace( Args&&... args );
```

Pushes a new element constructed from args into the container.

**Requirements:** the type T shall meet the EmplaceConstructible requirements from [container.requirements] and the MoveAssignable requirements from [moveassignable] ISO C++ Standard sections.

### 11.2.5.2.3.16 Popping elements

```
bool try_pop( value_type& value )
```

If the container is empty, does nothing.

Otherwise, copies the highest priority element from the container and assigns it to the value. The popped element is destroyed.

**Requirements:** the type T shall meet the MoveAssignable requirements from [moveassignable] ISO C++ Standard section.

**Returns:** true if the element was popped, false otherwise.

### 11.2.5.2.3.17 Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these methods with other (either concurrently safe) methods.

### 11.2.5.2.3.18 clear

```
void clear();
```

Removes all elements from the container.

### 11.2.5.2.3.19 swap

```
void swap( concurrent_priority_queue& other );
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is true.

Otherwise if `get_allocator() != other.get_allocator()` the behavior is undefined.

### 11.2.5.2.3.20 Non-member functions

These functions provides binary comparison and swap operations on `tbb::concurrent_priority_queue` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `tbb::concurrent_priority_queue` as a type alias for which the non-member functions are reachable only via argument dependent lookup.

```
template <typename T, typename Compare, typename Allocator>
void swap( concurrent_priority_queue<T, Compare, Allocator>& lhs,
            concurrent_priority_queue<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_priority_queue<T, Compare, Allocator>& lhs,
                    const concurrent_priority_queue<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_priority_queue<T, Compare, Allocator>& lhs,
                    const concurrent_priority_queue<T, Compare, Allocator>& rhs );
```

### 11.2.5.2.3.21 Non-member swap

```
template <typename T, typename Compare, typename Allocator>
void swap( concurrent_priority_queue<T, Compare, Allocator>& lhs,
           concurrent_priority_queue<T, Compare, Allocator>& rhs );
```

Equivalent to `lhs.swap(rhs)`.

### 11.2.5.2.3.22 Non-member binary comparisons

```
template <typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_priority_queue<T, Compare, Allocator>& lhs,
                   const concurrent_priority_queue<T, Compare, Allocator>& rhs );
```

Checks if `lhs` is equal to `rhs`, that is they have the same number of elements and `lhs` contains all elements from `rhs` with the same priority.

**Returns:** true if `lhs` is equal to `rhs`, false otherwise.

```
template <typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_priority_queue<T, Compare, Allocator>& lhs,
                   const concurrent_priority_queue<T, Compare, Allocator>& rhs );
```

Equivalent to `!(lhs == rhs)`.

**Returns:** true if `lhs` is not equal to `rhs`, false otherwise.

### 11.2.5.2.3.23 Other

#### 11.2.5.2.3.24 Deduction guides

Where possible, constructors of `tbb::concurrent_priority_queue` supports class template argument deduction (since C++17):

```
template <typename InputIterator>
concurrent_priority_queue( InputIterator, InputIterator )
-> concurrent_priority_queue<iterator_value_t<InputIterator>>;

template <typename InputIterator, typename Compare>
concurrent_priority_queue( InputIterator, InputIterator, const Compare& )
-> concurrent_priority_queue<iterator_value_t<InputIterator>,
                           Compare>;

template <typename InputIterator, typename Allocator>
concurrent_priority_queue( InputIterator, InputIterator, const Allocator& alloc )
-> concurrent_priority_queue<iterator_value_t<InputIterator>,
                           std::less<iterator_value_t<InputIterator>,
                           Allocator>;

template <typename InputIterator, typename Compare, typename Allocator>
concurrent_priority_queue( InputIterator, InputIterator, const Compare&,
                           const Allocator& )
-> concurrent_priority_queue<iterator_value_t<InputIterator>,
                           Compare, Allocator>;
```

(continues on next page)

(continued from previous page)

```

template <typename T>
concurrent_priority_queue( std::initializer_list<T> )
-> concurrent_priority_queue<T>;

template <typename T, typename Compare>
concurrent_priority_queue( std::initializer_list<T>, const Compare& )
-> concurrent_priority_queue<T, Compare>;

template <typename T, typename Allocator>
concurrent_priority_queue( std::initializer_list<T>, const Allocator& )
-> concurrent_priority_queue<T, std::less<T>, Allocator>;

template <typename T, typename Compare, typename Allocator>
concurrent_priority_queue( std::initializer_list<T>, const Compare&, const Allocator&  
↔ )
-> concurrent_priority_queue<T, Compare, Allocator>;

```

Where the type alias `iterator_value_t` defines as follows:

```

template <typename InputIterator>
using iterator_value_t = typename std::iterator_traits<InputIterator>::value_type;

```

### Example

```

#include <tbb/concurrent_priority_queue.h>
#include <vector>
#include <functional>

int main() {
    std::vector<int> vec;

    // Deduces cpq1 as tbb::concurrent_priority_queue<int>
    tbb::concurrent_priority_queue cpq1(vec.begin(), vec.end());

    // Deduces cpq2 as tbb::concurrent_priority_queue<int, std::greater>
    tbb::concurrent_priority_queue cpq2(vec.begin(), vec.end(), std::greater{});
}

```

## 11.2.5.3 Unordered associative containers

### 11.2.5.3.1 concurrent\_hash\_map

#### [containers.concurrent\_hash\_map]

`concurrent_hash_map` is a class template for an unordered associative container which holds key-value pairs with unique keys and supports concurrent insertion, lookup and erasure.

### 11.2.5.3.1.1 Class Template Synopsis

```
// Defined in header <tbb/concurrent_hash_map.h>

namespace tbb {

    template <typename Key, typename T,
              typename HashCompare = tbb_hash_compare<Key>,
              typename Allocator = tbb_allocator<std::pair<const Key, T>>>
    class concurrent_hash_map {
public:
    using key_type = Key;
    using mapped_type = T;
    using value_type = std::pair<const Key, T>;

    using reference = value_type&;
    using const_reference = const value_type&;
    using pointer = typename std::allocator_traits<Allocator>::pointer;
    using const_pointer = typename std::allocator_traits<Allocator>
        ::const_pointer;

    using allocator_type = Allocator;

    using size_type = <implementation-defined unsigned integer type>;
    using difference_type = <implementation-defined signed integer type>;
};

    using iterator = <implementation-defined ForwardIterator>;
    using const_iterator = <implementation-defined constant_
        ForwardIterator>;

    using range_type = <implementation-defined ContainerRange>;
    using const_range_type = <implementation-defined constant_
        ContainerRange>;

    class accessor;
    class const_accessor;

// Construction, destruction, copying
concurrent_hash_map();

    explicit concurrent_hash_map( const HashCompare& compare,
                                  const allocator_type& alloc =
        allocator_type() );

    explicit concurrent_hash_map( const allocator_type& alloc );

    concurrent_hash_map( size_type n, const HashCompare& compare,
                         const allocator_type& alloc = allocator_type() );
};

    concurrent_hash_map( size_type n, const allocator_type& alloc =
        allocator_type() );

    template <typename InputIterator>
    concurrent_hash_map( InputIterator first, InputIterator last,
                        const HashCompare& compare,
```

(continues on next page)

(continued from previous page)

```

        const allocator_type& alloc = allocator_type()_  

    );  
  

template <typename InputIterator>  
concurrent_hash_map( InputIterator first, InputIterator last,  

                     const allocator_type& alloc = allocator_type()_  

    );  
  

concurrent_hash_map( std::initializer_list<value_type> init,  

                     const HashCompare& compare,  

                     const allocator_type& alloc = allocator_type()_  

    );  
  

concurrent_hash_map( std::initializer_list<value_type> init,  

                     const allocator_type& alloc = allocator_type()_  

    );  
  

concurrent_hash_map( const concurrent_hash_map& other );  

concurrent_hash_map( const concurrent_hash_map& other,  

                     const allocator_type& alloc );  
  

concurrent_hash_map( concurrent_hash_map&& other );  

concurrent_hash_map( concurrent_hash_map&& other,  

                     const allocator_type& alloc );  
  

~concurrent_hash_map();  
  

concurrent_hash_map& operator=( const concurrent_hash_map& other );  

concurrent_hash_map& operator=( concurrent_hash_map&& other );  

concurrent_hash_map& operator=( std::initializer_list<value_type>_  

    init );  
  

allocator_type get_allocator() const;  
  

// Concurrently unsafe modifiers  

void clear();  
  

void swap( concurrent_hash_map& other );  
  

// Hash policy  

void rehash( size_type sz = 0 );  

size_type bucket_count() const;  
  

// Size and capacity  

size_type size() const;  

bool empty() const;  

size_type max_size() const;  
  

// Lookup  

bool find( const_accessor& result, const key_type& key ) const;  

bool find( accessor& result, const key_type& key );  
  

size_type count( const key_type& key ) const;  
  

// Modifiers  

bool insert( const_accessor& result, const key_type& key );  

bool insert( accessor& result, const key_type& key );

```

(continues on next page)

(continued from previous page)

```

bool insert( const_accessor& result, const value_type& value );
bool insert( accessor& result, const value_type& value );

bool insert( const_accessor& result, value_type&& value );
bool insert( accessor& result, value_type&& value );

bool insert( const value_type& value );
bool insert( value_type&& value );

template <typename InputIterator>
void insert( InputIterator first, InputIterator last );

void insert( std::initializer_list<value_type> init );

template <typename... Args>
bool emplace( const_accessor& result, Args&&... args );

template <typename... Args>
bool emplace( accessor& result, Args&&... args );

template <typename... Args>
bool emplace( Args&&... args );

bool erase( const key_type& key );

bool erase( const_accessor& item_accessor );
bool erase( accessor& item_accessor );

// Iterators
iterator begin();
const_iterator begin() const;
const_iterator cbegin() const;

iterator end();
const_iterator end() const;
const_iterator cend() const;

std::pair<iterator, iterator> equal_range( const key_type& key );
std::pair<const_iterator, const_iterator> equal_range( const key_
type& key ) const;

// Parallel iteration
range_type range( std::size_t grainsize = 1 );
const_range_type range( std::size_t grainsize = 1 ) const;
}; // class concurrent_hash_map

} // namespace tbb

```

Requirements:

- The expression `std::allocator_type<Allocator>::destroy(m, val)` where `m` is an object of the type `Allocator` and `val` is an object of type `value_type` should be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `HashCompare` shall meet the *HashCompare requirements*.
- The type `Allocator` shall meet the `Allocator` requirements from [allocator.requirements] ISO C++ Stan-

dard section.

### 11.2.5.3.1.2 Member classes

#### 11.2.5.3.1.3 accessor and const\_accessor

Member classes `concurrent_hash_map::accessor` and `concurrent_hash_map::const_accessor` are called *accessors*. Accessors allows multiple threads to concurrently access the key-value pairs in `concurrent_hash_map`. Accessor is called *empty* if it does not point to any item.

#### 11.2.5.3.1.4 accessor member class

Member class `concurrent_hash_map::accessor` provides read-write access to the key-value pair in `concurrent_hash_map`.

```
namespace tbb {

    template <typename Key, typename T, typename HashCompare, typename Allocator>
    class concurrent_hash_map<Key, T, HashCompare, Allocator>::accessor {
        using value_type = std::pair<const Key, T>

        accessor();
        ~accessor();

        bool empty() const;
        value_type& operator*() const;
        value_type* operator->() const;

        void release();
    }; // class accessor

} // namespace tbb
```

#### 11.2.5.3.1.5 const\_accessor member class

Member class `concurrent_hash_map::const_accessor` provides read only access to the key-value pair in `concurrent_hash_map`.

```
namespace tbb {

    template <typename Key, typename T, typename HashCompare, typename Allocator>
    class concurrent_hash_map<Key, T, HashCompare, Allocator>::const_accessor {
        using value_type = const std::pair<const Key, T>

        const_accessor();
        ~const_accessor();

        bool empty() const;
        value_type& operator*() const;
        value_type* operator->() const;

        void release();
    }; // class const_accessor

} // namespace tbb
```

(continues on next page)

(continued from previous page)

```
}; // class const_accessor  
} // namespace tbb
```

#### 11.2.5.3.1.6 Member functions

##### 11.2.5.3.1.7 Construction and destruction

```
accessor();  
const_accessor();
```

Constructs an empty accessor.

```
~accessor();  
~const_accessor();
```

Destroys the accessor. If `*this` is not empty - releases the ownership of the element.

##### 11.2.5.3.1.8 Emptiness

```
bool empty() const;
```

**Returns:** `true` if the accessor is empty, `false` otherwise.

##### 11.2.5.3.1.9 Key-value pair access

```
value_type& operator*() const;
```

**Returns:** a reference to the key-value pair on which the accessor points.

The behavior is undefined if the accessor is empty.

```
value_type* operator->() const;
```

**Returns:** a pointer to the key-value pair on which the accessor points.

The behavior is undefined if the accessor is empty.

### 11.2.5.3.1.10 Releasing

```
void release();
```

If `*this` is not empty, releases the ownership of the element. `*this` becomes empty.

### 11.2.5.3.1.11 Member functions

#### 11.2.5.3.1.12 Construction, destruction, copying

#### 11.2.5.3.1.13 Empty container constructors

```
concurrent_hash_map();

explicit concurrent_hash_map( const HashCompare& compare,
                               const allocator_type& alloc = allocator_type() );
                           ↵;

explicit concurrent_hash_map( const allocator_type& alloc );
```

Constructs an empty `concurrent_hash_map`. The initial number of buckets is unspecified.

If provided, uses the comparator `compare` to calculate hash codes and compare `key_type` objects for equality and the allocator `alloc` to allocate the memory.

---

```
concurrent_hash_map( size_type n, const HashCompare& compare,
                     const allocator_type& alloc = allocator_type() );

concurrent_hash_map( size_type n, const allocator_type& alloc = allocator_
                     ↵type() );
```

Constructs an empty `concurrent_hash_map` with `n` preallocated buckets.

If provided, uses the comparator `compare` to calculate hash codes and compare `key_type` objects for equality and the allocator `alloc` to allocate the memory.

#### 11.2.5.3.1.14 Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_hash_map( InputIterator first, InputIterator last,
                     const HashCompare& compare,
                     const allocator_type& alloc = allocator_type() );

template <typename InputIterator>
concurrent_hash_map( InputIterator first, InputIterator last,
                     const allocator_type& alloc = allocator_type() );
```

Constructs the `concurrent_hash_map` which contains the elements from the half-open interval `[first, last)`.

If the range `[first, last)` contains multiple elements with equal keys, it is unspecified which element would be inserted.

If provided, uses the comparator `compare` to calculate hash codes and compare `key_type` objects for equality and the allocator `alloc` to allocate the memory.

**Requirements:** the type `InputIterator` shall meet the requirements of `InputIterator` from [input.iterators] ISO C++ Standard section.

```
concurrent_hash_map( std::initializer_list<value_type> init,
                     const HashCompare& compare,
                     const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_hash_map(init.begin(), init.end(), compare, alloc)`.

```
concurrent_hash_map( std::initializer_list<value_type> init,
                     const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_hash_map(init.begin(), init.end(), alloc)`.

#### 11.2.5.3.1.15 Copying constructors

```
concurrent_hash_map( const concurrent_hash_map& other );
concurrent_hash_map( const concurrent_hash_map& other,
                     const allocator_type& alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

#### 11.2.5.3.1.16 Moving constructors

```
concurrent_hash_map( concurrent_hash_map&& other );
concurrent_hash_map( concurrent_hash_map&& other,
                     const allocator_type& alloc );
```

Constructs a `concurrent_hash_map` with the content of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

### 11.2.5.3.1.17 Destructor

```
~concurrent_hash_map();
```

Destroys the `concurrent_hash_map`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

### 11.2.5.3.1.18 Assignment operators

```
concurrent_hash_map& operator=( const concurrent_hash_map& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment` is true.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

**Returns:** a reference to `*this`.

```
concurrent_hash_map& operator=( concurrent_hash_map&& other );
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment` is true.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

**Returns:** a reference to `*this`.

```
concurrent_hash_map& operator=( std::initializer_list<value_type> init );
```

Replaces all elements in `*this` by the elements in `init`.

If `init` contains multiple elements with equal keys, it is unspecified which element would be inserted.

The behavior is undefined in case of concurrent operations with `*this`.

**Returns:** a reference to `*this`.

### 11.2.5.3.1.19 get\_allocator

```
allocator_type get_allocator() const;
```

**Returns:** a copy of the allocator associated with `*this`.

### 11.2.5.3.1.20 Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

#### 11.2.5.3.1.21 clear

```
void clear();
```

Removes all elements from the container.

#### 11.2.5.3.1.22 swap

```
void swap( concurrent_hash_map& other );
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is true.

Otherwise if `get_allocator() != other.get_allocator()` the behavior is undefined.

### 11.2.5.3.1.23 Hash policy

#### 11.2.5.3.1.24 Rehashing

```
void rehash( size_type n = 0 );
```

If `n > 0` sets the number of buckets to the value which is not less than `n`.

#### 11.2.5.3.1.25 bucket\_count

```
size_type bucket_count() const;
```

**Returns:** the number of buckets in the container.

#### 11.2.5.3.1.26 Size and capacity

#### 11.2.5.3.1.27 empty

```
bool empty() const;
```

**Returns:** true if the container is empty, false otherwise.

The result may differ with the actual container state in case of pending concurrent insertions or erasures.

### 11.2.5.3.1.28 size

```
size_type size() const;
```

**Returns:** the number of elements in the container.

The result may differ with the actual container state in case of pending concurrent insertions or erasures.

### 11.2.5.3.1.29 max\_size

```
size_type max_size() const;
```

**Returns:** The maximum number of elements that container can hold.

### 11.2.5.3.1.30 Lookup

All methods in this section can be executed concurrently with each other, and concurrently-safe modifiers.

#### 11.2.5.3.1.31 find

```
bool find( const_accessor& result, const key_type& key ) const;
bool find( accessor& result, const key_type& key );
```

If the accessor `result` is not empty - releases the `result`.

If an element with the key equal to `key` exists - sets the `result` to provide access to this element.

**Returns:** `true` if an element with the key equal to `key` was found, `false` otherwise.

#### 11.2.5.3.1.32 count

```
size_type count( const key_type& key ) const;
```

**Returns:** 1 if an element with the key equal to `key` exists, 0 otherwise.

#### 11.2.5.3.1.33 Concurrently safe modifiers

All methods in this section can be executed concurrently with each other, and lookup methods.

### 11.2.5.3.1.34 Inserting values

```
bool insert( const_accessor& result, const key_type& key );
bool insert( accessor& result, const key_type& key );
```

If the accessor `result` is not empty - releases the `result` and attempts to insert the value, constructed from `key`, `mapped_type()` into the container.

Sets the `result` to provide access to the inserted element or to the element with equal key which was already presented in the container.

**Requirements:**

- the type `value_type` shall meet the `EmplaceConstructible` requirements from [container.requirements] ISO C++ Standard section..
- the type `mapped_type` shall meet the `DefaultConstructible` requirements from [default-constructible] ISO C++ Standard section..

**Returns:** `true` if an element was inserted, `false` otherwise.

```
bool insert( const_accessor& result, const value_type& value );
bool insert( accessor& result, const value_type& value );
```

If the accessor `result` is not empty - releases the `result` and attempts to insert the value `value` into the container.

Sets the `result` to provide access to the inserted element or to the element with equal key which was already presented in the container.

**Requirements:** the type `value_type` shall meet the `CopyInsertable` requirements from [container.requirements] ISO C++ Standard section.

**Returns:** `true` if an element was inserted, `false` otherwise.

```
bool insert( const value_type& value );
```

Attempts to insert the value `value` into the container.

**Requirements:** the type `value_type` shall meet the `CopyInsertable` requirements from [container.requirements] ISO C++ Standard section.

**Returns:** `true` if an element was inserted, `false` otherwise.

```
bool insert( const_accessor& result, value_type&& value );
bool insert( accessor& result, value_type&& value );
```

If the accessor `result` is not empty - releases the `result` and attempts to insert the value `value` into the container using move semantics.

Sets the `result` to provide access to the inserted element or to the element with equal key which was already presented in the container.

`value` is left in a valid, but unspecified state.

**Requirements:** the type `value_type` shall meet the `MoveInsertable` requirements from [container.requirements] ISO C++ Standard section.

**Returns:** `true` if an element was inserted, `false` otherwise.

```
bool insert( value_type&& value );
```

Attempts to insert the value `value` into the container using move semantics.

**Requirements:** the type `value_type` shall meet the `MoveInsertable` requirements from [container.requirements] ISO C++ Standard section.

**Returns:** `true` if an element was inserted, `false` otherwise.

#### 11.2.5.3.1.35 Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last );
```

Attempts to insert all items from the half-open interval `[first, last)` into the container.

If the interval `[first, last)` contains multiple elements with equal keys, it is unspecified which element should be inserted.

**Requirements:** the type `InputIterator` must meet the requirements of `InputIterator` from [input.iterators] ISO C++ Standard section.

```
void insert( std::initializer_list<value_type> init );
```

Equivalent to `insert(init.begin(), init.end())`.

#### 11.2.5.3.1.36 Emplacing elements

```
template <typename... Args>
bool emplace( const_accessor& result, Args&&... args );

template <typename... Args>
bool emplace( accessor& result, Args&&... args );
```

If the accessor `result` is not empty - releases the `result` and attempts to insert an element constructed in-place from `args` into the container.

Sets the `result` to provide access to the inserted element or to the element with equal key which was already presented in the container.

**Requirements:** the type `value_type` shall meet the `EmplaceConstructible` requirements from [container.requirements] ISO C++ Standard section.

**Returns:** `true` if an element was inserted, `false` otherwise

```
template <typename... Args>
bool emplace( Args&&... args );
```

Attempts to insert an element constructed in-place from `args` into the container.

**Requirements:** the type `value_type` shall meet the `EmplaceConstructible` requirements from [container.requirements] ISO C++ Standard section.

**Returns:** `true` if an element was inserted, `false` otherwise

#### 11.2.5.3.1.37 Erasing elements

```
bool erase( const key_type& key );
```

If an element with the key equal to `key` exists - removes it from the container.

**Returns:** `true` if an element was removed, `false` otherwise.

```
bool erase( const_accessor& item_accessor );
bool erase( accessor& item_accessor );
```

Removes an element owned by `item_accessor` from the container.

**Requirements:** `item_accessor` should not be empty.

**Returns:** `true` if an element was removed by the current thread, `false` if it was removed by another thread.

#### 11.2.5.3.1.38 Iterators

The types `concurrent_hash_map::iterator` and `concurrent_hash_map::const_iterator` meets the requirements of `ForwardIterator` from [forward.iterators] ISO C++ Standard section.

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

#### 11.2.5.3.1.39 begin and cbegin

```
iterator begin();
const_iterator begin() const;
const_iterator cbegin() const;
```

**Returns:** an iterator to the first element in the container.

#### 11.2.5.3.1.40 end and cend

```
iterator end();
const_iterator end() const;
const_iterator cend() const;
```

**Returns:** an iterator to the element which follows the last element in the container.

#### 11.2.5.3.1.41 equal\_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );
std::pair<const_iterator, const_iterator> equal_range( const key_type& key )  
const;
```

If an element with the key equal to key exists in the container - a pair of iterators {f, l}, where f is an iterator to this element, l is std::next(f). Otherwise - {end(), end()}.

#### 11.2.5.3.1.42 Parallel iteration

Member types concurrent\_hash\_map::range\_type and concurrent\_hash\_map::const\_range\_type meets the *ContainerRange requirements*.

These types differ only in that the bounds for a concurrent\_hash\_map::const\_range\_type are of type concurrent\_hash\_map::const\_iterator, whereas the bounds for a concurrent\_hash\_map::range\_type are of type concurrent\_hash\_map::iterator.

Traversing the concurrent\_hash\_map is not thread safe. The behavior is undefined in case of concurrent execution of any member functions while traversing the range\_type or const\_range\_type.

#### 11.2.5.3.1.43 range member function

```
range_type range( std::size_t grainsize = 1 );
const_range_type range( std::size_t grainsize = 1 ) const;
```

**Returns:** a range object representing all elements in the container.

#### 11.2.5.3.1.44 Non member functions

These functions provides binary comparison and swap operations on tbb::concurrent\_hash\_map objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define tbb::concurrent\_hash\_map as a type alias for which the non-member functions are reachable only via argument dependent lookup.

```
template <typename Key, typename T, typename HashCompare, typename Allocator>
bool operator==( const concurrent_hash_map<Key, T, HashCompare, Allocator>& lhs,
                   const concurrent_hash_map<Key, T, HashCompare, Allocator>& rhs );

template <typename Key, typename T, typename HashCompare, typename Allocator>
bool operator!=( const concurrent_hash_map<Key, T, HashCompare, Allocator>& lhs,
                   const concurrent_hash_map<Key, T, HashCompare, Allocator>& rhs );

template <typename Key, typename T, typename HashCompare, typename Allocator>
void swap( concurrent_hash_map<Key, T, HashCompare, Allocator>& lhs,
            concurrent_hash_map<Key, T, HashCompare, Allocator>& rhs );
```

#### 11.2.5.3.1.45 Non-member swap

```
template <typename Key, typename T, typename HashCompare, typename Allocator>
void swap( concurrent_hash_map<Key, T, HashCompare, Allocator>& lhs,
            concurrent_hash_map<Key, T, HashCompare, Allocator>& rhs );
```

Equivalent to `lhs.swap(rhs)`.

#### 11.2.5.3.1.46 Non-member binary comparisons

Two objects of `concurrent_hash_map` are equal if the following conditions are true:

- They contains an equal number of elements.
- Each element from the one container also contains in the other.

```
template <typename Key, typename T, typename HashCompare, typename Allocator>
bool operator==( const concurrent_hash_map<Key, T, HashCompare, Allocator>& lhs,
                   const concurrent_hash_map<Key, T, HashCompare, Allocator>& rhs );
```

**Returns:** true if `lhs` is equal to `rhs`, false otherwise.

```
template <typename Key, typename T, typename HashCompare, typename Allocator>
bool operator!=( const concurrent_hash_map<Key, T, HashCompare, Allocator>& lhs,
                   const concurrent_hash_map<Key, T, HashCompare, Allocator>& rhs );
```

Equivalent to `!(lhs == rhs)`.

**Returns:** true if `lhs` is not equal to `rhs`, false otherwise.

#### 11.2.5.3.1.47 Other

#### 11.2.5.3.1.48 Deduction guides

Where possible, constructors of `concurrent_hash_map` supports class template argument deduction (since C++17):

```

template <typename InputIterator,
          typename HashCompare,
          typename Allocator = tbb_allocator<iterator_alloc_value_t<InputIterator>>>
concurrent_hash_map( InputIterator, InputIterator,
                     const HashCompare&,
                     const Allocator& = Allocator() )
-> concurrent_hash_map<iterator_key_t<InputIterator>,
                      iterator_mapped_t<InputIterator>,
                      HashCompare,
                      Allocator>;

template <typename InputIterator,
          typename Allocator = tbb_allocator<iterator_alloc_value_t<InputIterator>>>
concurrent_hash_map( InputIterator, InputIterator,
                     const Allocator& = Allocator() )
-> concurrent_hash_map<iterator_key_t<InputIterator>,
                      iterator_mapped_t<InputIterator>,
                      tbb_hash_compare<iterator_key_t<InputIterator>>,
                      Allocator>;

template <typename Key,
          typename T,
          typename HashCompare,
          typename Allocator = tbb_allocator<std::pair<const Key, T>>>
concurrent_hash_map( std::initializer_list<std::pair<const Key, T>>,
                     const HashCompare&,
                     const Allocator& = Allocator() )
-> concurrent_hash_map<Key,
                      T,
                      HashCompare,
                      Allocator>;

template <typename Key,
          typename T,
          typename Allocator = tbb_allocator<std::pair<const Key, T>>>
concurrent_hash_map( std::initializer_list<std::pair<const Key, T>>,
                     const Allocator& = Allocator() )
-> concurrent_hash_map<Key,
                      T,
                      tbb_hash_compare<Key>,
                      Allocator>;

```

Where the type aliases `iterator_key_t`, `iterator_mapped_t` and `iterator_alloc_value_t` defines as follows:

```

template <typename InputIterator>
using iterator_key_t = std::remove_const_t<typename std::iterator_traits
  ↪<InputIterator>::value_type::first_type>;

template <typename InputIterator>
using iterator_mapped_t = typename std::iterator_traits<InputIterator>::value_
  ↪type::second_type;

template <typename InputIterator>
using iterator_alloc_value_t = std::pair<std::add_const_t<iterator_key_t
  ↪<InputIterator>,
                                             iterator_mapped_t<InputIterator>>>;

```

## Example

```
#include <tbb/concurrent_hash_map.h>
#include <vector>

int main() {
    std::vector<std::pair<const int, float>> v;

    // Deduces chmap1 as tbb::concurrent_hash_map<int, float>
    tbb::concurrent_hash_map chmap1(v.begin(), v.end());

    std::allocator<std::pair<const int, float>> alloc;
    // Deduces chmap2 as tbb::concurrent_hash_map<int, float,
    //                                         tbb_hash_compare<int>,
    //                                         std::allocator<std::pair<const int,_
    ↵float>>>
    tbb::concurrent_hash_map chmap2(v.begin(), v.end(), alloc);
}
```

### 11.2.5.3.2 concurrent\_unordered\_map

#### [containers.concurrent\_unordered\_map]

`tbb::concurrent_unordered_map` is a class template represents an unordered associative container which stores key-value pairs with unique keys and supports concurrent insertion, lookup and traversal, but not concurrent erasure.

#### 11.2.5.3.2.1 Class Template Synopsis

```
// Defined in header <tbb/concurrent_unordered_map.h>

namespace tbb {

    template <typename Key,
              typename T,
              typename Hash = std::hash<Key>,
              typename KeyEqual = std::equal_to<Key>,
              typename Allocator = tbb_allocator<std::pair<const Key, T>>>
    class concurrent_unordered_map {
public:
    using key_type = Key;
    using mapped_type = T;
    using value_type = std::pair<const Key, T>;

    using size_type = <implementation-defined unsigned integer type>;
    using difference_type = <implementation-defined signed integer type>;

    using hasher = Hash;
    using key_equal = /*See below*/;

    using allocator_type = Allocator;

    using reference = value_type&;
    using const_reference = const value_type&;
}
```

(continues on next page)

(continued from previous page)

```

using pointer = typename std::allocator_traits<Allocator>::pointer;
using const_pointer = typename std::allocator_traits<Allocator>::const_
→pointer;

using iterator = <implementation-defined ForwardIterator>;
using const_iterator = <implementation-defined constant ForwardIterator>;

using local_iterator = <implementation-defined ForwardIterator>;
using const_local_iterator = <implementation-defined constant ForwardIterator>
→;

using node_type = <implementation-defined node handle>;

using range_type = <implementation-defined ContainerRange>;
using const_range_type = <implementation-defined constant ContainerRange>;

// Construction, destruction, copying
concurrent_unordered_map();

explicit concurrent_unordered_map( size_type bucket_count, const hasher& hash,
→= hasher(),
→                                const key_equal& equal = key_equal(),
→                                const allocator_type& alloc = allocator_
→type() );

concurrent_unordered_map( size_type bucket_count, const allocator_type& alloc,
→);

concurrent_unordered_map( size_type bucket_count, const hasher& hash,
→                                const allocator_type& alloc );

explicit concurrent_unordered_map( const allocator_type& alloc );

template <typename InputIterator>
concurrent_unordered_map( InputIterator first, InputIterator last,
→                                size_type bucket_count = /*implementation-defined*/,
→                                const hasher& hash = hasher(),
→                                const key_equal& equal = key_equal(),
→                                const allocator_type& alloc = allocator_type() );

template <typename Inputiterator>
concurrent_unordered_map( InputIterator first, InputIterator last,
→                                size_type bucket_count, const allocator_type& alloc,
→);

template <typename InputIterator>
concurrent_unordered_map( InputIterator first, InputIterator last,
→                                size_type bucket_count, const hasher& hash,
→                                const allocator_type& alloc );

concurrent_unordered_map( std::initializer_list<value_type> init,
→                                size_type bucket_count = /*implementation-defined*/,
→                                const hasher& hash = hasher(),
→                                const key_equal& equal = key_equal(),
→                                const allocator_type& alloc = allocator_type() );

concurrent_unordered_map( std::initializer_list<value_type> init,

```

(continues on next page)

(continued from previous page)

```

        size_type bucket_count, const allocator_type& alloc ↵);

concurrent_unordered_map( std::initializer_list<value_type> init,
                           size_type bucket_count, const hasher& hash,
                           const allocator_type& alloc );

concurrent_unordered_map( const concurrent_unordered_map& other );
concurrent_unordered_map( const concurrent_unordered_map& other,
                           const allocator_type& alloc );

concurrent_unordered_map( concurrent_unordered_map&& other );
concurrent_unordered_map( concurrent_unordered_map&& other,
                           const allocator_type& alloc );

~concurrent_unordered_map();

concurrent_unordered_map& operator=( const concurrent_unordered_map& other );
concurrent_unordered_map& operator=( concurrent_unordered_map&& other ) ↵
noexcept(/*See details*/);

concurrent_unordered_map& operator=( std::initializer_list<value_type> init );

allocator_type get_allocator() const;

// Iterators
iterator begin() noexcept;
const_iterator begin() const noexcept;
const_iterator cbegin() const noexcept;

iterator end() noexcept;
const_iterator end() const noexcept;
const_iterator cend() const noexcept;

// Size and capacity
bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// Concurrently safe modifiers
std::pair<iterator, boolinsert( const value_type& value );
iterator insert( const_iterator hint, const value_type& value );

template <typename P>
std::pair<iterator, boolinsert( P&& value );

template <typename P>
iterator insert( const_iterator hint, P&& value );

std::pair<iterator, boolinsert( value_type&& value );
iterator insert( const_iterator hint, value_type&& value );

template <typename InputIterator>
void insert( InputIterator first, InputIterator last );

void insert( std::initializer_list<value_type> init );

```

(continues on next page)

(continued from previous page)

```

std::pair<iterator, bool> insert( node_type&& nh );
iterator insert( const_iterator hint, node_type&& nh );

template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args );

template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_map<Key, T, SrcHash, SrcKeyEqual, Allocator>&
→ source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_map<Key, T, SrcHash, SrcKeyEqual, Allocator>&
→& source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_multimap<Key, T, SrcHash, SrcKeyEqual, Allocator>&
→ source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_multimap<Key, T, SrcHash, SrcKeyEqual, Allocator>&&
→ source );

// Concurrently unsafe modifiers
void clear() noexcept;

iterator unsafe_erase( const_iterator pos );
iterator unsafe_erase( iterator pos );

iterator unsafe_erase( const_iterator first, const_iterator last );

size_type unsafe_erase( const key_type& key );

template <typename K>
size_type unsafe_erase( const K& key );

node_type unsafe_extract( const_iterator pos );
node_type unsafe_extract( iterator pos );

node_type unsafe_extract( const key_type& key );

template <typename K>
node_type unsafe_extract( const K& key );

void swap( concurrent_unordered_map& other );

// Element access
mapped_type& at( const key_type& key );
const mapped_type& at( const key_type& key ) const;

mapped_type& operator[]( const key_type& key );
mapped_type& operator[]( key_type&& key );

// Lookup
size_type count( const key_type& key ) const;

```

(continues on next page)

(continued from previous page)

```

template <typename K>
size_type count( const K& key ) const;

iterator find( const key_type& key );
const_iterator find( const key_type& key ) const;

template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;

bool contains( const key_type& key ) const;

template <typename K>
bool contains( const K& key ) const;

std::pair<iterator, iterator> equal_range( const key_type& key );
std::pair<const_iterator, const_iterator> equal_range( const key_type& key ) ↵
const;

template <typename K>
std::pair<iterator, iterator> equal_range( const K& key );

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key ) const;

// Bucket interface
local_iterator unsafe_begin( size_type n );
const_local_iterator unsafe_begin( size_type n ) const;
const_local_iterator unsafe_cbegin( size_type n ) const;

local_iterator unsafe_end( size_type n );
const_local_iterator unsafe_end( size_type n ) const;
const_local_iterator unsafe_cend( size_type n ) const;

size_type unsafe_bucket_count() const;
size_type unsafe_max_bucket_bound() const;

size_type unsafe_bucket_size( size_type n ) const;

size_type unsafe_bucket( const key_type& key ) const;

// Hash policy
float load_factor() const;

float max_load_factor() const;
void max_load_factor( float ml );

void rehash( size_type count );

void reserve( size_type count );

// Observers
hasher hash_function() const;
key_equal key_eq() const;

```

(continues on next page)

(continued from previous page)

```

// Parallel iteration
range_type range();
const_range_type range() const;
}; // class concurrent_unordered_map

} // namespace tbb

```

Requirements:

- The expression `std::allocator_type<Allocator>::destroy(m, val)` where `m` is an object of the type `Allocator` and `val` is an object of type `value_type` should be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Hash` shall meet the `Hash` requirements from [hash] ISO C++ Standard section.
- The type `KeyEqual` shall meet the `BinaryPredicate` requirements from [algorithms.general] ISO C++ Standard section.
- The type `Allocator` shall meet the `Allocator` requirements from [allocator.requirements] ISO C++ Standard section.

### 11.2.5.3.2.2 Description

`tbb::concurrent_unordered_map` is an unordered associative container, which elements are organized into buckets. The value of the hash function `Hash` for `Key` object determines the number of the bucket in which the corresponding element would be placed.

If the qualified-id `Hash::transparent_key_equal` is valid and denotes a type, member type `concurrent_unordered_map::key_equal` defines as the value of this qualified-id. In this case the program is ill-formed if any of the following conditions are met:

- The template parameter `KeyEqual` is different from `std::equal_to<Key>`.
- Qualified-id `Hash::transparent_key_equal::is_transparent` is not valid or not denotes a type.

Otherwise, member type `concurrent_unordered_map::key_equal` defines as the value of the template parameter `KeyEqual`.

### 11.2.5.3.2.3 Member functions

#### 11.2.5.3.2.4 Construction, destruction, copying

#### 11.2.5.3.2.5 Empty container constructors

```

concurrent_unordered_map();

explicit concurrent_unordered_map( const allocator_type& alloc );

```

Constructs an empty `concurrent_unordered_map`. The initial number of buckets is unspecified.

If provided uses the allocator `alloc` to allocate the memory.

```

explicit concurrent_unordered_map( size_type bucket_count,
                                    const hasher& hash = hasher(),
                                    const key_equal& equal = key_equal(),
                                    const allocator_type& alloc = allocator_
                                     ↵type() );

concurrent_unordered_map( size_type bucket_count, const allocator_type&_
                         ↵alloc );

concurrent_unordered_map( size_type bucket_count, const hasher& hash,
                         const allocator_type& alloc );

```

Constructs an empty concurrent\_unordered\_map with bucket\_count buckets.

If provided uses the hash function hasher, predicate equal to compare key\_type objects for equality and the allocator alloc to allocate the memory.

#### 11.2.5.3.2.6 Constructors from the sequence of elements

```

template <typename InputIterator>
concurrent_unordered_map( InputIterator first, InputIterator last,
                          size_type bucket_count = /*implementation-defined*/
                           ↵,
                           const hasher& hash = hasher(),
                           const key_equal& equal = key_equal(),
                           const allocator_type& alloc = allocator_type() );

template <typename Inputiterator>
concurrent_unordered_map( InputIterator first, InputIterator last,
                          size_type bucket_count, const allocator_type&_
                           ↵alloc );

template <typename InputIterator>
concurrent_unordered_map( InputIterator first, InputIterator last,
                          size_type bucket_count, const hasher& hash,
                          const allocator_type& alloc );

```

Constructs the concurrent\_unordered\_map which contains the elements from the half-open interval [first, last).

If the range [first, last) contains multiple elements with equal keys, it is unspecified which element would be inserted.

If provided uses the hash function hasher, predicate equal to compare key\_type objects for equality and the allocator alloc to allocate the memory.

**Requirements:** the type InputIterator shall meet the requirements of InputIterator from [input.iterators] ISO C++ Standard section.

```

concurrent_unordered_map( std::initializer_list<value_type> init,
                          size_type bucket_count = /*implementation-defined*/
                           ↵,
                           const hasher& hash = hasher(),
                           const key_equal& equal = key_equal(),
                           const allocator_type& alloc = allocator_type() );

```

Equivalent to `concurrent_unordered_map(init.begin(), init.end(), bucket_count, hash, equal, alloc).`

```
concurrent_unordered_map( std::initializer_list<value_type> init,
                           size_type bucket_count, const allocator_type&u
                           alloc );
```

Equivalent to `concurrent_unordered_map(init.begin(), init.end(), bucket_count, alloc).`

```
concurrent_unordered_map( std::initializer_list<value_type> init,
                           size_type bucket_count, const hasher& hash,
                           const allocator_type& alloc );
```

Equivalent to `concurrent_unordered_map(init.begin(), init.end(), bucket_count, hash, alloc).`

#### 11.2.5.3.2.7 Copying constructors

```
concurrent_unordered_map( const concurrent_unordered_map& other );
concurrent_unordered_map( const concurrent_unordered_map& other,
                           const allocator_type& alloc );
```

Constructs a copy of `other`.

If the `allocator` argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

#### 11.2.5.3.2.8 Moving constructors

```
concurrent_unordered_map( concurrent_unordered_map&& other );
concurrent_unordered_map( concurrent_unordered_map&& other,
                           const allocator_type& alloc );
```

Constructs a `concurrent_unordered_map` with the contents of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the `allocator` argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

### 11.2.5.3.2.9 Destructor

```
~concurrent_unordered_map();
```

Destroys the `concurrent_unordered_map`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

### 11.2.5.3.2.10 Assignment operators

```
concurrent_unordered_map& operator=( const concurrent_unordered_map& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment` is true.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

**Returns:** a reference to `*this`.

```
concurrent_unordered_map& operator=( concurrent_unordered_map&& other )  
    noexcept /*See below*/;
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment` is true.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

**Returns:** a reference to `*this`.

**Exceptions:** `noexcept` specification:

```
noexcept (std::allocator_traits<allocator_type>::is_always_equal::value &&  
        std::is_nothrow_move_assignable<hasher>::value &&  
        std::is_nothrow_move_assignable<key_equal>::value)
```

```
concurrent_unordered_map& operator=( std::initializer_list<value_type> init );
```

Replaces all elements in `*this` by the elements in `init`.

If `init` contains multiple elements with equal keys, it is unspecified which element would be inserted.

The behavior is undefined in case of concurrent operations with `*this`.

**Returns:** a reference to `*this`.

### 11.2.5.3.2.11 Iterators

The types `concurrent_unordered_map::iterator` and `concurrent_unordered_map::const_iterator` meets the requirements of `ForwardIterator` from [forward.iterators] ISO C++ Standard section.

### 11.2.5.3.2.12 begin and cbegin

```
iterator begin();
const_iterator begin() const;
const_iterator cbegin() const;
```

**Returns:** an iterator to the first element in the container.

### 11.2.5.3.2.13 end and cend

```
iterator end();
const_iterator end() const;
const_iterator cend() const;
```

**Returns:** an iterator to the element which follows the last element in the container.

### 11.2.5.3.2.14 Size and capacity

#### 11.2.5.3.2.15 empty

```
bool empty() const;
```

**Returns:** `true` if the container is empty, `false` otherwise.

The result may differ with the actual container state in case of pending concurrent insertions.

#### 11.2.5.3.2.16 size

```
size_type size() const;
```

**Returns:** the number of elements in the container.

The result may differ with the actual container size in case of pending concurrent insertions.

### 11.2.5.3.2.17 max\_size

```
size_type max_size() const;
```

**Returns:** the maximum number of elements that container can hold.

### 11.2.5.3.2.18 Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other, lookup methods and while traversing the container.

### 11.2.5.3.2.19 Emplacing elements

```
template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args );
```

Attempts to insert an element ,constructed in-place from `args` into the container.

**Returns:** `std::pair<iterator, bool>` where `iterator` points to the inserted element or to an existing element with equal key. Boolean value is `true` if insertion took place, `false` otherwise.

**Requirements:** the type `value_type` shall meet the `EmplaceConstructible` requirements from [container.requirements] ISO C++ Standard section.

```
template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );
```

Attempts to insert an element ,constructed in-place from `args` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

**Returns:** an `iterator` to the inserted element or to an existing element with equal key.

**Requirements:** the type `value_type` shall meet the `EmplaceConstructible` requirements from [container.requirements] ISO C++ Standard section.

### 11.2.5.3.2.20 Inserting values

```
std::pair<iterator, bool> insert( const value_type& value );
```

Attempts to insert `value` into the container.

**Returns:** `std::pair<iterator, bool>` where `iterator` points to the inserted element or to an existing element with equal key. Boolean value is `true` if insertion took place, `false` otherwise.

**Requirements:** the type `value_type` shall meet the `CopyInsertable` requirements from [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, const value_type& value );
```

Attempts to insert `value` into the container.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

**Returns:** an `iterator` to the inserted element or to an existing element with equal key.

**Requirements:** the type `value_type` shall meet the `CopyInsertable` requirements from [container.requirements] ISO C++ Standard section.

```
template <typename P>
std::pair<iterator, bool> insert( P&& value );
```

Equivalent to `emplace(std::forward<P>(value))`.

This overload only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is `true`.

```
template <typename P>
iterator insert( const_iterator hint, P&& value );
```

Equivalent to `emplace_hint(hint, std::forward<P>(value))`.

This overload only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is `true`.

```
std::pair<iterator, bool> insert( value_type&& value );
```

Attempts to insert `value` into the container using move semantics.

`value` is left in a valid, but unspecified state.

**Returns:** `std::pair<iterator, bool>` where `iterator` points to the inserted element or to an existing element with equal key. Boolean value is `true` if insertion took place, `false` otherwise.

**Requirements:** the type `value_type` shall meet the `MoveInsertable` requirements from [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, value_type&& other );
```

Attempts to insert `value` into the container using move semantics.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

`value` is left in a valid, but unspecified state.

**Returns:** an `iterator` to the inserted element or to an existing element with equal key.

**Requirements:** the type `value_type` shall meet the `MoveInsertable` requirements from [container.requirements] ISO C++ Standard section.

### 11.2.5.3.2.21 Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last );
```

Attempts to insert all items from the half-open interval [first, last) into the container.

If the interval [first, last) contains multiple elements with equal keys, it is unspecified which element should be inserted.

**Requirements:** the type `InputIterator` must meet the requirements of `InputIterator` from [`input iterators`] ISO C++ Standard section.

```
void insert( std::initializer_list<value_type> init );
```

Equivalent to `insert(init.begin(), init.end())`.

### 11.2.5.3.2.22 Inserting nodes

```
std::pair<iterator, bool

```

If the node handle `nh` is empty, does nothing.

Otherwise - attempts to insert the node, owned by `nh` into the container.

If the insertion fails, node handle `nh` remains ownership of the node.

Otherwise - `nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

**Returns:** `std::pair<iterator, bool>` where `iterator` points to the inserted element or to an existing element with key equal to `nh.key()`. Boolean value is `true` if insertion took place, `false` otherwise.

```
iterator insert( const_iterator hint, node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise - attempts to insert the node, owned by `nh` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

If the insertion fails, node handle `nh` remains ownership of the node.

Otherwise - `nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

**Returns:** an iterator pointing to the inserted element or to an existing element with key equal to `nh.key()`.

## Merging containers

```

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_map<Key, T, SrcHash, SrcKeyEqual, Allocator>
↪& source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_map<Key, T, SrcHash, SrcKeyEqual, Allocator>
↪&& source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_multimap<Key, T, SrcHash, SrcKeyEqual, Allocator>& source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_multimap<Key, T, SrcHash, SrcKeyEqual, Allocator>&& source );

```

Transfers those elements from `source` which keys do not exist in the container.

In case of merging with the container with multiple elements with equal keys, it is unspecified which element would be transferred.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `get_allocator() != source.get_allocator()`.

#### 11.2.5.3.2.23 Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

#### 11.2.5.3.2.24 Clearing

```
void clear();
```

Removes all elements from the container.

#### 11.2.5.3.2.25 Erasing elements

```

iterator unsafe_erase( const_iterator pos );
iterator unsafe_erase( iterator pos );

```

Removes the element pointed to by `pos` from the container.

Invalidates all iterators and references to the removed element.

**Returns:** `iterator` which follows the removed element.

**Requirements:** the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
size_type unsafe_erase( const key_type& key );
```

Removes the element with the key equal with `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

**Returns:** 1 if an element with the key equal to `key` exists, 0 otherwise.

```
template <typename K>
size_type unsafe_erase( const K& key );
```

Removes the element with the key which compares equivalent with `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

This overload only participates in overload resolution if all of the following conditions are met:

- The qualified-id `hasher::transparent_key_equal` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

**Returns:** 1 if an element with the key which compares equivalent with `key` exists, 0 otherwise.

#### 11.2.5.3.2.26 Erasing sequences

```
iterator unsafe_erase( const_iterator first, const_iterator last );
```

Removes all elements from the half-open interval `[first, last)` from the container.

**Returns:** iterator which follows the last removed element.

**Requirements:** the range `[first, last)` must be a valid subrange in `*this`.

#### 11.2.5.3.2.27 Extracting nodes

```
node_type unsafe_extract( iterator pos );
node_type unsafe_extract( const_iterator pos );
```

Transfers ownership of the element pointed to by `pos` from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

**Returns:** the node handle that owns the extracted element.

**Requirements:** the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
node_type unsafe_extract( const key_type& key );
```

If an element with the key equal to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

**Returns:** the node handle that owns the extracted element or an empty node handle if an element with the key equal to `key` was not found.

```
template <typename K>
node_type unsafe_extract( const K& key );
```

If an element with the key which compares equivalent with `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

This overload only participates in overload resolution if all of the following conditions are met:

- The qualified-id `hasher::transparent_key_equal` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

**Returns:** the node handle that owns the extracted element or an empty node handle if an element with the key which compares equivalent with `key` was not found.

#### 11.2.5.3.2.28 swap

```
void swap( concurrent_unordered_map& other ) noexcept( /*See below*/ );
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise if `get_allocator() != other.get_allocator()` the behavior is undefined.

**Exceptions:** `noexcept` specification:

```
noexcept( std::allocator_traits<allocator_type>::is_always_
    ↵equal::value &&
        std::is_nothrow_swappable<hasher>::value &&
        std::is_nothrow_swappable<key_equal>::value )
```

#### 11.2.5.3.2.29 Element access

##### 11.2.5.3.2.30 at

```
value_type& at( const key_type& key );
const value_type& at( const key_type& key ) const;
```

**Returns:** a reference to `item.second` where `item` is the element with the key equal to `key`.

**Throws:** `std::out_of_range` exception if the element with the key equal to `key` is not presented in the container.

#### 11.2.5.3.2.31 operator[]

```
value_type& operator[]( const key_type& key );
```

If the element with the key equal to `key` is not presented in the container, inserts a new element, constructed in-place from `std::piecewise_construct`, `std::forward_as_tuple(key)`, `std::tuple<>()`.

**Requirements:** the type `value_type` shall meet the `EmplaceConstructible` requirements from [container.requirements] ISO C++ Standard section.

**Returns:** a reference to `item.second` where `item` is the element with the key equal to `key`.

```
value_type& operator[]( key_type&& key );
```

If the element with the key equal to `key` is not presented in the container, inserts a new element, constructed in-place from `std::piecewise_construct`, `std::forward_as_tuple(std::move(key))`, `std::tuple<>()`.

**Requirements:** the type `value_type` shall meet the `EmplaceConstructible` requirements from [container.requirements] ISO C++ Standard section.

**Returns:** a reference to `item.second` where `item` is the element with the key equal to `key`.

#### 11.2.5.3.2.32 Lookup

All methods in this section can be executed concurrently with each other, concurrently-safe modifiers and while traversing the container.

#### 11.2.5.3.2.33 count

```
size_type count( const key_type& key );
```

**Returns:** the number of elements with the key equal to `key`.

```
template <typename K>
size_type count( const K& key );
```

**Returns:** the number of elements with the key which compares equivalent with `key`.

This overload only participates in overload resolution if `qualified-id hasher::transparent_key_equal` is valid and denotes a type.

### 11.2.5.3.2.34 find

```
iterator find( const key_type& key );
const_iterator find( const key_type& key ) const;
```

**Returns:** an iterator to the element with the key equal to key or end() if no such element exists.

```
template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;
```

**Returns:** an iterator to the element with the key which compares equivalent with key or end() if no such element exists.

These overloads only participates in overload resolution if qualified-id hasher::transparent\_key\_equal is valid and denotes a type.

### 11.2.5.3.2.35 contains

```
bool contains( const key_type& key ) const;
```

**Returns:** true if an element with the key equal to key exists in the container, false otherwise.

```
template <typename K>
bool contains( const K& key ) const;
```

**Returns:** true if an element with the key which compares equivalent with key exists in the container, false otherwise.

This overload only participates in overload resolution if qualified-id hasher::transparent\_key\_equal is valid and denotes a type.

### 11.2.5.3.2.36 equal\_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );
std::pair<const_iterator, const_iterator> equal_range( const key_type& key ) const;
```

**Returns:** if an element with the key equal to key exists - a pair of iterators {f, l}, where f is an iterator to this element, l is std::next(f). Otherwise - {end(), end() }.

```
template <typename K>
std::pair<iterator, iterator> equal_range( const K& key )

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key )
```

**Returns:** if an element with the key which compares equivalent with `key` exists - a pair of iterators `{f, l}`, where `f` is an iterator to this element, `l` is `std::next(f)`. Otherwise - `{end(), end()}`.

These overloads only participates in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

#### 11.2.5.3.2.37 Bucket interface

The types `concurrent_unordered_map::local_iterator` and `concurrent_unordered_map::const_local_iterator` meets the requirements of `ForwardIterator` from [forward.iterators] ISO C++ Standard section.

These iterators are used to traverse the certain bucket.

All methods in this section can only be executed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

#### 11.2.5.3.2.38 Bucket begin and bucket end

```
local_iterator unsafe_begin( size_type n );
const_local_iterator unsafe_begin( size_type n ) const;
const_local_iterator unsafe_cbegin( size_type n ) const;
```

**Returns:** an iterator to the first element in the bucket number `n`.

```
local_iterator unsafe_end( size_type n );
const_local_iterator unsafe_end( size_type n ) const;
const_local_iterator unsafe_cend( size_type n ) const;
```

**Returns:** an iterator to the element which follows the last element in the bucket number `n`.

#### 11.2.5.3.2.39 The number of buckets

```
size_type unsafe_bucket_count() const;
```

**Returns:** the number of buckets in the container.

```
size_type unsafe_max_bucket_count() const;
```

**Returns:** the maximum number of buckets that container can hold.

#### 11.2.5.3.2.40 Size of the bucket

```
size_type unsafe_bucket_size( size_type n ) const;
```

**Returns:** the number of elements in the bucket number n.

#### 11.2.5.3.2.41 Bucket number

```
size_type unsafe_bucket( const key_type& key ) const;
```

**Returns:** the number of the bucket in which the element with the key key is stored.

#### 11.2.5.3.2.42 Hash policy

Hash policy of concurrent\_unordered\_map manages the number of buckets in the container and the allowed maximum number of elements per bucket (load factor). If the maximum load factor is exceeded, the container can automatically increase the number of buckets.

#### 11.2.5.3.2.43 Load factor

```
float load_factor() const;
```

**Returns:** the average number of elements per bucket, which is size() / unsafe\_bucket\_count().

```
float max_load_factor() const;
```

**Returns:** the maximum number of elements per bucket.

```
void max_load_factor( float ml );
```

Sets the maximum number of elements per bucket to ml.

#### 11.2.5.3.2.44 Manual rehashing

```
void rehash( size_type n );
```

Sets the number of buckets to n and rehashes the container.

```
void reserve( size_type n );
```

Sets the number of buckets to the value which is needed to store n elements.

#### 11.2.5.3.2.45 Observers

##### 11.2.5.3.2.46 get\_allocator

```
allocator_type get_allocator() const;
```

**Returns:** a copy of the allocator associated with `*this`.

##### 11.2.5.3.2.47 hash\_function

```
hasher hash_function() const;
```

**Returns:** a copy of the hash function associated with `*this`.

##### 11.2.5.3.2.48 key\_eq

```
key_equal key_eq() const;
```

**Returns:** a copy of the key equality predicate associated with `*this`.

#### 11.2.5.3.2.49 Parallel iteration

Member types `concurrent_unordered_map::range_type` and `concurrent_unordered_map::const_range_type` meets the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_unordered_map::const_range_type` are of type `concurrent_unordered_map::const_iterator`, whereas the bounds for a `concurrent_unordered_map::range_type` are of type `concurrent_unordered_map::iterator`.

#### 11.2.5.3.2.50 range member function

```
range_type range();
const_range_type range() const;
```

**Returns:** a range object representing all elements in the container.

#### 11.2.5.3.2.51 Non-member functions

These functions provides binary comparison and swap operations on `tbb::concurrent_unordered_map` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `tbb::concurrent_unordered_map` as a type alias for which the non-member functions are reachable only via argument dependent lookup.

```

template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
void swap( concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& lhs,
            concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& rhs );

template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& lhs,
                    const concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& rhs );

template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& lhs,
                    const concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& rhs );

```

### 11.2.5.3.2.52 Non-member swap

```

template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
void swap( concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& lhs,
            concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& rhs ) noexcept(noexcept(lhs.swap(rhs)));

```

Equivalent to `lhs.swap(rhs)`.

### 11.2.5.3.2.53 Non-member binary comparisons

Two objects of `concurrent_unordered_map` are equal if the following conditions are true:

- They contains an equal number of elements.
- Each element from the one container also contains in the other.

```

template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& lhs,
                    const concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& rhs );

```

**Returns:** true if `lhs` is equal to `rhs`, false otherwise.

```

template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator!=( const concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& lhs,
                    const concurrent_unordered_map<Key, T, Hash, KeyEqual, Allocator>& rhs );

```

Equivalent to `!(lhs == rhs)`.

**Returns:** true if lhs is not equal to rhs, false otherwise.

#### 11.2.5.3.2.54 Other

##### 11.2.5.3.2.55 Deduction guides

Where possible, constructors of `concurrent_unordered_map` supports class template argument deduction (since C++17):

```
template <typename InputIterator,
         typename Hash = std::hash<iterator_key_t<InputIterator>>,
         typename KeyEqual = std::equal_to<iterator_key_t<InputIterator>>,
         typename Allocator = tbb_allocator<iterator_alloc_value_t<InputIterator>>>
concurrent_unordered_map( InputIterator, InputIterator,
                         map_size_type = /*implementation_defined*/,
                         Hash = Hash(), KeyEqual = KeyEqual(),
                         Allocator = Allocator() )
-> concurrent_unordered_map<iterator_key_t<InputIterator>,
                           iterator_mapped_t<InputIterator>,
                           Hash, KeyEqual, Allocator>;
```

```
template <typename InputIterator,
         typename Allocator>
concurrent_unordered_map( InputIterator, InputIterator,
                         map_size_type,
                         Allocator )
-> concurrent_unordered_map<iterator_key_t<InputIterator>,
                           iterator_mapped_t<InputIterator>,
                           std::hash<iterator_key_t<InputIterator>>,
                           std::equal_to<iterator_key_t<InputIterator>>,
                           Allocator>;
```

```
template <typename InputIterator,
         typename Allocator>
concurrent_unordered_map( InputIterator, InputIterator, Allocator )
-> concurrent_unordered_map<iterator_key_t<InputIterator>,
                           iterator_mapped_t<InputIterator>,
                           std::hash<iterator_key_t<InputIterator>>,
                           std::equal_to<iterator_key_t<InputIterator>>,
                           Allocator>;
```

```
template <typename InputIterator,
         typename Hash,
         typename Allocator>
concurrent_unordered_map( InputIterator, InputIterator,
                         Hash, Allocator )
-> concurrent_unordered_map<iterator_key_t<InputIterator>,
                           iterator_mapped_t<InputIterator>,
                           Hash,
                           std::equal_to<iterator_key_t<InputIterator>>,
                           Allocator>;
```

```
template <typename Key,
         typename T,
```

(continues on next page)

(continued from previous page)

```

typename Hash = std::hash<Key>,
typename KeyEqual = std::equal_to<Key>,
typename Allocator = tbb_allocator<std::pair<Key, T>>>
concurrent_unordered_map( std::initializer_list<value_type>,
                           map_size_type = /*implementation-defined*/,
                           Hash = Hash(),
                           KeyEqual = KeyEqual(),
                           Allocator = Allocator() )
-> concurrent_unordered_map<Key, T,
                           Hash,
                           KeyEqual,
                           Allocator>;

template <typename Key,
          typename T,
          typename Allocator>
concurrent_unordered_map( std::initializer_list<value_type>,
                           map_size_type, Allocator )
-> concurrent_unordered_map<Key, T,
                           std::hash<Key>,
                           std::equal_to<Key>,
                           Allocator>;

template <typename Key,
          typename T,
          typename Hash,
          typename Allocator>
concurrent_unordered_map( std::initializer_list<value_type>,
                           map_size_type, Hash, Allocator )
-> concurrent_unordered_map<Key, T,
                           Hash,
                           std::equal_to<Key>,
                           Allocator>;

```

Where the type `map_size_type` refers to the `size_type` member type of the deduced `concurrent_unordered_map` and the type aliases `iterator_key_t`, `iterator_mapped_t` and `iterator_alloc_value_t` defines as follows:

```

template <typename InputIterator>
using iterator_key_t = std::remove_const_t<typename std::iterator_traits<
  ↵<InputIterator>::value_type::first_type>>;

template <typename InputIterator>
using iterator_mapped_t = typename std::iterator_traits<InputIterator>::value_
  ↵type::second_type;

template <typename InputIterator>
using iterator_alloc_value_t = std::pair<std::add_const_t<iterator_key_t
  ↵<InputIterator>,
                           iterator_mapped_t<InputIterator>>>;

```

### Example

```
#include <tbb/concurrent_unordered_map.h>
#include <vector>
#include <functional>
```

(continues on next page)

(continued from previous page)

```

struct CustomHasher {...};

int main() {
    std::vector<std::pair<int, float>> v;

    // Deduces m1 as concurrent_unordered_map<int, float>
    tbb::concurrent_unordered_map m1(v.begin(), v.end());

    // Deduces m2 as concurrent_unordered_map<int, float, CustomHasher>;
    tbb::concurrent_unordered_map m2(v.begin(), v.end(), CustomHasher{});
}

```

### 11.2.5.3.3 concurrent\_unordered\_multimap

#### [containers.concurrent\_unordered\_multimap]

`tbb::concurrent_unordered_multimap` is a class template represents an unordered associative container which stores key-value pairs and supports concurrent insertion, lookup and traversal, but not concurrent erasure. The container allows to store multiple elements with equal keys.

#### 11.2.5.3.3.1 Class Template Synopsis

```

// Defined in header <tbb/concurrent_unordered_map.h>

namespace tbb {
    template <typename Key,
              typename T,
              typename Hash = std::hash<Key>,
              typename KeyEqual = std::equal_to<Key>,
              typename Allocator = tbb_allocator<std::pair<const Key, T>>>
    class concurrent_unordered_multimap {
    public:
        using key_type = Key;
        using mapped_type = T;
        using value_type = std::pair<const Key, T>;

        using size_type = <implementation-defined unsigned integer type>;
        using difference_type = <implementation-defined signed integer type>;

        using hasher = Hash;
        using key_equal = /*See below*/;

        using allocator_type = Allocator;

        using reference = value_type&;
        using const_reference = const value_type&;

        using pointer = typename std::allocator_traits<Allocator>::pointer;
        using const_pointer = typename std::allocator_traits<Allocator>::const_
            ↵pointer;

        using iterator = <implementation-defined ForwardIterator>;

```

(continues on next page)

(continued from previous page)

```

using const_iterator = <implementation-defined constant ForwardIterator>;
using local_iterator = <implementation-defined ForwardIterator>;
using const_local_iterator = <implementation-defined constant ForwardIterator>;
using node_type = <implementation-defined node handle>;
using range_type = <implementation-defined ContainerRange>;
using const_range_type = <implementation-defined constant ContainerRange>;
// Construction, destruction, copying
concurrent_unordered_multimap();

explicit concurrent_unordered_multimap( size_type bucket_count, const hasher&
hash = hasher(),
const key_equal& equal = key_equal(),
const allocator_type& alloc =
allocator_type() );

concurrent_unordered_multimap( size_type bucket_count, const allocator_type&
alloc );

concurrent_unordered_multimap( size_type bucket_count, const hasher& hash,
const allocator_type& alloc );

explicit concurrent_unordered_multimap( const allocator_type& alloc );

template <typename InputIterator>
concurrent_unordered_multimap( InputIterator first, InputIterator last,
size_type bucket_count = /*implementation-
defined*/,
const hasher& hash = hasher(),
const key_equal& equal = key_equal(),
const allocator_type& alloc = allocator_type() );
template <typename Inputiterator>
concurrent_unordered_multimap( InputIterator first, InputIterator last,
size_type bucket_count, const allocator_type&
alloc );

template <typename InputIterator>
concurrent_unordered_multimap( InputIterator first, InputIterator last,
size_type bucket_count, const hasher& hash,
const allocator_type& alloc );

concurrent_unordered_multimap( std::initializer_list<value_type> init,
size_type bucket_count = /*implementation-
defined*/,
const hasher& hash = hasher(),
const key_equal& equal = key_equal(),
const allocator_type& alloc = allocator_type() );
concurrent_unordered_multimap( std::initializer_list<value_type> init,
size_type bucket_count, const allocator_type&
alloc );

```

(continues on next page)

(continued from previous page)

```

concurrent_unordered_multimap( std::initializer_list<value_type> init,
                               size_type bucket_count, const hasher& hash,
                               const allocator_type& alloc );

concurrent_unordered_multimap( const concurrent_unordered_multimap& other );
concurrent_unordered_multimap( const concurrent_unordered_multimap& other,
                               const allocator_type& alloc );

concurrent_unordered_multimap( concurrent_unordered_multimap&& other );
concurrent_unordered_multimap( concurrent_unordered_multimap&& other,
                               const allocator_type& alloc );

~concurrent_unordered_multimap();

concurrent_unordered_multimap& operator=( const concurrent_unordered_multimap&
→ other );
concurrent_unordered_multimap& operator=( concurrent_unordered_multimap&&→
other ) noexcept(/*See details*/);

concurrent_unordered_multimap& operator=( std::initializer_list<value_type>→
init );

allocator_type get_allocator() const;

// Iterators
iterator begin() noexcept;
const_iterator begin() const noexcept;
const_iterator cbegin() const noexcept;

iterator end() noexcept;
const_iterator end() const noexcept;
const_iterator cend() const noexcept;

// Size and capacity
bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// Concurrently safe modifiers
std::pair<iterator, boolinsert( const value_type& value );
iterator insert( const_iterator hint, const value_type& value );

template <typename P>
std::pair<iterator, boolinsert( P&& value );

template <typename P>
iterator insert( const_iterator hint, P&& value );

std::pair<iterator, boolinsert( value_type&& value );
iterator insert( const_iterator hint, value_type&& value );

template <typename InputIterator>
void insert( InputIterator first, InputIterator last );

void insert( std::initializer_list<value_type> init );

```

(continues on next page)

(continued from previous page)

```

std::pair<iterator, bool> insert( node_type&& nh );
iterator insert( const_iterator hint, node_type&& nh );

template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args );

template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_map<Key, T, SrcHash, SrcKeyEqual, Allocator>&
→ source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_map<Key, T, SrcHash, SrcKeyEqual, Allocator>&
→& source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_multimap<Key, T, SrcHash, SrcKeyEqual, Allocator>&
→ source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_multimap<Key, T, SrcHash, SrcKeyEqual, Allocator>&&
→ source );

// Concurrently unsafe modifiers
void clear() noexcept;

iterator unsafe_erase( const_iterator pos );
iterator unsafe_erase( iterator pos );

iterator unsafe_erase( const_iterator first, const_iterator last );

size_type unsafe_erase( const key_type& key );

template <typename K>
size_type unsafe_erase( const K& key );

node_type unsafe_extract( const_iterator pos );
node_type unsafe_extract( iterator pos );

node_type unsafe_extract( const key_type& key );

template <typename K>
node_type unsafe_extract( const K& key );

void swap( concurrent_unordered_multimap& other );

// Lookup
size_type count( const key_type& key ) const;

template <typename K>
size_type count( const K& key ) const;

iterator find( const key_type& key );
const_iterator find( const key_type& key ) const;

```

(continues on next page)

(continued from previous page)

```

template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;

bool contains( const key_type& key ) const;

template <typename K>
bool contains( const K& key ) const;

std::pair<iterator, iterator> equal_range( const key_type& key );
std::pair<const_iterator, const_iterator> equal_range( const key_type& key ) const;  

const;

template <typename K>
std::pair<iterator, iterator> equal_range( const K& key );

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key ) const;

// Bucket interface
local_iterator unsafe_begin( size_type n );
const_local_iterator unsafe_begin( size_type n ) const;
const_local_iterator unsafe_cbegin( size_type n ) const;

local_iterator unsafe_end( size_type n );
const_local_iterator unsafe_end( size_type n ) const;
const_local_iterator unsafe_cend( size_type n ) const;

size_type unsafe_bucket_count() const;
size_type unsafe_max_bucket_bound() const;

size_type unsafe_bucket_size( size_type n ) const;

size_type unsafe_bucket( const key_type& key ) const;

// Hash policy
float load_factor() const;

float max_load_factor() const;
void max_load_factor( float ml );

void rehash( size_type count );

void reserve( size_type count );

// Observers
hasher hash_function() const;
key_equal key_eq() const;

// Parallel iteration
range_type range();
const_range_type range() const;
}; // class concurrent_unordered_multimap
} // namespace tbb

```

Requirements:

- The expression `std::allocator_type<Allocator>::destroy(m, val)` where `m` is an object of the type `Allocator` and `val` is an object of type `value_type` should be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Hash` shall meet the `Hash` requirements from [hash] ISO C++ Standard section.
- The type `KeyEqual` shall meet the `BinaryPredicate` requirements from [algorithms.general] ISO C++ Standard section.
- The type `Allocator` shall meet the `Allocator` requirements from [allocator.requirements] ISO C++ Standard section.

#### 11.2.5.3.3.2 Description

`tbb::concurrent_unordered_multimap` is an unordered associative container, which elements are organized into buckets. The value of the hash function `Hash` for `Key` object determines the number of the bucket in which the corresponding element would be placed.

If the qualified-id `Hash::transparent_key_equal` is valid and denotes a type, member type `concurrent_unordered_multimap::key_equal` defines as the value of this qualified-id. In this case the program is ill-formed if any of the following conditions are met:

- The template parameter `KeyEqual` is different from `std::equal_to<Key>`.
- Qualified-id `Hash::transparent_key_equal::is_transparent` is not valid or not denotes a type.

Otherwise, member type `concurrent_unordered_multimap::key_equal` defines as the value of the template parameter `KeyEqual`.

#### 11.2.5.3.3 Member functions

##### 11.2.5.3.3.4 Construction, destruction, copying

##### 11.2.5.3.3.5 Empty container constructors

```
concurrent_unordered_multimap();
explicit concurrent_unordered_multimap( const allocator_type& alloc );
```

Constructs an empty `concurrent_unordered_multimap`. The initial number of buckets is unspecified.

If provided uses the allocator `alloc` to allocate the memory.

```
explicit concurrent_unordered_multimap( size_type bucket_count,
                                       const hasher& hash = hasher(),
                                       const key_equal& equal = key_equal(),
                                       const allocator_type& alloc =
                                         allocator_type() );
concurrent_unordered_multimap( size_type bucket_count, const allocator_type&_
  alloc );
```

(continues on next page)

(continued from previous page)

```
concurrent_unordered_multimap( size_type bucket_count, const hasher& hash,
                                const allocator_type& alloc );
```

Constructs an empty concurrent\_unordered\_multimap with bucket\_count buckets.

If provided uses the hash function hasher, predicate equal to compare key\_type objects for equality and the allocator alloc to allocate the memory.

#### 11.2.5.3.3.6 Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_unordered_multimap( InputIterator first, InputIterator last,
                               size_type bucket_count = /*implementation-
                               defined*/,
                               const hasher& hash = hasher(),
                               const key_equal& equal = key_equal(),
                               const allocator_type& alloc = allocator_
                               type() );

template <typename Inputiterator>
concurrent_unordered_multimap( InputIterator first, InputIterator last,
                               size_type bucket_count, const allocator_type&_
                               alloc );

template <typename InputIterator>
concurrent_unordered_multimap( InputIterator first, InputIterator last,
                               size_type bucket_count, const hasher& hash,
                               const allocator_type& alloc );
```

Constructs the concurrent\_unordered\_multimap which contains all elements from the half-open interval [first, last).

If provided uses the hash function hasher, predicate equal to compare key\_type objects for equality and the allocator alloc to allocate the memory.

**Requirements:** the type InputIterator shall meet the requirements of InputIterator from [input.iterators] ISO C++ Standard section.

```
concurrent_unordered_multimap( std::initializer_list<value_type> init,
                               size_type bucket_count = /*implementation-
                               defined*/,
                               const hasher& hash = hasher(),
                               const key_equal& equal = key_equal(),
                               const allocator_type& alloc = allocator_
                               type() );
```

Equivalent to concurrent\_unordered\_multimap(init.begin(), init.end(), bucket\_count, hash, equal, alloc).

```
concurrent_unordered_multimap( std::initializer_list<value_type> init,
                               size_type bucket_count, const allocator_type&_
                               alloc );
```

Equivalent to concurrent\_unordered\_multimap(init.begin(), init.end(), bucket\_count, alloc).

```
concurrent_unordered_multimap( std::initializer_list<value_type> init,
                               size_type bucket_count, const hasher& hash,
                               const allocator_type& alloc );
```

Equivalent to concurrent\_unordered\_multimap(init.begin(), init.end(), bucket\_count, hash, alloc).

#### 11.2.5.3.3.7 Copying constructors

```
concurrent_unordered_multimap( const concurrent_unordered_multimap& other );
concurrent_unordered_multimap( const concurrent_unordered_multimap& other,
                               const allocator_type& alloc );
```

Constructs a copy of other.

If the allocator argument is not provided, it is obtained by calling std::allocator\_traits<allocator\_type>::select\_on\_container\_copy\_construction(other.get\_allocator()).

The behavior is undefined in case of concurrent operations with other.

#### 11.2.5.3.3.8 Moving constructors

```
concurrent_unordered_multimap( concurrent_unordered_multimap&& other );
concurrent_unordered_multimap( concurrent_unordered_multimap&& other,
                               const allocator_type& alloc );
```

Constructs a concurrent\_unordered\_multimap with the contents of other using move semantics.

other is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling std::move(other.get\_allocator()).

The behavior is undefined in case of concurrent operations with other.

#### 11.2.5.3.3.9 Destructor

```
~concurrent_unordered_multimap();
```

Destroys the concurrent\_unordered\_multimap. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with \*this.

### 11.2.5.3.3.10 Assignment operators

```
concurrent_unordered_multimap& operator=( const concurrent_unordered_
    ↵multimap& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment` is true.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

**Returns:** a reference to `*this`.

```
concurrent_unordered_multimap& operator=( concurrent_unordered_multimap&&_
    ↵other ) noexcept( /*See below*/ );
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment` is true.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

**Returns:** a reference to `*this`.

**Exceptions:** noexcept specification:

```
noexcept( std::allocator_traits<allocator_type>::is_always_
    ↵equal::value &&
        std::is_nothrow_moveAssignable<hasher>::value &&
        std::is_nothrow_moveAssignable<key_equal>::value )
```

```
concurrent_unordered_multimap& operator=( std::initializer_list<value_type>_
    ↵init );
```

Replaces all elements in `*this` by the elements in `init`.

If `init` contains multiple elements with equal keys, it is unspecified which element would be inserted.

The behavior is undefined in case of concurrent operations with `*this`.

**Returns:** a reference to `*this`.

### 11.2.5.3.3.11 Iterators

The types `concurrent_unordered_multimap::iterator` and `concurrent_unordered_multimap::const_iterator` meets the requirements of `ForwardIterator` from [forward.iterators] ISO C++ Standard section.

### 11.2.5.3.3.12 begin and cbegin

```
iterator begin();
const_iterator begin() const;
const_iterator cbegin() const;
```

**Returns:** an iterator to the first element in the container.

### 11.2.5.3.3.13 end and cend

```
iterator end();
const_iterator end() const;
const_iterator cend() const;
```

**Returns:** an iterator to the element which follows the last element in the container.

### 11.2.5.3.3.14 Size and capacity

#### 11.2.5.3.3.15 empty

```
bool empty() const;
```

**Returns:** true if the container is empty, false otherwise.

The result may differ with the actual container state in case of pending concurrent insertions.

#### 11.2.5.3.3.16 size

```
size_type size() const;
```

**Returns:** the number of elements in the container.

The result may differ with the actual container size in case of pending concurrent insertions.

#### 11.2.5.3.3.17 max\_size

```
size_type max_size() const;
```

**Returns:** the maximum number of elements that container can hold.

### 11.2.5.3.3.18 Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other, lookup methods and while traversing the container.

### 11.2.5.3.3.19 Emplacing elements

```
template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args )
```

Inserts an element ,constructed in-place from `args` into the container.

**Returns:** `std::pair<iterator, bool>` where `iterator` points to the inserted element.  
Boolean value is always true.

```
template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args )
```

Inserts an element ,constructed in-place from `args` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

**Returns:** an `iterator` to the inserted element.

### 11.2.5.3.3.20 Inserting values

```
std::pair<iterator, bool> insert( const value_type& value )
```

Inserts the value `value` into the container.

**Returns:** `std::pair<iterator, bool>` where `iterator` points to the inserted element.  
Boolean value is always true.

```
iterator insert( const_iterator hint, const value_type& other )
```

Inserts the value `value` into the container.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

**Returns:** an `iterator` to the inserted element.

```
template <typename P>
std::pair<iterator, bool> insert( P&& value )
```

Equivalent to `emplace(std::forward<P>(value))`.

This overload only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is true.

```
template <typename P>
iterator insert( const_iterator hint, P&& value )
```

Equivalent to `emplace_hint(hint, std::forward<P>(value))`.

This overload only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is true.

```
std::pair<iterator, bool> insert( value_type&& value )
```

Inserts the value `value` into the container using move semantics.

`value` is left in a valid, but unspecified state.

**Returns:** `std::pair<iterator, bool>` where `iterator` points to the inserted element. Boolean value is always true.

```
iterator insert( const_iterator hint, value_type&& other )
```

Inserts the value `value` into the container using move semantics.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

`value` is left in a valid, but unspecified state.

**Returns:** an `iterator` to the inserted element.

### 11.2.5.3.3.21 Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last )
```

Inserts all items from the half-open interval `[first, last)` into the container.

**Requirements:** the type `InputIterator` must meet the requirements of `InputIterator` from [input iterators] ISO C++ Standard section.

```
void insert( std::initializer_list<value_type> init )
```

Equivalent to `insert(init.begin(), init.end())`.

### 11.2.5.3.3.22 Inserting nodes

```
std::pair<iterator, bool> insert( node_type&& nh )
```

If the node handle `nh` is empty, does nothing.

Otherwise - inserts the node, owned by `nh` into the container.

`nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if nh is not empty and `get_allocator() != nh.get_allocator()`.

**Returns:** `std::pair<iterator, bool>` where iterator points to the inserted element. Boolean value is always true.

```
iterator insert( const_iterator hint, node_type&& nh )
```

If the node handle nh is empty, does nothing.

Otherwise - inserts the node, owned by nh into the container.

Optionally uses the parameter hint as a suggestion to where the node should be placed.

nh is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if nh is not empty and `get_allocator() != nh.get_allocator()`.

**Returns:** an iterator pointing to the inserted element.

### 11.2.5.3.3.23 Merging containers

```
template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_map<Key, T, SrcHash, SrcKeyEqual, Allocator>
            & source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_map<Key, T, SrcHash, SrcKeyEqual, Allocator>
            && source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_multimap<Key, T, SrcHash, SrcKeyEqual, Allocator>& source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_multimap<Key, T, SrcHash, SrcKeyEqual, Allocator>&& source );
```

Transfers all elements from source to \*this.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `get_allocator() != source.get_allocator()`.

### 11.2.5.3.3.24 Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

### 11.2.5.3.3.25 Clearing

```
void clear();
```

Removes all elements from the container.

### 11.2.5.3.3.26 Erasing elements

```
iterator unsafe_erase( const_iterator pos );
iterator unsafe_erase( iterator pos );
```

Removes the element pointed to by `pos` from the container.

Invalidates all iterators and references to the removed element.

**Returns:** iterator which follows the removed element.

**Requirements:** the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
size_type unsafe_erase( const key_type& key );
```

Removes all elements with the key equal to `key` if it exists in the container.

Invalidates all iterators and references to the removed elements.

**Returns:** the number of removed elements.

```
template <typename K>
size_type unsafe_erase( const K& key );
```

Removes all elements with the key which compares equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed elements.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `hasher::transparent_key_equal` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

**Returns:** the number of removed elements.

### 11.2.5.3.3.27 Erasing sequences

```
iterator unsafe_erase( const_iterator first, const_iterator last );
```

Removes all elements from the half-open interval `[first, last)` from the container.

**Returns:** iterator which follows the last removed element.

**Requirements:** the range `[first, last)` must be a valid subrange in `*this`.

### 11.2.5.3.3.28 Extracting nodes

```
node_type unsafe_extract( iterator pos );
node_type unsafe_extract( const_iterator pos );
```

Transfers ownership of the element pointed to by `pos` from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

**Returns:** the node handle that owns the extracted element.

**Requirements:** the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
node_type unsafe_extract( const key_type& key );
```

If at least one element with the key equal to `key` exists, transfers ownership of one of these element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

If there are multiple elements with the key equal to `key` exists, it is unspecified which element should be transferred.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

**Returns:** the node handle that owns the extracted element or an empty node handle if an element with the key equal to `key` was not found.

```
template <typename K>
node_type unsafe_extract( const K& key );
```

If at least one element with the key which compares equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

If there are multiple elements with the key which compares equivalent with `key` exists, it is unspecified which element should be transferred.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `hasher::transparent_key_equal` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

**Returns:** the node handle that owns the extracted element or an empty node handle if an element with the key which compares equivalent to `key` was not found.

### 11.2.5.3.3.29 swap

```
void swap( concurrent_unordered_multimap& other ) noexcept(/*See below*/);
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is true.

Otherwise if `get_allocator() != other.get_allocator()` the behavior is undefined.

**Exceptions:** `noexcept` specification:

```
noexcept(std::allocator_traits<allocator_type>::is_always_equal::value &&
         std::is_nothrow_swappable<hasher>::value &&
         std::is_nothrow_swappable<key_equal>::value
```

### 11.2.5.3.3.30 Lookup

All methods in this section can be executed concurrently with each other, concurrently-safe modifiers and while traversing the container.

#### 11.2.5.3.3.31 count

```
size_type count( const key_type& key );
```

**Returns:** the number of elements with the key equal to `key`.

```
template <typename K>
size_type count( const K& key );
```

**Returns:** the number of elements with the key which compares equivalent with `key`.

This overload only participates in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

#### 11.2.5.3.3.32 find

```
iterator find( const key_type& key );
const_iterator find( const key_type& key ) const;
```

**Returns:** an iterator to the element with the key equal to `key` or `end()` if no such element exists.

If there are multiple elements with the key equal to `key` exists, it is unspecified which element should be found.

```
template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;
```

**Returns:** an iterator to the element with the key which compares equivalent with key or end() if no such element exists.

If there are multiple elements with the key which compares equivalent with key exists, it is unspecified which element should be found.

These overloads only participates in overload resolution if qualified-id hasher::transparent\_key\_equal is valid and denotes a type.

#### 11.2.5.3.3.33 contains

```
bool contains( const key_type& key ) const;
```

**Returns:** true if at least one element with the key equal to key exists in the container, false otherwise.

```
template <typename K>
bool contains( const K& key ) const;
```

**Returns:** true if at least one element with the key which compares equivalent with key exists in the container, false otherwise.

This overload only participates in overload resolution if qualified-id hasher::transparent\_key\_equal is valid and denotes a type.

#### 11.2.5.3.3.34 equal\_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );
std::pair<const_iterator, const_iterator> equal_range( const key_type& key ) const;
```

**Returns:** if at least one element with the key equal to key exists - a pair of iterators {f, l}, where f is an iterator to the first element with the key equal to key, l is an iterator to the element which follows the last element with the key equal to key. Otherwise - {end(), end() }.

```
template <typename K>
std::pair<iterator, iterator> equal_range( const K& key )

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key )
```

**Returns:** if at least one element with the key equal to key exists - a pair of iterators {f, l}, where f is an iterator to the first element with the key which compares equivalent with key, l is an iterator to the element which follows the last element with the key which compares equivalent with key. Otherwise - {end(), end() }.

These overloads only participates in overload resolution if qualified-id `hasher::transparent_key_equal` is valid and denotes a type.

#### 11.2.5.3.3.35 Bucket interface

The types `concurrent_unordered_multimap::local_iterator` and `concurrent_unordered_multimap::const_local_iterator` meets the requirements of `ForwardIterator` from [forward.iterators] ISO C++ Standard section.

These iterators are used to traverse the certain bucket.

All methods in this section can only be executed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

#### 11.2.5.3.3.36 Bucket begin and bucket end

```
local_iterator unsafe_begin( size_type n );
const_local_iterator unsafe_begin( size_type n ) const;
const_local_iterator unsafe_cbegin( size_type n ) const;
```

**Returns:** an iterator to the first element in the bucket number n.

```
local_iterator unsafe_end( size_type n );
const_local_iterator unsafe_end( size_type n ) const;
const_local_iterator unsafe_cend( size_type n ) const;
```

**Returns:** an iterator to the element which follows the last element in the bucket number n.

#### 11.2.5.3.3.37 The number of buckets

```
size_type unsafe_bucket_count() const;
```

**Returns:** the number of buckets in the container.

```
size_type unsafe_max_bucket_count() const;
```

**Returns:** the maximum number of buckets that container can hold.

#### 11.2.5.3.3.38 Size of the bucket

```
size_type unsafe_bucket_size( size_type n ) const;
```

**Returns:** the number of elements in the bucket number `n`.

#### 11.2.5.3.3.39 Bucket number

```
size_type unsafe_bucket( const key_type& key ) const;
```

**Returns:** the number of the bucket in which the element with the key `key` is stored.

#### 11.2.5.3.3.40 Hash policy

Hash policy of `concurrent_unordered_multimap` manages the number of buckets in the container and the allowed maximum number of elements per bucket (load factor). If the maximum load factor is exceeded, the container can automatically increase the number of buckets.

#### 11.2.5.3.3.41 Load factor

```
float load_factor() const;
```

**Returns:** the average number of elements per bucket, which is `size() / unsafe_bucket_count()`.

```
float max_load_factor() const;
```

**Returns:** the maximum number of elements per bucket.

```
void max_load_factor( float ml );
```

Sets the maximum number of elements per bucket to `ml`.

#### 11.2.5.3.3.42 Manual rehashing

```
void rehash( size_type n );
```

Sets the number of buckets to `n` and rehashes the container.

```
void reserve( size_type n );
```

Sets the number of buckets to the value which is needed to store `n` elements.

#### 11.2.5.3.3.43 Observers

##### 11.2.5.3.3.44 get\_allocator

```
allocator_type get_allocator() const;
```

**Returns:** a copy of the allocator associated with `*this`.

##### 11.2.5.3.3.45 hash\_function

```
hasher hash_function() const;
```

**Returns:** a copy of the hash function associated with `*this`.

##### 11.2.5.3.3.46 key\_eq

```
key_equal key_eq() const;
```

**Returns:** a copy of the key equality predicate associated with `*this`.

#### 11.2.5.3.3.47 Parallel iteration

Member types `concurrent_unordered_multimap::range_type` and `concurrent_unordered_multimap::const_range_type` meets the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_unordered_multimap::const_range_type` are of type `concurrent_unordered_multimap::const_iterator`, whereas the bounds for a `concurrent_unordered_multimap::range_type` are of type `concurrent_unordered_multimap::iterator`.

##### 11.2.5.3.3.48 range member function

```
range_type range();
const_range_type range() const;
```

**Returns:** a range object representing all elements in the container.

#### 11.2.5.3.3.49 Non-member functions

These functions provides binary comparison and swap operations on `tbb::concurrent_unordered_multimap` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `tbb::concurrent_unordered_multimap` as a type alias for which the non-member functions are reachable only via argument dependent lookup.

```

template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
void swap( concurrent_unordered_multimap<Key, T, Hash, KeyEqual, Allocator>& lhs,
            concurrent_unordered_multimap<Key, T, Hash, KeyEqual, Allocator>& rhs );

template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_multimap<Key, T, Hash, KeyEqual,_
          Allocator>& lhs,
                  const concurrent_unordered_multimap<Key, T, Hash, KeyEqual,_
          Allocator>& rhs );

template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_multimap<Key, T, Hash, KeyEqual,_
          Allocator>& lhs,
                  const concurrent_unordered_multimap<Key, T, Hash, KeyEqual,_
          Allocator>& rhs );

```

### 11.2.5.3.3.50 Non-member swap

```

template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
void swap( concurrent_unordered_multimap<Key, T, Hash, KeyEqual, Allocator>& lhs,
            concurrent_unordered_multimap<Key, T, Hash, KeyEqual, Allocator>& rhs )_
          noexcept(noexcept(lhs.swap(rhs)));

```

Equivalent to `lhs.swap(rhs)`.

### 11.2.5.3.3.51 Non-member binary comparisons

Two objects of `concurrent_unordered_multimap` are equal if the following conditions are true:

- They contains an equal number of elements.
- Each group of elements with the same key in one container has the corresponding group of equivalent elements in the other container (not necessary in the same order).

```

template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_multimap<Key, T, Hash, KeyEqual,_
          Allocator>& lhs,
                  const concurrent_unordered_multimap<Key, T, Hash, KeyEqual,_
          Allocator>& rhs );

```

**Returns:** `true` if `lhs` is equal to `rhs`, `false` otherwise.

```

template <typename Key, typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator!=( const concurrent_unordered_multimap<Key, T, Hash, KeyEqual,_
          Allocator>& lhs,
                  const concurrent_unordered_multimap<Key, T, Hash, KeyEqual,_
          Allocator>& rhs );

```

Equivalent to `!(lhs == rhs)`.

**Returns:** true if lhs is not equal to rhs, false otherwise.

### 11.2.5.3.3.52 Other

#### 11.2.5.3.3.53 Deduction guides

Where possible, constructors of `concurrent_unordered_multimap` supports class template argument deduction (since C++17):

```
template <typename InputIterator,
         typename Hash = std::hash<iterator_key_t<InputIterator>>,
         typename KeyEqual = std::equal_to<iterator_key_t<InputIterator>>,
         typename Allocator = tbb_allocator<iterator_alloc_value_t<InputIterator>>>
concurrent_unordered_multimap( InputIterator, InputIterator,
                               map_size_type = /*implementation defined*/,
                               Hash = Hash(), KeyEqual = KeyEqual(),
                               Allocator = Allocator() )
-> concurrent_unordered_multimap<iterator_key_t<InputIterator>,
                                         iterator_mapped_t<InputIterator>,
                                         Hash, KeyEqual, Allocator>;
```

```
template <typename InputIterator,
         typename Allocator>
concurrent_unordered_multimap( InputIterator, InputIterator,
                               map_size_type,
                               Allocator )
-> concurrent_unordered_multimap<iterator_key_t<InputIterator>,
                                         iterator_mapped_t<InputIterator>,
                                         std::hash<iterator_key_t<InputIterator>>,
                                         std::equal_to<iterator_key_t<InputIterator>>,
                                         Allocator>;
```

```
template <typename InputIterator,
         typename Allocator>
concurrent_unordered_multimap( InputIterator, InputIterator, Allocator )
-> concurrent_unordered_multimap<iterator_key_t<InputIterator>,
                                         iterator_mapped_t<InputIterator>,
                                         std::hash<iterator_key_t<InputIterator>>,
                                         std::equal_to<iterator_key_t<InputIterator>>,
                                         Allocator>;
```

```
template <typename InputIterator,
         typename Hash,
         typename Allocator>
concurrent_unordered_multimap( InputIterator, InputIterator,
                               Hash, Allocator )
-> concurrent_unordered_multimap<iterator_key_t<InputIterator>,
                                         iterator_mapped_t<InputIterator>,
                                         Hash,
                                         std::equal_to<iterator_key_t<InputIterator>>,
                                         Allocator>;
```

```
template <typename Key,
         typename T,
```

(continues on next page)

(continued from previous page)

```

typename Hash = std::hash<Key>,
typename KeyEqual = std::equal_to<Key>,
typename Allocator = tbb_allocator<std::pair<Key, T>>>
concurrent_unordered_multimap( std::initializer_list<value_type>,
                               map_size_type = /*implementation-defined*/,
                               Hash = Hash(),
                               KeyEqual = KeyEqual(),
                               Allocator = Allocator() )
-> concurrent_unordered_multimap<Key, T,
                                    Hash,
                                    KeyEqual,
                                    Allocator>;

template <typename Key,
          typename T,
          typename Allocator>
concurrent_unordered_multimap( std::initializer_list<value_type>,
                               map_size_type, Allocator )
-> concurrent_unordered_multimap<Key, T,
                                    std::hash<Key>,
                                    std::equal_to<Key>,
                                    Allocator>;

template <typename Key,
          typename T,
          typename Hash,
          typename Allocator>
concurrent_unordered_multimap( std::initializer_list<value_type>,
                               map_size_type, Hash, Allocator )
-> concurrent_unordered_multimap<Key, T,
                                    Hash,
                                    std::equal_to<Key>,
                                    Allocator>;

```

Where the type `map_size_type` refers to the `size_type` member type of the deduced `concurrent_multimap` and the type aliases `iterator_key_t`, `iterator_mapped_t` and `iterator_alloc_value_t` defines as follows:

```

template <typename InputIterator>
using iterator_key_t = std::remove_const_t<typename std::iterator_traits<
  ↵<InputIterator>::value_type::first_type>>;
template <typename InputIterator>
using iterator_mapped_t = typename std::iterator_traits<InputIterator>::value_
  ↵type::second_type;
template <typename InputIterator>
using iterator_alloc_value_t = std::pair<std::add_const_t<iterator_key_t
  ↵<InputIterator>,
  iterator_mapped_t<InputIterator>>>;

```

### Example

```
#include <tbb/concurrent_unordered_map.h>
#include <vector>
#include <functional>
```

(continues on next page)

(continued from previous page)

```

struct CustomHasher {...};

int main() {
    std::vector<std::pair<int, float>> v;

    // Deduces m1 as concurrent_unordered_multimap<int, float>
    tbb::concurrent_unordered_multimap m1(v.begin(), v.end());

    // Deduces m2 as concurrent_unordered_multimap<int, float, CustomHasher>;
    tbb::concurrent_unordered_multimap m2(v.begin(), v.end(), CustomHasher{});
}

```

#### 11.2.5.3.4 concurrent\_unordered\_set

[**containers.concurrent\_unordered\_set**]

**tbb::concurrent\_unordered\_set** is a class template represents an unordered sequence of unique elements which supports concurrent insertion, lookup and traversal, but not concurrent erasure.

##### 11.2.5.3.4.1 Class Template Synopsis

```

// Defined in header <tbb/concurrent_unordered_set.h>

namespace tbb {
    template <typename T,
              typename Hash = std::hash<Key>,
              typename KeyEqual = std::equal_to<Key>,
              typename Allocator = tbb_allocator<std::pair<const Key, T>>>
    class concurrent_unordered_set {
    public:
        using key_type = Key;
        using value_type = Key;

        using size_type = <implementation-defined unsigned integer type>;
        using difference_type = <implementation-defined signed integer type>;

        using hasher = Hash;
        using key_equal = /*See below*/;

        using allocator_type = Allocator;

        using reference = value_type&;
        using const_reference = const value_type&;

        using pointer = typename std::allocator_traits<Allocator>::pointer;
        using const_pointer = typename std::allocator_traits<Allocator>::const_
        ↪pointer;

        using iterator = <implementation-defined ForwardIterator>;
        using const_iterator = <implementation-defined constant ForwardIterator>;

        using local_iterator = <implementation-defined ForwardIterator>;

```

(continues on next page)

(continued from previous page)

```

using const_local_iterator = <implementation-defined constant ForwardIterator>
 $\leftrightarrow$ ;

using node_type = <implementation-defined node handle>;

using range_type = <implementation-defined ContainerRange>;
using const_range_type = <implementation-defined constant ContainerRange>;

// Construction, destruction, copying
concurrent_unordered_set();

explicit concurrent_unordered_set( size_type bucket_count, const hasher& hash_
 $\leftrightarrow$ = hasher(),
 $\quad\quad\quad$  const key_equal& equal = key_equal(),
 $\quad\quad\quad$  const allocator_type& alloc = allocator_
 $\leftrightarrow$ type() );

concurrent_unordered_set( size_type bucket_count, const allocator_type& alloc_
 $\leftrightarrow$ );

concurrent_unordered_set( size_type bucket_count, const hasher& hash,
 $\quad\quad\quad$  const allocator_type& alloc );

explicit concurrent_unordered_set( const allocator_type& alloc );

template <typename InputIterator>
concurrent_unordered_set( InputIterator first, InputIterator last,
 $\quad\quad\quad$  size_type bucket_count = /*implementation-defined*/,
 $\quad\quad\quad$  const hasher& hash = hasher(),
 $\quad\quad\quad$  const key_equal& equal = key_equal(),
 $\quad\quad\quad$  const allocator_type& alloc = allocator_type() );

template <typename Inputiterator>
concurrent_unordered_set( InputIterator first, InputIterator last,
 $\quad\quad\quad$  size_type bucket_count, const allocator_type& alloc_
 $\leftrightarrow$ );

template <typename InputIterator>
concurrent_unordered_set( InputIterator first, InputIterator last,
 $\quad\quad\quad$  size_type bucket_count, const hasher& hash,
 $\quad\quad\quad$  const allocator_type& alloc );

concurrent_unordered_set( std::initializer_list<value_type> init,
 $\quad\quad\quad$  size_type bucket_count = /*implementation-defined*/,
 $\quad\quad\quad$  const hasher& hash = hasher(),
 $\quad\quad\quad$  const key_equal& equal = key_equal(),
 $\quad\quad\quad$  const allocator_type& alloc = allocator_type() );

concurrent_unordered_set( std::initializer_list<value_type> init,
 $\quad\quad\quad$  size_type bucket_count, const allocator_type& alloc_
 $\leftrightarrow$ );

concurrent_unordered_set( std::initializer_list<value_type> init,
 $\quad\quad\quad$  size_type bucket_count, const hasher& hash,
 $\quad\quad\quad$  const allocator_type& alloc );

concurrent_unordered_set( const concurrent_unordered_set& other );

```

(continues on next page)

(continued from previous page)

```

concurrent_unordered_set( const concurrent_unordered_set& other,
                         const allocator_type& alloc );

concurrent_unordered_set( concurrent_unordered_set&& other );
concurrent_unordered_set( concurrent_unordered_set&& other,
                         const allocator_type& alloc );

~concurrent_unordered_set();

concurrent_unordered_set& operator=( const concurrent_unordered_set& other );
concurrent_unordered_set& operator=( concurrent_unordered_set&& other )  

→noexcept(/*See details*/);

concurrent_unordered_set& operator=( std::initializer_list<value_type> init );

allocator_type get_allocator() const;

// Iterators
iterator begin() noexcept;
const_iterator begin() const noexcept;
const_iterator cbegin() const noexcept;

iterator end() noexcept;
const_iterator end() const noexcept;
const_iterator cend() const noexcept;

// Size and capacity
bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// Concurrently safe modifiers
std::pair<iterator, bool> insert( const value_type& value );
iterator insert( const_iterator hint, const value_type& value );

std::pair<iterator, bool> insert( value_type&& value );
iterator insert( const_iterator hint, value_type&& value );

template <typename InputIterator>
void insert( InputIterator first, InputIterator last );

void insert( std::initializer_list<value_type> init );

std::pair<iterator, bool> insert( node_type&& nh );
iterator insert( const_iterator hint, node_type&& nh );

template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args );

template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_set<T, SrcHash, SrcKeyEqual, Allocator>&  

→source );

template <typename SrcHash, typename SrcKeyEqual>

```

(continues on next page)

(continued from previous page)

```

void merge( concurrent_unordered_set<T, SrcHash, SrcKeyEqual, Allocator>&&_  

source );  
  

template <typename SrcHash, typename SrcKeyEqual>  

void merge( concurrent_unordered_multiset<T, SrcHash, SrcKeyEqual, Allocator>&  

source );  
  

template <typename SrcHash, typename SrcKeyEqual>  

void merge( concurrent_unordered_multiset<T, SrcHash, SrcKeyEqual, Allocator>&  

source );  
  

// Concurrently unsafe modifiers  

void clear() noexcept;  
  

iterator unsafe_erase( const_iterator pos );  

iterator unsafe_erase( iterator pos );  
  

iterator unsafe_erase( const_iterator first, const_iterator last );  
  

size_type unsafe_erase( const key_type& key );  
  

template <typename K>  

size_type unsafe_erase( const K& key );  
  

node_type unsafe_extract( const_iterator pos );  

node_type unsafe_extract( iterator pos );  
  

node_type unsafe_extract( const key_type& key );  
  

template <typename K>  

node_type unsafe_extract( const K& key );  
  

void swap( concurrent_unordered_set& other );  
  

// Lookup  

size_type count( const key_type& key ) const;  
  

template <typename K>  

size_type count( const K& key ) const;  
  

iterator find( const key_type& key );  

const_iterator find( const key_type& key ) const;  
  

template <typename K>  

iterator find( const K& key );  
  

template <typename K>  

const_iterator find( const K& key ) const;  
  

bool contains( const key_type& key ) const;  
  

template <typename K>  

bool contains( const K& key ) const;  
  

std::pair<iterator, iterator> equal_range( const key_type& key );  

std::pair<const_iterator, const_iterator> equal_range( const key_type& key )  

const;
```

(continues on next page)

(continued from previous page)

```

template <typename K>
std::pair<iterator, iterator> equal_range( const K& key );

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key ) const;

// Bucket interface
local_iterator unsafe_begin( size_type n );
const_local_iterator unsafe_begin( size_type n ) const;
const_local_iterator unsafe_cbegin( size_type n ) const;

local_iterator unsafe_end( size_type n );
const_local_iterator unsafe_end( size_type n ) const;
const_local_iterator unsafe_cend( size_type n ) const;

size_type unsafe_bucket_count() const;
size_type unsafe_max_bucket_bound() const;

size_type unsafe_bucket_size( size_type n ) const;

size_type unsafe_bucket( const key_type& key ) const;

// Hash policy
float load_factor() const;

float max_load_factor() const;
void max_load_factor( float ml );

void rehash( size_type count );

void reserve( size_type count );

// Observers
hasher hash_function() const;
key_equal key_eq() const;

// Parallel iteration
range_type range();
const_range_type range() const;
};

// class concurrent_unordered_set
} // namespace tbb

```

#### Requirements:

- The expression `std::allocator_type<Allocator>::destroy(m, val)` where `m` is an object of the type `Allocator` and `val` is an object of type `value_type` should be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Hash` shall meet the `Hash` requirements from [hash] ISO C++ Standard section.
- The type `KeyEqual` shall meet the `BinaryPredicate` requirements from [algorithms.general] ISO C++ Standard section.
- The type `Allocator` shall meet the `Allocator` requirements from [allocator.requirements] ISO C++ Standard section.

#### 11.2.5.3.4.2 Description

`tbb::concurrent_unordered_set` is an unordered sequence, which elements are organized into buckets. The value of the hash function `Hash` for `Key` object determines the number of the bucket in which the corresponding element would be placed.

If the qualified-id `Hash::transparent_key_equal` is valid and denotes a type, member type `concurrent_unordered_set::key_equal` defines as the value of this qualified-id. In this case the program is ill-formed if any of the following conditions are met:

- The template parameter `KeyEqual` is different from `std::equal_to<Key>`.
- Qualified-id `Hash::transparent_key_equal::is_transparent` is not valid or not denotes a type.

Otherwise, member type `concurrent_unordered_set::key_equal` defines as the value of the template parameter `KeyEqual`.

#### 11.2.5.3.4.3 Member functions

##### 11.2.5.3.4.4 Construction, destruction, copying

##### 11.2.5.3.4.5 Empty container constructors

```
concurrent_unordered_set();

explicit concurrent_unordered_set( const allocator_type& alloc );
```

Constructs an empty `concurrent_unordered_set`. The initial number of buckets is unspecified.

If provided uses the allocator `alloc` to allocate the memory.

```
explicit concurrent_unordered_set( size_type bucket_count,
                                    const hasher& hash = hasher(),
                                    const key_equal& equal = key_equal(),
                                    const allocator_type& alloc = allocator_
                                     ↵type() );

concurrent_unordered_set( size_type bucket_count, const allocator_type&↳
                         ↵alloc );

concurrent_unordered_set( size_type bucket_count, const hasher& hash,
                         const allocator_type& alloc );
```

Constructs an empty `concurrent_unordered_set` with `bucket_count` buckets.

If provided uses the hash function `hasher`, predicate `equal` to compare `key_type` objects for equality and the allocator `alloc` to allocate the memory.

#### 11.2.5.3.4.6 Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_unordered_set( InputIterator first, InputIterator last,
                         size_type bucket_count = /*implementation-defined*/
                         ↵,
                         const hasher& hash = hasher(),
                         const key_equal& equal = key_equal(),
                         const allocator_type& alloc = allocator_type() );

template <typename InputIterator>
concurrent_unordered_set( InputIterator first, InputIterator last,
                         size_type bucket_count, const allocator_type&_
                         ↵alloc );

template <typename InputIterator>
concurrent_unordered_set( InputIterator first, InputIterator last,
                         size_type bucket_count, const hasher& hash,
                         const allocator_type& alloc );
```

Constructs the concurrent\_unordered\_set which contains the elements from the half-open interval [first, last).

If the range [first, last) contains multiple equal elements, it is unspecified which element would be inserted.

If provided uses the hash function hasher, predicate equal to compare key\_type objects for equality and the allocator alloc to allocate the memory.

**Requirements:** the type InputIterator shall meet the requirements of InputIterator from [input.iterators] ISO C++ Standard section.

```
concurrent_unordered_set( std::initializer_list<value_type> init,
                         size_type bucket_count = /*implementation-defined*/
                         ↵,
                         const hasher& hash = hasher(),
                         const key_equal& equal = key_equal(),
                         const allocator_type& alloc = allocator_type() );
```

Equivalent to concurrent\_unordered\_set(init.begin(), init.end(),  
bucket\_count, hash, equal, alloc).

```
concurrent_unordered_set( std::initializer_list<value_type> init,
                         size_type bucket_count, const allocator_type&_
                         ↵alloc );
```

Equivalent to concurrent\_unordered\_set(init.begin(), init.end(),  
bucket\_count, alloc).

```
concurrent_unordered_set( std::initializer_list<value_type> init,
                         size_type bucket_count, const hasher& hash,
                         const allocator_type& alloc );
```

Equivalent to concurrent\_unordered\_set(init.begin(), init.end(),  
bucket\_count, hash, alloc).

#### 11.2.5.3.4.7 Copying constructors

```
concurrent_unordered_set( const concurrent_unordered_set& other );
concurrent_unordered_set( const concurrent_unordered_set& other,
                        const allocator_type& alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

#### 11.2.5.3.4.8 Moving constructors

```
concurrent_unordered_set( concurrent_unordered_set&& other );
concurrent_unordered_set( concurrent_unordered_set&& other,
                        const allocator_type& alloc );
```

Constructs a `concurrent_unordered_set` with the contents of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

#### 11.2.5.3.4.9 Destructor

```
~concurrent_unordered_set();
```

Destroys the `concurrent_unordered_set`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

#### 11.2.5.3.4.10 Assignment operators

```
concurrent_unordered_set& operator=( const concurrent_unordered_set& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment` is true.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

**Returns:** a reference to `*this`.

```
concurrent_unordered_set& operator=( concurrent_unordered_set&& other )  
→ noexcept /*See below*/;
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment` is true.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

**Returns:** a reference to `*this`.

**Exceptions:** noexcept specification:

```
noexcept (std::allocator_traits<allocator_type>::is_always_  
→ equal::value &&  
         std::is_nothrow_moveAssignable<hasher>::value &&  
         std::is_nothrow_moveAssignable<key_equal>::value)
```

```
concurrent_unordered_set& operator=( std::initializer_list<value_type> init  
→ );
```

Replaces all elements in `*this` by the elements in `init`.

If `init` contains multiple equal elements, it is unspecified which element would be inserted.

The behavior is undefined in case of concurrent operations with `*this`.

**Returns:** a reference to `*this`.

#### 11.2.5.3.4.11 Iterators

The types `concurrent_unordered_set::iterator` and `concurrent_unordered_set::const_iterator` meets the requirements of `ForwardIterator` from [forward.iterators] ISO C++ Standard section.

#### 11.2.5.3.4.12 begin and cbegin

```
iterator begin();  
  
const_iterator begin() const;  
  
const_iterator cbegin() const;
```

**Returns:** an iterator to the first element in the container.

#### 11.2.5.3.4.13 end and cend

```
iterator end();
const_iterator end() const;
const_iterator cend() const;
```

**Returns:** an iterator to the element which follows the last element in the container.

#### 11.2.5.3.4.14 Size and capacity

#### 11.2.5.3.4.15 empty

```
bool empty() const;
```

**Returns:** true if the container is empty, false otherwise.

The result may differ with the actual container state in case of pending concurrent insertions.

#### 11.2.5.3.4.16 size

```
size_type size() const;
```

**Returns:** the number of elements in the container.

The result may differ with the actual container size in case of pending concurrent insertions.

#### 11.2.5.3.4.17 max\_size

```
size_type max_size() const;
```

**Returns:** the maximum number of elements that container can hold.

#### 11.2.5.3.4.18 Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other, lookup methods and while traversing the container.

#### 11.2.5.3.4.19 Inserting values

```
std::pair<iterator, bool> insert( const value_type& value );
```

Attempts to insert the value value into the container.

**Returns:** std::pair<iterator, bool> where iterator points to the inserted element or to an existing equal element. Boolean value is true if insertion took place, false otherwise.

**Requirements:** the type value\_type shall meet the CopyInsertable requirements from [container.requirements] ISO C++ Standard section.

---

```
iterator insert( const_iterator hint, const value_type& other );
```

Attempts to insert the value `value` into the container.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

**Returns:** an `iterator` to the inserted element or to an existing equal element.

**Requirements:** the type `value_type` shall meet the `CopyInsertable` requirements from [container.requirements] ISO C++ Standard section.

---

```
std::pair<iterator, bool

```

Attempts to insert the value `value` into the container using move semantics.

`value` is left in a valid, but unspecified state.

**Returns:** `std::pair<iterator, bool>` where `iterator` points to the inserted element or to an existing equal element. Boolean value is `true` if insertion took place, `false` otherwise.

**Requirements:** the type `value_type` shall meet the `MoveInsertable` requirements from [container.requirements] ISO C++ Standard section.

---

```
iterator insert( const_iterator hint, value_type&& other );
```

Attempts to insert the value `value` into the container using move semantics.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

`value` is left in a valid, but unspecified state.

**Returns:** an `iterator` to the inserted element or to an existing equal element.

**Requirements:** the type `value_type` shall meet the `MoveInsertable` requirements from [container.requirements] ISO C++ Standard section.

#### 11.2.5.3.4.20 Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last );
```

Attempts to insert all items from the half-open interval `[first, last)` into the container.

If the interval `[first, last)` contains multiple equal elements, it is unspecified which element should be inserted.

**Requirements:** the type `InputIterator` must meet the requirements of `InputIterator` from [input.iterators] ISO C++ Standard section.

---

```
void insert( std::initializer_list<value_type> init );
```

Equivalent to `insert(init.begin(), init.end())`.

#### 11.2.5.3.4.21 Inserting nodes

```
std::pair<iterator, bool> insert( node_type&& nh );
```

If the node handle nh is empty, does nothing.

Otherwise - attempts to insert the node, owned by nh into the container.

If the insertion fails, node handle nh remains ownership of the node.

Otherwise - nh is left in an empty state.

No copy or move constructors of value\_type are performed.

The behavior is undefined if nh is not empty and get\_allocator() != nh.get\_allocator().

**Returns:** std::pair<iterator, bool> where iterator points to the inserted element or to an existing element equal to nh.value(). Boolean value is true if insertion took place, false otherwise.

```
iterator insert( const_iterator hint, node_type&& nh );
```

If the node handle nh is empty, does nothing.

Otherwise - attempts to insert the node, owned by nh into the container.

Optionally uses the parameter hint as a suggestion to where the node should be placed.

If the insertion fails, node handle nh remains ownership of the node.

Otherwise - nh is left in an empty state.

No copy or move constructors of value\_type are performed.

The behavior is undefined if nh is not empty and get\_allocator() != nh.get\_allocator().

**Returns:** an iterator pointing to the inserted element or to an existing element equal to nh.value().

#### 11.2.5.3.4.22 Emplacing elements

```
template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args );
```

Attempts to insert an element ,constructed in-place from args into the container.

**Returns:** std::pair<iterator, bool> where iterator points to the inserted element or to an existing equal element. Boolean value is true if insertion took place, false otherwise.

**Requirements:** the type value\_type shall meet the EmplaceConstructible requirements from [container.requirements] ISO C++ Standard section.

```
template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );
```

Attempts to insert an element ,constructed in-place from args into the container.

Optionally uses the parameter hint as a suggestion to where the node should be placed.

**Returns:** an iterator to the inserted element or to an existing equal element.

**Requirements:** the type `value_type` shall meet the `EmplaceConstructible` requirements from [container.requirements] ISO C++ Standard section.

#### 11.2.5.3.4.23 Merging containers

```
template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_set<T, SrcHash, SrcKeyEqual, Allocator>&,
            source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_set<T, SrcHash, SrcKeyEqual, Allocator>&&,
            source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_multiset<T, SrcHash, SrcKeyEqual, Allocator>,
            & source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_multiset<T, SrcHash, SrcKeyEqual, Allocator>,
            && source );
```

Transfers those elements from `source` which do not exist in the container.

In case of merging with the container with multiple equal elements, it is unspecified which element would be transferred.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `get_allocator() != source.get_allocator()`.

#### 11.2.5.3.4.24 Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

#### 11.2.5.3.4.25 Clearing

```
void clear();
```

Removes all elements from the container.

#### 11.2.5.3.4.26 Erasing elements

```
iterator unsafe_erase( const_iterator pos );
iterator unsafe_erase( iterator pos );
```

Removes the element pointed to by `pos` from the container.

Invalidates all iterators and references to the removed element.

**Returns:** iterator which follows the removed element.

**Requirements:** the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
size_type unsafe_erase( const key_type& key );
```

Removes the element equal to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

**Returns:** 1 if an element equal to `key` exists, 0 otherwise.

```
template <typename K>
size_type unsafe_erase( const K& key );
```

Removes the element which compares equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

This overload only participates in overload resolution if all of the following statements are true:

- The qualified-id `hasher::transparent_key_equal` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

**Returns:** 1 if an element which compares equivalent to `key` exists, 0 otherwise.

#### 11.2.5.3.4.27 Erasing sequences

```
iterator unsafe_erase( const_iterator first, const_iterator last );
```

Removes all elements from the half-open interval `[first, last)` from the container.

**Returns:** iterator which follows the last removed element.

**Requirements:** the range `[first, last)` must be a valid subrange in `*this`.

#### 11.2.5.3.4.28 Extracting nodes

```
node_type unsafe_extract( iterator pos );
node_type unsafe_extract( const_iterator pos );
```

Transfers ownership of the element pointed to by `pos` from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

**Returns:** the node handle that owns the extracted element.

**Requirements:** the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
node_type unsafe_extract( const key_type& key );
```

If an element equal to key exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of value\_type are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

**Returns:** the node handle that owns the extracted element or an empty node handle if an element equal to key was not found.

```
template <typename K>
node_type unsafe_extract( const K& key );
```

If an element which compares equivalent to key exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of value\_type are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

This overload only participates in overload resolution if all of the following statements are true:

- The qualified-id hasher::transparent\_key\_equal is valid and denotes a type.
- std::is\_convertible<K, iterator>::value is false.
- std::is\_convertible<K, const\_iterator>::value is false.

**Returns:** the node handle that owns the extracted element or an empty node handle if an element which compares equivalent to key was not found.

#### 11.2.5.3.4.29 swap

```
void swap( concurrent_unordered_set& other ) noexcept(/*See below*/);
```

Swaps contents of \*this and other.

Swaps allocators if std::allocator\_traits<allocator\_type>::propagate\_on\_container\_swap::value is true.

Otherwise if get\_allocator() != other.get\_allocator() the behavior is undefined.

**Exceptions:** noexcept specification:

```
noexcept(std::allocator_traits<allocator_type>::is_always_
equal::value &&
    std::is_nothrow_swappable<hasher>::value &&
    std::is_nothrow_swappable<key_equal>::value)
```

### 11.2.5.3.4.30 Lookup

All methods in this section can be executed concurrently with each other, concurrently-safe modifiers and while traversing the container.

#### 11.2.5.3.4.31 count

```
size_type count( const key_type& key );
```

**Returns:** the number of elements equal to key.

```
template <typename K>
size_type count( const K& key );
```

**Returns:** the number of elements which compares equivalent with key.

This overload only participates in overload resolution if qualified-id hasher::transparent\_key\_equal is valid and denotes a type.

#### 11.2.5.3.4.32 find

```
iterator find( const key_type& key );
const_iterator find( const key_type& key ) const;
```

**Returns:** an iterator to the element equal to key or end() if no such element exists.

```
template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;
```

**Returns:** an iterator to the element which compares equivalent with key or end() if no such element exists.

These overloads only participates in overload resolution if qualified-id hasher::transparent\_key\_equal is valid and denotes a type.

#### 11.2.5.3.4.33 contains

```
bool contains( const key_type& key ) const;
```

**Returns:** true if an element equal to key exists in the container, false otherwise.

```
template <typename K>
bool contains( const K& key ) const;
```

**Returns:** true if an element which compares equivalent with key exists in the container, false otherwise.

This overload only participates in overload resolution if qualified-id hasher::transparent\_key\_equal is valid and denotes a type.

#### 11.2.5.3.4.34 equal\_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );
std::pair<const_iterator, const_iterator> equal_range( const key_type& key ) const;
```

**Returns:** if an element equal to key exists - a pair of iterators {f, l}, where f is an iterator to this element, l is std::next(f). Otherwise - {end(), end() }.

```
template <typename K>
std::pair<iterator, iterator> equal_range( const K& key )

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key ) const
```

**Returns:** if an element which compares equivalent with key exists - a pair of iterators {f, l}, where f is an iterator to this element, l is std::next(f). Otherwise - {end(), end() }.

These overloads only participates in overload resolution if qualified-id hasher::transparent\_key\_equal is valid and denotes a type.

#### 11.2.5.3.4.35 Bucket interface

The types concurrent\_unordered\_set::local\_iterator and concurrent\_unordered\_set::const\_local\_iterator meets the requirements of ForwardIterator from [forward.iterators] ISO C++ Standard section.

These iterators are used to traverse the certain bucket.

All methods in this section can only be executed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

#### 11.2.5.3.4.36 Bucket begin and bucket end

```
local_iterator unsafe_begin( size_type n );
const_local_iterator unsafe_begin( size_type n ) const;
const_local_iterator unsafe_cbegin( size_type n ) const;
```

**Returns:** an iterator to the first element in the bucket number n.

```
local_iterator unsafe_end( size_type n );
const_local_iterator unsafe_end( size_type n ) const;
const_local_iterator unsafe_cend( size_type n ) const;
```

**Returns:** an iterator to the element which follows the last element in the bucket number n.

#### 11.2.5.3.4.37 The number of buckets

```
size_type unsafe_bucket_count() const;
```

**Returns:** the number of buckets in the container.

```
size_type unsafe_max_bucket_count() const;
```

**Returns:** the maximum number of buckets that container can hold.

#### 11.2.5.3.4.38 Size of the bucket

```
size_type unsafe_bucket_size( size_type n ) const;
```

**Returns:** the number of elements in the bucket number n.

#### 11.2.5.3.4.39 Bucket number

```
size_type unsafe_bucket( const key_type& key ) const;
```

**Returns:** the number of the bucket in which the element with the key key is stored.

#### 11.2.5.3.4.40 Hash policy

Hash policy of concurrent\_unordered\_set manages the number of buckets in the container and the allowed maximum number of elements per bucket (load factor). If the maximum load factor is exceeded, the container can automatically increase the number of buckets.

#### 11.2.5.3.4.41 Load factor

```
float load_factor() const;
```

**Returns:** the average number of elements per bucket, which is size() / unsafe\_bucket\_count().

```
float max_load_factor() const;
```

**Returns:** the maximum number of elements per bucket.

---

```
void max_load_factor( float ml );
```

Sets the maximum number of elements per bucket to ml.

#### 11.2.5.3.4.42 Manual rehashing

```
void rehash( size_type n );
```

Sets the number of buckets to n and rehashes the container.

```
void reserve( size_type n );
```

Sets the number of buckets to the value which is needed to store n elements.

#### 11.2.5.3.4.43 Observers

##### 11.2.5.3.4.44 get\_allocator

```
allocator_type get_allocator() const;
```

**Returns:** a copy of the allocator associated with \*this.

##### 11.2.5.3.4.45 hash\_function

```
hasher hash_function() const;
```

**Returns:** a copy of the hash function associated with \*this.

##### 11.2.5.3.4.46 key\_eq

```
key_equal key_eq() const;
```

**Returns:** a copy of the key equality predicate associated with \*this.

#### 11.2.5.3.4.47 Parallel iteration

Member types concurrent\_unordered\_set::range\_type and concurrent\_unordered\_set::const\_range\_type meets the *ContainerRange requirements*.

These types differ only in that the bounds for a concurrent\_unordered\_set::const\_range\_type are of type concurrent\_unordered\_set::const\_iterator, whereas the bounds for a concurrent\_unordered\_set::range\_type are of type concurrent\_unordered\_set::iterator.

#### 11.2.5.3.4.48 range member function

```
range_type range();
const_range_type range() const;
```

**Returns:** a range object representing all elements in the container.

#### 11.2.5.3.4.49 Non-member functions

These functions provides binary comparison and swap operations on `tbb::concurrent_unordered_set` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `tbb::concurrent_unordered_set` as a type alias for which the non-member functions are reachable only via argument dependent lookup.

```
template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
void swap( concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& lhs,
            concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& rhs );

template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& lhs,
                    const concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& rhs );

template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& lhs,
                    const concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& rhs );
```

#### 11.2.5.3.4.50 Non-member swap

```
template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
void swap( concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& lhs,
            concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& rhs ) ↳
→noexcept(noexcept(lhs.swap(rhs)));
```

Equivalent to `lhs.swap(rhs)`.

### 11.2.5.3.4.51 Non-member binary comparisons

Two objects of concurrent\_unordered\_set are equal if the following conditions are true:

- They contains an equal number of elements.
- Each element from the one container also contains in the other.

```
template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& lhs,
                    const concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& rhs );
```

**Returns:** true if lhs is equal to rhs, false otherwise.

```
template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator!=( const concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& lhs,
                    const concurrent_unordered_set<T, Hash, KeyEqual, Allocator>& rhs );
```

Equivalent to !(lhs == rhs).

**Returns:** true if lhs is not equal to rhs, false otherwise.

### 11.2.5.3.4.52 Other

#### 11.2.5.3.4.53 Deduction guides

Where possible, constructors of concurrent\_unordered\_set supports class template argument deduction (since C++17):

```
template <typename InputIterator,
          typename Hash = std::hash<iterator_value_t<InputIterator>>,
          typename KeyEqual = std::equal_to<iterator_value_t<InputIterator>>,
          typename Allocator = tbb_allocator<iterator_value_t<InputIterator>>>
concurrent_unordered_set( InputIterator, InputIterator,
                         map_size_type = /*implementation_defined*/,
                         Hash = Hash(), KeyEqual = KeyEqual(),
                         Allocator = Allocator() )
-> concurrent_unordered_set<iterator_value_t<InputIterator>,
                           Hash, KeyEqual, Allocator>;
```

```
template <typename InputIterator,
          typename Allocator>
concurrent_unordered_set( InputIterator, InputIterator,
                         map_size_type,
                         Allocator )
-> concurrent_unordered_set<iterator_value_t<InputIterator>,
                           std::hash<iterator_value_t<InputIterator>>,
                           std::equal_to<iterator_value_t<InputIterator>>,
                           Allocator>;
```

```
template <typename InputIterator,
          typename Allocator>
concurrent_unordered_set( InputIterator, InputIterator, Allocator )
```

(continues on next page)

(continued from previous page)

```

-> concurrent_unordered_set<iterator_value_t<InputIterator>,
    std::hash<iterator_value_t<InputIterator>>,
    std::equal_to<iterator_key_t<InputIterator>>,
    Allocator>;

template <typename InputIterator,
         typename Hash,
         typename Allocator>
concurrent_unordered_set( InputIterator, InputIterator,
                         Hash, Allocator )
-> concurrent_unordered_set<iterator_value_t<InputIterator>,
    Hash,
    std::equal_to<iterator_value_t<InputIterator>>,
    Allocator>;

template <typename T,
         typename Hash = std::hash<Key>,
         typename KeyEqual = std::equal_to<Key>,
         typename Allocator = tbb_allocator<std::pair<Key, T>>>
concurrent_unordered_set( std::initializer_list<value_type>,
                         map_size_type = /*implementation-defined*/,
                         Hash = Hash(),
                         KeyEqual = KeyEqual(),
                         Allocator = Allocator() )
-> concurrent_unordered_set<T,
    Hash,
    KeyEqual,
    Allocator>;

template <typename T,
         typename Allocator>
concurrent_unordered_set( std::initializer_list<value_type>,
                         map_size_type, Allocator )
-> concurrent_unordered_set<T,
    std::hash<Key>,
    std::equal_to<Key>,
    Allocator>;

template <typename T,
         typename Hash,
         typename Allocator>
concurrent_unordered_set( std::initializer_list<value_type>,
                         map_size_type, Hash, Allocator )
-> concurrent_unordered_set<T,
    Hash,
    std::equal_to<Key>,
    Allocator>;

```

Where the type `map_size_type` refers to the `size_type` member type of the deduced `concurrent_unordered_set` and the type alias `iterator_value_t` defines as follows:

```

template <typename InputIterator>
using iterator_value_t = typename std::iterator_traits<InputIterator>::value_type;

```

### Example

```
#include <tbb/concurrent_unordered_set.h>
#include <vector>
#include <functional>

struct CustomHasher {...};

int main() {
    std::vector<int> v;

    // Deduces s1 as concurrent_unordered_set<int>
    tbb::concurrent_unordered_set s1(v.begin(), v.end());

    // Deduces s2 as concurrent_unordered_set<int, CustomHasher>;
    tbb::concurrent_unordered_set s2(v.begin(), v.end(), CustomHasher{});
}
```

### 11.2.5.3.5 concurrent\_unordered\_multiset

#### [containers.concurrent\_unordered\_multiset]

`tbb::concurrent_unordered_multiset` is a class template represents an unordered sequence of elements which supports concurrent insertion, lookup and traversal, but not concurrent erasure. The container allows to store multiple equivalent elements.

#### 11.2.5.3.5.1 Class Template Synopsis

```
// Defined in header <tbb/concurrent_unordered_set.h>

namespace tbb {
    template <typename T,
              typename Hash = std::hash<Key>,
              typename KeyEqual = std::equal_to<Key>,
              typename Allocator = tbb_allocator<std::pair<const Key, T>>>
    class concurrent_unordered_multiset {
    public:
        using key_type = Key;
        using value_type = Key;

        using size_type = <implementation-defined unsigned integer type>;
        using difference_type = <implementation-defined signed integer type>;

        using hasher = Hash;
        using key_equal = /*See below*/;

        using allocator_type = Allocator;

        using reference = value_type&;
        using const_reference = const value_type&;

        using pointer = typename std::allocator_traits<Allocator>::pointer;
        using const_pointer = typename std::allocator_traits<Allocator>::const_
            ↪pointer;

        using iterator = <implementation-defined ForwardIterator>;
    }
```

(continues on next page)

(continued from previous page)

```

using const_iterator = <implementation-defined constant ForwardIterator>;
using local_iterator = <implementation-defined ForwardIterator>;
using const_local_iterator = <implementation-defined constant ForwardIterator>;
using node_type = <implementation-defined node handle>;
using range_type = <implementation-defined ContainerRange>;
using const_range_type = <implementation-defined constant ContainerRange>;
// Construction, destruction, copying
concurrent_unordered_multiset();

explicit concurrent_unordered_multiset( size_type bucket_count, const hasher&
hash = hasher(),
const key_equal& equal = key_equal(),
const allocator_type& alloc =
allocator_type() );

concurrent_unordered_multiset( size_type bucket_count, const allocator_type&
alloc );

concurrent_unordered_multiset( size_type bucket_count, const hasher& hash,
const allocator_type& alloc );

explicit concurrent_unordered_multiset( const allocator_type& alloc );

template <typename InputIterator>
concurrent_unordered_multiset( InputIterator first, InputIterator last,
size_type bucket_count = /*implementation-
defined*/,
const hasher& hash = hasher(),
const key_equal& equal = key_equal(),
const allocator_type& alloc = allocator_type() );
template <typename Inputiterator>
concurrent_unordered_multiset( InputIterator first, InputIterator last,
size_type bucket_count, const allocator_type&
alloc );

template <typename InputIterator>
concurrent_unordered_multiset( InputIterator first, InputIterator last,
size_type bucket_count, const hasher& hash,
const allocator_type& alloc );

concurrent_unordered_multiset( std::initializer_list<value_type> init,
size_type bucket_count = /*implementation-
defined*/,
const hasher& hash = hasher(),
const key_equal& equal = key_equal(),
const allocator_type& alloc = allocator_type() );
concurrent_unordered_multiset( std::initializer_list<value_type> init,
size_type bucket_count, const allocator_type&
alloc );

```

(continues on next page)

(continued from previous page)

```

concurrent_unordered_multiset( std::initializer_list<value_type> init,
                               size_type bucket_count, const hasher& hash,
                               const allocator_type& alloc );

concurrent_unordered_multiset( const concurrent_unordered_multiset& other );
concurrent_unordered_multiset( const concurrent_unordered_multiset& other,
                               const allocator_type& alloc );

concurrent_unordered_multiset( concurrent_unordered_multiset&& other );
concurrent_unordered_multiset( concurrent_unordered_multiset&& other,
                               const allocator_type& alloc );

~concurrent_unordered_multiset();

concurrent_unordered_multiset& operator=( const concurrent_unordered_multiset&
→ other );
concurrent_unordered_multiset& operator=( concurrent_unordered_multiset&&_
→other ) noexcept(/*See details*/);

concurrent_unordered_multiset& operator=( std::initializer_list<value_type>_
→init );

allocator_type get_allocator() const;

// Iterators
iterator begin() noexcept;
const_iterator begin() const noexcept;
const_iterator cbegin() const noexcept;

iterator end() noexcept;
const_iterator end() const noexcept;
const_iterator cend() const noexcept;

// Size and capacity
bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// Concurrently safe modifiers
std::pair<iterator, boolconst value_type& value );
iterator insert( const_iterator hint, const value_type& value );

std::pair<iterator, booltemplate <typename InputIterator>
void insert( InputIterator first, InputIterator last );

void insert( std::initializer_list<value_type> init );

std::pair<iterator, booltemplate <typename... Args>
std::pair<iterator, bool

```

(continues on next page)

(continued from previous page)

```

template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_set<T, SrcHash, SrcKeyEqual, Allocator>&_
source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_set<T, SrcHash, SrcKeyEqual, Allocator>&&_
source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_multiset<T, SrcHash, SrcKeyEqual, Allocator>&_
source );

template <typename SrcHash, typename SrcKeyEqual>
void merge( concurrent_unordered_multiset<T, SrcHash, SrcKeyEqual, Allocator>&&_
source );

// Concurrently unsafe modifiers
void clear() noexcept;

iterator unsafe_erase( const_iterator pos );
iterator unsafe_erase( iterator pos );

iterator unsafe_erase( const_iterator first, const_iterator last );

size_type unsafe_erase( const key_type& key );

template <typename K>
size_type unsafe_erase( const K& key );

node_type unsafe_extract( const_iterator pos );
node_type unsafe_extract( iterator pos );

node_type unsafe_extract( const key_type& key );

template <typename K>
node_type unsafe_extract( const K& key );

void swap( concurrent_unordered_multiset& other );

// Lookup
size_type count( const key_type& key ) const;

template <typename K>
size_type count( const K& key ) const;

iterator find( const key_type& key );
const_iterator find( const key_type& key ) const;

template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;

```

(continues on next page)

(continued from previous page)

```

bool contains( const key_type& key ) const;

template <typename K>
bool contains( const K& key ) const;

std::pair<iterator, iterator> equal_range( const key_type& key );
std::pair<const_iterator, const_iterator> equal_range( const key_type& key ) const;

template <typename K>
std::pair<iterator, iterator> equal_range( const K& key );

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key ) const;

// Bucket interface
local_iterator unsafe_begin( size_type n );
const_local_iterator unsafe_begin( size_type n ) const;
const_local_iterator unsafe_cbegin( size_type n ) const;

local_iterator unsafe_end( size_type n );
const_local_iterator unsafe_end( size_type n ) const;
const_local_iterator unsafe_cend( size_type n ) const;

size_type unsafe_bucket_count() const;
size_type unsafe_max_bucket_bound() const;

size_type unsafe_bucket_size( size_type n ) const;

size_type unsafe_bucket( const key_type& key ) const;

// Hash policy
float load_factor() const;

float max_load_factor() const;
void max_load_factor( float ml );

void rehash( size_type count );

void reserve( size_type count );

// Observers
hasher hash_function() const;
key_equal key_eq() const;

// Parallel iteration
range_type range();
const_range_type range() const;
};

// class concurrent_unordered_multiset
} // namespace tbb

```

Requirements:

- The expression `std::allocator_type<Allocator>::destroy(m, val)` where `m` is an object of the type `Allocator` and `val` is an object of type `value_type` should be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Hash` shall meet the `Hash` requirements from [hash] ISO C++ Standard section.

- The type `KeyEqual` shall meet the `BinaryPredicate` requirements from [algorithms.general] ISO C++ Standard section.
- The type `Allocator` shall meet the `Allocator` requirements from [allocator.requirements] ISO C++ Standard section.

### 11.2.5.3.5.2 Description

`tbb::concurrent_unordered_multiset` is an unordered sequence, which elements are organized into buckets. The value of the hash function `Hash` for `Key` object determines the number of the bucket in which the corresponding element would be placed.

If the qualified-id `Hash::transparent_key_equal` is valid and denotes a type, member type `concurrent_unordered_multiset::key_equal` defines as the value of this qualified-id. In this case the program is ill-formed if any of the following conditions are met:

- The template parameter `KeyEqual` is different from `std::equal_to<Key>`.
- Qualified-id `Hash::transparent_key_equal::is_transparent` is not valid or not denotes a type.

Otherwise, member type `concurrent_unordered_multiset::key_equal` defines as the value of the template parameter `KeyEqual`.

### 11.2.5.3.5.3 Member functions

#### 11.2.5.3.5.4 Construction, destruction, copying

##### 11.2.5.3.5.5 Empty container constructors

```
concurrent_unordered_multiset();
explicit concurrent_unordered_multiset( const allocator_type& alloc );
```

Constructs an empty `concurrent_unordered_multiset`. The initial number of buckets is unspecified.

If provided uses the allocator `alloc` to allocate the memory.

```
explicit concurrent_unordered_multiset( size_type bucket_count,
                                       const hasher& hash = hasher(),
                                       const key_equal& equal = key_equal(),
                                       const allocator_type& alloc =
                                         allocator_type() );
concurrent_unordered_multiset( size_type bucket_count, const allocator_type&_
  alloc );
concurrent_unordered_multiset( size_type bucket_count, const hasher& hash,
                               const allocator_type& alloc );
```

Constructs an empty `concurrent_unordered_multiset` with `bucket_count` buckets.

If provided uses the hash function `hasher`, predicate `equal` to compare `key_type` objects for equality and the allocator `alloc` to allocate the memory.

### 11.2.5.3.5.6 Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_unordered_multiset( InputIterator first, InputIterator last,
                               size_type bucket_count = /*implementation-
                               defined*/,
                               const hasher& hash = hasher(),
                               const key_equal& equal = key_equal(),
                               const allocator_type& alloc = allocator_
                               type() );

template <typename Inputiterator>
concurrent_unordered_multiset( InputIterator first, InputIterator last,
                               size_type bucket_count, const allocator_type&_
                               alloc );

template <typename InputIterator>
concurrent_unordered_multiset( InputIterator first, InputIterator last,
                               size_type bucket_count, const hasher& hash,
                               const allocator_type& alloc );
```

Constructs the concurrent\_unordered\_multiset which contains the elements from the half-open interval [first, last)`.

If provided uses the hash function hasher, predicate equal to compare key\_type objects for equality and the allocator alloc to allocate the memory.

**Requirements:** the type InputIterator shall meet the requirements of InputIterator from [input.iterators] ISO C++ Standard section.

```
concurrent_unordered_multiset( std::initializer_list<value_type> init,
                               size_type bucket_count = /*implementation-
                               defined*/,
                               const hasher& hash = hasher(),
                               const key_equal& equal = key_equal(),
                               const allocator_type& alloc = allocator_
                               type() );
```

Equivalent to concurrent\_unordered\_multiset(init.begin(), init.end(), bucket\_count, hash, equal, alloc).

```
concurrent_unordered_multiset( std::initializer_list<value_type> init,
                               size_type bucket_count, const allocator_type&_
                               alloc );
```

Equivalent to concurrent\_unordered\_multiset(init.begin(), init.end(), bucket\_count, alloc).

```
concurrent_unordered_multiset( std::initializer_list<value_type> init,
                               size_type bucket_count, const hasher& hash,
                               const allocator_type& alloc );
```

Equivalent to concurrent\_unordered\_multiset(init.begin(), init.end(), bucket\_count, hash, alloc).

### 11.2.5.3.5.7 Copying constructors

```
concurrent_unordered_multiset( const concurrent_unordered_multiset& other );
concurrent_unordered_multiset( const concurrent_unordered_multiset& other,
const allocator_type& alloc );
```

Constructs a copy of other.

If the allocator argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with other.

### 11.2.5.3.5.8 Moving constructors

```
concurrent_unordered_multiset( concurrent_unordered_multiset&& other );
concurrent_unordered_multiset( concurrent_unordered_multiset&& other,
const allocator_type& alloc );
```

Constructs a `concurrent_unordered_multiset` with the contents of other using move semantics.

other is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with other.

### 11.2.5.3.5.9 Destructor

```
~concurrent_unordered_multiset();
```

Destroys the `concurrent_unordered_multiset`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

### 11.2.5.3.5.10 Assignment operators

```
concurrent_unordered_multiset& operator=( const concurrent_unordered_
multiset& other );
```

Replaces all elements in `*this` by the copies of the elements in other.

Copy assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment` is true.

The behavior is undefined in case of concurrent operations with `*this` and other.

**Returns:** a reference to `*this`.

```
concurrent_unordered_multiset& operator=( concurrent_unordered_multiset&&_  
other ) noexcept /*See below*/;
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment` is true.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

**Returns:** a reference to `*this`.

**Exceptions:** noexcept specification:

```
noexcept (std::allocator_traits<allocator_type>::is_always_  
equal::value &&  
    std::is_nothrow_moveAssignable<hasher>::value &&  
    std::is_nothrow_moveAssignable<key_equal>::value)
```

```
concurrent_unordered_multiset& operator=( std::initializer_list<value_type>_  
init );
```

Replaces all elements in `*this` by the elements in `init`.

The behavior is undefined in case of concurrent operations with `*this`.

**Returns:** a reference to `*this`.

### 11.2.5.3.5.11 Iterators

The types `concurrent_unordered_multiset::iterator` and `concurrent_unordered_multiset::const_iterator` meets the requirements of `ForwardIterator` from [forward.iterators] ISO C++ Standard section.

### 11.2.5.3.5.12 begin and cbegin

```
iterator begin();  
const_iterator begin() const;  
const_iterator cbegin() const;
```

**Returns:** an iterator to the first element in the container.

### 11.2.5.3.5.13 end and cend

```
iterator end();
const_iterator end() const;
const_iterator cend() const;
```

**Returns:** an iterator to the element which follows the last element in the container.

### 11.2.5.3.5.14 Size and capacity

#### 11.2.5.3.5.15 empty

```
bool empty() const;
```

**Returns:** true if the container is empty, false otherwise.

The result may differ with the actual container state in case of pending concurrent insertions.

#### 11.2.5.3.5.16 size

```
size_type size() const;
```

**Returns:** the number of elements in the container.

The result may differ with the actual container size in case of pending concurrent insertions.

#### 11.2.5.3.5.17 max\_size

```
size_type max_size() const;
```

**Returns:** the maximum number of elements that container can hold.

### 11.2.5.3.5.18 Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other, lookup methods and while traversing the container.

#### 11.2.5.3.5.19 Inserting values

```
std::pair<iterator, bool> insert( const value_type& value )
```

Inserts the value value into the container.

**Returns:** std::pair<iterator, bool> where iterator points to the inserted element. Boolean value is always true.

```
iterator insert( const_iterator hint, const value_type& other )
```

Inserts the value `value` into the container.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

**Returns:** an `iterator` to the inserted element.

```
std::pair<iterator, bool

```

Inserts the value `value` into the container using move semantics.

`value` is left in a valid, but unspecified state.

**Returns:** `std::pair<iterator, bool>` where `iterator` points to the inserted element.  
Boolean value is always `true`.

```
iterator insert( const_iterator hint, value_type&& other )
```

Inserts the value `value` into the container using move semantics.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

`value` is left in a valid, but unspecified state.

**Returns:** an `iterator` to the inserted element.

#### 11.2.5.3.5.20 Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last )
```

Inserts all items from the half-open interval `[first, last)` into the container.

**Requirements:** the type `InputIterator` must meet the requirements of `InputIterator` from [`input iterators`] ISO C++ Standard section.

```
void insert( std::initializer_list<value_type> init )
```

Equivalent to `insert(init.begin(), init.end())`.

#### 11.2.5.3.5.21 Inserting nodes

```
std::pair<iterator, bool

```

If the node handle `nh` is empty, does nothing.

Otherwise - inserts the node, owned by `nh` into the container.

`nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if nh is not empty and `get_allocator() != nh.get_allocator()`.

**Returns:** `std::pair<iterator, bool>` where iterator points to the inserted element. Boolean value is always true.

```
iterator insert( const_iterator hint, node_type&& nh )
```

If the node handle nh is empty, does nothing.

Otherwise - inserts the node, owned by nh into the container.

Optionally uses the parameter hint as a suggestion to where the node should be placed.

nh is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if nh is not empty and `get_allocator() != nh.get_allocator()`.

**Returns:** an iterator pointing to the inserted element.

#### 11.2.5.3.5.22 Emplacing elements

```
template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args )
```

Inserts an element ,constructed in-place from args into the container.

**Returns:** `std::pair<iterator, bool>` where iterator points to the inserted element. Boolean value is always true.

```
template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args )
```

Inserts an element ,constructed in-place from args into the container.

Optionally uses the parameter hint as a suggestion to where the node should be placed.

**Returns:** an iterator to the inserted element.

#### 11.2.5.3.5.23 Merging containers

```
template <typename SrcHash, SrcKeyEqual>
void merge( concurrent_unordered_set<T, SrcHash, SrcKeyEqual, Allocator>& source )

template <typename SrcHash, SrcKeyEqual>
void merge( concurrent_unordered_set<T, SrcHash, SrcKeyEqual, Allocator>&& source )

template <typename SrcHash, SrcKeyEqual>
void merge( concurrent_unordered_multiset<T, SrcHash, SrcKeyEqual, Allocator>& source )
```

(continues on next page)

(continued from previous page)

```
template <typename SrcHash, SrcKeyEqual>
void merge( concurrent_unordered_multiset<T, SrcHash, SrcKeyEqual, Allocator>
    ↪&& source )
```

Transfers all elements from `source` to `*this`.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `get_allocator() != source.get_allocator()`.

#### 11.2.5.3.5.24 Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

#### 11.2.5.3.5.25 Clearing

```
void clear();
```

Removes all elements from the container.

#### 11.2.5.3.5.26 Erasing elements

```
iterator unsafe_erase( const_iterator pos );
iterator unsafe_erase( iterator pos );
```

Removes the element pointed to by `pos` from the container.

Invalidates all iterators and references to the removed element.

**Returns:** iterator which follows the removed element.

**Requirements:** the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
size_type unsafe_erase( const key_type& key );
```

Removes the element equal to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

**Returns:** the number of removed elements.

```
template <typename K>
size_type unsafe_erase( const K& key );
```

Removes the element which compares equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

This overload only participates in overload resolution if all of the following conditions are met:

- The qualified-id `hasher::transparent_key_equal` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is false.
- `std::is_convertible<K, const_iterator>::value` is false.

**Returns:** the number of removed elements.

#### 11.2.5.3.5.27 Erasing sequences

```
iterator unsafe_erase( const_iterator first, const_iterator last );
```

Removes all elements from the half-open interval `[first, last)` from the container.

**Returns:** iterator which follows the last removed element.

**Requirements:** the range `[first, last)` must be a valid subrange in `*this`.

#### 11.2.5.3.5.28 Extracting nodes

```
node_type unsafe_extract( iterator pos );
node_type unsafe_extract( const_iterator pos );
```

Transfers ownership of the element pointed to by `pos` from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

**Returns:** the node handle that owns the extracted element.

**Requirements:** the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
node_type unsafe_extract( const key_type& key );
```

If an element equal to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

If there are multiple elements equal to `key` exists, it is unspecified which element should be transferred.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

**Returns:** the node handle that owns the extracted element or an empty node handle if an element equal to `key` was not found.

```
template <typename K>
node_type unsafe_extract( const K& key );
```

If an element which compares equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

If there are multiple elements which compares equivalent with `key` exists, it is unspecified which element should be transferred.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

This overload only participates in overload resolution if all of the following conditions are met:

- The qualified-id `hasher::transparent_key_equal` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

**Returns:** the node handle that owns the extracted element or an empty node handle if an element which compares equivalent to `key` was not found.

#### 11.2.5.3.5.29 swap

```
void swap( concurrent_unordered_multiset& other ) noexcept(/*See below*/);
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise if `get_allocator() != other.get_allocator()` the behavior is undefined.

**Exceptions:** `noexcept` specification:

```
noexcept(std::allocator_traits<allocator_type>::is_always_
    equal::value &&
    std::is_nothrow_swappable<hasher>::value &&
    std::is_nothrow_swappable<key_equal>::value
```

#### 11.2.5.3.5.30 Lookup

All methods in this section can be executed concurrently with each other, concurrently-safe modifiers and while traversing the container.

#### 11.2.5.3.5.31 count

```
size_type count( const key_type& key );
```

**Returns:** the number of elements equal to `key`.

```
template <typename K>
size_type count( const K& key );
```

**Returns:** the number of elements which compares equivalent with key.

This overload only participates in overload resolution if qualified-id hasher::transparent\_key\_equal is valid and denotes a type.

#### 11.2.5.3.5.32 find

```
iterator find( const key_type& key );
const_iterator find( const key_type& key ) const;
```

**Returns:** an iterator to the element equal to key or end() if no such element exists.

If there are multiple elements equal to key exists, it is unspecified which element should be found.

```
template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;
```

**Returns:** an iterator to the element which compares equivalent with key or end() if no such element exists.

If there are multiple elements which compares equivalent with key exists, it is unspecified which element should be found.

These overloads only participates in overload resolution if qualified-id hasher::transparent\_key\_equal is valid and denotes a type.

#### 11.2.5.3.5.33 contains

```
bool contains( const key_type& key ) const;
```

**Returns:** true if at least one element equal to key exists in the container, false otherwise.

```
template <typename K>
bool contains( const K& key ) const;
```

**Returns:** true if at least one element which compares equivalent with key exists in the container, false otherwise.

This overload only participates in overload resolution if qualified-id hasher::transparent\_key\_equal is valid and denotes a type.

### 11.2.5.3.5.34 equal\_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );
std::pair<const_iterator, const_iterator> equal_range( const key_type& key ) ↳ const;
```

**Returns:** if at least one element with the key equal to key exists - a pair of iterators {f, l}, where f is an iterator to the first element equal to key, l is an iterator to the element which follows the last element equal to key. Otherwise - {end(), end() }.

```
template <typename K>
std::pair<iterator, iterator> equal_range( const K& key )

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key )
```

**Returns:** if at least one element with the key which compares equivalent with key exists - a pair of iterators {f, l}, where f is an iterator to the first element which compares equivalent with key, l is an iterator to the element which follows the last element which compares equivalent with key. Otherwise - {end(), end() }.

These overloads only participates in overload resolution if qualified-id hasher::transparent\_key\_equal is valid and denotes a type.

### 11.2.5.3.5.35 Bucket interface

The types concurrent\_unordered\_multiset::local\_iterator and concurrent\_unordered\_multiset::const\_local\_iterator meets the requirements of ForwardIterator from [forward.iterators] ISO C++ Standard section.

These iterators are used to traverse the certain bucket.

All methods in this section can only be executed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

### 11.2.5.3.5.36 Bucket begin and bucket end

```
local_iterator unsafe_begin( size_type n );
const_local_iterator unsafe_begin( size_type n ) const;
const_local_iterator unsafe_cbegin( size_type n ) const;
```

**Returns:** an iterator to the first element in the bucket number n.

```
local_iterator unsafe_end( size_type n );
const_local_iterator unsafe_end( size_type n ) const;
const_local_iterator unsafe_cend( size_type n ) const;
```

**Returns:** an iterator to the element which follows the last element in the bucket number n.

### 11.2.5.3.5.37 The number of buckets

```
size_type unsafe_bucket_count() const;
```

**Returns:** the number of buckets in the container.

```
size_type unsafe_max_bucket_count() const;
```

**Returns:** the maximum number of buckets that container can hold.

### 11.2.5.3.5.38 Size of the bucket

```
size_type unsafe_bucket_size( size_type n ) const;
```

**Returns:** the number of elements in the bucket number n.

### 11.2.5.3.5.39 Bucket number

```
size_type unsafe_bucket( const key_type& key ) const;
```

**Returns:** the number of the bucket in which the element with the key key is stored.

### 11.2.5.3.5.40 Hash policy

Hash policy of concurrent\_unordered\_multiset manages the number of buckets in the container and the allowed maximum number of elements per bucket (load factor). If the maximum load factor is exceeded, the container can automatically increase the number of buckets.

### 11.2.5.3.5.41 Load factor

```
float load_factor() const;
```

**Returns:** the average number of elements per bucket, which is `size() / unsafe_bucket_count()`.

```
float max_load_factor() const;
```

**Returns:** the maximum number of elements per bucket.

```
void max_load_factor( float ml );
```

Sets the maximum number of elements per bucket to ml.

#### 11.2.5.3.5.42 Manual rehashing

```
void rehash( size_type n );
```

Sets the number of buckets to n and rehashes the container.

```
void reserve( size_type n );
```

Sets the number of buckets to the value which is needed to store n elements.

#### 11.2.5.3.5.43 Observers

##### 11.2.5.3.5.44 get\_allocator

```
allocator_type get_allocator() const;
```

**Returns:** a copy of the allocator associated with \*this.

##### 11.2.5.3.5.45 hash\_function

```
hasher hash_function() const;
```

**Returns:** a copy of the hash function associated with \*this.

##### 11.2.5.3.5.46 key\_eq

```
key_equal key_eq() const;
```

**Returns:** a copy of the key equality predicate associated with \*this.

##### 11.2.5.3.5.47 Parallel iteration

Member types concurrent\_unordered\_multiset::range\_type and concurrent\_unordered\_multiset::const\_range\_type meets the *ContainerRange requirements*.

These types differ only in that the bounds for a concurrent\_unordered\_multiset::const\_range\_type are of type concurrent\_unordered\_multiset::const\_iterator, whereas the bounds for a concurrent\_unordered\_multiset::range\_type are of type concurrent\_unordered\_multiset::iterator.

#### 11.2.5.3.5.48 range member function

```
range_type range();
const_range_type range() const;
```

**Returns:** a range object representing all elements in the container.

#### 11.2.5.3.5.49 Non-member functions

These functions provides binary comparison and swap operations on `tbb::concurrent_unordered_multiset` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `tbb::concurrent_unordered_multiset` as a type alias for which the non-member functions are reachable only via argument dependent lookup.

```
template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
void swap( concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& lhs,
           concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& rhs );

template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& lhs,
                   const concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& rhs );

template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& lhs,
                   const concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& rhs );
```

#### 11.2.5.3.5.50 Non-member swap

```
template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
void swap( concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& lhs,
           concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& rhs ) noexcept(noexcept(lhs.swap(rhs)));
```

Equivalent to `lhs.swap(rhs)`.

### 11.2.5.3.5.51 Non-member binary comparisons

Two objects of concurrent\_unordered\_multiset are equal if the following conditions are true:

- They contains an equal number of elements.
- Each group of elements with the same key in one container has the corresponding group of equivalent elements in the other container (not necessary in the same order).

```
template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator==( const concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& lhs,
                   const concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& rhs );
```

**Returns:** true if lhs is equal to rhs, false otherwise.

```
template <typename T, typename Hash,
          typename KeyEqual, typename Allocator>
bool operator!=( const concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& lhs,
                   const concurrent_unordered_multiset<T, Hash, KeyEqual, Allocator>& rhs );
```

Equivalent to !(lhs == rhs).

**Returns:** true if lhs is not equal to rhs, false otherwise.

### 11.2.5.3.5.52 Other

#### 11.2.5.3.5.53 Deduction guides

Where possible, constructors of concurrent\_unordered\_multiset supports class template argument deduction (since C++17):

```
template <typename InputIterator,
          typename Hash = std::hash<iterator_value_t<InputIterator>>,
          typename KeyEqual = std::equal_to<iterator_value_t<InputIterator>>,
          typename Allocator = tbb_allocator<iterator_value_t<InputIterator>>>
concurrent_unordered_multiset( InputIterator, InputIterator,
                               map_size_type = /*implementation_defined*/,
                               Hash = Hash(), KeyEqual = KeyEqual(),
                               Allocator = Allocator() )
-> concurrent_unordered_multiset<iterator_value_t<InputIterator>,
                                         Hash, KeyEqual, Allocator>;
```

```
template <typename InputIterator,
          typename Allocator>
concurrent_unordered_multiset( InputIterator, InputIterator,
                               map_size_type,
                               Allocator )
-> concurrent_unordered_multiset<iterator_value_t<InputIterator>,
                                         std::hash<iterator_value_t<InputIterator>>,
                                         std::equal_to<iterator_value_t<InputIterator>>,
```

(continues on next page)

(continued from previous page)

```

Allocator>;

template <typename InputIterator,
          typename Allocator>
concurrent_unordered_multiset( InputIterator, InputIterator, Allocator )
-> concurrent_unordered_multiset<iterator_value_t<InputIterator>,
                                std::hash<iterator_value_t<InputIterator>>,
                                std::equal_to<iterator_key_t<InputIterator>>,
                                Allocator>;

template <typename InputIterator,
          typename Hash,
          typename Allocator>
concurrent_unordered_multiset( InputIterator, InputIterator,
                               Hash, Allocator )
-> concurrent_unordered_multiset<iterator_value_t<InputIterator>,
                                Hash,
                                std::equal_to<iterator_value_t<InputIterator>>,
                                Allocator>;

template <typename T,
          typename Hash = std::hash<Key>,
          typename KeyEqual = std::equal_to<Key>,
          typename Allocator = tbb_allocator<std::pair<Key, T>>>
concurrent_unordered_multiset( std::initializer_list<value_type>,
                               map_size_type = /*implementation-defined*/,
                               Hash = Hash(),
                               KeyEqual = KeyEqual(),
                               Allocator = Allocator() )
-> concurrent_unordered_multiset<T,
                                Hash,
                                KeyEqual,
                                Allocator>;

template <typename T,
          typename Allocator>
concurrent_unordered_multiset( std::initializer_list<value_type>,
                               map_size_type, Allocator )
-> concurrent_unordered_multiset<T,
                                std::hash<Key>,
                                std::equal_to<Key>,
                                Allocator>;

template <typename T,
          typename Hash,
          typename Allocator>
concurrent_unordered_multiset( std::initializer_list<value_type>,
                               map_size_type, Hash, Allocator )
-> concurrent_unordered_multiset<T,
                                Hash,
                                std::equal_to<Key>,
                                Allocator>;

```

where the type `map_size_type` refers to the `size_type` member type of the deduced `concurrent_unordered_multiset`. and the type alias `iterator_value_t` defines as follows:

```
template <typename InputIterator>
using iterator_value_t = typename std::iterator_traits<InputIterator>::value_type;
```

### Example

```
#include <tbb/concurrent_unordered_set.h>
#include <vector>
#include <functional>

struct CustomHasher {...};

int main() {
    std::vector<int> v;

    // Deduces s1 as concurrent_unordered_multiset<int>
    tbb::concurrent_unordered_multiset s1(v.begin(), v.end());

    // Deduces s2 as concurrent_unordered_multiset<int, CustomHasher>;
    tbb::concurrent_unordered_multiset s2(v.begin(), v.end(), CustomHasher{});
}
```

## 11.2.5.4 Ordered associative containers

### 11.2.5.4.1 concurrent\_map

#### [containers.concurrent\_map]

tbb::concurrent\_map is a class template represents a sorted associative container which stores unique elements and supports concurrent insertion, lookup and traversal, but not concurrent erasure.

#### 11.2.5.4.1.1 Class Template Synopsis

```
namespace tbb {

    template <typename Key,
              typename T,
              typename Compare = std::less<Key>,
              typename Allocator = tbb_allocator<std::pair<const Key, T>>
    class concurrent_map {
    public:
        using key_type = Key;
        using mapped_type = T;
        using value_type = std::pair<const Key, T>;

        using size_type = <implementation-defined unsigned integer type>;
        using difference_type = <implementation-defined signed integer type>;

        using key_compare = Compare;
        using allocator_type = Allocator;

        using reference = value_type&;
        using const_reference = const value_type&;
        using pointer = std::allocator_traits<Allocator>::pointer;
        using const_pointer = std::allocator_traits<Allocator>::const_pointer;
    }
```

(continues on next page)

(continued from previous page)

```

using iterator = <implementation-defined ForwardIterator>;
using const_iterator = <implementation-defined constant ForwardIterator>;

using node_type = <implementation-defined node handle>

using range_type = <implementation-defined range>;
using const_range_type = <implementation-defined constant node handle>

class value_compare;

// Construction, destruction, copying
concurrent_map();
explicit concurrent_map( const key_compare& comp,
                           const allocator_type& alloc = allocator_type() );
explicit concurrent_map( const allocator_type& alloc );

template <typename InputIterator>
concurrent_map( InputIterator first, InputIterator last,
                const key_compare& comp = key_compare(),
                const allocator_type& alloc = allocator_type() );

template <typename InputIterator>
concurrent_map( InputIterator first, InputIterator last,
                const allocator_type& alloc );

concurrent_map( std::initializer_list<value_type> init,
                const key_compare& comp = key_compare(),
                const allocator_type& alloc = allocator_type() );

concurrent_map( std::initializer_list<value_type> init, const allocator_type& alloc );
~alloc );

concurrent_map( const concurrent_map& other );
concurrent_map( const concurrent_map& other,
                const allocator_type& alloc );

concurrent_map( concurrent_map&& other );
concurrent_map( concurrent_map&& other,
                const allocator_type& alloc );

~concurrent_map();

concurrent_map& operator=( const concurrent_map& other );
concurrent_map& operator=( concurrent_map&& other );
concurrent_map& operator=( std::initializer_list<value_type> init );

allocator_type get_allocator() const;

// Element access
value_type& at( const key_type& key );
const value_type& at( const key_type& key ) const;

value_type& operator[]( const key_type& key );
value_type& operator[]( key_type&& key );

```

(continues on next page)

(continued from previous page)

```

// Iterators
iterator begin();
const_iterator begin() const;
const_iterator cbegin() const;

iterator end();
const_iterator end() const;
const_iterator cend() const;

// Size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// Concurrently safe modifiers
std::pair<iterator, bool> insert( const value_type& value );

iterator insert( const_iterator hint, const value_type& value );

template <typename P>
std::pair<iterator, bool> insert( P&& value );

template <typename P>
iterator insert( const_iterator hint, P&& value );

std::pair<iterator, bool> insert( value_type&& value );

iterator insert( const_iterator hint, value_type&& value );

template <typename InputIterator>
void insert( InputIterator first, InputIterator last );

void insert( std::initializer_list<value_type> init );

std::pair<iterator, bool> insert( node_type&& nh );
iterator insert( const_iterator hint, node_type&& nh );

template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args );

template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );

template <typename SrcCompare>
void merge( concurrent_map<Key, T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_map<Key, T, SrcCompare, Allocator>&& source );

template <typename SrcCompare>
void merge( concurrent_multimap<Key, T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_multimap<Key, T, SrcCompare, Allocator>&& source );

// Concurrently unsafe modifiers
void clear();

```

(continues on next page)

(continued from previous page)

```

iterator unsafe_erase( const_iterator pos );
iterator unsafe_erase( iterator pos );

iterator unsafe_erase( const_iterator first, const_iterator last );

size_type unsafe_erase( const key_type& key );

template <typename K>
size_type unsafe_erase( const K& key );

node_type unsafe_extract( const_iterator pos );
node_type unsafe_extract( iterator pos );

node_type unsafe_extract( const key_type& key );

template <typename K>
node_type unsafe_extract( const K& key );

void swap( concurrent_map& other );

// Lookup
size_type count( const key_type& key );

template <typename K>
size_type count( const K& key );

iterator find( const key_type& key );
const_iterator find( const key_type& key ) const;

template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;

bool contains( const key_type& key ) const;

template <typename K>
bool contains( const K& key ) const;

std::pair<iterator, iterator> equal_range( const key_type& key );
std::pair<const_iterator, const_iterator> equal_range( const key_type& key ) const,  

const;

template <typename K>
std::pair<iterator, iterator> equal_range( const K& key );
std::pair<const_iterator, const_iterator> equal_range( const K& key ) const,  

const;

iterator lower_bound( const key_type& key );
const_iterator lower_bound( const key_type& key ) const;

template <typename K>
iterator lower_bound( const K& key );

template <typename K>
const_iterator lower_bound( const K& key ) const;

```

(continues on next page)

(continued from previous page)

```

iterator upper_bound( const key_type& key );
const_iterator upper_bound( const key_type& key ) const;

template <typename K>
iterator upper_bound( const K& key );

template <typename K>
const_iterator upper_bound( const K& key ) const;

// Observers
key_compare key_comp() const;

value_compare value_comp() const;

// Parallel iteration
range_type range();
const_range_type range() const;
};

// class concurrent_map

} // namespace tbb
}

```

Requirements:

- The expression `std::allocator_traits<Allocator>::destroy(m, val)` where `m` is an object of the type `Allocator` and `val` is an object of the type `value_type` should be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Compare` shall meet the `Compare` requirements from [alg.sorting] ISO C++ Standard section.
- The type `Allocator` shall meet the `Allocator` requirements from [allocator.requirements] ISO C++ Standard section.

#### 11.2.5.4.1.2 Member classes

##### 11.2.5.4.1.3 value\_compare

`concurrent_map::value_compare` is a function object which is used to compare `concurrent_map::value_type` objects by comparing their first components.

##### 11.2.5.4.1.4 Class Synopsis

```

namespace tbb {

template <typename Key, typename T,
          typename Compare, typename Allocator>
class concurrent_map<Key, T, Compare, Allocator>::value_compare {
protected:
    key_compare comp;

    value_compare( key_compare c );

public:
}

```

(continues on next page)

(continued from previous page)

```
    bool operator() ( const value_type& lhs, const value_type& rhs ) const;
} // class value_compare
} // namespace tbb
```

#### 11.2.5.4.1.5 Member objects

```
key_compare comp;
```

The key comparison function object.

#### 11.2.5.4.1.6 Member functions

```
value_compare( key_compare c );
```

Constructs a `value_compare` with the stored key comparison function object `c`.

```
    bool operator() ( const value_type& lhs, const value_type& rhs ) const;
```

Compares `lhs.first` and `rhs.first` by calling the stored key comparison function `comp`.

**Returns:** true if first components of `lhs` and `rhs` are equal, false otherwise.

#### 11.2.5.4.1.7 Member functions

##### 11.2.5.4.1.8 Construction, destruction, copying

##### 11.2.5.4.1.9 Empty container constructors

```
concurrent_map();

explicit concurrent_map( const key_compare& comp,
                           const allocator_type& alloc = allocator_type() );
explicit concurrent_map( const allocator_type& alloc );
```

Constructs empty `concurrent_map`.

If provided uses the comparison function object `comp` for all `key_type` comparisons and the allocator `alloc` to allocate the memory.

#### 11.2.5.4.1.10 Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_map( InputIterator first, InputIterator last,
                const key_compare& comp = key_compare(),
                const allocator_type& alloc = allocator_type() );

template <typename InputIterator>
concurrent_map( InputIterator first, InputIterator last,
                const allocator_type& alloc = allocator_type() );
```

Constructs the concurrent\_map which contains the elements from the half-open interval [first, last).

If the range [first, last) contains multiple elements with equal keys, it is unspecified which element would be inserted.

If provided uses the comparison function object comp for all key\_type comparisons and the allocator alloc to allocate the memory.

**Requirements:** the type InputIterator shall meet the requirements of *InputIterator* from [input iterators] ISO C++ Standard section.

```
concurrent_map( std::initializer_list<value_type> init, const key_compare& comp =
                comp = key_compare(),
                const allocator_type& alloc = allocator_type() );
```

Equivalent to concurrent\_map(init.begin(), init.end(), comp, alloc).

```
concurrent_map( std::initializer_list<value_type> init,
                const allocator_type& alloc );
```

Equivalent to concurrent\_map(init.begin(), init.end(), alloc).

#### 11.2.5.4.1.11 Copying constructors

```
concurrent_map( const concurrent_map& other );
concurrent_map( const concurrent_map& other, const allocator_type& alloc );
```

Constructs a copy of other.

If the allocator argument is not provided, it is obtained by calling std::allocator\_traits<allocator\_type>::select\_on\_container\_copy\_construction(other.get\_allocator()).

The behavior is undefined in case of concurrent operations with other.

#### 11.2.5.4.1.12 Moving constructors

```
concurrent_map( concurrent_map&& other );
concurrent_map( concurrent_map&& other, const allocator_type& alloc );
```

Constructs a `concurrent_map` with the contents of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

#### 11.2.5.4.1.13 Destructor

```
~concurrent_map();
```

Destroys the `concurrent_map`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

#### 11.2.5.4.1.14 Assignment operators

```
concurrent_map& operator=( const concurrent_map& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment` is true.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

**Returns:** a reference to `*this`.

```
concurrent_map& operator=( concurrent_map&& other );
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment` is true.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

**Returns:** a reference to `*this`.

```
concurrent_map& operator=( std::initializer_list<value_type> init );
```

Replaces all elements in `*this` by the elements in `init`.

If `init` contains multiple elements with equal keys, it is unspecified which element would be inserted.

The behavior is undefined in case of concurrent operations with `*this`.

**Returns:** a reference to `*this`.

#### 11.2.5.4.1.15 Element access

##### 11.2.5.4.1.16 `at`

```
value_type& at( const key_type& key );
const value_type& at( const key_type& key ) const;
```

**Returns:** a reference to `item.second` where `item` is the element with the key equal to `key`.

**Throws:** `std::out_of_range` exception if the element with the key equal to `key` is not presented in the container.

##### 11.2.5.4.1.17 `operator[]`

```
value_type& operator[]( const key_type& key );
```

If the element with the key equal to `key` is not presented in the container, inserts a new element, constructed in-place from `std::piecewise_construct`, `std::forward_as_tuple(key)`, `std::tuple<>()`.

**Requirements:** the type `value_type` shall meet the `EmplaceConstructible` requirements from [container.requirements] ISO C++ section.

**Returns:** a reference to `item.second` where `item` is the element with the key equal to `key`.

```
value_type& operator[]( key_type&& key );
```

If the element with the key equal to `key` is not presented in the container, inserts a new element, constructed in-place from `std::piecewise_construct`, `std::forward_as_tuple(std::move(key))`, `std::tuple<>()`.

**Requirements:** the type `value_type` shall meet the `EmplaceConstructible` requirements from [container.requirements] ISO C++ section.

**Returns:** a reference to `item.second` where `item` is the element with the key equal to `key`.

#### 11.2.5.4.1.18 Iterators

The types `concurrent_map::iterator` and `concurrent_map::const_iterator` meets the requirements of `ForwardIterator` from [forward.iterators] ISO C++ standard section.

#### 11.2.5.4.1.19 begin and cbegin

```
iterator begin();
const_iterator begin() const;
const_iterator cbegin() const;
```

**Returns:** an iterator to the first element in the container.

#### 11.2.5.4.1.20 end and cend

```
iterator end();
const_iterator end() const;
const_iterator cend() const;
```

**Returns:** an iterator to the element which follows the last element in the container.

#### 11.2.5.4.1.21 Size and capacity

#### 11.2.5.4.1.22 empty

```
bool empty() const;
```

**Returns:** true if the container is empty, false otherwise.

The result may differ with the actual container state in case of pending concurrent insertions.

#### 11.2.5.4.1.23 size

```
size_type size() const;
```

**Returns:** the number of elements in the container.

The result may differ with the actual container size in case of pending concurrent insertions.

#### 11.2.5.4.1.24 max\_size

```
size_type max_size() const;
```

**Returns:** the maximum number of elements that container can hold.

#### 11.2.5.4.1.25 Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other, lookup methods and while traversing the container.

#### 11.2.5.4.1.26 Inserting values

```
std::pair<iterator, bool> insert( const value_type& value );
```

Attempts to insert the value `value` into the container.

**Returns:** `std::pair<iterator, bool>` where `iterator` points to the inserted element or to an existing element with equal key. Boolean value is `true` if insertion took place, `false` otherwise.

**Requirements:** the type `value_type` shall meet the `CopyInsertable` requirements from [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, const value_type& other );
```

Attempts to insert the value `value` into the container.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

**Returns:** an `iterator` to the inserted element or to an existing element with equal key.

**Requirements:** the type `value_type` shall meet the `CopyInsertable` requirements from [container.requirements] ISO C++ Standard section.

```
template <typename P>
std::pair<iterator, bool> insert( P&& value );
```

Equivalent to `emplace(std::forward<P>(value))`.

This overload only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is `true`.

```
template <typename P>
iterator insert( const_iterator hint, P&& value );
```

Equivalent to `emplace_hint(hint, std::forward<P>(value))`.

This overload only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is `true`.

```
std::pair<iterator, bool> insert( value_type&& value );
```

Attempts to insert the value `value` into the container using move semantics.

`value` is left in a valid, but unspecified state.

**Returns:** `std::pair<iterator, bool` where `iterator` points to the inserted element or to an existing element with equal key. Boolean value is `true` if insertion took place, `false` otherwise.

**Requirements:** the type `value_type` shall meet the `MoveInsertable` requirements from [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, value_type&& other );
```

Attempts to insert the value `value` into the container using move semantics.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

`value` is left in a valid, but unspecified state.

**Returns:** an `iterator` to the inserted element or to an existing element with equal key.

**Requirements:** the type `value_type` shall meet the `MoveInsertable` requirements from [container.requirements] ISO C++ Standard section.

#### 11.2.5.4.1.27 Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last );
```

Attempts to insert all items from the half-open interval `[first, last)` into the container.

If the interval `[first, last)` contains multiple elements with equal keys, it is unspecified which element should be inserted.

**Requirements:** the type `InputIterator` must meet the requirements of `InputIterator` from [input.iterators] ISO C++ Standard section.

```
void insert( std::initializer_list<value_type> init );
```

Equivalent to `insert(init.begin(), init.end())`.

#### 11.2.5.4.1.28 Inserting nodes

```
std::pair<iterator, bool> insert( node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise - attempts to insert the node, owned by `nh` into the container.

If the insertion fails, node handle `nh` remains ownership of the node.

Otherwise - `nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if nh is not empty and `get_allocator() != nh.get_allocator()`.

**Returns:** `std::pair<iterator, bool>` where iterator points to the inserted element or to an existing element with key equal to `nh.key()`. Boolean value is true if insertion took place, false otherwise.

```
iterator insert( const_iterator hint, node_type&& nh );
```

If the node handle nh is empty, does nothing.

Otherwise - attempts to insert the node, owned by nh into the container.

Optionally uses the parameter hint as a suggestion to where the node should be placed.

If the insertion fails, node handle nh remains ownership of the node.

Otherwise - nh is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if nh is not empty and `get_allocator() != nh.get_allocator()`.

**Returns:** an iterator pointing to the inserted element or to an existing element with key equal to `nh.key()`.

#### 11.2.5.4.1.29 Emplacing elements

```
template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args );
```

Attempts to insert an element ,constructed in-place from args into the container.

**Returns:** `std::pair<iterator, bool>` where iterator points to the inserted element or to an existing element with equal key. Boolean value is true if insertion took place, false otherwise.

**Requirements:** the type `value_type` shall meet the `EmplaceConstructible` requirements from [container.requirements] ISO C++ section.

```
template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );
```

Attempts to insert an element ,constructed in-place from args into the container.

Optionally uses the parameter hint as a suggestion to where the node should be placed.

**Returns:** an iterator to the inserted element or to an existing element with equal key.

**Requirements:** the type `value_type` shall meet the `EmplaceConstructible` requirements from [container.requirements] ISO C++ section.

#### Merging containers

```
template <typename SrcCompare>
void merge( concurrent_map<Key, T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_map<Key, T, SrcCompare, Allocator>&& source );
```

(continues on next page)

(continued from previous page)

```
template <typename SrcCompare>
void merge( concurrent_multimap<Key, T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_multimap<Key, T, SrcCompare, Allocator>&& source );
```

Transfers those elements from `source` which keys do not exist in the container.

In case of merging with the container with multiple elements with equal keys, it is unspecified which element would be transferred.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `get_allocator() != source.get_allocator()`.

#### 11.2.5.4.1.30 Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

#### 11.2.5.4.1.31 Clearing

```
void clear();
```

Removes all elements from the container.

#### 11.2.5.4.1.32 Erasing elements

```
iterator unsafe_erase( const_iterator pos );
iterator unsafe_erase( iterator pos );
```

Removes the element pointed to by `pos` from the container.

Invalidates all iterators and references to the removed element.

**Returns:** iterator which follows the removed element.

**Requirements:** the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
size_type unsafe_erase( const key_type& key );
```

Removes the element with the key equal to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

**Returns:** 1 if an element with the key equal to `key` exists, 0 otherwise.

```
template <typename K>
size_type unsafe_erase( const K& key );
```

Removes the element with the key which compares equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `key_compare::is_transparent` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

**Returns:** 1 if an element with the key which compares equivalent to `key` exists, 0 otherwise.

#### 11.2.5.4.1.33 Erasing sequences

```
iterator unsafe_erase( const_iterator first, const_iterator last );
```

Removes all elements from the half-open interval `[first, last)` from the container.

**Returns:** iterator which follows the last removed element.

**Requirements:** the range `[first, last)` must be a valid subrange in `*this`.

#### 11.2.5.4.1.34 Extracting nodes

```
node_type unsafe_extract( iterator pos );
node_type unsafe_extract( const_iterator pos );
```

Transfers ownership of the element pointed to by `pos` from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

**Returns:** the node handle that owns the extracted element.

**Requirements:** the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
node_type unsafe_extract( const key_type& key );
```

If an element with the key equal to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

**Returns:** the node handle that owns the extracted element or an empty node handle if an element with the key equal to `key` was not found.

```
template <typename K>
node_type unsafe_extract( const K& key );
```

If an element with the key which compares equivalent to key exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of value\_type are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

This overload only participates in overload resolution if all of the following statements are true:

- The qualified-id `key_compare::is_transparent` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

**Returns:** the node handle that owns the extracted element or an empty node handle if an element with the key which compares equivalent to key was not found.

#### 11.2.5.4.1.35 swap

```
void swap( concurrent_map& other );
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise if `get_allocator() != other.get_allocator()` the behavior is undefined.

#### 11.2.5.4.1.36 Lookup

All methods in this section can be executed concurrently with each other, concurrently-safe modifiers and while traversing the container.

#### 11.2.5.4.1.37 count

```
size_type count( const key_type& key );
```

**Returns:** the number of elements with the key equal to key.

```
template <typename K>
size_type count( const K& key );
```

**Returns:** the number of elements with the key which compares equivalent with key.

This overload only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

#### 11.2.5.4.1.38 find

```
iterator find( const key_type& key );
const_iterator find( const key_type& key ) const;
```

**Returns:** an iterator to the element with the key equal to key or end() if no such element exists.

```
template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;
```

**Returns:** an iterator to the element with the key which compares equivalent with key or end() if no such element exists.

These overloads only participates in overload resolution if qualified-id key\_compare::is\_transparent is valid and denotes a type.

#### 11.2.5.4.1.39 contains

```
bool contains( const key_type& key ) const;
```

**Returns:** true if an element with the key equal to key exists in the container, false otherwise.

```
template <typename K>
bool contains( const K& key ) const;
```

**Returns:** true if an element with the key which compares equivalent with key exists in the container, false otherwise.

This overload only participates in overload resolution if qualified-id key\_compare::is\_transparent is valid and denotes a type.

#### 11.2.5.4.1.40 lower\_bound

```
iterator lower_bound( const key_type& key );
const_iterator lower_bound( const key_type& key ) const;
```

**Returns:** an iterator to the first element in the container with the key which is *not less* than key.

```
template <typename K>
iterator lower_bound( const K& key )

template <typename K>
const_iterator lower_bound( const K& key ) const
```

**Returns:** an iterator to the first element in the container with the key which compares *not less* with key.  
 These overloads only participates in overload resolution if qualified-id key\_compare::is\_transparent is valid and denotes a type.

#### 11.2.5.4.1.41 upper\_bound

```
iterator upper_bound( const key_type& key );
const_iterator upper_bound( const key_type& key ) const;
```

**Returns:** an iterator to the first element in the container with the key which is *greater* than key.

```
template <typename K>
iterator upper_bound( const K& key );

template <typename K>
const_iterator upper_bound( const K& key ) const;
```

**Returns:** an iterator to the first element in the container with the key which compares greater with key.

These overloads only participates in overload resolution if qualified-id key\_compare::is\_transparent is valid and denotes a type.

#### 11.2.5.4.1.42 equal\_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );
std::pair<const_iterator, const_iterator> equal_range( const key_type& key ) const;
```

**Returns:** if an element with the key equal to key exists - a pair of iterators {f, l}, where f is an iterator to this element, l is std::next(f). Otherwise - {end(), end() }.

```
template <typename K>
std::pair<iterator, iterator> equal_range( const K& key )

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key )
```

**Returns:** if an element with the key which compares equivalent with key exists - a pair of iterators {f, l}, where f is an iterator to this element, l is std::next(f). Otherwise - {end(), end() }.

These overloads only participates in overload resolution if qualified-id key\_compare::is\_transparent is valid and denotes a type.

#### 11.2.5.4.1.43 Observers

##### 11.2.5.4.1.44 get\_allocator

```
allocator_type get_allocator() const;
```

**Returns:** a copy of the allocator associated with `*this`.

##### 11.2.5.4.1.45 key\_comp

```
key_compare key_comp() const;
```

**Returns:** a copy of the key comparison functor associated with `*this`.

##### 11.2.5.4.1.46 value\_comp

```
value_compare value_comp() const;
```

**Returns:** an object of the `value_compare` class which is used to compare `value_type` objects.

#### 11.2.5.4.1.47 Parallel iteration

Member types `concurrent_map::range_type` and `concurrent_map::const_range_type` meets the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_map::const_range_type` are of type `concurrent_map::const_iterator`, whereas the bounds for a `concurrent_map::range_type` are of type `concurrent_map::iterator`.

#### 11.2.5.4.1.48 range member function

```
range_type range();
const_range_type range() const;
```

**Returns:** a range object representing all elements in the container.

#### 11.2.5.4.1.49 Non-member functions

These functions provides binary and lexicographical comparison and swap operations on `tbb::concurrent_map` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `tbb::concurrent_map` as a type alias for which the non-member functions are reachable only via argument dependent lookup.

```

template <typename Key, typename T, typename Compare, typename Allocator>
void swap( concurrent_map<Key, T, Compare, Allocator>& lhs,
           concurrent_map<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                   const concurrent_map<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                   const concurrent_map<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator<( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                   const concurrent_map<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                   const concurrent_map<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator<=( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                   const concurrent_map<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>=( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                   const concurrent_map<Key, T, Compare, Allocator>& rhs );

```

#### 11.2.5.4.1.50 Non-member swap

```

template <typename Key, typename T, typename Compare, typename Allocator>
void swap( concurrent_map<Key, T, Compare, Allocator>& lhs,
           concurrent_map<Key, T, Compare, Allocator>& rhs );

```

Equivalent to `lhs.swap(rhs)`.

#### 11.2.5.4.1.51 Non-member binary comparisons

Two `tbb::concurrent_map` objects are equal if they have the same number of elements and each element in one container is equal to the element in other container on the same position.

```

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                   const concurrent_map<Key, T, Compare, Allocator>& rhs )

```

**Returns:** true if `lhs` is equal to `rhs`, false otherwise.

```

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                   const concurrent_map<Key, T, Compare, Allocator>& rhs )

```

**Returns:** true if `lhs` is not equal to `rhs`, false otherwise.

#### 11.2.5.4.1.52 Non-member lexicographical comparisons

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator<( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                   const concurrent_map<Key, T, Compare, Allocator>& rhs )
```

**Returns:** true if lhs is lexicographically *less* than rhs.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator<=( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                   const concurrent_map<Key, T, Compare, Allocator>& rhs )
```

**Returns:** true if lhs is lexicographically *less or equal* than rhs.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                   const concurrent_map<Key, T, Compare, Allocator>& rhs )
```

**Returns:** true if lhs is lexicographically *greater* than rhs.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>=( const concurrent_map<Key, T, Compare, Allocator>& lhs,
                   const concurrent_map<Key, T, Compare, Allocator>& rhs )
```

**Returns:** true if lhs is lexicographically *greater or equal* than rhs.

#### 11.2.5.4.1.53 Other

##### 11.2.5.4.1.54 Deduction guides

Where possible, constructors of `concurrent_map` supports class template argument deduction (since C++17):

```
template <typename InputIterator,
          typename Compare = std::less<iterator_key_t<InputIterator>>,
          typename Allocator = tbb_allocator<iterator_alloc_value_t<InputIterator>>>
concurrent_map( InputIterator, InputIterator, Compare = Compare(), Allocator = _  
_Allocator() )
-> concurrent_map<iterator_key_t<InputIterator>,
                  iterator_mapped_t<InputIterator>,
                  Compare,
                  Allocator>;
```

```
template <typename InputIterator,
          typename Allocator>
concurrent_map( InputIterator, InputIterator, Allocator )
-> concurrent_map<iterator_key_t<InputIterator>,
                  iterator_mapped_t<InputIterator>,
                  std::less<iterator_key_t<InputIterator>>,
                  Allocator>;
```

(continues on next page)

(continued from previous page)

```

template <typename Key,
          typename T,
          typename Compare = std::less<Key>,
          typename Allocator = tbb_allocator<std::pair<const Key, T>>>
concurrent_map( std::initializer_list<std::pair<Key, T>>, Compare = Compare(), _
    ↵Allocator = Allocator() )
-> concurrent_map<Key, T, Compare, Allocator>;

template <typename Key,
          typename T,
          typename Allocator>
concurrent_map( std::initializer_list<std::pair<Key, T>>, Allocator )
-> concurrent_map<Key, T, std::less<Key>, Allocator>;

```

where the type aliases `iterator_key_t`, `iterator_mapped_t`, `iterator_alloc_value_t` defines as follows:

```

template <typename InputIterator>
using iterator_key_t = std::remove_const_t<typename std::iterator_traits<
    ↵<InputIterator>::value_type::first_type>;

template <typename InputIterator>
using iterator_mapped_t = typename std::iterator_traits<InputIterator>::value_
    ↵type::second_type;

template <typename InputIterator>
using iterator_alloc_value_t = std::pair<std::add_const_t<iterator_key_t
    ↵<InputIterator>>,
                                         iterator_mapped_t<InputIterator>>;

```

## Example

```

#include <tbb/concurrent_map.h>
#include <vector>

int main() {
    std::vector<std::pair<int, float>> v;

    // Deduces cm1 as concurrent_map<int, float>
    tbb::concurrent_map cm1(v.begin(), v.end());

    // Deduces cm2 as concurrent_map<int, float>
    tbb::concurrent_map cm2({std::pair(1, 2f), std::pair(2, 3f)});}

}

```

### 11.2.5.4.2 concurrent\_multimap

#### [containers.concurrent\_multimap]

`tbb::concurrent_multimap` is a class template represents a sorted associative container which supports concurrent insertion, lookup and traversal, but not concurrent erasure. The container allows to store multiple elements with equal keys.

#### 11.2.5.4.2.1 Class Template Synopsis

```
namespace tbb {

    template <typename Key,
              typename T,
              typename Compare = std::less<Key>,
              typename Allocator = tbb_allocator<std::pair<const Key, T>>
    class concurrent_multimap {
public:
    using key_type = Key;
    using mapped_type = T;
    using value_type = std::pair<const Key, T>;

    using size_type = <implementation-defined unsigned integer type>;
    using difference_type = <implementation-defined signed integer type>;

    using key_compare = Compare;
    using allocator_type = Allocator;

    using reference = value_type&;
    using const_reference = const value_type&;
    using pointer = std::allocator_traits<Allocator>::pointer;
    using const_pointer = std::allocator_traits<Allocator>::const_pointer;

    using iterator = <implementation-defined ForwardIterator>;
    using const_iterator = <implementation-defined constant ForwardIterator>;

    using node_type = <implementation-defined node handle>;

    using range_type = <implementation-defined range>;
    using const_range_type = <implementation-defined constant node handle>;

    class value_compare;

    // Construction, destruction, copying
    concurrent_multimap();
    explicit concurrent_multimap( const key_compare& comp,
                                const allocator_type& alloc = allocator_type() );
    explicit concurrent_multimap( const allocator_type& alloc );

    template <typename InputIterator>
    concurrent_multimap( InputIterator first, InputIterator last,
                        const key_compare& comp = key_compare(),
                        const allocator_type& alloc = allocator_type() );

```

(continues on next page)

(continued from previous page)

```

template <typename InputIterator>
concurrent_multimap( InputIterator first, InputIterator last,
                     const allocator_type& alloc );

concurrent_multimap( std::initializer_list<value_type> init,
                     const key_compare& comp = key_compare(),
                     const allocator_type& alloc = allocator_type() );

concurrent_multimap( std::initializer_list<value_type> init, const allocator_
                     ↪type& alloc );

concurrent_multimap( const concurrent_multimap& other );
concurrent_multimap( const concurrent_multimap& other,
                     const allocator_type& alloc );

concurrent_multimap( concurrent_multimap&& other );
concurrent_multimap( concurrent_multimap&& other,
                     const allocator_type& alloc );

~concurrent_multimap();

concurrent_multimap& operator= ( const concurrent_multimap& other );
concurrent_multimap& operator= ( concurrent_multimap&& other );
concurrent_multimap& operator= ( std::initializer_list<value_type> init );

allocator_type get_allocator() const;

// Iterators
iterator begin();
const_iterator begin() const;
const_iterator cbegin() const;

iterator end();
const_iterator end() const;
const_iterator cend() const;

// Size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// Concurrently safe modifiers
std::pair<iterator, bool> insert( const value_type& value );

iterator insert( const_iterator hint, const value_type& value );

template <typename P>
std::pair<iterator, bool> insert( P&& value );

template <typename P>
iterator insert( const_iterator hint, P&& value );

std::pair<iterator, bool> insert( value_type&& value );

iterator insert( const_iterator hint, value_type&& value );

template <typename InputIterator>

```

(continues on next page)

(continued from previous page)

```

void insert( InputIterator first, InputIterator last );

void insert( std::initializer_list<value_type> init );

std::pair<iterator, bool> insert( node_type&& nh );
iterator insert( const_iterator hint, node_type&& nh );

template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args );

template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );

template <typename SrcCompare>
void merge( concurrent_map<Key, T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_map<Key, T, SrcCompare, Allocator>&& source );

template <typename SrcCompare>
void merge( concurrent_multimap<Key, T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_multimap<Key, T, SrcCompare, Allocator>&& source );

// Concurrently unsafe modifiers
void clear();

iterator unsafe_erase( const_iterator pos );
iterator unsafe_erase( iterator pos );

iterator unsafe_erase( const_iterator first, const_iterator last );

size_type unsafe_erase( const key_type& key );

template <typename K>
size_type unsafe_erase( const K& key );

node_type unsafe_extract( const_iterator pos );
node_type unsafe_extract( iterator pos );

node_type unsafe_extract( const key_type& key );

template <typename K>
node_type unsafe_extract( const K& key );

void swap( concurrent_multimap& other );

// Lookup
size_type count( const key_type& key );

template <typename K>
size_type count( const K& key );

iterator find( const key_type& key );
const_iterator find( const key_type& key ) const;

```

(continues on next page)

(continued from previous page)

```

template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;

bool contains( const key_type& key ) const;

template <typename K>
bool contains( const K& key ) const;

std::pair<iterator, iterator> equal_range( const key_type& key );
std::pair<const_iterator, const_iterator> equal_range( const key_type& key )  
↳const;

template <typename K>
std::pair<iterator, iterator> equal_range( const K& key );
std::pair<const_iterator, const_iterator> equal_range( const K& key ) const;

iterator lower_bound( const key_type& key );
const_iterator lower_bound( const key_type& key ) const;

template <typename K>
iterator lower_bound( const K& key );

template <typename K>
const_iterator lower_bound( const K& key ) const;

iterator upper_bound( const key_type& key );
const_iterator upper_bound( const key_type& key ) const;

template <typename K>
iterator upper_bound( const K& key );

template <typename K>
const_iterator upper_bound( const K& key ) const;

// Observers
key_compare key_comp() const;

value_compare value_comp() const;

// Parallel iteration
range_type range();
const_range_type range() const;
};

// class concurrent_multimap

} // namespace tbb

```

Requirements:

- The expression `std::allocator_traits<Allocator>::destroy(m, val)` where `m` is an object of the type `Allocator` and `val` is an object of the type `value_type` should be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Compare` shall meet the `Compare` requirements from [alg.sorting] ISO C++ Standard section.
- The type `Allocator` shall meet the `Allocator` requirements from [allocator.requirements] ISO C++ Stan-

dard section.

#### 11.2.5.4.2.2 Member classes

##### 11.2.5.4.2.3 value\_compare

`concurrent_multimap::value_compare` is a function object which is used to compare `concurrent_multimap::value_type` objects by comparing their first components.

##### 11.2.5.4.2.4 Class Synopsis

```
namespace tbb {

    template <typename Key, typename T,
              typename Compare, typename Allocator>
    class concurrent_multimap<Key, T, Compare, Allocator>::value_compare {
protected:
    key_compare comp;

    value_compare( key_compare c );

public:
    bool operator()( const value_type& lhs, const value_type& rhs ) const;
    // class value_compare
} // namespace tbb
```

##### 11.2.5.4.2.5 Member objects

```
key_compare comp;
```

The key comparison function object.

##### 11.2.5.4.2.6 Member functions

```
value_compare( key_compare c );
```

Constructs a `value_compare` with the stored key comparison function object `c`.

```
bool operator()( const value_type& lhs, const value_type& rhs ) const;
```

Compares `lhs.first` and `rhs.first` by calling the stored key comparison function `comp`.

**Returns:** true if first components of `lhs` and `rhs` are equal, false otherwise.

#### 11.2.5.4.2.7 Member functions

#### 11.2.5.4.2.8 Construction, destruction, copying

#### 11.2.5.4.2.9 Empty container constructors

```
concurrent_multimap();

explicit concurrent_multimap( const key_compare& comp,
                                const allocator_type& alloc = allocator_type() );
                                ↵;

explicit concurrent_multimap( const allocator_type& alloc );
```

Constructs empty concurrent\_multimap.

If provided uses the comparison function object `comp` for all `key_type` comparisons and the allocator `alloc` to allocate the memory.

#### 11.2.5.4.2.10 Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_multimap( InputIterator first, InputIterator last,
                     const key_compare& comp = key_compare(),
                     const allocator_type& alloc = allocator_type() );

template <typename InputIterator>
concurrent_multimap( InputIterator first, InputIterator last,
                     const allocator_type& alloc = allocator_type() );
```

Constructs the concurrent\_multimap which contains all elements from the half-open interval [first, last].

If provided uses the comparison function object `comp` for all `key_type` comparisons and the allocator `alloc` to allocate the memory.

**Requirements:** the type `InputIterator` shall meet the requirements of `InputIterator` from [`input iterators`] ISO C++ Standard section.

```
concurrent_multimap( std::initializer_list<value_type> init, const key_
                     ↵compare& comp = key_compare(),
                     const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_multimap(init.begin(), init.end(), comp, alloc)`.

```
concurrent_multimap( std::initializer_list<value_type> init,
                     const allocator_type& alloc );
```

Equivalent to `concurrent_multimap(init.begin(), init.end(), alloc)`.

#### 11.2.5.4.2.11 Copying constructors

```
concurrent_multimap( const concurrent_multimap& other );
concurrent_multimap( const concurrent_multimap& other, const allocator_type&✓  
alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

#### 11.2.5.4.2.12 Moving constructors

```
concurrent_multimap( concurrent_multimap&& other );
concurrent_multimap( concurrent_multimap&& other, const allocator_type&✓  
alloc );
```

Constructs a `concurrent_multimap` with the contents of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

#### 11.2.5.4.2.13 Destructor

```
~concurrent_multimap();
```

Destroys the `concurrent_multimap`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

#### 11.2.5.4.2.14 Assignment operators

```
concurrent_multimap& operator=( const concurrent_multimap& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment` is true.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

**Returns:** a reference to `*this`.

```
concurrent_multimap& operator=( concurrent_multimap&& other );
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment` is true.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

**Returns:** a reference to `*this`.

```
concurrent_multimap& operator=( std::initializer_list<value_type> init );
```

Replaces all elements in `*this` by the elements in `init`.

If `init` contains multiple elements with equal keys, it is unspecified which element would be inserted.

The behavior is undefined in case of concurrent operations with `*this`.

**Returns:** a reference to `*this`.

#### 11.2.5.4.2.15 Iterators

The types `concurrent_multimap::iterator` and `concurrent_multimap::const_iterator` meets the requirements of `ForwardIterator` from [forward.iterators] ISO C++ standard section.

#### 11.2.5.4.2.16 begin and cbegin

```
iterator begin();
const_iterator begin() const;
const_iterator cbegin() const;
```

**Returns:** an iterator to the first element in the container.

#### 11.2.5.4.2.17 end and cend

```
iterator end();
const_iterator end() const;
const_iterator cend() const;
```

**Returns:** an iterator to the element which follows the last element in the container.

#### 11.2.5.4.2.18 Size and capacity

#### 11.2.5.4.2.19 empty

```
bool empty() const;
```

**Returns:** true if the container is empty, false otherwise.

The result may differ with the actual container state in case of pending concurrent insertions.

#### 11.2.5.4.2.20 size

```
size_type size() const;
```

**Returns:** the number of elements in the container.

The result may differ with the actual container size in case of pending concurrent insertions.

#### 11.2.5.4.2.21 max\_size

```
size_type max_size() const;
```

**Returns:** the maximum number of elements that container can hold.

#### 11.2.5.4.2.22 Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other, lookup methods and while traversing the container.

#### 11.2.5.4.2.23 Emplacing elements

```
template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args );
```

Inserts an element ,constructed in-place from args into the container.

**Returns:** std::pair<iterator, bool> where iterator points to the inserted element. Boolean value is always true.

**Requirements:** the type value\_type shall meet the EmplaceConstructible requirements from [container.requirements] ISO C++ section.

```
template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );
```

Inserts an element ,constructed in-place from args into the container.

Optionally uses the parameter hint as a suggestion to where the node should be placed.

**Returns:** an iterator to the inserted element.

**Requirements:** the type `value_type` shall meet the `EmplaceConstructible` requirements from [container.requirements] ISO C++ section.

#### 11.2.5.4.2.24 Inserting values

```
std::pair<iterator, bool> insert( const value_type& value );
```

Inserts the value `value` into the container.

**Returns:** `std::pair<iterator, bool>` where `iterator` points to the inserted element. Boolean value is always true.

**Requirements:** the type `value_type` shall meet the `CopyInsertable` requirements from [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, const value_type& other );
```

Inserts the value `value` into the container.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

**Returns:** an `iterator` to the inserted element.

**Requirements:** the type `value_type` shall meet the `CopyInsertable` requirements from [container.requirements] ISO C++ Standard section.

```
template <typename P>
std::pair<iterator, bool> insert( P&& value );
```

Equivalent to `emplace(std::forward<P>(value))`.

This overload only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is true.

```
template <typename P>
iterator insert( const_iterator hint, P&& value );
```

Equivalent to `emplace_hint(hint, std::forward<P>(value))`.

This overload only participates in overload resolution if `std::is_constructible<value_type, P&&>::value` is true.

```
std::pair<iterator, bool> insert( value_type&& value );
```

Inserts the value `value` into the container using move semantics.

`value` is left in a valid, but unspecified state.

**Returns:** `std::pair<iterator, bool>` where `iterator` points to the inserted element. Boolean value is always true.

**Requirements:** the type `value_type` shall meet the `MoveInsertable` requirements from [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, value_type&& other );
```

Inserts the value `value` into the container using move semantics.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

`value` is left in a valid, but unspecified state.

**Returns:** an `iterator` to the inserted element.

**Requirements:** the type `value_type` shall meet the `MoveInsertable` requirements from [container.requirements] ISO C++ Standard section.

#### 11.2.5.4.2.25 Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last );
```

Inserts all items from the half-open interval `[first, last)` into the container.

**Requirements:** the type `InputIterator` must meet the requirements of `InputIterator` from [input.iterators] ISO C++ Standard section.

```
void insert( std::initializer_list<value_type> init );
```

Equivalent to `insert(init.begin(), init.end())`.

#### 11.2.5.4.2.26 Inserting nodes

```
std::pair<iterator, bool> insert( node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise - inserts the node, owned by `nh` into the container.

`nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

**Returns:** `std::pair<iterator, bool>` where `iterator` points to the inserted element. Boolean value is always `true`.

```
iterator insert( const_iterator hint, node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise - inserts the node, owned by `nh` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

`nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

**Returns:** an iterator pointing to the inserted element.

#### 11.2.5.4.2.27 Merging containers

```
template <typename SrcCompare>
void merge( concurrent_map<Key, T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_map<Key, T, SrcCompare, Allocator>&& source );

template <typename SrcCompare>
void merge( concurrent_multimap<Key, T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_multimap<Key, T, SrcCompare, Allocator>&& source );
```

Transfers all elements from `source` to `*this`.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `get_allocator() != source.get_allocator()`.

#### 11.2.5.4.2.28 Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

#### 11.2.5.4.2.29 Clearing

```
void clear();
```

Removes all elements from the container.

#### 11.2.5.4.2.30 Erasing elements

```
iterator unsafe_erase( const_iterator pos );
iterator unsafe_erase( iterator pos );
```

Removes the element pointed to by `pos` from the container.

Invalidates all iterators and references to the removed element.

**Returns:** iterator which follows the removed element.

**Requirements:** the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
size_type unsafe_erase( const key_type& key );
```

Removes all element with the key equal to key if it exists in the container.

Invalidates all iterators and references to the removed elements.

**Returns:** the number of removed elements.

```
template <typename K>
size_type unsafe_erase( const K& key );
```

Removes all elements with the key which compares equivalent to key if it exists in the container.

Invalidates all iterators and references to the removed elements.

This overload only participates in overload resolution if all of the following statements are true:

- The qualified-id `key_compare::is_transparent` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

**Returns:** the number of removed elements.

#### 11.2.5.4.2.31 Erasing sequences

```
iterator unsafe_erase( const_iterator first, const_iterator last );
```

Removes all elements from the half-open interval [first, last) from the container.

**Returns:** iterator which follows the last removed element.

**Requirements:** the range [first, last) must be a valid subrange in `*this`.

#### 11.2.5.4.2.32 Extracting nodes

```
node_type unsafe_extract( iterator pos );
node_type unsafe_extract( const_iterator pos );
```

Transfers ownership of the element pointed to by pos from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

**Returns:** the node handle that owns the extracted element.

**Requirements:** the iterator pos should be valid, dereferenceable and point to the element in `*this`.

```
node_type unsafe_extract( const key_type& key );
```

If at least one element with the key equal to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

If there are multiple elements with the key equal to `key` exists, it is unspecified which element should be transferred.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

**Returns:** the node handle that owns the extracted element or an empty node handle if an element with the key equal to `key` was not found.

```
template <typename K>
node_type unsafe_extract( const K& key );
```

If at least one element with the key which compares equivalent to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

If there are multiple elements with the key which compares equivalent with `key` exists, it is unspecified which element should be transferred.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

This overload only participates in overload resolution if all of the following statements are true:

- The qualified-id `key_compare::is_transparent` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

**Returns:** the node handle that owns the extracted element or an empty node handle if an element with the key which compares equivalent to `key` was not found.

#### 11.2.5.4.2.33 swap

```
void swap( concurrent_multimap& other );
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`.

Otherwise if `get_allocator() != other.get_allocator()` the behavior is undefined.

#### 11.2.5.4.2.34 Lookup

All methods in this section can be executed concurrently with each other, concurrently-safe modifiers and while traversing the container.

#### 11.2.5.4.2.35 count

```
size_type count( const key_type& key );
```

**Returns:** the number of elements with the key equal to key.

```
template <typename K>
size_type count( const K& key );
```

**Returns:** the number of elements with the key which compares equivalent with key.

This overload only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

#### 11.2.5.4.2.36 find

```
iterator find( const key_type& key );
const_iterator find( const key_type& key ) const;
```

**Returns:** an iterator to the element with the key equal to key or `end()` if no such element exists.

If there are multiple elements with the key equal to key exists, it is unspecified which element should be found.

```
template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;
```

**Returns:** an iterator to the element with the key which compares equivalent with key or `end()` if no such element exists.

If there are multiple elements with the key which compares equivalent with key exists, it is unspecified which element should be found.

These overloads only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

#### 11.2.5.4.2.37 contains

```
bool contains( const key_type& key ) const;
```

**Returns:** true if an element with the key equal to key exists in the container, false otherwise.

```
template <typename K>
bool contains( const K& key ) const;
```

**Returns:** true if an element with the key which compares equivalent with key exists in the container, false otherwise.

This overload only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

#### 11.2.5.4.2.38 lower\_bound

```
iterator lower_bound( const key_type& key );
const_iterator lower_bound( const key_type& key ) const;
```

**Returns:** an iterator to the first element in the container with the key which is *not less* than key.

```
template <typename K>
iterator lower_bound( const K& key )

template <typename K>
const_iterator lower_bound( const K& key ) const
```

**Returns:** an iterator to the first element in the container with the key which compares *not less* with key.

These overloads only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

#### 11.2.5.4.2.39 upper\_bound

```
iterator upper_bound( const key_type& key );
const_iterator upper_bound( const key_type& key ) const;
```

**Returns:** an iterator to the first element in the container with the key which is *greater* than key.

```
template <typename K>
iterator upper_bound( const K& key );

template <typename K>
const_iterator upper_bound( const K& key ) const;
```

**Returns:** an iterator to the first element in the container with the key which compares greater with key.

These overloads only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

#### 11.2.5.4.2.40 equal\_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );
std::pair<const_iterator, const_iterator> equal_range( const key_type& key ) ↳ const;
```

**Returns:** if at least one element with the key equal to key exists - a pair of iterators {f, l}, where f is an iterator to the first element with the key equal to key, l is an iterator to the element which follows the last element with the key equal to key. Otherwise - {end(), end() }.

```
template <typename K>
std::pair<iterator, iterator> equal_range( const K& key )

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key )
```

**Returns:** if at least one element with the key equal to key exists - a pair of iterators {f, l}, where f is an iterator to the first element with the key which compares equivalent with key, l is an iterator to the element which follows the last element with the key which compares equivalent with key. Otherwise - {end(), end() }.

These overloads only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

#### 11.2.5.4.2.41 Observers

##### 11.2.5.4.2.42 get\_allocator

```
allocator_type get_allocator() const;
```

**Returns:** a copy of the allocator associated with `*this`.

##### 11.2.5.4.2.43 key\_comp

```
key_compare key_comp() const;
```

**Returns:** a copy of the key comparison functor associated with `*this`.

#### 11.2.5.4.2.44 value\_comp

```
value_compare value_comp() const;
```

**Returns:** an object of the `value_compare` class which is used to compare `value_type` objects.

#### 11.2.5.4.2.45 Parallel iteration

Member types `concurrent_multimap::range_type` and `concurrent_multimap::const_range_type` meets the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_multimap::const_range_type` are of type `concurrent_multimap::const_iterator`, whereas the bounds for a `concurrent_multimap::range_type` are of type `concurrent_multimap::iterator`.

#### 11.2.5.4.2.46 range member function

```
range_type range();
const_range_type range() const;
```

**Returns:** a range object representing all elements in the container.

#### 11.2.5.4.2.47 Non-member functions

These functions provides binary and lexicographical comparison and swap operations on `tbb::concurrent_multimap` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `tbb::concurrent_multimap` as a type alias for which the non-member functions are reachable only via argument dependent lookup.

```
template <typename Key, typename T, typename Compare, typename Allocator>
void swap( concurrent_multimap<Key, T, Compare, Allocator>& lhs,
           concurrent_multimap<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                    const concurrent_multimap<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                    const concurrent_multimap<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator<( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                   const concurrent_multimap<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                   const concurrent_multimap<Key, T, Compare, Allocator>& rhs );
```

(continues on next page)

(continued from previous page)

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator<=( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                  const concurrent_multimap<Key, T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>=( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                  const concurrent_multimap<Key, T, Compare, Allocator>& rhs );
```

#### 11.2.5.4.2.48 Non-member swap

```
template <typename Key, typename T, typename Compare, typename Allocator>
void swap( concurrent_multimap<Key, T, Compare, Allocator>& lhs,
           concurrent_multimap<Key, T, Compare, Allocator>& rhs );
```

Equivalent to `lhs.swap(rhs)`.

#### 11.2.5.4.2.49 Non-member binary comparisons

Two `tbb::concurrent_multimap` objects are equal if they have the same number of elements and each element in one container is equal to the element in other container on the same position.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                   const concurrent_multimap<Key, T, Compare, Allocator>& rhs )
```

**Returns:** true if `lhs` is equal to `rhs`, false otherwise.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                   const concurrent_multimap<Key, T, Compare, Allocator>& rhs )
```

**Returns:** true if `lhs` is not equal to `rhs`, false otherwise.

#### 11.2.5.4.2.50 Non-member lexicographical comparisons

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator<( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                  const concurrent_multimap<Key, T, Compare, Allocator>& rhs )
```

**Returns:** true if `lhs` is lexicographically *less* than `rhs`.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator<=( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                  const concurrent_multimap<Key, T, Compare, Allocator>& rhs )
```

**Returns:** true if `lhs` is lexicographically *less or equal* than `rhs`.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                    const concurrent_multimap<Key, T, Compare, Allocator>& rhs )
```

**Returns:** true if lhs is lexicographically *greater* than rhs.

```
template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>=( const concurrent_multimap<Key, T, Compare, Allocator>& lhs,
                     const concurrent_multimap<Key, T, Compare, Allocator>& rhs )
```

**Returns:** true if lhs is lexicographically *greater or equal* than rhs.

#### 11.2.5.4.2.51 Other

##### 11.2.5.4.2.52 Deduction guides

Where possible, constructors of concurrent\_multimap supports class template argument deduction (since C++17):

```
template <typename InputIterator,
          typename Compare = std::less<iterator_key_t<InputIterator>>,
          typename Allocator = tbb_allocator<iterator_alloc_value_t<InputIterator>>>
concurrent_multimap( InputIterator, InputIterator, Compare = Compare(), Allocator = Allocator() )
-> concurrent_multimap<iterator_key_t<InputIterator>,
                      iterator_mapped_t<InputIterator>,
                      Compare,
                      Allocator>;

template <typename InputIterator,
          typename Allocator>
concurrent_multimap( InputIterator, InputIterator, Allocator )
-> concurrent_multimap<iterator_key_t<InputIterator>,
                      iterator_mapped_t<InputIterator>,
                      std::less<iterator_key_t<InputIterator>>,
                      Allocator>;

template <typename Key,
          typename T,
          typename Compare = std::less<Key>,
          typename Allocator = tbb_allocator<std::pair<const Key, T>>>
concurrent_multimap( std::initializer_list<std::pair<Key, T>>, Compare = Compare(), Allocator = Allocator() )
-> concurrent_multimap<Key, T, Compare, Allocator>;

template <typename Key,
          typename T,
          typename Allocator>
concurrent_multimap( std::initializer_list<std::pair<Key, T>>, Allocator )
-> concurrent_multimap<Key, T, std::less<Key>, Allocator>;
```

where the type aliases iterator\_key\_t, iterator\_mapped\_t, iterator\_alloc\_value\_t defines as follows:

```

template <typename InputIterator>
using iterator_key_t = std::remove_const_t<typename std::iterator_traits<
    <InputIterator>::value_type::first_type>;
```

```

template <typename InputIterator>
using iterator_mapped_t = typename std::iterator_traits<InputIterator>::value_
    type::second_type;
```

```

template <typename InputIterator>
using iterator_alloc_value_t = std::pair<std::add_const_t<iterator_key_t
    <InputIterator>>,
```

iterator\_mapped\_t<InputIterator>>;

### Example

```

#include <tbb/concurrent_map.h>
#include <vector>

int main() {
    std::vector<std::pair<int, float>> v;

    // Deduces cm1 as concurrent_multimap<int, float>
    tbb::concurrent_multimap cm1(v.begin(), v.end());

    // Deduces cm2 as concurrent_multimap<int, float>
    tbb::concurrent_multimap cm2({std::pair(1, 2f), std::pair(2, 3f)});
```

}

### 11.2.5.4.3 concurrent\_set

#### [containers.concurrent\_set]

`tbb::concurrent_set` is a class template represents an unordered sequence of unique elements which supports concurrent insertion, lookup and traversal, but not concurrent erasure.

#### 11.2.5.4.3.1 Class Template Synopsis

```

// Defined in header <tbb/concurrent_set.h>

namespace tbb {

    template <typename T,
        typename Compare = std::less<T>,
        typename Allocator = tbb_allocator<T>>
    class concurrent_set {
    public:
        using key_type = T;
        using value_type = T;

        using size_type = <implementation-defined unsigned integer type>;
        using difference_type = <implementation-defined signed integer type>;

        using key_compare = Compare;
        using value_compare = Compare;
```

(continues on next page)

(continued from previous page)

```

using allocator_type = Allocator;

using reference = value_type&;
using const_reference = const value_type&;
using pointer = std::allocator_traits<Allocator>::pointer;
using const_pointer = std::allocator_traits<Allocator>::const_pointer;

using iterator = <implementation-defined ForwardIterator>;
using const_iterator = <implementation-defined constant ForwardIterator>;

using node_type = <implementation-defined node handle>;

using range_type = <implementation-defined range>;
using const_range_type = <implementation-defined constant node handle>;

// Construction, destruction, copying
concurrent_set();
explicit concurrent_set( const key_compare& comp,
                           const allocator_type& alloc = allocator_type() );
explicit concurrent_set( const allocator_type& alloc );

template <typename InputIterator>
concurrent_set( InputIterator first, InputIterator last,
                const key_compare& comp = key_compare(),
                const allocator_type& alloc = allocator_type() );

template <typename InputIterator>
concurrent_set( InputIterator first, InputIterator last,
                const allocator_type& alloc );

concurrent_set( std::initializer_list<value_type> init,
                const key_compare& comp = key_compare(),
                const allocator_type& alloc = allocator_type() );

concurrent_set( std::initializer_list<value_type> init, const allocator_type& alloc );
→alloc );

concurrent_set( const concurrent_set& other );
concurrent_set( const concurrent_set& other,
                const allocator_type& alloc );

concurrent_set( concurrent_set&& other );
concurrent_set( concurrent_set&& other,
                const allocator_type& alloc );

~concurrent_set();

concurrent_set& operator=( const concurrent_set& other );
concurrent_set& operator=( concurrent_set&& other );
concurrent_set& operator=( std::initializer_list<value_type> init );

allocator_type get_allocator() const;

// Iterators
iterator begin();

```

(continues on next page)

(continued from previous page)

```

const_iterator begin() const;
const_iterator cbegin() const;

iterator end();
const_iterator end() const;
const_iterator cend() const;

// Size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// Concurrently safe modifiers
std::pair<iterator, bool> insert( const value_type& value );

iterator insert( const_iterator hint, const value_type& value );

std::pair<iterator, bool> insert( value_type&& value );

iterator insert( const_iterator hint, value_type&& value );

template <typename InputIterator>
void insert( InputIterator first, InputIterator last );

void insert( std::initializer_list<value_type> init );

std::pair<iterator, bool> insert( node_type&& nh );
iterator insert( const_iterator hint, node_type&& nh );

template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args );

template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );

template <typename SrcCompare>
void merge( concurrent_set<T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_set<T, SrcCompare, Allocator>&& source );

template <typename SrcCompare>
void merge( concurrent_multiset<T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_multiset<T, SrcCompare, Allocator>&& source );

// Concurrently unsafe modifiers
void clear();

iterator unsafe_erase( const_iterator pos );
iterator unsafe_erase( iterator pos );

iterator unsafe_erase( const_iterator first, const_iterator last );

size_type unsafe_erase( const key_type& key );

```

(continues on next page)

(continued from previous page)

```

template <typename K>
size_type unsafe_erase( const K& key );

node_type unsafe_extract( const_iterator pos );
node_type unsafe_extract( iterator pos );

node_type unsafe_extract( const key_type& key );

template <typename K>
node_type unsafe_extract( const K& key );

void swap( concurrent_set& other );

// Lookup
size_type count( const key_type& key );

template <typename K>
size_type count( const K& key );

iterator find( const key_type& key );
const_iterator find( const key_type& key ) const;

template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;

bool contains( const key_type& key ) const;

template <typename K>
bool contains( const K& key ) const;

std::pair<iterator, iterator> equal_range( const key_type& key );
std::pair<const_iterator, const_iterator> equal_range( const key_type& key )  

const;  

template <typename K>
std::pair<iterator, iterator> equal_range( const K& key );
std::pair<const_iterator, const_iterator> equal_range( const K& key ) const;  

iterator lower_bound( const key_type& key );
const_iterator lower_bound( const key_type& key ) const;  

template <typename K>
iterator lower_bound( const K& key );  

template <typename K>
const_iterator lower_bound( const K& key ) const;  

iterator upper_bound( const key_type& key );
const_iterator upper_bound( const key_type& key ) const;  

template <typename K>
iterator upper_bound( const K& key );  

template <typename K>

```

(continues on next page)

(continued from previous page)

```

const_iterator upper_bound( const K& key ) const;

// Observers
key_compare key_comp() const;

value_compare value_comp() const;

// Parallel iteration
range_type range();
const_range_type range() const;
};

// class concurrent_set

} // namespace tbb

```

#### Requirements:

- The expression `std::allocator_traits<Allocator>::destroy(m, val)` where `m` is an object of the type `Allocator` and `val` is an object of the type `value_type` should be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Compare` shall meet the `Compare` requirements from [alg.sorting] ISO C++ Standard section.
- The type `Allocator` shall meet the `Allocator` requirements from [allocator.requirements] ISO C++ Standard section.

#### 11.2.5.4.3.2 Member functions

#### 11.2.5.4.3.3 Construction, destruction, copying

#### 11.2.5.4.3.4 Empty container constructors

```

concurrent_set();

explicit concurrent_set( const key_compare& comp,
                         const allocator_type& alloc = allocator_type() );

explicit concurrent_set( const allocator_type& alloc );

```

Constructs empty `concurrent_set`.

If provided uses the comparison function object `comp` for all `key_type` comparisons and the allocator `alloc` to allocate the memory.

#### 11.2.5.4.3.5 Constructors from the sequence of elements

```

template <typename InputIterator>
concurrent_set( InputIterator first, InputIterator last,
                const key_compare& comp = key_compare(),
                const allocator_type& alloc = allocator_type() );

template <typename InputIterator>
concurrent_set( InputIterator first, InputIterator last,
                const allocator_type& alloc = allocator_type() );

```

Constructs the `concurrent_set` which contains the elements from the half-open interval `[first, last)`.

If the range `[first, last)` contains multiple equal elements, it is unspecified which element would be inserted.

If provided uses the comparison function object `comp` for all `key_type` comparisons and the allocator `alloc` to allocate the memory.

**Requirements:** the type `InputIterator` shall meet the requirements of `InputIterator` from [`input iterators`] ISO C++ Standard section.

```
concurrent_set( std::initializer_list<value_type> init, const key_compare& comp = key_compare(),
const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_set(init.begin(), init.end(), comp, alloc)`.

```
concurrent_set( std::initializer_list<value_type> init,
const allocator_type& alloc );
```

Equivalent to `concurrent_set(init.begin(), init.end(), alloc)`.

#### 11.2.5.4.3.6 Copying constructors

```
concurrent_set( const concurrent_set& other );
concurrent_set( const concurrent_set& other, const allocator_type& alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

#### 11.2.5.4.3.7 Moving constructors

```
concurrent_set( concurrent_set&& other );
concurrent_set( concurrent_set&& other, const allocator_type& alloc );
```

Constructs a `concurrent_set` with the contents of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

#### 11.2.5.4.3.8 Destructor

```
~concurrent_set();
```

Destroys the concurrent\_set. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

#### 11.2.5.4.3.9 Assignment operators

```
concurrent_set& operator=( const concurrent_set& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment` is true.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

**Returns:** a reference to `*this`.

```
concurrent_set& operator=( concurrent_set&& other );
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment` is true.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

**Returns:** a reference to `*this`.

```
concurrent_set& operator=( std::initializer_list<value_type> init );
```

Replaces all elements in `*this` by the elements in `init`.

If `init` contains multiple elements with equal keys, it is unspecified which element would be inserted.

The behavior is undefined in case of concurrent operations with `*this`.

**Returns:** a reference to `*this`.

#### 11.2.5.4.3.10 Iterators

The types `concurrent_set::iterator` and `concurrent_set::const_iterator` meets the requirements of `ForwardIterator` from [forward.iterators] ISO C++ standard section.

#### 11.2.5.4.3.11 begin and cbegin

```
iterator begin();
const_iterator begin() const;
const_iterator cbegin() const;
```

**Returns:** an iterator to the first element in the container.

#### 11.2.5.4.3.12 end and cend

```
iterator end();
const_iterator end() const;
const_iterator cend() const;
```

**Returns:** an iterator to the element which follows the last element in the container.

#### 11.2.5.4.3.13 Size and capacity

##### 11.2.5.4.3.14 empty

```
bool empty() const;
```

**Returns:** true if the container is empty, false otherwise.

The result may differ with the actual container state in case of pending concurrent insertions.

##### 11.2.5.4.3.15 size

```
size_type size() const;
```

**Returns:** the number of elements in the container.

The result may differ with the actual container size in case of pending concurrent insertions.

##### 11.2.5.4.3.16 max\_size

```
size_type max_size() const;
```

**Returns:** the maximum number of elements that container can hold.

#### 11.2.5.4.3.17 Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other, lookup methods and while traversing the container.

#### 11.2.5.4.3.18 Inserting values

```
std::pair<iterator, bool> insert( const value_type& value );
```

Attempts to insert the value `value` into the container.

**Returns:** `std::pair<iterator, bool` where `iterator` points to the inserted element or to an existing element with equal key. Boolean value is `true` if insertion took place, `false` otherwise.

**Requirements:** the type `value_type` shall meet the `CopyInsertable` requirements from [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, const value_type& other );
```

Attempts to insert the value `value` into the container.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

**Returns:** an `iterator` to the inserted element or to an existing element with equal key.

**Requirements:** the type `value_type` shall meet the `CopyInsertable` requirements from [container.requirements] ISO C++ Standard section.

```
std::pair<iterator, bool> insert( value_type&& value );
```

Attempts to insert the value `value` into the container using move semantics.

`value` is left in a valid, but unspecified state.

**Returns:** `std::pair<iterator, bool` where `iterator` points to the inserted element or to an existing element with equal key. Boolean value is `true` if insertion took place, `false` otherwise.

**Requirements:** the type `value_type` shall meet the `MoveInsertable` requirements from [container.requirements] ISO C++ Standard section.

```
iterator insert( const_iterator hint, value_type&& other );
```

Attempts to insert the value `value` into the container using move semantics.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

`value` is left in a valid, but unspecified state.

**Returns:** an `iterator` to the inserted element or to an existing element with equal key.

**Requirements:** the type `value_type` shall meet the `MoveInsertable` requirements from [container.requirements] ISO C++ Standard section.

#### 11.2.5.4.3.19 Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last );
```

Attempts to insert all items from the half-open interval [first, last) into the container.

If the interval [first, last) contains multiple equal elements, it is unspecified which element should be inserted.

**Requirements:** the type `InputIterator` must meet the requirements of `InputIterator` from [`input iterators`] ISO C++ Standard section.

```
void insert( std::initializer_list<value_type> init );
```

Equivalent to `insert(init.begin(), init.end())`.

#### 11.2.5.4.3.20 Inserting nodes

```
std::pair<iterator, bool

```

If the node handle `nh` is empty, does nothing.

Otherwise - attempts to insert the node, owned by `nh` into the container.

If the insertion fails, node handle `nh` remains ownership of the node.

Otherwise - `nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

**Returns:** `std::pair<iterator, bool` where `iterator` points to the inserted element or to an existing element equal to `nh.value()`. Boolean value is `true` if insertion took place, `false` otherwise.

```
iterator insert( const_iterator hint, node_type&& nh );
```

If the node handle `nh` is empty, does nothing.

Otherwise - attempts to insert the node, owned by `nh` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

If the insertion fails, node handle `nh` remains ownership of the node.

Otherwise - `nh` is left in an empty state.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `nh` is not empty and `get_allocator() != nh.get_allocator()`.

**Returns:** an iterator pointing to the inserted element or to an existing element equal to `nh.value()`.

#### 11.2.5.4.3.21 Emplacing elements

```
template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args );
```

Attempts to insert an element ,constructed in-place from `args` into the container.

**Returns:** `std::pair<iterator, bool` where `iterator` points to the inserted element or to an existing element with equal key. Boolean value is `true` if insertion took place, `false` otherwise.

**Requirements:** the type `value_type` shall meet the `EmplaceConstructible` requirements from [container.requirements] ISO C++ section.

```
template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );
```

Attempts to insert an element ,constructed in-place from `args` into the container.

Optionally uses the parameter `hint` as a suggestion to where the node should be placed.

**Returns:** an `iterator` to the inserted element or to an existing element with equal key.

**Requirements:** the type `value_type` shall meet the `EmplaceConstructible` requirements from [container.requirements] ISO C++ section.

#### 11.2.5.4.3.22 Merging containers

```
template <typename SrcCompare>
void merge( concurrent_set<T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_set<T, SrcCompare, Allocator>&& source );

template <typename SrcCompare>
void merge( concurrent_multiset<T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_multiset<T, SrcCompare, Allocator>&& source );
```

Transfers those elements from `source` which keys do not exist in the container.

In case of merging with the container with multiple equal elements, it is unspecified which element would be transferred.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `get_allocator() != source.get_allocator()`.

#### 11.2.5.4.3.23 Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

#### 11.2.5.4.3.24 Clearing

```
void clear();
```

Removes all elements from the container.

#### 11.2.5.4.3.25 Erasing elements

```
iterator unsafe_erase( const_iterator pos );
iterator unsafe_erase( iterator pos );
```

Removes the element pointed to by `pos` from the container.

Invalidates all iterators and references to the removed element.

**Returns:** iterator which follows the removed element.

**Requirements:** the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
size_type unsafe_erase( const key_type& key );
```

Removes the element equal to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

**Returns:** 1 if an element equal to `key` exists, 0 otherwise.

```
template <typename K>
size_type unsafe_erase( const K& key );
```

Removes the element which compares equivalent to `key` if it exists in the container.

Invalidates all iterators and references to the removed element.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `key_compare::is_transparent` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

**Returns:** 1 if an element which compares equivalent to `key` exists, 0 otherwise.

#### 11.2.5.4.3.26 Erasing sequences

```
iterator unsafe_erase( const_iterator first, const_iterator last );
```

Removes all elements from the half-open interval [first, last) from the container.

**Returns:** iterator which follows the last removed element.

**Requirements:** the range [first, last) must be a valid subrange in \*this.

#### 11.2.5.4.3.27 Extracting nodes

```
node_type unsafe_extract( iterator pos );
node_type unsafe_extract( const_iterator pos );
```

Transfers ownership of the element pointed to by pos from the container to the node handle.

No copy or move constructors of value\_type are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

**Returns:** the node handle that owns the extracted element.

**Requirements:** the iterator pos should be valid, dereferenceable and point to the element in \*this.

```
node_type unsafe_extract( const key_type& key );
```

If an element equal to key exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of value\_type are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

**Returns:** the node handle that owns the extracted element or an empty node handle if an element equal to key was not found.

```
template <typename K>
node_type unsafe_extract( const K& key );
```

If an element which compares equivalent to key exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of value\_type are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

This overload only participates in overload resolution if all of the following statements are true:

- The qualified-id `key_compare::is_transparent` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is false.

- `std::is_convertible<K, const_iterator>::value` is false.

**Returns:** the node handle that owns the extracted element or an empty node handle if an element which compares equivalent to key was not found.

#### 11.2.5.4.3.28 swap

```
void swap( concurrent_set& other );
```

Swaps contents of `*this` and `other`.

Swaps allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap::value` is true.

Otherwise if `get_allocator() != other.get_allocator()` the behavior is undefined.

#### 11.2.5.4.3.29 Lookup

All methods in this section can be executed concurrently with each other, concurrently-safe modifiers and while traversing the container.

#### 11.2.5.4.3.30 count

```
size_type count( const key_type& key );
```

**Returns:** the number of elements equal to key.

```
template <typename K>
size_type count( const K& key );
```

**Returns:** the number of elements which compares equivalent with key.

This overload only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

#### 11.2.5.4.3.31 find

```
iterator find( const key_type& key );
const_iterator find( const key_type& key ) const;
```

**Returns:** an iterator to the element equal to key or `end()` if no such element exists.

```
template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;
```

**Returns:** an iterator to the element which compares equivalent with `key` or `end()` if no such element exists.

These overloads only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

#### 11.2.5.4.3.32 `contains`

```
bool contains( const key_type& key ) const;
```

**Returns:** `true` if an element equal to `key` exists in the container, `false` otherwise.

```
template <typename K>
bool contains( const K& key ) const;
```

**Returns:** `true` if an element which compares equivalent with `key` exists in the container, `false` otherwise.

This overload only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

#### 11.2.5.4.3.33 `lower_bound`

```
iterator lower_bound( const key_type& key );
const_iterator lower_bound( const key_type& key ) const;
```

**Returns:** an iterator to the first element in the container which is *not less* than `key`.

```
template <typename K>
iterator lower_bound( const K& key )

template <typename K>
const_iterator lower_bound( const K& key ) const
```

**Returns:** an iterator to the first element in the container which compares *not less* with `key`.

These overloads only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

#### 11.2.5.4.3.34 `upper_bound`

```
iterator upper_bound( const key_type& key );
const_iterator upper_bound( const key_type& key ) const;
```

**Returns:** an iterator to the first element in the container which is *greater* than `key`.

```
template <typename K>
iterator upper_bound( const K& key );

template <typename K>
const_iterator upper_bound( const K& key ) const;
```

**Returns:** an iterator to the first element in the container which compares greater with key.

These overloads only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

#### 11.2.5.4.3.35 equal\_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );
std::pair<const_iterator, const_iterator> equal_range( const key_type& key ) const;
```

**Returns:** if an element equal to key exists - a pair of iterators {f, l}, where f is an iterator to this element, l is `std::next(f)`. Otherwise - {end(), end() }.

```
template <typename K>
std::pair<iterator, iterator> equal_range( const K& key )

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key )
```

**Returns:** if an element which compares equivalent with key exists - a pair of iterators {f, l}, where f is an iterator to this element, l is `std::next(f)`. Otherwise - {end(), end() }.

These overloads only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

#### 11.2.5.4.3.36 Observers

##### 11.2.5.4.3.37 get\_allocator

```
allocator_type get_allocator() const;
```

**Returns:** a copy of the allocator associated with `*this`.

##### 11.2.5.4.3.38 key\_comp

```
key_compare key_comp() const;
```

**Returns:** a copy of the key comparison functor associated with `*this`.

#### 11.2.5.4.3.39 value\_comp

```
value_compare value_comp() const;
```

**Returns:** an object of the `value_compare` class which is used to compare `value_type` objects.

#### 11.2.5.4.3.40 Parallel iteration

Member types `concurrent_set::range_type` and `concurrent_set::const_range_type` meets the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_set::const_range_type` are of type `concurrent_set::const_iterator`, whereas the bounds for a `concurrent_set::range_type` are of type `concurrent_set::iterator`.

#### 11.2.5.4.3.41 range member function

```
range_type range();
const_range_type range() const;
```

**Returns:** a range object representing all elements in the container.

#### 11.2.5.4.3.42 Non-member functions

These functions provides binary and lexicographical comparison and swap operations on `tbb::concurrent_set` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `tbb::concurrent_set` as a type alias for which the non-member functions are reachable only via argument dependent lookup.

```
template <typename T, typename Compare, typename Allocator>
void swap( concurrent_set<T, Compare, Allocator>& lhs,
           concurrent_set<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_set<T, Compare, Allocator>& lhs,
                    const concurrent_set<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_set<T, Compare, Allocator>& lhs,
                    const concurrent_set<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator<( const concurrent_set<T, Compare, Allocator>& lhs,
                   const concurrent_set<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator>( const concurrent_set<T, Compare, Allocator>& lhs,
                   const concurrent_set<T, Compare, Allocator>& rhs );
```

(continues on next page)

(continued from previous page)

```
template <typename T, typename Compare, typename Allocator>
bool operator<=( const concurrent_set<T, Compare, Allocator>& lhs,
                  const concurrent_set<T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>=( const concurrent_set<T, Compare, Allocator>& lhs,
                  const concurrent_set<T, Compare, Allocator>& rhs );
```

#### 11.2.5.4.3.43 Non-member swap

```
template <typename T, typename Compare, typename Allocator>
void swap( concurrent_set<T, Compare, Allocator>& lhs,
           concurrent_set<T, Compare, Allocator>& rhs );
```

Equivalent to `lhs.swap(rhs)`.

#### 11.2.5.4.3.44 Non-member binary comparisons

Two `tbb::concurrent_set` objects are equal if they have the same number of elements and each element in one container is equal to the element in other container on the same position.

```
template <typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_set<T, Compare, Allocator>& lhs,
                   const concurrent_set<T, Compare, Allocator>& rhs )
```

**Returns:** true if `lhs` is equal to `rhs`, false otherwise.

```
template <typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_set<T, Compare, Allocator>& lhs,
                   const concurrent_set<T, Compare, Allocator>& rhs )
```

**Returns:** true if `lhs` is not equal to `rhs`, false otherwise.

#### 11.2.5.4.3.45 Non-member lexicographical comparisons

```
template <typename T, typename Compare, typename Allocator>
bool operator<( const concurrent_set<T, Compare, Allocator>& lhs,
                  const concurrent_set<T, Compare, Allocator>& rhs )
```

**Returns:** true if `lhs` is lexicographically *less* than `rhs`.

```
template <typename T, typename Compare, typename Allocator>
bool operator<=( const concurrent_set<T, Compare, Allocator>& lhs,
                  const concurrent_set<T, Compare, Allocator>& rhs )
```

**Returns:** true if `lhs` is lexicographically *less or equal* than `rhs`.

```
template <typename T, typename Compare, typename Allocator>
bool operator>( const concurrent_set<T, Compare, Allocator>& lhs,
                    const concurrent_set<T, Compare, Allocator>& rhs )
```

**Returns:** true if lhs is lexicographically *greater* than rhs.

```
template <typename T, typename Compare, typename Allocator>
bool operator>=( const concurrent_set<T, Compare, Allocator>& lhs,
                     const concurrent_set<T, Compare, Allocator>& rhs )
```

**Returns:** true if lhs is lexicographically *greater or equal* than rhs.

#### 11.2.5.4.3.46 Other

##### 11.2.5.4.3.47 Deduction guides

Where possible, constructors of `concurrent_set` supports class template argument deduction (since C++17):

```
template <typename InputIterator,
          typename Compare = std::less<iterator_value_t<InputIterator>>,
          typename Allocator = tbb_allocator<iterator_value_t<InputIterator>>>
concurrent_set( InputIterator, InputIterator, Compare = Compare(), Allocator = Allocator() )
-> concurrent_set<iterator_value_t<InputIterator>,
               Compare,
               Allocator>;

template <typename InputIterator,
          typename Allocator>
concurrent_set( InputIterator, InputIterator, Allocator )
-> concurrent_set<iterator_value_t<InputIterator>,
               std::less<iterator_key_t<InputIterator>>,
               Allocator>;

template <typename T,
          typename Compare = std::less<T>,
          typename Allocator = tbb_allocator<T>>
concurrent_set( std::initializer_list<T>, Compare = Compare(), Allocator = Allocator() )
-> concurrent_set<T, Compare, Allocator>;

template <typename T,
          typename Allocator>
concurrent_set( std::initializer_list<T>, Allocator )
-> concurrent_set<T, std::less<Key>, Allocator>;
```

Where the type alias `iterator_value_t` defines as follows:

```
template <typename InputIterator>
using iterator_value_t = typename std::iterator_traits<InputIterator>::value_type;
```

#### Example

```
#include <tbb/concurrent_set.h>
#include <vector>

int main() {
    std::vector<int> v;

    // Deduces cs1 as concurrent_set<int>
    tbb::concurrent_set cs1(v.begin(), v.end());

    // Deduces cs2 as concurrent_set<int>
    tbb::concurrent_set cs2({1, 2, 3});
}
```

#### 11.2.5.4.4 concurrent\_multiset

##### [containers.concurrent\_multiset]

`tbb::concurrent_multiset` is a class template represents an unordered sequence of unique elements which supports concurrent insertion, lookup and traversal, but not concurrent erasure. The container allows to store multiple equivalent elements.

##### 11.2.5.4.4.1 Class Template Synopsis

```
// Defined in header <tbb/concurrent_set.h>

namespace tbb {

    template <typename T,
              typename Compare = std::less<T>,
              typename Allocator = tbb_allocator<T>>
    class concurrent_multiset {
public:
    using key_type = T;
    using value_type = T;

    using size_type = <implementation-defined unsigned integer type>;
    using difference_type = <implementation-defined signed integer type>;

    using key_compare = Compare;
    using value_compare = Compare;

    using allocator_type = Allocator;

    using reference = value_type&;
    using const_reference = const value_type&;
    using pointer = std::allocator_traits<Allocator>::pointer;
    using const_pointer = std::allocator_traits<Allocator>::const_pointer;

    using iterator = <implementation-defined ForwardIterator>;
    using const_iterator = <implementation-defined constant ForwardIterator>;

    using node_type = <implementation-defined node handle>;

    using range_type = <implementation-defined range>;
}
```

(continues on next page)

(continued from previous page)

```

using const_range_type = <implementation-defined constant node handle>;

// Construction, destruction, copying
concurrent_multiset();
explicit concurrent_multiset( const key_compare& comp,
                               const allocator_type& alloc = allocator_type() );
→;

explicit concurrent_multiset( const allocator_type& alloc );

template <typename InputIterator>
concurrent_multiset( InputIterator first, InputIterator last,
                     const key_compare& comp = key_compare(),
                     const allocator_type& alloc = allocator_type() );

template <typename InputIterator>
concurrent_multiset( InputIterator first, InputIterator last,
                     const allocator_type& alloc );

concurrent_multiset( std::initializer_list<value_type> init,
                     const key_compare& comp = key_compare(),
                     const allocator_type& alloc = allocator_type() );

concurrent_multiset( std::initializer_list<value_type> init, const allocator_
→type& alloc );

concurrent_multiset( const concurrent_multiset& other );
concurrent_multiset( const concurrent_multiset& other,
                     const allocator_type& alloc );

concurrent_multiset( concurrent_multiset&& other );
concurrent_multiset( concurrent_multiset&& other,
                     const allocator_type& alloc );

~concurrent_multiset();

concurrent_multiset& operator=( const concurrent_multiset& other );
concurrent_multiset& operator=( concurrent_multiset&& other );
concurrent_multiset& operator=( std::initializer_list<value_type> init );

allocator_type get_allocator() const;

// Iterators
iterator begin();
const_iterator begin() const;
const_iterator cbegin() const;

iterator end();
const_iterator end() const;
const_iterator cend() const;

// Size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// Concurrently safe modifiers

```

(continues on next page)

(continued from previous page)

```

std::pair<iterator, bool> insert( const value_type& value );

iterator insert( const_iterator hint, const value_type& value );

std::pair<iterator, bool> insert( value_type&& value );

iterator insert( const_iterator hint, value_type&& value );

template <typename InputIterator>
void insert( InputIterator first, InputIterator last );

void insert( std::initializer_list<value_type> init );

std::pair<iterator, bool> insert( node_type&& nh );
iterator insert( const_iterator hint, node_type&& nh );

template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args );

template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );

template <typename SrcCompare>
void merge( concurrent_set<T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_set<T, SrcCompare, Allocator>&& source );

template <typename SrcCompare>
void merge( concurrent_multiset<T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_multiset<T, SrcCompare, Allocator>&& source );

// Concurrently unsafe modifiers
void clear();

iterator unsafe_erase( const_iterator pos );
iterator unsafe_erase( iterator pos );

iterator unsafe_erase( const_iterator first, const_iterator last );

size_type unsafe_erase( const key_type& key );

template <typename K>
size_type unsafe_erase( const K& key );

node_type unsafe_extract( const_iterator pos );
node_type unsafe_extract( iterator pos );

node_type unsafe_extract( const key_type& key );

template <typename K>
node_type unsafe_extract( const K& key );

void swap( concurrent_multiset& other );

```

(continues on next page)

(continued from previous page)

```

// Lookup
size_type count( const key_type& key );

template <typename K>
size_type count( const K& key );

iterator find( const key_type& key );
const_iterator find( const key_type& key ) const;

template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;

bool contains( const key_type& key ) const;

template <typename K>
bool contains( const K& key ) const;

std::pair<iterator, iterator> equal_range( const key_type& key );
std::pair<const_iterator, const_iterator> equal_range( const key_type& key ) ↵
const;

template <typename K>
std::pair<iterator, iterator> equal_range( const K& key );
std::pair<const_iterator, const_iterator> equal_range( const K& key ) const;

iterator lower_bound( const key_type& key );
const_iterator lower_bound( const key_type& key ) const;

template <typename K>
iterator lower_bound( const K& key );

template <typename K>
const_iterator lower_bound( const K& key ) const;

iterator upper_bound( const key_type& key );
const_iterator upper_bound( const key_type& key ) const;

template <typename K>
iterator upper_bound( const K& key );

template <typename K>
const_iterator upper_bound( const K& key ) const;

// Observers
key_compare key_comp() const;

value_compare value_comp() const;

// Parallel iteration
range_type range();
const_range_type range() const;
}; // class concurrent_multiset

} // namespace tbb

```

Requirements:

- The expression `std::allocator_traits<Allocator>::destroy(m, val)` where `m` is an object of the type `Allocator` and `val` is an object of the type `value_type` should be well-formed. Member functions can impose stricter requirements depending on the type of the operation.
- The type `Compare` shall meet the `Compare` requirements from [alg.sorting] ISO C++ Standard section.
- The type `Allocator` shall meet the `Allocator` requirements from [allocator.requirements] ISO C++ Standard section.

#### 11.2.5.4.4.2 Member functions

#### 11.2.5.4.4.3 Construction, destruction, copying

#### 11.2.5.4.4.4 Empty container constructors

```
concurrent_multiset();

explicit concurrent_multiset( const key_compare& comp,
                               const allocator_type& alloc = allocator_type() );
explicit concurrent_multiset( const allocator_type& alloc );
```

Constructs empty `concurrent_multiset`.

If provided uses the comparison function object `comp` for all `key_type` comparisons and the allocator `alloc` to allocate the memory.

#### 11.2.5.4.4.5 Constructors from the sequence of elements

```
template <typename InputIterator>
concurrent_multiset( InputIterator first, InputIterator last,
                     const key_compare& comp = key_compare(),
                     const allocator_type& alloc = allocator_type() );

template <typename InputIterator>
concurrent_multiset( InputIterator first, InputIterator last,
                     const allocator_type& alloc = allocator_type() );
```

Constructs the `concurrent_multiset` which contains all elements from the half-open interval `[first, last]`.

If provided uses the comparison function object `comp` for all `key_type` comparisons and the allocator `alloc` to allocate the memory.

**Requirements:** the type `InputIterator` shall meet the requirements of `InputIterator` from [input iterators] ISO C++ Standard section.

```
concurrent_multiset( std::initializer_list<value_type> init, const key_
                     compare& comp = key_compare(),
                     const allocator_type& alloc = allocator_type() );
```

Equivalent to `concurrent_multiset(init.begin(), init.end(), comp, alloc)`.

```
concurrent_multiset( std::initializer_list<value_type> init,
                      const allocator_type& alloc );
```

Equivalent to `concurrent_multiset(init.begin(), init.end(), alloc)`.

#### 11.2.5.4.4.6 Copying constructors

```
concurrent_multiset( const concurrent_multiset& other );
concurrent_multiset( const concurrent_multiset& other, const allocator_type&  
                     ↵alloc );
```

Constructs a copy of `other`.

If the allocator argument is not provided, it is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

#### 11.2.5.4.4.7 Moving constructors

```
concurrent_multiset( concurrent_multiset&& other );
concurrent_multiset( concurrent_multiset&& other, const allocator_type&  
                     ↵alloc );
```

Constructs a `concurrent_multiset` with the contents of `other` using move semantics.

`other` is left in a valid, but unspecified state.

If the allocator argument is not provided, it is obtained by calling `std::move(other.get_allocator())`.

The behavior is undefined in case of concurrent operations with `other`.

#### 11.2.5.4.4.8 Destructor

```
~concurrent_multiset();
```

Destroys the `concurrent_multiset`. Calls destructors of the stored elements and deallocates the used storage.

The behavior is undefined in case of concurrent operations with `*this`.

#### 11.2.5.4.4.9 Assignment operators

```
concurrent_multiset& operator=( const concurrent_multiset& other );
```

Replaces all elements in `*this` by the copies of the elements in `other`.

Copy assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_copy_assignment` is true.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

**Returns:** a reference to `*this`.

```
concurrent_multiset& operator=( concurrent_multiset&& other );
```

Replaces all elements in `*this` by the elements in `other` using move semantics.

`other` is left in a valid, but unspecified state.

Move assigns allocators if `std::allocator_traits<allocator_type>::propagate_on_container_move_assignment` is true.

The behavior is undefined in case of concurrent operations with `*this` and `other`.

**Returns:** a reference to `*this`.

```
concurrent_multiset& operator=( std::initializer_list<value_type> init );
```

Replaces all elements in `*this` by the elements in `init`.

If `init` contains multiple elements with equal keys, it is unspecified which element would be inserted.

The behavior is undefined in case of concurrent operations with `*this`.

**Returns:** a reference to `*this`.

#### 11.2.5.4.4.10 Iterators

The types `concurrent_multiset::iterator` and `concurrent_multiset::const_iterator` meets the requirements of `ForwardIterator` from [forward.iterators] ISO C++ standard section.

#### 11.2.5.4.4.11 begin and cbegin

```
iterator begin();
const_iterator begin() const;
const_iterator cbegin() const;
```

**Returns:** an iterator to the first element in the container.

#### 11.2.5.4.4.12 end and cend

```
iterator end();
const_iterator end() const;
const_iterator cend() const;
```

**Returns:** an iterator to the element which follows the last element in the container.

#### 11.2.5.4.4.13 Size and capacity

#### 11.2.5.4.4.14 empty

```
bool empty() const;
```

**Returns:** true if the container is empty, false otherwise.

The result may differ with the actual container state in case of pending concurrent insertions.

#### 11.2.5.4.4.15 size

```
size_type size() const;
```

**Returns:** the number of elements in the container.

The result may differ with the actual container size in case of pending concurrent insertions.

#### 11.2.5.4.4.16 max\_size

```
size_type max_size() const;
```

**Returns:** the maximum number of elements that container can hold.

#### 11.2.5.4.4.17 Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other, lookup methods and while traversing the container.

#### 11.2.5.4.4.18 Inserting values

```
std::pair<iterator, bool> insert( const value_type& value );
```

Inserts the value value into the container.

**Returns:** std::pair<iterator, bool> where iterator points to the inserted element. Boolean value is always true.

**Requirements:** the type value\_type shall meet the CopyInsertable requirements from [container.requirements] ISO C++ Standard section.

---

```
iterator insert( const_iterator hint, const value_type& other );
```

Inserts the value `value` into the container.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

**Returns:** an `iterator` to the inserted element.

**Requirements:** the type `value_type` shall meet the `CopyInsertable` requirements from [container.requirements] ISO C++ Standard section.

---

```
std::pair<iterator, bool> insert( value_type&& value );
```

Inserts the value `value` into the container using move semantics.

`value` is left in a valid, but unspecified state.

**Returns:** `std::pair<iterator, bool>` where `iterator` points to the inserted element. Boolean value is always `true`.

**Requirements:** the type `value_type` shall meet the `MoveInsertable` requirements from [container.requirements] ISO C++ Standard section.

---

```
iterator insert( const_iterator hint, value_type&& other );
```

Inserts the value `value` into the container using move semantics.

Optionally uses the parameter `hint` as a suggestion to where the element should be placed.

`value` is left in a valid, but unspecified state.

**Returns:** an `iterator` to the inserted element.

**Requirements:** the type `value_type` shall meet the `MoveInsertable` requirements from [container.requirements] ISO C++ Standard section.

#### 11.2.5.4.4.19 Inserting sequences of elements

```
template <typename InputIterator>
void insert( InputIterator first, InputIterator last );
```

Inserts all items from the half-open interval `[first, last)` into the container.

**Requirements:** the type `InputIterator` must meet the requirements of `InputIterator` from [input.iterators] ISO C++ Standard section.

---

```
void insert( std::initializer_list<value_type> init );
```

Equivalent to `insert(init.begin(), init.end())`.

#### 11.2.5.4.4.20 Inserting nodes

```
std::pair<iterator, bool> insert( node_type&& nh );
```

If the node handle nh is empty, does nothing.

Otherwise - inserts the node, owned by nh into the container.

nh is left in an empty state.

No copy or move constructors of value\_type are performed.

The behavior is undefined if nh is not empty and get\_allocator() != nh.get\_allocator().

**Returns:** std::pair<iterator, bool> where iterator points to the inserted element. Boolean value is always true.

```
iterator insert( const_iterator hint, node_type&& nh );
```

If the node handle nh is empty, does nothing.

Otherwise - inserts the node, owned by nh into the container.

Optionally uses the parameter hint as a suggestion to where the node should be placed.

nh is left in an empty state.

No copy or move constructors of value\_type are performed.

The behavior is undefined if nh is not empty and get\_allocator() != nh.get\_allocator().

**Returns:** an iterator pointing to the inserted element.

#### 11.2.5.4.4.21 Emplacing elements

```
template <typename... Args>
std::pair<iterator, bool> emplace( Args&&... args );
```

Inserts an element, constructed in-place from args into the container.

**Returns:** std::pair<iterator, bool> where iterator points to the inserted element. Boolean value is always true.

**Requirements:** the type value\_type shall meet the `EmplaceConstructible` requirements from [container.requirements] ISO C++ section.

```
template <typename... Args>
iterator emplace_hint( const_iterator hint, Args&&... args );
```

Inserts an element, constructed in-place from args into the container.

Optionally uses the parameter hint as a suggestion to where the node should be placed.

**Returns:** an iterator to the inserted element.

**Requirements:** the type value\_type shall meet the `EmplaceConstructible` requirements from [container.requirements] ISO C++ section.

#### 11.2.5.4.4.22 Merging containers

```
template <typename SrcCompare>
void merge( concurrent_set<T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_set<T, SrcCompare, Allocator>&& source );

template <typename SrcCompare>
void merge( concurrent_multiset<T, SrcCompare, Allocator>& source );

template <typename SrcCompare>
void merge( concurrent_multiset<T, SrcCompare, Allocator>&& source );
```

Transfers all elements from `source` to `*this`.

No copy or move constructors of `value_type` are performed.

The behavior is undefined if `get_allocator() != source.get_allocator()`.

#### 11.2.5.4.4.23 Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these member functions with other (either concurrently safe) methods.

#### 11.2.5.4.4.24 Clearing

```
void clear();
```

Removes all elements from the container.

#### 11.2.5.4.4.25 Erasing elements

```
iterator unsafe_erase( const_iterator pos );
iterator unsafe_erase( iterator pos );
```

Removes the element pointed to by `pos` from the container.

Invalidates all iterators and references to the removed element.

**Returns:** iterator which follows the removed element.

**Requirements:** the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
size_type unsafe_erase( const key_type& key );
```

Removes all elements equal to `key` if they exists in the container.

Invalidates all iterators and references to the removed element.

**Returns:** the number of removed elements.

```
template <typename K>
size_type unsafe_erase( const K& key );
```

Removes all elements which compares equivalent to `key` if they exists in the container.

Invalidates all iterators and references to the removed element.

This overload only participates in overload resolution if all of the following statements are `true`:

- The qualified-id `key_compare::is_transparent` is valid and denotes a type.
- `std::is_convertible<K, iterator>::value` is `false`.
- `std::is_convertible<K, const_iterator>::value` is `false`.

**Returns:** the number of removed elements.

#### 11.2.5.4.4.26 Erasing sequences

```
iterator unsafe_erase( const_iterator first, const_iterator last );
```

Removes all elements from the half-open interval `[first, last)` from the container.

**Returns:** iterator which follows the last removed element.

**Requirements:** the range `[first, last)` must be a valid subrange in `*this`.

#### 11.2.5.4.4.27 Extracting nodes

```
node_type unsafe_extract( iterator pos );
node_type unsafe_extract( const_iterator pos );
```

Transfers ownership of the element pointed to by `pos` from the container to the node handle.

No copy or move constructors of `value_type` are performed.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

**Returns:** the node handle that owns the extracted element.

**Requirements:** the iterator `pos` should be valid, dereferenceable and point to the element in `*this`.

```
node_type unsafe_extract( const key_type& key );
```

If an element equal to `key` exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of `value_type` are performed.

If there are multiple elements equal to `key` exists, it is unspecified which element should be transferred.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

**Returns:** the node handle that owns the extracted element or an empty node handle if an element equal to key was not found.

```
template <typename K>
node_type unsafe_extract( const K& key );
```

If an element which compares equivalent to key exists, transfers ownership of this element from the container to the node handle.

No copy or move constructors of value\_type are performed.

If there are multiple elements which compares equivalent with key exists, it is unspecified which element should be transferred.

Invalidates all iterators to the extracted element. Pointers and references to the extracted element remain valid.

This overload only participates in overload resolution if all of the following statements are true:

- The qualified-id key\_compare::is\_transparent is valid and denotes a type.
- std::is\_convertible<K, iterator>::value is false.
- std::is\_convertible<K, const\_iterator>::value is false.

**Returns:** the node handle that owns the extracted element or an empty node handle if an element which compares equivalent to key was not found.

#### 11.2.5.4.4.28 swap

```
void swap( concurrent_multiset& other );
```

Swaps contents of \*this and other.

Swaps allocators if std::allocator\_traits<allocator\_type>::propagate\_on\_container\_swap::value is true.

Otherwise if get\_allocator() != other.get\_allocator() the behavior is undefined.

#### 11.2.5.4.4.29 Lookup

All methods in this section can be executed concurrently with each other, concurrently-safe modifiers and while traversing the container.

#### 11.2.5.4.4.30 count

```
size_type count( const key_type& key );
```

**Returns:** the number of elements equal to key.

```
template <typename K>
size_type count( const K& key );
```

**Returns:** the number of elements which compares equivalent with `key`.

This overload only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

#### 11.2.5.4.4.31 `find`

```
iterator find( const key_type& key );
const_iterator find( const key_type& key ) const;
```

**Returns:** an iterator to the element equal to `key` or `end()` if no such element exists.

If there are multiple elements equal to `key` exists, it is unspecified which element should be found.

```
template <typename K>
iterator find( const K& key );

template <typename K>
const_iterator find( const K& key ) const;
```

**Returns:** an iterator to the element which compares equivalent with `key` or `end()` if no such element exists.

If there are multiple elements which compares equivalent with `key` exists, it is unspecified which element should be found.

These overloads only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

#### 11.2.5.4.4.32 `contains`

```
bool contains( const key_type& key ) const;
```

**Returns:** `true` if at least one element equal to `key` exists in the container, `false` otherwise.

```
template <typename K>
bool contains( const K& key ) const;
```

**Returns:** `true` if at least one element which compares equivalent with `key` exists in the container, `false` otherwise.

This overload only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

#### 11.2.5.4.4.33 lower\_bound

```
iterator lower_bound( const key_type& key );
const_iterator lower_bound( const key_type& key ) const;
```

**Returns:** an iterator to the first element in the container which is *not less* than key.

```
template <typename K>
iterator lower_bound( const K& key )

template <typename K>
const_iterator lower_bound( const K& key ) const
```

**Returns:** an iterator to the first element in the container which compares *not less* with key.

These overloads only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

#### 11.2.5.4.4.34 upper\_bound

```
iterator upper_bound( const key_type& key );
const_iterator upper_bound( const key_type& key ) const;
```

**Returns:** an iterator to the first element in the container which is *greater* than key.

```
template <typename K>
iterator upper_bound( const K& key );

template <typename K>
const_iterator upper_bound( const K& key ) const;
```

**Returns:** an iterator to the first element in the container which compares greater with key.

These overloads only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

#### 11.2.5.4.4.35 equal\_range

```
std::pair<iterator, iterator> equal_range( const key_type& key );
std::pair<const_iterator, const_iterator> equal_range( const key_type& key ) const;
```

**Returns:** if at least one element equal to key exists - a pair of iterators {f, l}, where f is an iterator to the first element equal to key, l is an iterator to the element which follows the last element equal to key. Otherwise - {end(), end() }.

```
template <typename K>
std::pair<iterator, iterator> equal_range( const K& key )

template <typename K>
std::pair<const_iterator, const_iterator> equal_range( const K& key )
```

**Returns:** if at least one element which compares equivalent with key exists - a pair of iterators {f, l}, where f is an iterator to the first element which compares equivalent with key, l is an iterator to the element which follows the last element which compares equivalent with key. Otherwise - {end(), end()}.

These overloads only participates in overload resolution if qualified-id `key_compare::is_transparent` is valid and denotes a type.

#### 11.2.5.4.4.36 Observers

##### 11.2.5.4.4.37 get\_allocator

```
allocator_type get_allocator() const;
```

**Returns:** a copy of the allocator associated with `*this`.

##### 11.2.5.4.4.38 key\_comp

```
key_compare key_comp() const;
```

**Returns:** a copy of the key comparison functor associated with `*this`.

##### 11.2.5.4.4.39 value\_comp

```
value_compare value_comp() const;
```

**Returns:** an object of the `value_compare` class which is used to compare `value_type` objects.

#### 11.2.5.4.4.40 Parallel iteration

Member types `concurrent_multiset::range_type` and `concurrent_multiset::const_range_type` meets the *ContainerRange requirements*.

These types differ only in that the bounds for a `concurrent_multiset::const_range_type` are of type `concurrent_multiset::const_iterator`, whereas the bounds for a `concurrent_multiset::range_type` are of type `concurrent_multiset::iterator`.

#### 11.2.5.4.4.41 range member function

```
range_type range();
const_range_type range() const;
```

**Returns:** a range object representing all elements in the container.

#### 11.2.5.4.4.42 Non-member functions

These functions provides binary and lexicographical comparison and swap operations on `tbb::concurrent_multiset` objects.

The exact namespace where these functions are defined is unspecified, as long as they may be used in respective comparison operations. For example, an implementation may define the classes and functions in the same internal namespace and define `tbb::concurrent_multiset` as a type alias for which the non-member functions are reachable only via argument dependent lookup.

```
template <typename T, typename Compare, typename Allocator>
void swap( concurrent_multiset<T, Compare, Allocator>& lhs,
           concurrent_multiset<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_multiset<T, Compare, Allocator>& lhs,
                   const concurrent_multiset<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_multiset<T, Compare, Allocator>& lhs,
                   const concurrent_multiset<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator<( const concurrent_multiset<T, Compare, Allocator>& lhs,
                   const concurrent_multiset<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator>( const concurrent_multiset<T, Compare, Allocator>& lhs,
                   const concurrent_multiset<T, Compare, Allocator>& rhs );

template <typename T, typename Compare, typename Allocator>
bool operator<=( const concurrent_multiset<T, Compare, Allocator>& lhs,
                   const concurrent_multiset<T, Compare, Allocator>& rhs );

template <typename Key, typename T, typename Compare, typename Allocator>
bool operator>=( const concurrent_multiset<T, Compare, Allocator>& lhs,
                   const concurrent_multiset<T, Compare, Allocator>& rhs );
```

#### 11.2.5.4.4.43 Non-member swap

```
template <typename T, typename Compare, typename Allocator>
void swap( concurrent_multiset<T, Compare, Allocator>& lhs,
           concurrent_multiset<T, Compare, Allocator>& rhs );
```

Equivalent to `lhs.swap(rhs)`.

#### 11.2.5.4.4.44 Non-member binary comparisons

Two `tbb::concurrent_multiset` objects are equal if they have the same number of elements and each element in one container is equal to the element in other container on the same position.

```
template <typename T, typename Compare, typename Allocator>
bool operator==( const concurrent_multiset<T, Compare, Allocator>& lhs,
                   const concurrent_multiset<T, Compare, Allocator>& rhs )
```

**Returns:** true if `lhs` is equal to `rhs`, false otherwise.

```
template <typename T, typename Compare, typename Allocator>
bool operator!=( const concurrent_multiset<T, Compare, Allocator>& lhs,
                   const concurrent_multiset<T, Compare, Allocator>& rhs )
```

**Returns:** true if `lhs` is not equal to `rhs`, false otherwise.

#### 11.2.5.4.4.45 Non-member lexicographical comparisons

```
template <typename T, typename Compare, typename Allocator>
bool operator<( const concurrent_multiset<T, Compare, Allocator>& lhs,
                   const concurrent_multiset<T, Compare, Allocator>& rhs )
```

**Returns:** true if `lhs` is lexicographically *less* than `rhs`.

```
template <typename T, typename Compare, typename Allocator>
bool operator<=( const concurrent_multiset<T, Compare, Allocator>& lhs,
                   const concurrent_multiset<T, Compare, Allocator>& rhs )
```

**Returns:** true if `lhs` is lexicographically *less or equal* than `rhs`.

```
template <typename T, typename Compare, typename Allocator>
bool operator>( const concurrent_multiset<T, Compare, Allocator>& lhs,
                   const concurrent_multiset<T, Compare, Allocator>& rhs )
```

**Returns:** true if `lhs` is lexicographically *greater* than `rhs`.

```
template <typename T, typename Compare, typename Allocator>
bool operator>=( const concurrent_multiset<T, Compare, Allocator>& lhs,
                   const concurrent_multiset<T, Compare, Allocator>& rhs )
```

**Returns:** true if `lhs` is lexicographically *greater or equal* than `rhs`.

#### 11.2.5.4.4.46 Other

#### 11.2.5.4.4.47 Deduction guides

Where possible, constructors of concurrent\_multiset supports class template argument deduction (since C++17):

```
template <typename InputIterator,
          typename Compare = std::less<iterator_value_t<InputIterator>>,
          typename Allocator = tbb_allocator<iterator_value_t<InputIterator>>>
concurrent_multiset( InputIterator, InputIterator, Compare = Compare(), Allocator = Allocator() )
-> concurrent_multiset<iterator_value_t<InputIterator>,
                        Compare,
                        Allocator>;

template <typename InputIterator,
          typename Allocator>
concurrent_multiset( InputIterator, InputIterator, Allocator )
-> concurrent_multiset<iterator_value_t<InputIterator>,
                        std::less<iterator_key_t<InputIterator>>,
                        Allocator>;

template <typename T,
          typename Compare = std::less<T>,
          typename Allocator = tbb_allocator<T>>
concurrent_multiset( std::initializer_list<T>, Compare = Compare(), Allocator = Allocator() )
-> concurrent_multiset<T, Compare, Allocator>;

template <typename T,
          typename Allocator>
concurrent_multiset( std::initializer_list<T>, Allocator )
-> concurrent_multiset<T, std::less<Key>, Allocator>;
```

Where the type alias iterator\_value\_t defines as follows:

```
template <typename InputIterator>
using iterator_value_t = typename std::iterator_traits<InputIterator>::value_type;
```

#### Example

```
#include <tbb/concurrent_set.h>
#include <vector>

int main() {
    std::vector<int> v;

    // Deduces cs1 as concurrent_multiset<int>
    tbb::concurrent_multiset cs1(v.begin(), v.end());

    // Deduces cs2 as concurrent_multiset<int>
    tbb::concurrent_multiset cs2({1, 2, 3});
}
```

### 11.2.5.5 Auxiliary classes

#### 11.2.5.5.1 tbb\_hash\_compare

##### [containers.tbb\_hash\_compare]

`tbb::tbb_hash_compare` is a class template for hash support. It is used with the `tbb::concurrent_hash_map` associative container to calculate hash codes and compare keys for equality.

`tbb_hash_compare` meets the *HashCompare requirements*.

##### 11.2.5.5.1.1 Class Template Synopsis

```
// Defined in header <tbb/concurrent_hash_map.h>

namespace tbb {

    template <typename Key>
    class tbb_hash_compare {
        static std::size_t hash( const Key& k );
        static bool equal( const Key& k1, const Key& k2 );
    }; // class tbb_hash_compare

} // namespace tbb
```

##### 11.2.5.5.1.2 Member functions

```
static std::size_t hash( const Key& k );
```

**Returns:** a hash code for a key `k`.

```
static bool equal( const Key& k1, const Key& k2 );
```

Equivalent to `k1 == k2`.

**Returns:** `true` if the keys are equal, `false` otherwise.

#### 11.2.5.5.2 Node handles

##### [containers.node\_handles]

Concurrent associative containers in oneAPI Threading Building Blocks (`concurrent_map`, `concurrent_multimap`, `concurrent_set`, `concurrent_multiset`, `concurrent_unordered_map`, `concurrent_unordered_multimap`, `concurrent_unordered_set`, and `concurrent_unordered_multiset`) stores elements in individually allocated, connected nodes. It makes possible to transfer data between containers with compatible node types by changing the connections, without copying or moving the actual data.

### 11.2.5.5.2.1 Class synopsis

```

class node_handle { // Exposition-only name
public:
    using key_type = <container-specific>; // Only for maps
    using mapped_type = <container-specific>; // Only for maps
    using value_type = <container-specific>; // Only for sets
    using allocator_type = <container-specific>;

    node_handle();
    node_handle( node_handle&& other );

    ~node_handle();

    node_handle& operator= ( node_handle&& other );

    void swap( node_handle& nh );

    bool empty() const;
    explicit operator bool() const;

    key_type& key() const; // Only for maps
    mapped_type& mapped() const; // Only for maps
    value_type& value() const; // Only for sets

    allocator_type get_allocator() const;
};

```

A node handle is a container-specific move-only nested type (exposed as *container::node\_type*) that represents a node outside of any container instance. It allows reading and modifying the data stored in the node, and inserting the node into a compatible container instance. The following containers have compatible node types and may exchange nodes:

- concurrent\_map and concurrent\_multimap with the same key\_type, mapped\_type and allocator\_type.
- concurrent\_set and concurrent\_multiset with the same value\_type and allocator\_type.
- concurrent\_unordered\_map and concurrent\_unordered\_multimap with the same key\_type, mapped\_type and allocator\_type.
- concurrent\_unordered\_set and concurrent\_unordered\_multiset with the same value\_type and allocator\_type.

Default or moved-from node handles are *empty*, i.e. do not represent a valid node. A non-empty node handle is typically created when a node is extracted out of a container, e.g. with the *unsafe\_extract* method. It stores the node along with a copy of the container's allocator. Upon assignment or destruction a non-empty node handle destroys the stored data and deallocates the node.

### 11.2.5.5.2.2 Member functions

#### 11.2.5.5.2.3 Constructors

```
node-handle();
```

Constructs an empty node handle.

```
node-handle( node-handle&& other );
```

Constructs a node handle that takes ownership of the node from `other`.

`other` is left in an empty state.

#### 11.2.5.5.2.4 Assignment

```
node-handle& operator=( node-handle&& other );
```

Transfers ownership of the node from `other` to `*this`. If `*this` was not empty before transferring, destroys and deallocates the stored node.

Move assignes the stored allocator if `std::allocator_traits<allocator_type>::propagate_on_container_is_true` is true.

`other` is left in an empty state.

#### 11.2.5.5.2.5 Destructor

```
~node-handle();
```

Destroys the node handle. If it is not empty, destroys and deallocates the owned node.

#### 11.2.5.5.2.6 Swap

```
void swap( node-handle& other )
```

Exchanges the nodes owned by `*this` and `other`.

Swaps the stored allocators if `std::allocator_traits<allocator_type>::propagate_on_container_swap_is_true` is true.

### 11.2.5.5.2.7 State

```
bool empty() const;
```

**Returns:** `true` if the node handle is empty, `false` otherwise.

```
explicit operator bool() const;
```

Equivalent to `!empty()`.

### 11.2.5.5.2.8 Access to the stored element

```
key_type& key() const;
```

Available only for map node handles.

**Returns:** a reference to the key of the element stored in the owned node.

The behavior is undefined if the node handle is empty.

```
mapped_type& mapped() const;
```

Available only for map node handles.

**Returns:** a reference to the value of the element stored in the owned node.

The behavior is undefined if the node handle is empty.

```
value_type& value() const;
```

Available only for set node handles.

**Returns:** a reference to the element stored in the owned node.

The behavior is undefined if the node handle is empty.

### 11.2.5.5.2.9 get\_allocator

```
allocator_type get_allocator() const;
```

**Returns:** a copy of the allocator stored in the node handle.

The behavior is undefined if the node handle is empty.

## 11.2.6 Thread Local Storage

### [thread\_local\_storage]

oneAPI Threading Building Blocks provides class templates for thread local storage. Each provides a thread-local element per thread and lazily creates elements on demand.

#### 11.2.6.1 combinable

##### [tls.combinable]

A class template for holding thread-local values during a parallel computation that will be merged into a final value.

A combinable provides each thread with its own instance of type T.

```
// Defined in header <tbb/combinable.h>

namespace tbb {
    template <typename T>
    class combinable {
        public:
            combinable();
            combinable(const combinable& other);
            combinable(combinable&& other);

            template <typename FInit>
            explicit combinable(FInit init);

            ~combinable();

            combinable& operator=( const combinable& other);
            combinable& operator=( combinable&& other);

            void clear();

            T& local();
            T& local(bool & exists);

            template<typename BinaryFunc> T combine(BinaryFunc f);
            template<typename UnaryFunc> void combine_each(UnaryFunc f);
    };
}
```

#### 11.2.6.1.1 Member functions

##### combinable()

Constructs combinable such that thread-local instances of T will be default-constructed.

##### template<typename FInit>

##### explicit combinable(FInit init)

Constructs combinable such that thread-local elements will be created by copying the result of *init()*.

**Caution:** The expression *init()* must be safe to evaluate concurrently by multiple threads. It is evaluated each time a new thread-local element is created.

**combinable (const combinable &other)**

Constructs a copy of *other*, so that it has copies of each element in *other* with the same thread mapping.

**combinable (combinable &&other)**

Constructs *combinable* by moving the content of *other* intact. *other* is left in an unspecified state but can be safely destroyed.

**~combinable ()**

Destroys all elements in *\*this*.

**combinable &operator= (const combinable &other)**

Sets *\*this* to be a copy of *other*. Returns a reference to *\*this*.

**combinable &operator= (combinable &&other)**

Moves the content of *other* to *\*this* intact. *other* is left in an unspecified state but can be safely destroyed.

Returns a reference to *\*this*.

**void clear ()**

Removes all elements from *\*this*.

**T &local ()**

If an element does not exist for the current thread, creates it.

**Returns:** Reference to thread-local element.

**T &local (bool &exists)**

Similar to *local ()*, except that *exists* is set to true if an element was already present for the current thread; false otherwise.

**Returns:** Reference to thread-local element.

**template<typename BinaryFunc>****T combine (BinaryFunc f)**

**Requires:** A *BinaryFunc* shall meet the *Function Objects* requirements from [function.objects] ISO C++ Standard section. Specifically, the type should be an associative binary functor with the signature *T BinaryFunc (T, T)* or *T BinaryFunc (const T&, const T&)*. A *T* type must be the same as a corresponding template parameter for *combinable* object.

**Effects:** Computes a reduction over all elements using binary functor *f*. All evaluations of *f* are done sequentially in the calling thread. If there are no elements, creates the result using the same rules as for creating a new element.

**Returns:** Result of the reduction.

**template<typename UnaryFunc>****void combine\_each (UnaryFunc f)**

**Requires:** An *UnaryFunc* shall meet the *Function Objects* requirements from [function.objects] ISO C++ Standard section. Specifically, the type should be an unary functor with the signature *void UnaryFunc (T)* or *void UnaryFunc (T&)* or *void UnaryFunc (const T&)*. A *T* type must be the same as a corresponding template parameter for *enumerable\_thread\_specific* object.

**Effects:** Evaluates *f(x)* for each thread-local element *x* in *\*this*. All evaluations are done sequentially in the calling thread.

---

**Note:** Methods of class *combinable* are not thread-safe, except for *local*.

---

### 11.2.6.2 enumerable\_thread\_specific

#### [tls.enumerable\_thread\_specific]

A class template for thread local storage.

```
// Defined in header <tbb/enumerable_thread_specific.h>

namespace tbb {

    enum ets_key_usage_type {
        ets_key_per_instance,
        ets_no_key,
        ets_suspend_aware
    };

    template <typename T,
              typename Allocator=cache_aligned_allocator<T>,
              ets_key_usage_type ETS_key_type=ets_no_key >
    class enumerable_thread_specific {
public:
    // Basic types
    using value_type = T;
    using reference = T&;
    using const_reference = const T&;
    using pointer = T*;
    using size_type = /* implementation-defined */;
    using difference_type = /* implementation-defined */;
    using allocator_type = Allocator;

    // Iterator types
    using iterator = /* implementation-defined */;
    using const_iterator = /* implementation-defined */;

    // Parallel range types
    using range_type = /* implementation-defined */;
    using const_range_type = /* implementation-defined */;

    // Construction
    enumerable_thread_specific();
    template <typename Finit>
    explicit enumerable_thread_specific( Finit finit );
    explicit enumerable_thread_specific( const T& exemplar );
    explicit enumerable_thread_specific( T&& exemplar );
    template <typename... Args>
    enumerable_thread_specific( Args&&... args );

    // Destruction
    ~enumerable_thread_specific();

    // Copy constructors
    enumerable_thread_specific( const enumerable_thread_specific& other );
    template<typename Alloc, ets_key_usage_type Cachetype>
    enumerable_thread_specific( const enumerable_thread_specific<T, Alloc,_
    ↪Cachetype>& other );
    // Copy assignments
    enumerable_thread_specific& operator=( const enumerable_thread_specific&_
    ↪other );

```

(continues on next page)

(continued from previous page)

```

template<typename Alloc, ets_key_usage_type Cachetype>
enumerable_thread_specific& operator=(const enumerable_thread_specific<T,_
→Alloc, Cachetype>& other);

// Move constructors
enumerable_thread_specific( enumerable_thread_specific&& other);
template<typename Alloc, ets_key_usage_type Cachetype>
enumerable_thread_specific( enumerable_thread_specific<T, Alloc, Cachetype>&&_
→other);
// Move assignments
enumerable_thread_specific& operator=(enumerable_thread_specific&& other );
template<typename Alloc, ets_key_usage_type Cachetype>
enumerable_thread_specific& operator=(enumerable_thread_specific<T, Alloc,_
→Cachetype>&& other );

// Other whole container operations
void clear();

// Concurrent operations
reference local();
reference local(bool& exists );
size_type size() const;
bool empty() const;

// Combining
template<typename BinaryFunc> T combine( BinaryFunc f );
template<typename UnaryFunc> void combine_each( UnaryFunc f );

// Parallel iteration
range_type range(size_t grainsize=1 );
const_range_type range(size_t grainsize=1 ) const;

// Iterators
iterator begin();
iterator end();
const_iterator begin() const;
const_iterator end() const;
};

} // namespace tbb

```

A class template `enumerable_thread_specific` provides thread local storage (TLS) for elements of type `T`. A class template `enumerable_thread_specific` acts as a container by providing iterators and ranges across all of the thread-local elements.

The thread-local elements are created lazily. A freshly constructed `enumerable_thread_specific` has no elements. When a thread requests access to an `enumerable_thread_specific`, it creates an element corresponding to that thread. The number of elements is equal to the number of distinct threads that have accessed the `enumerable_thread_specific` and not necessarily the number of threads in use by the application. Clearing an `enumerable_thread_specific` removes all its elements.

The ETS\_key\_usage\_type parameter type can be used to select an underlying implementation.

**Caution:** `enumerable_thread_specific` uses the OS-specific value returned by `std::this_thread::get_id()` to identify threads. This value is not guaranteed to be unique except for the life of the thread. A newly created thread may get an OS-specific ID equal to that of an already destroyed

thread. The number of elements of the `enumerable_thread_specific` may therefore be less than the number of actual distinct threads that have called `local()`, and the element returned by the first reference by a thread to the `enumerable_thread_specific` may not be newly-constructed.

### 11.2.6.2.1 Member functions

#### 11.2.6.2.1.1 Construction, destruction, copying

#### 11.2.6.2.1.2 Empty container constructors

```
enumerable_thread_specific();
```

Constructs an `enumerable_thread_specific` where each thread-local element will be default-constructed.

```
template<typename Finit> explicit enumerable_thread_specific( Finit finit );
```

Constructs an `enumerable_thread_specific` such that any thread-local element will be created by copying the result of `finit()`.

---

**Note:** The expression `finit()` must be safe to evaluate concurrently by multiple threads. It is evaluated each time a thread-local element is created.

---

```
explicit enumerable_thread_specific( const T& exemplar );
```

Constructs an `enumerable_thread_specific` where each thread-local element will be copy-constructed from `exemplar`.

```
explicit enumerable_thread_specific( T&& exemplar );
```

Constructs an `enumerable_thread_specific` object, move constructor of `T` can be used to store `exemplar` internally, however thread-local elements are always copy-constructed.

```
template <typename... Args> enumerable_thread_specific( Args&&... args );
```

Constructs `enumerable_thread_specific` such that any thread-local element will be constructed by invoking `T(args...)`.

---

**Note:** This constructor does not participate in overload resolution if the type of the first argument in `args...` is `T`, or `enumerable_thread_specific<T>`, or `foo()` is a valid expression for a value `foo` of that type.

---

### 11.2.6.2.1.3 Copying constructors

```
enumerable_thread_specific ( const enumerable_thread_specific& other );  
  

template<typename Alloc, ets_key_usage_type Cachetype> enumerable_thread_specific ( _  

_>const enumerable_thread_specific <T, Alloc, Cachetype>& other );
```

Constructs an `enumerable_thread_specific` as a copy of `other`. The values are copy-constructed from the values in `other` and have same thread correspondence.

### 11.2.6.2.1.4 Moving constructors

```
enumerable_thread_specific ( enumerable_thread_specific&& other )
```

Constructs an `enumerable_thread_specific` by moving the content of `other` intact. `other` is left in an unspecified state, but can be safely destroyed.

```
template<typename Alloc, ets_key_usage_type Cachetype> enumerable_thread_specific ( _  

_>enumerable_thread_specific <T, Alloc, Cachetype>&& other )
```

Constructs an `enumerable_thread_specific` using per-element move construction from the values in `other`, and keeping their thread correspondence. `other` is left in an unspecified state, but can be safely destroyed.

### 11.2.6.2.1.5 Destructor

```
~enumerable_thread_specific()
```

Destroys all elements in `*this`. Destroys any native TLS keys that were created for this instance.

### 11.2.6.2.1.6 Assignment operators

```
enumerable_thread_specific& operator=( const enumerable_thread_specific& other );
```

Copies the content of `other` to `*this`. Returns a reference to `this*`.

```
template<typename Alloc, ets_key_usage_type Cachetype>  
enumerable_thread_specific& operator=( const enumerable_thread_specific<T, Alloc, _  

_>Cachetype>& other );
```

Copies the content of `other` to `*this`. Returns a reference to `this*`.

---

**Note:** The allocator and key usage specialization is unchanged by this call.

---

```
enumerable_thread_specific& operator=( enumerable_thread_specific&& other );
```

Moves the content of `other` to `*this` intact. An `other` is left in an unspecified state, but can be safely destroyed. Returns a reference to `this*`.

```
template<typename Alloc, ets_key_usage_type Cachetype>
enumerable_thread_specific<operator=( enumerable_thread_specific<T, Alloc, Cachetype>
~&& other );
```

Moves the content of `other` to `*this` using per-element move construction and keeping thread correspondence. An `other` is left in an unspecified state, but can be safely destroyed. Returns a reference to `this*`.

---

**Note:** The allocator and key usage specialization is unchanged by this call.

---

#### 11.2.6.2.1.7 Concurrently safe modifiers

All member functions in this section can be performed concurrently with each other.

reference **local ()**

If there is no current element corresponding to the current thread, then this method constructs a new element. A new element is copy-constructed if an exemplar was provided to the constructor for `*this`, otherwise a new element is default constructed.

**Returns:** A reference to the element of `*this` that corresponds to the current thread.

reference **local (bool &exists)**

Similar to `local ()`, except that `exists` is set to true if an element was already present for the current thread; false otherwise.

**Returns:** Reference to thread-local element.

#### 11.2.6.2.1.8 Concurrently unsafe modifiers

All member functions in this section can only be performed serially. The behavior is undefined in case of concurrent execution of these methods with other (either concurrently safe) methods.

#### 11.2.6.2.1.9 clear

```
void clear();
```

Destroys all elements in `*this`.

#### 11.2.6.2.1.10 Size and capacity

**size\_type size () const**

Returns the number of elements in `*this`. The value is equal to the number of distinct threads that have called `local ()` after `*this` was constructed or most recently cleared.

**bool empty () const**

Returns `true` if the container is empty, `false` otherwise.

### 11.2.6.2.1.11 Iteration

Class template `enumerable_thread_specific` supports random access iterators, which enable iteration over the set of all elements in the container.

`iterator begin()`

Returns iterator pointing to the beginning of the set of elements.

`iterator end()`

Returns iterator pointing to the end of the set of elements.

`const_iterator begin() const`

Returns const\_iterator pointing to the beginning of the set of elements.

`const_iterator end() const`

Returns const\_iterator pointing to the end of the set of elements.

Class template `enumerable_thread_specific` supports `const_range_type` and `range_type` types, which model the *ContainerRange requirement*. The types differ only in that the bounds for a `const_range_type` are of type `const_iterator`, whereas the bounds for a `range_type` are of type `iterator`.

`const_range_type range(size_t grainsize = 1) const`

**Returns:** A `const_range_type` representing all elements in `*this`. The parameter `grainsize` is in units of elements.

`range_type range(size_t grainsize = 1)`

**Returns:** A `range_type` representing all elements in `*this`. The parameter `grainsize` is in units of elements.

### 11.2.6.2.1.12 Combining

The member functions in this section iterate across the entire container sequentially in the calling thread.

`template<typename BinaryFunc>`

`T combine(BinaryFunc f)`

**Requires:** A `BinaryFunc` shall meet the *Function Objects* requirements from [function.objects] ISO C++ Standard section. Specifically, the type should be an associative binary functor with the signature `T BinaryFunc(T, T)` or `T BinaryFunc(const T&, const T&)`. A `T` type must be the same as a corresponding template parameter for `enumerable_thread_specific` object.

**Effects:** Computes reduction over all elements using binary functor `f`. If there are no elements, creates the result using the same rules as for creating a thread-local element.

**Returns:** Result of the reduction.

`template<typename UnaryFunc>`

`void combine_each(UnaryFunc f)`

**Requires:** An `UnaryFunc` shall meet the *Function Objects* requirements from [function.objects] ISO C++ Standard section. Specifically, the type should be an unary functor with the signature `void UnaryFunc(T)` or `void UnaryFunc(T&)` or `void UnaryFunc(const T&)`. A `T` type must be the same as a corresponding template parameter for `enumerable_thread_specific` object.

**Effects:** Evaluates `f(x)` for each instance `x` of `T` in `*this`.

### 11.2.6.2.2 Non-member types and constants

**enum ets\_key\_usage\_type::ets\_key\_per\_instance**

Enumeration parameter type used to select an implementation that consumes 1 native TLS key per enumerable\_thread\_specific instance. The number of native TLS keys may be limited and can be fairly small.

**enum ets\_key\_usage\_type::ets\_no\_key**

Enumeration parameter type used to select an implementation that consumes no native TLS keys. If no ets\_key\_usage\_type parameter type is provided, ets\_no\_key is used by default.

**enum ets\_key\_usage\_type::ets\_suspend\_aware**

tbb::task::suspend function can change the value of enumerable\_thread\_specific object. In order to avoid this problem, ets\_suspend\_aware enumeration parameter type should be used. The local() value can be the same for different threads, but no two distinct threads can access the same value simultaneously.

This section also describes class template flattened2d, which assists a common idiom here an enumerable\_thread\_specific represents a container partitioner across threads.

### 11.2.6.3 flattened2d

[**tls.flattened2d**]

The class template flattened2d is an adaptor that provides a flattened view of a container of containers.

```
// Defined in header <tbb/enumerable_thread_specific.h>

namespace tbb {

    template<typename Container>
    class flattened2d {
    public:
        // Basic types
        using size_type = /* implementation-defined */;
        using difference_type = /* implementation-defined */;
        using allocator_type = /* implementation-defined */;
        using value_type = /* implementation-defined */;
        using reference = /* implementation-defined */;
        using const_reference = /* implementation-defined */;
        using pointer = /* implementation-defined */;
        using const_pointer = /* implementation-defined */;

        using iterator = /* implementation-defined */;
        using const_iterator = /* implementation-defined */;

        explicit flattened2d( const Container& c );

        flattened2d( const Container& c,
                    typename Container::const_iterator first,
                    typename Container::const_iterator last );

        iterator begin();
        iterator end();
        const_iterator begin() const;
        const_iterator end() const;
    };
}
```

(continues on next page)

(continued from previous page)

```

        size_type size() const;
};

template <typename Container>
flattened2d<Container> flatten2d(const Container &c);

template <typename Container>
flattened2d<Container> flatten2d(
    const Container &c,
    const typename Container::const_iterator first,
    const typename Container::const_iterator last);

} // namespace tbb

```

#### Requirements:

- A Container type shall meet the container requirements from [container.requirements.general] ISO C++ section.

Iterating from `begin()` to `end()` visits all of the elements in the inner containers. The class template supports forward iterators only.

The utility function `flatten2d` creates a `flattened2d` object from a specified container.

#### 11.2.6.3.1 Member functions

##### `explicit flattened2d(const Container &c)`

Constructs a `flattened2d` representing the sequence of elements in the inner containers contained by outer container `c`.

**Safety:** these operations must not be invoked concurrently on the same `flattened2d`.

##### `flattened2d(const Container &c, typename Container::const_iterator first, typename Container::const_iterator last)`

Constructs a `flattened2d` representing the sequence of elements in the inner containers in the half-open interval `[first, last)` of a container `c`.

**Safety:** these operations must not be invoked concurrently on the same `flattened2d`.

##### `size_type size() const`

Returns the sum of the sizes of the inner containers that are viewable in the `flattened2d`.

**Safety:** These operations may be invoked concurrently on the same `flattened2d`.

##### `iterator begin()`

Returns iterator pointing to the beginning of the set of local copies.

##### `iterator end()`

Returns iterator pointing to the end of the set of local copies.

##### `const_iterator begin() const`

Returns `const_iterator` pointing to the beginning of the set of local copies.

##### `const_iterator end() const`

Returns `const_iterator` pointing to the end of the set of local copies.

### 11.2.6.3.2 Non-member functions

```
template<typename Container>
flattened2d<Container> flatten2d(const Container &c, const typename Container::const_iterator b,
                                    const typename Container::const_iterator e)
Constructs and returns a flattened2d object that provides iterators that traverse the elements in the containers within the half-open range [b, e) of a container c.
```

```
template<typename Container>
flattened2d(const Container &c)
Constructs and returns a flattened2d that provides iterators that traverse the elements in all of the containers within a container c.
```

## 11.3 oneTBB Auxiliary Interfaces

### 11.3.1 Memory Allocation

#### [memory\_allocation]

This section describes classes and functions related to memory allocation.

#### 11.3.1.1 Allocators

oneAPI Threading Building Blocks implements several classes that meet the allocator requirements from the [allocator.requirements] ISO C++ Standard section.

##### 11.3.1.1.1 tbb\_allocator

#### [memory\_allocation.tbb\_allocator]

A `tbb_allocator` is a class template that models the allocator requirements from the [allocator.requirements] ISO C++ section.

The `tbb_allocator` allocates and frees memory via the oneAPI Threading Building Blocks malloc library if it is available, otherwise it reverts to using `std::malloc` and `std::free`.

```
// Defined in header <tbb/tbb_allocator.h>

namespace tbb {
    template<typename T> class tbb_allocator {
    public:
        using value_type = T;
        using size_type = std::size_t;
        using propagate_on_container_move_assignment = std::true_type;
        using is_always_equal = std::true_type;

        enum malloc_type {
            scalable,
            standard
        };

        tbb_allocator() = default;
        template<typename U>
```

(continues on next page)

(continued from previous page)

```
tbb_allocator(const tbb_allocator<U>&) noexcept;

T* allocate(size_type);
void deallocate(T*, size_type);

static malloc_type allocator_type();
};

}
```

### 11.3.1.1.1.1 Member Functions

**T \*allocate (size\_type n)**

Allocates  $n * \text{sizeof}(T)$  bytes. Returns a pointer to the allocated memory.

**void deallocate (T \*p, size\_type n)**

Deallocates memory pointed to by p. The behavior is undefined if the pointer p is not the result of the allocate(n) method. The behavior is undefined if the memory has been already deallocated.

**static malloc\_type allocator\_type ()**

Returns the enumeration type malloc\_type::scalable if the oneAPI TBB malloc library is available and malloc\_type::standard otherwise.

### 11.3.1.1.1.2 Non-member Functions

These functions provide comparison operations between two tbb\_allocator instances.

```
template<typename T, typename U>
bool operator==(const tbb_allocator<T>&, const tbb_allocator<U>&) noexcept;

template<typename T, typename U>
bool operator!=(const tbb_allocator<T>&, const tbb_allocator<U>&) noexcept;
```

The namespace where these functions are defined is unspecified, as long as they may be used in respective binary operation expressions on tbb\_allocator objects. For example, an implementation may define the classes and functions in the same unspecified internal namespace, and define tbb::tbb\_allocator as a type alias for which the non-member functions are reachable only via argument dependent lookup.

**template<typename T, typename U>**

**bool operator==(const tbb\_allocator<T>&, const tbb\_allocator<U>&) noexcept**

Returns true.

**template<typename T, typename U>**

**bool operator!=(const tbb\_allocator<T>&, const tbb\_allocator<U>&) noexcept**

Returns false.

### 11.3.1.1.2 scalable\_allocator

#### [memory\_allocation.scalable\_allocator]

A `scalable_allocator` is a class template that models the allocator requirements from the [allocator.requirements] ISO C++ section.

The `scalable_allocator` allocates and frees memory in a way that scales with the number of processors. Memory allocated by a `scalable_allocator` should be freed by a `scalable_allocator`, not by a `std::allocator`.

```
// Defined in header <tbb/scalable_allocator.h>

namespace tbb {
    template<typename T> class scalable_allocator {
    public:
        using value_type = T;
        using size_type = std::size_t;
        using propagate_on_container_move_assignment = std::true_type;
        using is_always_equal = std::true_type;

        scalable_allocator() = default;
        template<typename U>
        scalable_allocator(const scalable_allocator<U>&) noexcept;

        T* allocate(size_type);
        void deallocate(T*, size_type);
    };
}
```

**Caution:** The `scalable_allocator` requires the memory allocator library. If the library is missing, calls to the scalable allocator fail. In contrast, if the memory allocator library is not available, `tbb_allocator` falls back on `std::malloc` and `std::free`.

#### 11.3.1.1.2.1 Member Functions

`value_type *allocate (size_type n)`

Allocates `n * sizeof(T)` bytes of memory. Returns a pointer to the allocated memory.

`void deallocate (value_type *p, size_type n)`

Deallocates memory pointed to by `p`. The behavior is undefined if the pointer `p` is not the result of the `allocate(n)` method. The behavior is undefined if the memory has been already deallocated.

#### 11.3.1.1.2.2 Non-member Functions

These functions provide comparison operations between two `scalable_allocator` instances.

```
namespace tbb {
    template<typename T, typename U>
    bool operator==(const scalable_allocator<T>&, const scalable_allocator<U>&) noexcept;

    template<typename T, typename U>
```

(continues on next page)

(continued from previous page)

```

bool operator!=(const scalable_allocator<T>&,
                  const scalable_allocator<U>&) noexcept;
}

```

The namespace where these functions are defined is unspecified, as long as they may be used in respective binary operation expressions on `scalable_allocator` objects. For example, an implementation may define the classes and functions in the same unspecified internal namespace, and define `tbb::scalable_allocator` as a type alias for which the non-member functions are reachable only via argument dependent lookup.

`template<typename T, typename U>`  
`bool operator==(const scalable_allocator<T>&, const scalable_allocator<U>&) noexcept`  
 Returns **true**.

`template<typename T, typename U>`  
`bool operator!=(const scalable_allocator<T>&, const scalable_allocator<U>&) noexcept`  
 Returns **false**.

### 11.3.1.1.3 `cache_aligned_allocator`

#### [`memory_allocation.cache_aligned_allocator`]

A `cache_aligned_allocator` is a class template that models the allocator requirements from the [allocator.requirements] ISO C++ section.

The `cache_aligned_allocator` allocates memory on cache line boundaries, in order to avoid false sharing and potentially improve performance. False sharing is a situation when logically distinct items occupy the same cache line, which can hurt performance if multiple threads attempt to access the different items simultaneously. Even though the items are logically separate, the processor hardware may have to transfer the cache line between the processors as if they were sharing a location. The net result can be much more memory traffic than if the logically distinct items were on different cache lines.

However, this class is sometimes an inappropriate replacement for default allocator, because the benefit of allocating on a cache line comes at the price that `cache_aligned_allocator` implicitly adds pad memory. Hence allocating many small objects with `cache_aligned_allocator` may increase memory usage.

```

// Defined in header <tbb/cache_aligned_allocator.h>

namespace tbb {
    template<typename T> class cache_aligned_allocator {
        public:
            using value_type = T;
            using size_type = std::size_t;
            using propagate_on_container_move_assignment = std::true_type;
            using is_always_equal = std::true_type;

            cache_aligned_allocator() = default;
            template<typename U>
            cache_aligned_allocator(const cache_aligned_allocator<U>&) noexcept;

            T* allocate(size_type);
            void deallocate(T*, size_type);
            size_type max_size() const noexcept;
    };
}

```

### 11.3.1.1.3.1 Member Functions

`T *allocate (size_type n)`

Returns a pointer to the allocated `n * sizeof (T)` bytes of memory, aligned on a cache-line boundary. The allocation may include extra hidden padding.

`void deallocate (T *p, size_type n)`

Deallocates memory pointed to by `p`. The deallocation also deallocates any extra hidden padding. The behavior is undefined if the pointer `p` is not the result of the `allocate (n)` method. The behavior is undefined if the memory has been already deallocated.

`size_type max_size () const noexcept`

Returns the largest value `n` for which the call `allocate (n)` might succeed with cache alignment constraints.

### 11.3.1.1.3.2 Non-member Functions

These functions provide comparison operations between two `cache_aligned_allocator` instances.

```
template<typename T, typename U>
bool operator==(const cache_aligned_allocator<T>&, const cache_aligned_allocator<U>&) noexcept;

template<typename T, typename U>
bool operator!=(const cache_aligned_allocator<T>&, const cache_aligned_allocator<U>&) noexcept;
```

The namespace where these functions are defined is unspecified, as long as they may be used in respective binary operation expressions on `cache_aligned_allocator` objects. For example, an implementation may define the classes and functions in the same unspecified internal namespace, and define `tbb::cache_aligned_allocator` as a type alias for which the non-member functions are reachable only via argument dependent lookup.

`template<typename T, typename U>
bool operator==(const cache_aligned_allocator<T>&, const cache_aligned_allocator<U>&) noexcept`

Returns true.

`template<typename T, typename U>
bool operator!=(const cache_aligned_allocator<T>&, const cache_aligned_allocator<U>&) noexcept`

Returns false.

### 11.3.1.2 Memory Resources

Starting from C++17, the standard library provides a `std::pmr::polymorphic_allocator` class that allocates memory from a supplied memory resource (see the [mem.poly\_allocator.class] ISO/IEC 14882:2017 section). Class `std::pmr::memory_resource` is an abstract interface for user-side implementation of different allocation strategies. For details, see [mem.res.class] ISO/IEC 14882:2017 standard section.

oneAPI Threading Building Blocks provides a set of `std::pmr::memory_resource` implementations.

### 11.3.1.2.1 cache\_aligned\_resource

#### [memory\_allocation.cache\_aligned\_resource]

A `cache_aligned_resource` is a general-purpose memory resource class, which acts as a wrapper to another memory resource to ensure that all allocations are aligned on cache line boundaries to avoid false sharing.

See the [cache\\_aligned\\_allocator template class](#) section for more information about false sharing avoidance.

```
// Defined in header <tbb/cache_aligned_allocator.h>

namespace tbb {
    class cache_aligned_resource {
public:
    cache_aligned_resource();
    explicit cache_aligned_resource( std::pmr::memory_resource* );

    std::pmr::memory_resource* upstream_resource() const;

private:
    void* do_allocate(size_t n, size_t alignment) override;
    void do_deallocate(void* p, size_t n, size_t alignment) override;
    bool do_is_equal(const std::pmr::memory_resource& other) const noexcept
override;
    };
}
```

#### 11.3.1.2.1.1 Member Functions

##### `cache_aligned_resource()`

Constructs a `cache_aligned_resource` over `std::pmr::get_default_resource()`.

##### `explicit cache_aligned_resource(std::pmr::memory_resource *r)`

Constructs a `cache_aligned_resource` over the memory resource `r`.

##### `std::pmr::memory_resource *upstream_resource() const`

Returns the pointer to the underlying memory resource.

##### `void *do_allocate(size_t n, size_t alignment) override`

Allocates `n` bytes of memory on a cache-line boundary, with alignment not less than requested. The allocation may include extra memory for padding. Returns pointer to the allocated memory.

##### `void do_deallocate(void *p, size_t n, size_t alignment) override`

Deallocates memory pointed to by `p` and any extra padding. Pointer `p` must be obtained with `do_allocate(n, alignment)`. The memory must not be deallocated beforehand.

##### `bool do_is_equal(const std::pmr::memory_resource &other) const noexcept override`

Compares upstream memory resources of `*this` and `other`. If `other` is not a `cache_aligned_resource`, returns false.

### 11.3.1.2.2 scalable\_memory\_resource

#### [memory\_allocation.scalable\_memory\_resource]

A `tbb::scalable_memory_resource()` is a function that returns a memory resource for scalable memory allocation.

The `scalable_memory_resource()` function returns the pointer to the memory resource managed by the TBB scalable memory allocator. In particular, its `allocate` method uses `scalable_aligned_malloc()`, and `deallocate` uses `scalable_free()`. The memory resources returned by this function compare equal.

`std::pmr::polymorphic_allocator` instantiated with `tbb::scalable_memory_resource()` behaves like `tbb::scalable_allocator`.

```
// Defined in header <tbb/scalable_allocator.h>

std::pmr::memory_resource* scalable_memory_resource();
```

### 11.3.1.3 Library Functions

#### 11.3.1.3.1 C Interface to Scalable Allocator

#### [memory\_allocation.scalable\_alloc\_c\_interface]

Low level interface for scalable memory allocation.

```
// Defined in header <tbb/scalable_allocator.h>

extern "C" {
    // Scalable analogs of C memory allocator
    void* scalable_malloc( size_t size );
    void  scalable_free( void* ptr );
    void* scalable_calloc( size_t nobj, size_t size );
    void* scalable_realloc( void* ptr, size_t size );

    // Analog of _msize/malloc_size/malloc_usable_size.
    size_t scalable_msize( void* ptr );

    // Scalable analog of posix_memalign
    int  scalable_posix_memalign( void** memptr, size_t alignment, size_t size );

    // Aligned allocation
    void* scalable_aligned_malloc( size_t size, size_t alignment);
    void  scalable_aligned_free( void* ptr );
    void* scalable_aligned_realloc( void* ptr, size_t size, size_t alignment );

    // Return values for scalable_allocation_* functions
    typedef enum {
        TBBMALLOC_OK,
        TBBMALLOC_INVALID_PARAM,
        TBBMALLOC_UNSUPPORTED,
        TBBMALLOC_NO_MEMORY,
        TBBMALLOC_NO_EFFECT
    } ScalableAllocationResult;

    typedef enum {
```

(continues on next page)

(continued from previous page)

```

// To turn on/off the use of huge memory pages
TBBMALLOC_USE_HUGE_PAGES,
// To set a threshold for the allocator memory usage.
// Exceeding it will forcefully clean internal memory buffers
TBBMALLOC_SET_SOFT_HEAP_LIMIT,
// Lower bound for the size (Bytes), that is interpreted as huge
// and not released during regular cleanup operations
TBBMALLOC_SET_HUGE_SIZE_THRESHOLD
} AllocationModeParam;

// Set allocator-specific allocation modes.
int scalable_allocation_mode(int param, intptr_t value);

typedef enum {
    // Clean internal allocator buffers for all threads.
    TBBMALLOC_CLEAN_ALL_BUFFERS,
    // Clean internal allocator buffer for current thread only.
    TBBMALLOC_CLEAN_THREAD_BUFFERS
} ScalableAllocationCmd;

// Call allocator-specific commands.
int scalable_allocation_command(int cmd, void *param);
}

```

These functions provide a C level interface to the scalable allocator. With the exception of `scalable_allocation_mode` and `scalable_allocation_command`, each routine `scalable_x` behaves analogously to library function `x`. The routines form the two families shown in the table below, “C Interface to Scalable Allocator”. Storage allocated by a `scalable_x` function in one family must be freed or resized by a `scalable_x` function in the same family, not by a C standard library function. Likewise storage allocated by a C standard library function should not be freed or resized by a `scalable_x` function.

Table 5: C Interface to Scalable Allocator

Allocation Routine	Deallocation Routine	Analogous Library
<code>scalable_malloc</code>	<code>scalable_free</code>	C standard library
<code>scalable_calloc</code>		
<code>scalable_realloc</code>		
<code>scalable_posix_memalign</code>		POSIX*
<code>scalable_aligned_malloc</code>	<code>scalable_aligned_free</code>	Microsoft* C run-time library
<code>scalable_aligned_realloc</code>		

The following functions do not allocate or free memory but allow to obtain useful information or to influence behavior of the memory allocator.

`size_t scalable_mszie(void *ptr)`

**Returns:** The usable size of the memory block pointed to by `ptr` if it was allocated by the scalable allocator. Returns zero if `ptr` does not point to such a block.

`int scalable_allocation_mode(int mode, intptr_t value)`

This function may be used to adjust behavior of the scalable memory allocator.

**Returns:** `TBBMALLOC_OK` if the operation succeeded, `TBBMALLOC_INVALID_PARAM` if `mode` is not one of the described below, or if `value` is not valid for the given mode. Other return values are possible, as described below.

#### scalable\_allocation\_mode Parameters: Parameter, Description

**TBBMALLOC\_USE\_HUGE\_PAGES**

`scalable_allocation_mode(TBBMALLOC_USE_HUGE_PAGES, 1)` tells the allocator to use huge pages if enabled by the operating system. `scalable_allocation_mode(TBBMALLOC_USE_HUGE_PAGES, 0)` disables it. Setting `TBB_MALLOC_USE_HUGE_PAGES` environment variable to 1 has the same effect as `scalable_allocation_mode(TBBMALLOC_USE_HUGE_PAGES, 1)`. The mode set with `scalable_allocation_mode()` takes priority over the environment variable.

**May return:** `TBBMALLOC_NO_EFFECT` if huge pages are not supported on the platform.

For now, this allocation mode is only supported for Linux\* OS. It works with both explicitly configured and transparent huge pages. For information about enabling and configuring huge pages, refer to OS documentation or ask your system administrator.

**TBBMALLOC\_SET\_SOFT\_HEAP\_LIMIT**

`scalable_allocation_mode(TBBMALLOC_SET_SOFT_HEAP_LIMIT, size)` sets a threshold of `size` bytes on the amount of memory the allocator takes from OS. Exceeding the threshold will urge the allocator to release memory from its internal buffers; however it does not prevent from requesting more memory if needed.

**TBBMALLOC\_SET\_HUGE\_SIZE\_THRESHOLD**

`scalable_allocation_mode(TBBMALLOC_SET_HUGE_SIZE_THRESHOLD, size)` sets a lower bound threshold (with no upper limit) of `size` bytes. Any object that is bigger than this threshold becomes huge and doesn't participate in internal periodic cleanup logic. However, it doesn't affect the logic of `TBBMALLOC_SET_SOFT_HEAP_LIMIT` mode as well as `TBBMALLOC_CLEAN_ALL_BUFFERS` operation.

Setting `TBB_MALLOC_SET_HUGE_SIZE_THRESHOLD` environment variable to the `size` value has the same effect, but is limited to the `LONG_MAX` value. The mode set with `scalable_allocation_mode` takes priority over the environment variable.

**int scalable\_allocation\_command(int cmd, void \*reserved)**

This function may be used to command the scalable memory allocator to perform an action specified by the first parameter. The second parameter is reserved and must be set to 0.

**Returns:** `TBBMALLOC_OK` if the operation succeeded, `TBBMALLOC_INVALID_PARAM` if `cmd` is not one of the described below, or if `reserved` is not equal to 0.

**scalable\_allocation\_command Parameters: Parameter, Description****TBBMALLOC\_CLEAN\_ALL\_BUFFERS**

`scalable_allocation_command(TBBMALLOC_CLEAN_ALL_BUFFERS, 0)` cleans internal memory buffers of the allocator, and possibly reduces memory footprint. It may result in increased time for subsequent memory allocation requests. The command is not designed for frequent use, and careful evaluation of the performance impact is recommended.

**May return:** `TBBMALLOC_NO_EFFECT` if no buffers were released.

**Note:** It is not guaranteed that the call will release all unused memory.

**TBBMALLOC\_CLEAN\_THREAD\_BUFFERS**

`scalable_allocation_command(TBBMALLOC_CLEAN_THREAD_BUFFERS, 0)` cleans internal memory buffers but only for the calling thread.

**May return:** `TBBMALLOC_NO_EFFECT` if no buffers were released.

## 11.3.2 Mutual Exclusion

### [mutex]

The library provides a set of mutual exclusion primitives to simplify writing race-free code. A mutex object facilitates protection against data races and allows safe synchronization of data between threads.

#### 11.3.2.1 Mutex Classes

##### 11.3.2.1.1 spin\_mutex

###### [mutex.spin\_mutex]

A `spin_mutex` is a class that models the *Mutex requirement* using a spin lock. The `spin_mutex` class satisfies all requirements of mutex type from the [thread.mutex.requirements] ISO C++ section. The `spin_mutex` class is not fair or recursive.

```
// Defined in header <tbb/spin_mutex.h>

namespace tbb {
    class spin_mutex {
    public:
        spin_mutex() noexcept;
        ~spin_mutex();

        spin_mutex(const spin_mutex&) = delete;
        spin_mutex& operator=(const spin_mutex&) = delete;

        class scoped_lock;

        void lock();
        bool try_lock();
        void unlock();

        static constexpr bool is_rw_mutex = false;
        static constexpr bool is_recursive_mutex = false;
        static constexpr bool is_fair_mutex = false;
    };
}
```

### 11.3.2.1.1.1 Member classes

#### `class scoped_lock`

Corresponding `scoped_lock` class. See the *Mutex requirement*.

### 11.3.2.1.1.2 Member functions

#### `spin_mutex()`

Constructs `spin_mutex` with unlocked state.

#### `~spin_mutex()`

Destroys an unlocked `spin_mutex`.

#### `void lock()`

Acquires a lock. Spins if the lock is taken.

#### `bool try_lock()`

Attempts to acquire a lock (non-blocking). Returns `true` if lock is acquired; `false` otherwise.

#### `void unlock()`

Releases a lock, held by a current thread.

### 11.3.2.1.2 spin\_rw\_mutex

#### [mutex.spin\_rw\_mutex]

A `spin_rw_mutex` is a class that models the *ReaderWriterMutex requirement* and satisfies all requirements of shared mutex type from the [thread.sharedmutex.requirements] ISO C++ section.

The `spin_rw_mutex` class is unfair spinning reader-writer lock with backoff and writer-preference.

```
// Defined in header <tbb/spin_rw_mutex.h>

namespace tbb {
    class spin_rw_mutex {
        public:
            spin_rw_mutex() noexcept;
            ~spin_rw_mutex();

            spin_rw_mutex(const spin_rw_mutex&) = delete;
            spin_rw_mutex& operator=(const spin_rw_mutex&) = delete;

            class scoped_lock;

            // exclusive ownership
            void lock();
            bool try_lock();
            void unlock();

            // shared ownership
            void lock_shared();
            bool try_lock_shared();
            void unlock_shared();

            static constexpr bool is_rw_mutex = true;
            static constexpr bool is_recursive_mutex = false;
    };
}
```

(continues on next page)

(continued from previous page)

```

    static constexpr bool is_fair_mutex = false;
};

}

```

### 11.3.2.1.2.1 Member classes

#### `class scoped_lock`

Corresponding scoped-lock class. See the *ReaderWriterMutex requirement*.

### 11.3.2.1.2.2 Member functions

#### `spin_rw_mutex()`

Constructs unlocked `spin_rw_mutex`.

#### `~spin_rw_mutex()`

Destroys unlocked `spin_rw_mutex`.

#### `void lock()`

Acquires a lock. Spins if the lock is taken.

#### `bool try_lock()`

Attempts to acquire a lock (non-blocking) on write. Returns true if the lock is acquired on write; false otherwise.

#### `void unlock()`

Releases a write lock, held by the current thread.

#### `void lock_shared()`

Acquires a lock on read. Spins if the lock is taken on write already.

#### `bool try_lock_shared()`

Attempts to acquire the lock (non-blocking) on read. Returns true if the lock is acquired on read; false otherwise.

#### `void unlock_shared()`

Releases a read lock, held by the current thread.

### 11.3.2.1.3 speculative\_spin\_mutex

#### [mutex.speculative\_spin\_mutex]

A `speculative_spin_mutex` is a class that models the *Mutex requirement* using a spin lock, and for processors which support hardware transactional memory (such as Intel® Transactional Synchronization Extensions (Intel® TSX)) may be implemented in a way that allows non-contending changes to the protected data to proceed in parallel.

The `speculative_spin_mutex` is not fair and not recursive. The `speculative_spin_mutex` is like a `spin_mutex`, but it may provide better throughput than non-speculative mutexes when the following conditions are met:

- Running on a processor that supports hardware transactional memory;
- multiple threads can concurrently execute the critical section(s) protected by the mutex, mostly without conflicting.

Otherwise it performs like a `spin_mutex`, possibly with worse throughput.

```
// Defined in header <tbb/spin_mutex.h>

namespace tbb {
    class speculative_spin_mutex {
public:
    speculative_spin_mutex() noexcept;
    ~speculative_spin_mutex();

    speculative_spin_mutex(const speculative_spin_mutex&) = delete;
    speculative_spin_mutex& operator=(const speculative_spin_mutex&) = delete;

    class scoped_lock;

    static constexpr bool is_rw_mutex = false;
    static constexpr bool is_recursive_mutex = false;
    static constexpr bool is_fair_mutex = false;
};

}
```

### 11.3.2.1.3.1 Member classes

#### `class scoped_lock`

Corresponding `scoped_lock` class. See the [Mutex requirement](#).

### 11.3.2.1.3.2 Member functions

#### `speculative_spin_mutex()`

Constructs `speculative_spin_mutex` with unlocked state.

#### `~speculative_spin_mutex()`

Destroys an unlocked `speculative_spin_mutex`.

### 11.3.2.1.4 `speculative_spin_rw_mutex`

#### [`mutex.speculative_spin_rw_mutex`]

A `speculative_spin_rw_mutex` is a class that models the [\*ReaderWriterMutex requirement\*](#), and for processors which support hardware transactional memory (such as Intel® Transactional Synchronization Extensions (Intel® TSX)) may be implemented in a way that allows non-contending changes to the protected data to proceed in parallel.

The `speculative_spin_rw_mutex` class is not fair and not recursive. The `speculative_spin_rw_mutex` class is like a `spin_rw_mutex`, but it may provide better throughput than non-speculative mutexes when the following conditions are met:

- Running on a processor that supports hardware transactional memory;
- multiple threads can concurrently execute the critical section(s) protected by the mutex, mostly without conflicting.

Otherwise it performs like a `spin_rw_mutex`, possibly with worse throughput.

For processors that support hardware transactional memory, `speculative_spin_rw_mutex` may be implemented in a way that

- speculative readers and writers do not block each other;

- a non-speculative reader blocks writers but allows speculative readers;
- a non-speculative writer blocks all readers and writers.

```
// Defined in header <tbb/spin_rw_mutex.h>

namespace tbb {
    class speculative_spin_rw_mutex {
public:
    speculative_spin_rw_mutex() noexcept;
    ~speculative_spin_rw_mutex();

    speculative_spin_rw_mutex(const speculative_spin_rw_mutex&) = delete;
    speculative_spin_rw_mutex& operator=(const speculative_spin_rw_mutex&) = delete;

    class scoped_lock;

    static constexpr bool is_rw_mutex = true;
    static constexpr bool is_recursive_mutex = false;
    static constexpr bool is_fair_mutex = false;
};

}
```

#### 11.3.2.1.4.1 Member classes

##### `class scoped_lock`

Corresponding `scoped_lock` class. See the *ReaderWriterMutex requirement*.

#### 11.3.2.1.4.2 Member functions

##### `speculative_spin_rw_mutex()`

Constructs `speculative_spin_rw_mutex` with unlocked state.

##### `~speculative_spin_rw_mutex()`

Destroys an unlocked `speculative_spin_rw_mutex`.

#### 11.3.2.1.5 queuing\_mutex

##### [mutex.queuing\_mutex]

A `queuing_mutex` is a class that models the *Mutex requirement*. The `queuing_mutex` is not recursive. The `queuing_mutex` is fair, threads acquire a lock on a mutex in the order that they request it.

```
// Defined in header <tbb/queuing_mutex.h>

namespace tbb {
    class queuing_mutex {
public:
    queuing_mutex() noexcept;
    ~queuing_mutex();

    queuing_mutex(const queuing_mutex&) = delete;
    queuing_mutex& operator=(const queuing_mutex&) = delete;
};
```

(continues on next page)

(continued from previous page)

```

    class scoped_lock;

    static constexpr bool is_rw_mutex = false;
    static constexpr bool is_recursive_mutex = false;
    static constexpr bool is_fair_mutex = true;
};

}

```

### 11.3.2.1.5.1 Member classes

#### `class scoped_lock`

Corresponding `scoped_lock` class. See the *Mutex requirement*.

### 11.3.2.1.5.2 Member functions

#### `queueing_mutex()`

Construct unlocked `queueing_mutex`.

#### `~queueing_mutex()`

Destroys unlocked `queueing_mutex`.

### 11.3.2.1.6 `queueing_rwlock`

#### [mutex.queueing\_rwlock]

A `queueing_rwlock` is a class that models the *ReaderWriterMutex requirement* concept. The `queueing_rwlock` is not recursive. The `queueing_rwlock` is fair, threads acquire a lock on a mutex in the order that they request it.

```

// Defined in header <tbb/queueing_rwlock.h>

namespace tbb {
    class queueing_rwlock {
public:
    queueing_rwlock() noexcept;
    ~queueing_rwlock();

    queueing_rwlock(const queueing_rwlock&) = delete;
    queueing_rwlock& operator=(const queueing_rwlock&) = delete;

    class scoped_lock;

    static constexpr bool is_rw_mutex = true;
    static constexpr bool is_recursive_mutex = false;
    static constexpr bool is_fair_mutex = true;
};
}

```

### 11.3.2.1.6.1 Member classes

#### `class scoped_lock`

Corresponding `scoped_lock` class. See the [ReaderWriterMutex requirement](#).

### 11.3.2.1.6.2 Member functions

#### `queueing_rw_mutex()`

Construct unlocked `queueing_rw_mutex`.

#### `~queueing_rw_mutex()`

Destroys unlocked `queueing_rw_mutex`.

### 11.3.2.1.7 null\_mutex

#### [mutex.null\_mutex]

A `null_mutex` is a class that models the [Mutex requirement](#) concept syntactically, but does nothing. It is useful for instantiating a template that expects a Mutex, but no mutual exclusion is actually needed for that instance.

```
// Defined in header <tbb/null_mutex.h>

namespace tbb {
    class null_mutex {
    public:
        constexpr null_mutex() noexcept;
        ~null_mutex();

        null_mutex(const null_mutex&) = delete;
        null_mutex& operator=(const null_mutex&) = delete;

        class scoped_lock;

        void lock();
        bool try_lock();
        void unlock();

        static constexpr bool is_rw_mutex = false;
        static constexpr bool is_recursive_mutex = true;
        static constexpr bool is_fair_mutex = true;
    };
}
```

### 11.3.2.1.7.1 Member classes

#### `class scoped_lock`

Corresponding `scoped_lock` class. See the [Mutex requirement](#).

### 11.3.2.1.7.2 Member functions

```
null_mutex()
    Construct unlocked mutex.

~null_mutex()
    Destroy unlocked mutex.

void lock()
    Acquire lock.

bool try_lock()
    Try acquiring lock (non-blocking)

void unlock()
    Release the lock.
```

### 11.3.2.1.8 null\_rw\_mutex

#### [mutex.null\_rw\_mutex]

A `null_rw_mutex` is a class that models the *ReaderWriterMutex requirement* syntactically, but does nothing. The `null_rw_mutex` class also satisfies all syntactic requirements of shared mutex type from the [thread.sharedmutex.requirements] ISO C++ section, but does nothing. It is useful for instantiating a template that expects a ReaderWriterMutex, but no mutual exclusion is actually needed for that instance.

```
// Defined in header <tbb/null_rw_mutex.h>

namespace tbb {
    class null_rw_mutex {
    public:
        constexpr null_rw_mutex() noexcept;
        ~null_rw_mutex();

        null_rw_mutex(const null_rw_mutex&) = delete;
        null_rw_mutex& operator=(const null_rw_mutex&) = delete;

        class scoped_lock;

        void lock();
        bool try_lock();
        void unlock();

        void lock_shared();
        bool try_lock_shared();
        void unlock_shared();

        static constexpr bool is_rw_mutex = true;
        static constexpr bool is_recursive_mutex = true;
        static constexpr bool is_fair_mutex = true;
    };
}
```

### 11.3.2.1.8.1 Member classes

#### `class scoped_lock`

Corresponding `scoped_lock` class. See the [ReaderWriterMutex requirement](#).

### 11.3.2.1.8.2 Member functions

#### `null_rw_mutex()`

Construct unlocked mutex.

#### `~null_rw_mutex()`

Destroy unlocked mutex.

#### `void lock()`

Acquires a lock.

#### `bool try_lock()`

Attempts to acquire a lock (non-blocking) on write. Returns `true`.

#### `void unlock()`

Releases a write lock, held by the current thread.

#### `void lock_shared()`

Acquires a lock on read.

#### `bool try_lock_shared()`

Attempts to acquire the lock (non-blocking) on read. Returns `true`.

#### `void unlock_shared()`

Releases a read lock, held by the current thread.

## 11.3.3 Timing

### [timing]

Parallel programming is about speeding up *wall clock* time, which is the real time that it takes a program or function to run. The library provides API to simplify timing within an application.

### 11.3.3.1 Syntax

```
// Declared in tick_count.h

class tick_count;

class tick_count::interval_t;
```

### 11.3.3.2 Classes

#### 11.3.3.2.1 tick\_count class

##### [timing.tick\_count]

A `tick_count` is an absolute wall clock timestamp. Two `tick_count` objects may be subtracted to compute wall clock duration `tick_count::interval_t`, which can be converted to seconds.

```
namespace tbb {

    class tick_count {
    public:
        class interval_t;
        tick_count();
        tick_count( const tick_count& );
        ~tick_count();
        tick_count& operator=( const tick_count& );
        static tick_count now();
        static double resolution();
    };

} // namespace tbb
```

`tick_count()` Constructs `tick_count` with an unspecified wall clock timestamp.

`tick_count( const tick_count& )` Constructs `tick_count` with the timestamp of the given `tick_count`.

`~tick_count()` Destructor.

`tick_count& operator=( const tick_count& )` Assigns the timestamp of one `tick_count` to another.

`static tick_count now()` Returns a `tick_count` object that represents the current wall clock timestamp.

`static double resolution()` Returns the resolution of the clock used by `tick_count`, in seconds.

#### 11.3.3.2.2 tick\_count::interval\_t class

##### [timing.tick\_count.interval\_t]

A `tick_count::interval_t` represents wall clock duration.

```
namespace tbb {

    class tick_count::interval_t {
    public:
        interval_t();
        explicit interval_t( double );
        ~interval_t();
        interval_t& operator=( const interval_t& );
        interval_t& operator+=( const interval_t& );
        interval_t& operator-=( const interval_t& );
        double seconds() const;
    };

} // namespace tbb
```

```

interval_t() Constructs interval_t representing zero time duration.

explicit interval_t( double ) Constructs interval_t representing the specified number of seconds.

~interval_t() Destructor.

interval_t& operator=( const interval_t& ) Assigns the wall clock duration of one interval_t
to another.

interval_t& operator+=( const interval_t& ) Increases the duration to the given interval_t,
and returns *this.

interval_t& operator-=( const interval_t& ) Decreases the duration to the given interval_t,
and returns *this.

double seconds() const Returns the duration measured in seconds.

```

### 11.3.3.2.3 Non-member functions

#### [timing.tick\_count.nonmember]

These functions provide arithmetic binary operations with wall clock timestamps and durations.

```
tbb::tick_count::interval_t operator-( const tbb::tick_count&, const tbb::tick_count&  
↔ );
tbb::tick_count::interval_t operator+( const tbb::tick_count::interval_t&, const  
↔tbb::tick_count::interval_t& );
tbb::tick_count::interval_t operator-( const tbb::tick_count::interval_t&, const  
↔tbb::tick_count::interval_t& );
```

The namespace where these functions are defined is unspecified, as long as they may be used in respective binary operation expressions on tick\_count and tick\_count::interval\_t objects. For example, an implementation may define the classes and functions in the same unspecified internal namespace, and define tbb::tick\_count as a type alias for which the non-member functions are reachable only via argument dependent lookup.

```
tbb::tick_count::interval_t operator-( const tbb::tick_count&, const tbb::tick_count& )
    Returns interval_t representing the duration between two given wall clock timestamps.

tbb::tick_count::interval_t operator+( const tbb::tick_count::interval_t&, const tbb::tick_
    Returns interval_t representing the sum of two given intervals.

tbb::tick_count::interval_t operator-( const tbb::tick_count::interval_t&, const tbb::tick_
    Returns interval_t representing the difference of two given intervals.
```

The oneAPI Video Processing Library (oneVPL) is a programming interface for video decoding, encoding, and processing to build portable media pipelines on CPUs, GPUs, and other accelerators. It provides device discovery and selection in media centric and video analytics workloads and API primitives for zero-copy buffer sharing. oneVPL is backwards and cross-architecture compatible to ensure optimal execution on current and next generation hardware without source code changes.

## 12.1 Architecture

SDK functions fall into the following categories:

**DECODE** Functions that decode compressed video streams into raw video frames

**ENCODE** Functions that encode raw video frames into compressed bitstreams

**VPP** Functions that perform video processing on raw video frames

**CORE** Auxiliary functions in the SDK for synchronization

**Misc** Global auxiliary functions

With the exception of the global auxiliary functions, SDK functions are named after their functioning domain and category, as shown in Figure 1. The oneVPL SDK exposes video domain functions.

MFVideoDecode_DecodeFrameAsync		
Prefix	Domain	Class
		Name

Fig. 1: Figure 1: SDK function name notation

Applications use SDK functions by linking with the SDK dispatcher library, as shown in Figure 2. The dispatcher library identifies the hardware acceleration device on the running platform, determines the most suitable platform library for the identified hardware acceleration, and then redirects function calls accordingly.

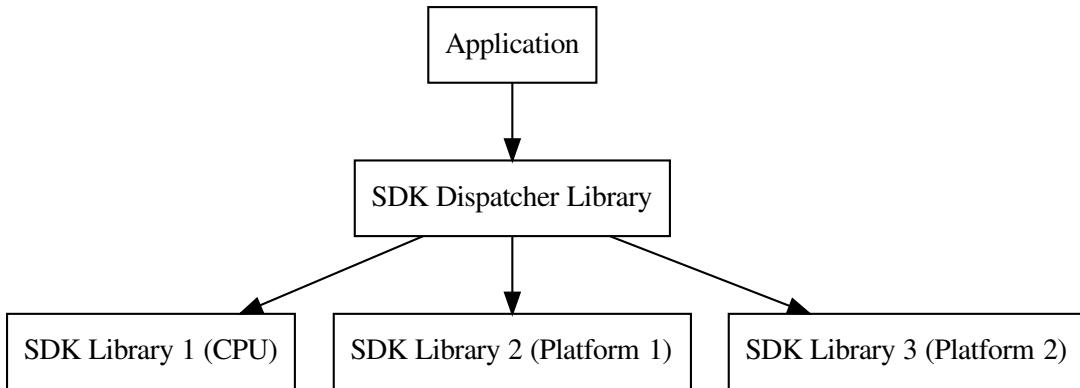


Fig. 2: Figure 2: SDK library dispatching mechanism

### 12.1.1 Video Decoding

The *DECODE* class of functions take a compressed bitstream as input and converts it to raw frames as output.

*DECODE* processes only pure or elementary video streams with the exception of AV1/VP9/VP8 decoders which accept IVF container. The library can process bitstreams that reside in the IVF container format, but cannot process bitstreams that reside in any other container format, such as MP4 or MPEG.

The application must first demultiplex the bitstreams. Demultiplexing extracts pure video streams out of the container format. The application can provide the input bitstream as one complete frame of data, a partial frame (less than one complete frame), or as multiple frames. If only a partial frame is provided, *DECODE* internally constructs one frame of data before decoding it.

The time stamp of a bitstream buffer must be accurate to the first byte of the frame data. For H.264, the first byte of the frame data comes from the NAL unit in the video coding layer. For MPEG-2 or VC-1, the first byte of the frame data comes from the picture header.

*DECODE* passes the time stamp to the output surface for audio and video multiplexing or synchronization.

Decoding the first frame is a special case, since *DECODE* does not provide enough configuration parameters to correctly process the bitstream. *DECODE* searches for the sequence header (a sequence parameter set in H.264, or a sequence header in MPEG-2 and VC-1) that contains the video configuration parameters used to encode subsequent video frames. The decoder skips any bitstream prior to the sequence header. In the case of multiple sequence headers in the bitstream, *DECODE* adopts the new configuration parameters, ensuring proper decoding of subsequent frames.

*DECODE* supports repositioning of the bitstream at any time during decoding. Because there is no way to obtain the correct sequence header associated with the specified bitstream position after a position change, the application must supply *DECODE* with a sequence header before the decoder can process the next frame at the new position. If the sequence header required to correctly decode the bitstream at the new position is not provided by the application, *DECODE* treats the new location as a new “first frame” and follows the procedure for decoding first frames.

## 12.1.2 Video Encoding

The [ENCODE](#) class of functions takes raw frames as input and compresses them into a bitstream.

Input frames usually come encoded in a repeated pattern called the Group of Picture ([GOP](#)) sequence. For example, a GOP sequence can start from an I-frame, followed by a few B-frames, a P-frame, and so on. ENCODE uses an MPEG-2 style GOP sequence structure that can specify the length of the sequence and the distance between two keyframes (I- or P-frames). A GOP sequence ensures that the segments of a bitstream do not completely depend upon each other. It also enables decoding applications to reposition the bitstream.

ENCODE processes input frames either by display order or by encoded order.

With display order processing, ENCODE receives input frames in the display order. GOP structure parameters specify the GOP sequence during ENCODE initialization. Scene changes resulting from the video processing stage of a pipeline can alter the GOP sequence.

With encoded order processing, ENCODE receives input frames in their encoding order. The application must specify the exact input frame type for encoding. ENCODE references GOP parameters to determine when to insert information, such as an end-of-sequence, into the bitstream.

ENCODE output consists of one frame of a bitstream with the time stamp passed from the input frame. The time stamp is used for multiplexing subsequent video with other associated data such as audio. The SDK library provides only pure video stream encoding. The application must provide its own multiplexing.

ENCODE supports the following bitrate control algorithms: constant bitrate, variable bitrate (VBR), and constant quantization parameter (QP). In the constant bitrate mode, ENCODE performs stuffing when the size of the least-compressed frame is smaller than what is required to meet the hypothetical reference decoder (HRD) buffer (or VBR) requirements. Stuffing is a process that appends zeros to the end of encoded frames.

## 12.1.3 Video Processing

Video processing functions ([VPP](#)) take raw frames as input and provide raw frames as output.

The actual conversion process is a chain operation with many single-function filters, as shown in Figure 3.



Fig. 3: Figure 3: Video processing operation pipeline

The application specifies the input and output format and the SDK configures the pipeline accordingly. The application can also attach one or more hint structures to configure individual filters or turn them on and off. Unless specifically instructed, the SDK builds the pipeline in a way that best utilizes hardware acceleration or generates the best video processing quality.

The [Video Processing Features table](#) shows SDK video processing features. The application can configure supported video processing features through the video processing I/O parameters. The application can also configure optional features through hints. See [Video Processing Procedures](#) for more details on how to configure optional filters.

Table 1: Video Processing Features

Video Processing Features	Configuration
Convert color format from input to output	I/O parameters
De-interlace to produce progressive frames at the output	I/O parameters
Crop and resize the input frames	I/O parameters
Convert input frame rate to match the output	I/O parameters
Perform inverse telecine operations	I/O parameters
Fields weaving	I/O parameters
Fields splitting	I/O parameters
Remove noise	hint (optional feature)
Enhance picture details/edges	hint (optional feature)
Adjust the brightness, contrast, saturation, and hue settings	hint (optional feature)
Perform image stabilization	hint (optional feature)
Convert input frame rate to match the output, based on frame interpolation	hint (optional feature)
Perform detection of picture structure	hint (optional feature)

### 12.1.3.1 Color Conversion Support

The *Color Conversion Support table* shows color conversion supported by VPP. ‘X’ indicates a supported function.

Table 2: Color Conversion Support

Output Color>						
Input Color	NV12	RGB32	P010	P210	NV16	A2RGB10
RGB4 (RGB32)	X (limited)	X (limited)				
NV12	X	X	X		X	
YV12	X	X				
UYVY	X					
YUY2	X	X				
P010	X		X	X		X
P210	X		X	X	X	X
NV16	X			X	X	

---

**Note:** The SDK video processing pipeline supports limited functionality for RGB4 input. Only filters that are required to convert input format to output one are included in the pipeline. All optional filters are skipped. See description of the `MFX_WRN_FILTER_SKIPPED` warning in the `mfxStatus` enum for more details on how to retrieve list of active filters.

---

### 12.1.3.2 Deinterlacing/Inverse Telecine Support

The *Deinterlacing/Inverse Telecine Support table* shows deinterlacing/inverse telecine support in *VPP*. ‘X’ indicates a supported function.

Table 3: Deinterlacing/Inverse Telecine Support

Input Field Rate (fps) Interlaced	Output Frame Rate (fps)	Progressive					
•	23.976	25	29.97	30	50	59.94	60
29.97	Inverse Telecine		X				
50		X			X		
59.94			X			X	
60				X			X

The table describes a pure deinterlacing algorithm. The application can combine the algorithm with frame rate conversion to achieve any desirable input or output frame rate ratio. Note that in this table input rate is field rate, which is the number of video fields in one second of video. Because the SDK uses frame rate in all configuration parameters, this input field rate should be divided by two during the SDK configuration. For example, a 60i to 60p conversion is shown in the right bottom cell of the table. It should be described in the `mfxVideoParam` structure as input frame rate equal to 30 and output 60.

The SDK supports two hardware accelerated deinterlacing algorithms: BOB DI (in Linux\* libVA terms, VAProcDeinterlacingBob) and Advanced DI (VAProcDeinterlacingMotionAdaptive). Default is ADI (Advanced DI) which uses reference frames and has better quality. BOB DI is faster than ADI mode. The user can select between speed and quality as needed.

Configure DI modes with the `mfxExtVPPDeinterlacing` structure.

There is one special mode of deinterlacing available in combination with frame rate conversion. If VPP input frame is interlaced (TFF or BFF), output is progressive, and the ratio between source frame rate and destination frame rate is  $\frac{1}{2}$  (for example 30 to 60, 29.97 to 59.94, 25 to 50), a special mode of VPP is turned on: For 30 interlaced input frames, the application will get 60 different progressive output frames.

### 12.1.3.3 Color Format Support

The `VPP Filter Color Format Support table` shows color formats supported by `VPP` filters. ‘X’ indicates a supported function.

Table 4: VPP Filter Color Format Support

Color>							
Filter	RGB4 (RGB32)	NV12	YV12	YUY2	P010	P210	NV1
Denoise		X					
MCTF		X					
Deinterlace		X					
Image stabilization		X					
Frame rate conversion		X					
Resize		X			X	X	X
Detail		X					
Color conversion	X	X	X	X	X	X	X
Composition	X	X					
Field copy		X					
Fields weaving		X					
Fields splitting		X					

---

**Note:** The SDK video processing pipeline supports limited hardware acceleration for the P010 format. A zero value for the `mfxFrameInfo::Shift` field leads to partial acceleration.

The SDK video processing pipeline does not support hardware acceleration for the P210 format.

---

## 12.2 Programming Guide

This chapter describes the concepts used in programming with the oneVPL SDK.

The application must use the include file, `mfxvideo.h` for C/C++ programming and link the SDK dispatcher library, `libmfx.so`.

Include these files:

```
#include "mfxvideo.h" /* The SDK include file */
```

Link this library:

```
libmfx.so /* The SDK dynamic dispatcher library (Linux\*) */
```

### 12.2.1 Status Codes

The SDK functions are organized into categories for easy reference. The categories include `ENCODE` (encoding functions), `DECODE` (decoding functions), and `VPP` (video processing functions).

`Init`, `Reset`, and `Close` are member functions within the `ENCODE`, `DECODE`, and `VPP` classes that initialize, restart, and deinitialize specific operations defined for the class. Call all other member functions within a given class, except `Query` and `QueryIOSurf`, within the **Init - Reset - Close** sequence (Reset functions are optional within the sequence).

The `Init` and `Reset` member functions set up necessary internal structures for media processing. `Init` functions allocate memory and `Reset` functions only reuse allocated internal memory. Therefore, `Reset` can fail if the SDK needs to allocate additional memory. `Reset` functions can also fine-tune `ENCODE` and `VPP` parameters during those processes or reposition a bitstream during `DECODE`.

All SDK functions return status codes to indicate if an operation succeeded or failed. See the `mfxStatus` enumerator for all defined status codes. The status code `mfxStatus::MFX_ERR_NONE` indicates that the function successfully completed its operation. Error status codes are less than `mfxStatus::MFX_ERR_NONE` and warning status codes are greater than `mfxStatus::MFX_ERR_NONE`.

If an SDK function returns a warning, it has sufficiently completed its operation, although the output of the function might not be strictly reliable. The application must check the validity of the output generated by the function.

If an SDK function returns an error (except `mfxStatus::MFX_ERR_MORE_DATA`, `mfxStatus::MFX_ERR_MORE_SURFACE`, or `mfxStatus::MFX_ERR_MORE_BITSTREAM`), the function aborts the operation. The application must call either the `Reset` function to reset the class back to a clean state or the `Close` function to terminate the operation. The behavior is undefined if the application continues to call any class member functions without a `Reset` or `Close`. To avoid memory leaks, always call the `Close` function after `Init`.

## 12.2.2 SDK Session

Before calling any SDK functions, the application must initialize the SDK library and create an SDK session. An SDK session maintains context for the use of any of *DECODE*, *ENCODE*, or *VPP* functions.

### 12.2.2.1 Intel® Media Software Development Kit Dispatcher (Legacy)

The function *MFXInit()* starts (initializes) an SDK session. *MFXClose()* closes (de-initializes) the SDK session. To avoid memory leaks, always call *MFXClose()* after *MFXInit()*.

The application can initialize a session as a software-based session (*MFX\_IMPL\_SOFTWARE*) or a hardware-based session (*MFX\_IMPL\_HARDWARE*). In a software-based session, the SDK functions execute on a CPU. In a hardware-base session, the SDK functions use platform acceleration capabilities. For platforms that expose multiple graphic devices, the application can initialize the SDK session on any alternative graphic device using the *MFX\_IMPL\_HARDWARE*, *MFX\_IMPL\_HARDWARE2*, *MFX\_IMPL\_HARDWARE3*, or *MFX\_IMPL\_HARDWARE4* values of *mfxImpl*

The application can also initialize a session to be automatic (*MFX\_IMPL\_AUTO* or *MFX\_IMPL\_AUTO\_ANY*), instructing the dispatcher library to detect the platform capabilities and choose the best SDK library available. After initialization, the SDK returns the actual implementation through the *MFXQueryImpl()* function.

Internally, the dispatcher works as follows:

1. It searches for the shared library with the specific name:

OS	Name	Description
Linux*	libmfxsw64.so.1	64-bit software-based implementation
Linux	libmfxsw32.so.1	32-bit software-based implementation
Linux	libmfhw64.so.1	64-bit hardware-based implementation
Linux	libmfhw64.so.1	32-bit hardware-based implementation
Windows*	libmfxsw32.dll	64-bit software-based implementation
Windows	libmfxsw32.dll	32-bit software-based implementation
Windows	libmfhw64.dll	64-bit hardware-based implementation
Windows	libmfhw64.dll	32-bit hardware-based implementation

2. Once the library is loaded, the dispatcher obtains addresses for each SDK function. See the *Exported Functions/API Version table* for the list of functions to export.

Shared library with the implementation search strategy:

- **Windows** Dispatcher goes through the following list in specified order until it finds implementstion library:
  1. **Driver Store** directory for the current adapter (more about Driver Store). **This is directory which library can be installed in by all types of graphics driver.**
  2. The directory which is specified for the current hardware under the registry key `HKEY_CURRENT_USER\Software\Intel\MediaSDK\Dispatch`.
  3. The directory which is specified for the current hardware under the registry key `HKEY_LOCAL_MACHINE\Software\Intel\MediaSDK\Dispatch`.
  4. The actual directory which is stored in these registry keys is `C:\Program Files\Intel\Media SDK`. **This is directory which library is installed in by Legacy graphics driver.**
  5. The directory where the current module (module that links dispatcher) is located (only if the current module is a dll).

After the main search part, dispatcher additionally checks:

1. The directory of the exe file of the current process, where it looks for software implementation only, no matter which implementation is requested by application.
  2. Default dll search, this way provides loading from directory of application's exe file and from **Sys-tem32/SysWOW64** directories ([more about default dll search order](#)).
  3. **System32** and **SysWOW64** are directories which library is installed in by DCH graphics driver.
- **Linux** Dispatcher goes through the following list in specified order until it finds implementstion library:
    1. Directories provided by the environment variable **LD\_LIBRARY\_PATH**.
    2. Content of the cache file /etc/ld.so.cache.
    3. Default path /lib, and then /usr/lib or /lib64, and then /usr/lib64 on some 64 bit OSes. On Debian: /usr/lib/x86\_64-linux-gnu.
    4. SDK installation folder.

### 12.2.2.2 oneVPL Dispatcher

The oneVPL dispatcher extends the legacy dispatcher by providing additional ability to select the appropriate implementation based on the implementation capabilities. Implementation capabilities include information about supported decoders, encoders, and VPP filters. For each supported encoder, decoder, and filter, capabilities include information about supported memory types, color formats, and image (frame) size in pixels.

The recommended approach to configure the dispatcher's capabilities search filters and to create a session based on suitable implementation is as follows:

1. Create loader (dispatcher function [\*MFXLoad\(\)\*](#)).
2. Create loader's configuration (dispatcher function [\*MFXCreateConfig\(\)\*](#)).
3. Add configuration properties (dispatcher function [\*MFXSetConfigFilterProperty\(\)\*](#)).
4. Explore available implementations (dispatcher function [\*MFXEnumImplementations\(\)\*](#)).
5. Create suitable session (dispatcher function [\*MFXCreateSession\(\)\*](#)).

The procedure to terminate an application is as follows:

1. Destroy session (function [\*MFXClose\(\)\*](#)).
2. Destroy loader (dispatcher function [\*MFXUnload\(\)\*](#)).

---

**Note:** Multiple loader instances can be created.

---



---

**Note:** Each loader may have multiple configuration objects associated with it.

---



---

**Important:** One configuration object can handle only one filter property.

---



---

**Note:** Multiple sessions can be created by using one loader object.

---

When the dispatcher searches for the implementation, it uses the following priority rules:

1. Hardware implementation has priority over software implementation.

2. General hardware implementation has priority over VSI hardware implementation.
3. Highest API version has higher priority over lower API version.

---

**Note:** Implementation has priority over the API version. In other words, the dispatcher must return the implementation with the highest API priority (greater than or equal to the implementation requested).

---

The dispatcher searches for the implementation in the following folders at runtime (in priority order):

1. User-defined search folders.
2. oneVPL package, including default system folders.
3. Standalone Intel® Media Software Development Kit package (or driver).

For more details, see legacy dispatcher search order.

A user has the ability to develop their own implementation and guide the oneVPL dispatcher to load their implementation by providing a list of search folders. The specific steps depend on which OS is used.

- Linux: User can provide a colon separated list of folders in ONEVPL\_SEARCH\_PATH environmental variable.
- Windows: User can provide semicolon separated list of folders in ONEVPL\_SEARCH\_PATH environmental variable. Alternatively, the user can use the Windows registry.

The dispatcher supports different software implementations. The user can use the `mfxImplDescription::VendorID` field, the `mfxImplDescription::VendorImplID` field, or the `mfxImplDescription::ImplName` field to search for the specific implementation.

Internally, the dispatcher works as follows:

1. Dispatcher loads any shared library within the given search folders.
2. For each loaded library, the dispatcher tries to resolve address of the `MFXQueryImplsCapabilities()` function to collect the implementation's capabilities.
3. Once the user has requested to create the session based on this implementation, the dispatcher obtains addresses of each SDK function. See the *Exported Functions/API Version table* for the list of functions to export.

This table summarizes the list of environmental variables used to control the dispatcher behavior:

Variable	Purpose
ONEVPL_SEARCH_PATH	List of user-defined search folders.

---

**Note:** Each implementation must support both dispatchers for backward compatibility with existing applications.

---

### 12.2.2.3 Multiple Sessions

Each SDK session can run exactly one instance of the DECODE, ENCODE, and VPP functions. This is good for a simple transcoding operation. If the application needs more than one instance of DECODE, ENCODE, or VPP in a complex transcoding setting or needs more simultaneous transcoding operations to balance CPU/GPU workloads, the application can initialize multiple SDK sessions. Each independent SDK session can be a software-based session or hardware-based session.

The application can use multiple SDK sessions independently or run a “joined” session. Independently operated SDK sessions cannot share data unless the application explicitly synchronizes session operations. This is to ensure that data is valid and complete before passing from the source to the destination session.

To join two sessions together, the application can use the function `MFXJoinSession()`. Alternatively, the application can use the `MFXCloneSession()` function to duplicate an existing session. Joined SDK sessions work together as a single session, sharing all session resources, threading control, and prioritization operations except hardware acceleration devices and external allocators. When joined, the first session (first join) serves as the parent session and will schedule execution resources with all other child sessions. Child sessions rely on the parent session for resource management.

With joined sessions, the application can set the priority of session operations through the `MFXSetPriority()` function. A lower priority session receives fewer CPU cycles. Session priority does not affect hardware accelerated processing.

After the completion of all session operations, the application can use the `MFXDisjoinSession()` function to remove the joined state of a session. Do not close the parent session until all child sessions are disjoined or closed.

## 12.2.3 Frame and Fields

In SDK terminology, a frame (also referred to as frame surface) contains either a progressive frame or a complementary field pair. If the frame is a complementary field pair, the odd lines of the surface buffer store the top fields and the even lines of the surface buffer store the bottom fields.

### 12.2.3.1 Frame Surface Locking

During encoding, decoding, or video processing, cases arise that require reserving input or output frames for future use. For example, when decoding, a frame that is ready for output must remain as a reference frame until the current sequence pattern ends. The usual approach to manage this is to cache the frames internally. This method requires a copy operation, which can significantly reduce performance.

SDK functions define a frame-locking mechanism to avoid the need for copy operations. The frame-locking mechanism works as follows:

1. The application allocates a pool of frame surfaces large enough to include SDK function I/O frame surfaces and internal cache needs. Each frame surface maintains a `Locked` counter, which is part of the `mfxFrameData` structure. The `Locked` counter is initially set to zero.
2. The application calls an SDK function with frame surfaces from the pool, whose `Locked` counter is set as appropriate for the desired operation. For decoding or video processing operations, where the SDK uses the surfaces to write, the `Locked` counter should be equal to zero. If the SDK function needs to reserve any frame surface, the SDK function increases the `Locked` counter of the frame surface. A non-zero `Locked` counter indicates that the calling application must treat the frame surface as “in use.” When the frame surface is in use, the application can read, but cannot alter, move, delete, or free the frame surface.
3. In subsequent SDK executions, if the frame surface is no longer in use, the SDK decreases the `Locked` counter. When the `Locked` counter reaches zero, the application is free to do as it wishes with the frame surface.

In general, the application must not increase or decrease the `Locked` counter, since the SDK manages this field. If, for some reason, the application needs to modify the `Locked` counter, the operation must be atomic to avoid a race condition.

**Attention:** Modifying the `Locked` counter is not recommended.

API version 2.0 introduces the `mfxFrameSurfaceInterface` structure which provides a set of callback functions for the `mfxFrameSurface1` to work with frames. This interface defines `mfxFrameSurface1` as a reference counted object which can be allocated by the SDK or application. The application must follow the general rules of operations with reference counted objects. For example, when surfaces are allocated by the SDK during `MFXVideoDECODE_DecodeFrameAsync()` or with the help of

`MFXMemory_GetSurfaceForVPP()` or `MFXMemory_GetSurfaceForEncode()`, the application must call the corresponding `mfxFrameSurfaceInterface->(*Release)` for the surfaces that are no longer in use.

**Attention:** Note that the Locked counter defines read/write access policies and the reference counter is responsible for managing a frame's lifetime.

**Note:** All `mfxFrameSurface1` structures starting from `mfxFrameSurface1::mfxStructVersion = {1,1}` support the `mfxFrameSurfaceInterface`.

## 12.2.4 Decoding Procedures

The following pseudo code shows the decoding procedure:

```

1 MFXVideoDECODE_DecodeHeader(session, bitstream, &init_param);
2 MFXVideoDECODE_QueryIOSurf(session, &init_param, &request);
3 allocate_pool_of_frame_surfaces(request.NumFrameSuggested);
4 MFXVideoDECODE_Init(session, &init_param);
5 sts=MFX_ERR_MORE_DATA;
6 for (;;) {
7     if (sts==MFX_ERR_MORE_DATA && !end_of_stream())
8         append_more_bitstream(bitstream);
9     find_unlocked_surface_from_the_pool(&work);
10    bits=(end_of_stream())?NULL:bitstream;
11    sts=MFXVideoDECODE_DecodeFrameAsync(session,bits,work,&disp,&syncp);
12    if (sts==MFX_ERR_MORE_SURFACE) continue;
13    if (end_of_stream() && sts==MFX_ERR_MORE_DATA) break;
14    if (sts==MFX_ERR_REALLOC_SURFACE) {
15        MFXVideoDECODE_GetVideoParam(session, &param);
16        realloc_surface(work, param.mfx.FrameInfo);
17        continue;
18    }
19    // skipped other error handling
20    if (sts==MFX_ERR_NONE) {
21        MFXVideoCORE_SyncOperation(session, syncp, INFINITE);
22        do_something_with_decoded_frame(disp);
23    }
24 }
25 MFXVideoDECODE_Close(session);
26 free_pool_of_frame_surfaces();

```

Note the following key points about the example:

- The application can use the `MFXVideoDECODE_DecodeHeader()` function to retrieve decoding initialization parameters from the bitstream. This step is optional if the data is retrievable from other sources such as an audio/video splitter.
- The application can use the `MFXVideoDECODE_QueryIOSurf()` function to obtain the number of working frame surfaces required to reorder output frames. This step is required if the application is responsible for memory allocation. Use of this function is not required if the SDK is responsible for memory allocation.
- The application calls the `MFXVideoDECODE_DecodeFrameAsync()` function for a decoding operation with the bitstream buffer (bits) and an unlocked working frame surface (work) as input parameters.

**Attention:** Starting with API version 2.0, the application can provide NULL as the working frame surface what leads to internal memory allocation.

- If decoding output is not available, the function returns a status code requesting additional bitstream input or working frame surfaces as follows:
  - `mfxStatus::MFX_ERR_MORE_DATA`: The function needs additional bitstream input. The existing buffer contains less than a frame's worth of bitstream data.
  - `mfxStatus::MFX_ERR_MORE_SURFACE`: The function needs one more frame surface to produce any output.
  - `mfxStatus::MFX_ERR_REALLOC_SURFACE`: Dynamic resolution change case - the function needs bigger working frame surface (work).
- Upon successful decoding, the `MFXVideoDECODE_DecodeFrameAsync()` function returns `mfxStatus::MFX_ERR_NONE`. However, the decoded frame data (identified by the `surface_out` pointer) is not yet available because the `MFXVideoDECODE_DecodeFrameAsync()` function is asynchronous. The application must use the `MFXVideoCORE_SyncOperation()` or `mfxFrameSurfaceInterface` interface to synchronize the decoding operation before retrieving the decoded frame data.
- At the end of the bitstream, the application continuously calls the `MFXVideoDECODE_DecodeFrameAsync()` function with a NULL bitstream pointer to drain any remaining frames cached within the SDK decoder, until the function returns `mfxStatus::MFX_ERR_MORE_DATA`.

The following example shows the simplified decoding procedure:

```

1 sts=MFX_ERR_MORE_DATA;
2 for (;;) {
3     if (sts==MFX_ERR_MORE_DATA && !end_of_stream())
4         append_more_bitstream(bitstream);
5     bits=(end_of_stream())?NULL:bitstream;
6     sts=MFXVideoDECODE_DecodeFrameAsync(session,bits,NULL,&disp,&syncp);
7     if (sts==MFX_ERR_MORE_SURFACE) continue;
8     if (end_of_stream() && sts==MFX_ERR_MORE_DATA) break;
9     // skipped other error handling
10    if (sts==MFX_ERR_NONE) {
11        MFXVideoCORE_SyncOperation(session, syncp, INFINITE);
12        do_something_with_decoded_frame(disp);
13        release_surface(disp);
14    }
15 }
```

API version 2.0 introduces a new decoding approach. For simple use cases, when the user wants to decode a stream and doesn't want to set additional parameters, the simplified procedure for the decoder's initialization has been proposed. For such situations it is possible to skip explicit stages of a stream's header decoding and the decoder's initialization and instead to perform it implicitly during decoding of the first frame. This change also requires additional field in the `mfxBitstream` object to indicate codec type. In that mode the decoder allocates `mfxFrameSurface1` internally, so users should set input surface to zero.

#### 12.2.4.1 Bitstream Repositioning

The application can use the following procedure for bitstream repositioning during decoding:

1. Use the `MFXVideoDECODE_Reset()` function to reset the SDK decoder.
2. Optional: If the application maintains a sequence header that correctly decodes the bitstream at the new position, the application may insert the sequence header to the bitstream buffer.
3. Append the bitstream from the new location to the bitstream buffer.
4. Resume the decoding procedure. If the sequence header is not inserted in the previous steps, the SDK decoder searches for a new sequence header before starting decoding.

#### 12.2.4.2 Broken Streams Handling

Robustness and the capability to handle broken input stream is an important part of the decoder.

First, the start code prefix (ITU-T\* H.264 3.148 and ITU-T H.265 3.142) is used to separate NAL units. Then all syntax elements in the bitstream are parsed and verified. If any of the elements violate the specification, the input bitstream is considered invalid and the decoder tries to re-sync (find the next start code). Subsequent decoder behavior is dependent on which syntax element is broken:

- SPS header – return `mfxStatus::MFX_ERR_INCOMPATIBLE_VIDEO_PARAM` (HEVC decoder only, AVC decoder uses last valid)
- PPS header – re-sync, use last valid PPS for decoding
- Slice header – skip this slice, re-sync
- Slice data - Corruption flags are set on output surface

---

**Note:** Some requirements are relaxed because there are many streams which violate the strict standard but can be decoded without errors.

---

- Many streams have IDR frames with `frame_num != 0` while the specification says that “If the current picture is an IDR picture, `frame_num` shall be equal to 0” (ITU-T H.265 7.4.3).
- VUI is also validated, but errors don’t invalidate the whole SPS. The decoder either doesn’t use the corrupted VUI (AVC) or resets incorrect values to default (HEVC).

Corruption at the reference frame is spread over all inter-coded pictures that use the reference frame for prediction. To cope with this problem you must either periodically insert I-frames (intra-coded) or use the ‘intra-refresh’ technique. The ‘intra refresh’ technique allows the recovery from corruptions within a predefined time interval. The main point of intra-refresh is to insert a cyclic intra-coded pattern (usually a row) of macroblocks into the inter-coded pictures, restricting motion vectors accordingly. Intra-refresh is often used in combination with recovery point SEI, where the `recovery_frame_cnt` is derived from the intra-refresh interval. The recovery point SEI message is well described at ITU-T H.264 D.2.7 and ITU-T H.265 D.2.8. If decoding starts from AU associated with this SEI message, then the message can be used by the decoder to determine from which picture all subsequent pictures have no errors. In comparison to IDR, the recovery point message doesn’t mark reference pictures as ‘unused for reference’.

Besides validation of syntax elements and their constraints, the decoder also uses various hints to handle broken streams:

- If there are no valid slices for the current frame, then the whole frame is skipped.
- The slices which violate slice segment header semantics (ITU-T H.265 7.4.7.1) are skipped. Only `slice_temporal_mvp_enabled_flag` is checked for now.

- Since LTR (Long Term Reference) stays at DPB until it is explicitly cleared by IDR or MMCO, the incorrect LTR could cause long standing visual artifacts. AVC decoder uses the following approaches to handle this:
  - When there is a DPB overflow in the case of an incorrect MMCO command that marks the reference picture as LT, the operation is rolled back.
  - An IDR frame with frame\_num != 0 can't be LTR.
- If the decoder detects frame gapping, it inserts ‘fake’ (marked as non-existing) frames, updates FrameNumWrap (ITU-T H.264 8.2.4.1) for reference frames, and applies the Sliding Window (ITU-T H.264 8.2.5.3) marking process. ‘Fake’ frames are marked as reference, but since they are marked as non-existing, they are not used for inter-prediction.

#### 12.2.4.3 VP8 Specific Details

Unlike other supported by SDK decoders, VP8 can accept only a complete frame as input. The application should provide the complete frame accompanied by the `MFX_BITSTREAM_COMPLETE_FRAME` flag. This is the single specific difference.

#### 12.2.4.4 JPEG

The application can use the same decoding procedures for JPEG/motion JPEG decoding, as shown in the pseudo code below:

```
// optional; retrieve initialization parameters
MFXVideoDECODE_DecodeHeader(...);
// decoder initialization
MFXVideoDECODE_Init(...);
// single frame/picture decoding
MFXVideoDECODE_DecodeFrameAsync(...);
MFXVideoCORE_SyncOperation(...);
// optional; retrieve meta-data
MFXVideoDECODE_GetUserData(...);
// close
MFXVideoDECODE_Close(...);
```

`DECODE` supports JPEG baseline profile decoding as follows:

- DCT-based process
- Source image: 8-bit samples within each component
- Sequential
- Huffman coding: 2 AC and 2 DC tables
- 3 loadable quantization matrices
- Interleaved and non-interleaved scans
- Single and multiple scans
  - Chroma sub-sampling ratios:
    - \* Chroma 4:0:0 (grey image)
    - \* Chroma 4:1:1
    - \* Chroma 4:2:0
    - \* Chroma horizontal 4:2:2

- \* Chroma vertical 4:2:2
- \* Chroma 4:4:4
- 3 channel images

The `MFXVideoDECODE_Query()` function will return `mfxStatus::MFX_ERR_UNSUPPORTED` if the input bitstream contains unsupported features.

For still picture JPEG decoding, the input can be any JPEG bitstreams that conform to the ITU-T Recommendation T.81 with an EXIF or JFIF header. For motion JPEG decoding, the input can be any JPEG bitstreams that conform to the ITU-T Recommendation T.81.

Unlike other SDK decoders, JPEG decoding supports three different output color formats: `NV12`, `YUY2`, and `RGB32`. This support sometimes requires internal color conversion and more complicated initialization. The color format of the input bitstream is described by the `JPEGChromaFormat` and `JPEGColorFormat` fields in the `mfxInfoMFX` structure. The `MFXVideoDECODE_DecodeHeader()` function usually fills them in. If the JPEG bitstream does not contain color format information, the application should provide it. Output color format is described by general SDK parameters: the `FourCC` and `ChromaFormat` fields in the `mfxFrameInfo` structure.

Motion JPEG supports interlaced content by compressing each field (a half-height frame) individually. This behavior is incompatible with the rest of the SDK transcoding pipeline, where the SDK requires fields to be in odd and even lines of the same frame surface. The decoding procedure is modified as follows:

- The application calls the `MFXVideoDECODE_DecodeHeader()` function with the first field JPEG bitstream to retrieve initialization parameters.
- The application initializes the SDK JPEG decoder with the following settings:
  - The `PicStruct` field of the `mfxVideoParam` structure set to the correct interlaced type, `MFX_PICSTRUCT_FIELD_TFF` or `MFX_PICSTRUCT_FIELD_BFF`, from the motion JPEG header.
  - Double the `Height` field in the `mfxVideoParam` structure as the value returned by the `MFXVideoDECODE_DecodeHeader()` function describes only the first field. The actual frame surface should contain both fields.
- During decoding, the application sends both fields for decoding in the same `mfxBitstream`. The application should also set `DataFlag` in the `mfxBitstream` structure to `MFX_BITSTREAM_COMPLETE_FRAME`. The SDK decodes both fields and combines them into odd and even lines according to the SDK convention.

The SDK supports JPEG picture rotation, in multiple of 90 degrees, as part of the decoding operation. By default, the `MFXVideoDECODE_DecodeHeader()` function returns the Rotation parameter so that after rotation, the pixel at the first row and first column is at the top left. The application can overwrite the default rotation before calling `MFXVideoDECODE_Init()`.

The application may specify Huffman and quantization tables during decoder initialization by attaching `mfxExtJPEGQuantTables` and `mfxExtJPEGHuffmanTables` buffers to the `mfxVideoParam` structure. In this case, the decoder ignores tables from bitstream and uses the tables specified by the application. The application can also retrieve these tables by attaching the same buffers to `mfxVideoParam` and calling `MFXVideoDECODE_GetVideoParam()` or `MFXVideoDECODE_DecodeHeader()` functions.

### 12.2.4.5 Multi-view Video Decoding

The SDK MVC decoder operates on complete MVC streams that contain all view and temporal configurations. The application can configure the SDK decoder to generate a subset at the decoding output. To do this, the application must understand the stream structure and use the stream information to configure the SDK decoder for target views.

The decoder initialization procedure is as follows:

1. The application calls the `MFXVideoDECODE_DecodeHeader()` function to obtain the stream structural information. This is done in two steps:
  1. The application calls the `MFXVideoDECODE_DecodeHeader()` function with the `mfxExtMVCSeqDesc` structure attached to the `mfxVideoParam` structure. At this point, do not allocate memory for the arrays in the `mfxExtMVCSeqDesc` structure. Set the View, ViewId, and OP pointers to NULL and set NumViewAlloc, NumViewIdAlloc, and NumOPAlloc to zero. The function parses the bitstream and returns `mfxStatus::MFX_ERR_NOT_ENOUGH_BUFFER` with the correct values for NumView, NumViewId, and NumOP. This step can be skipped if the application is able to obtain the NumView, NumViewId, and NumOP values from other sources.
  2. The application allocates memory for the View, ViewId, and OP arrays and calls the `MFXVideoDECODE_DecodeHeader()` function again. The function returns the MVC structural information in the allocated arrays.
2. The application fills the `mfxExtMVCTargetViews` structure to choose the target views, based on information described in the `mfxExtMVCSeqDesc` structure.
3. The application initializes the SDK decoder using the `MFXVideoDECODE_Init()` function. The application must attach both the `mfxExtMVCSeqDesc` structure and the `mfxExtMVCTargetViews` structure to the `mfxVideoParam` structure.

In the above steps, do not modify the values of the `mfxExtMVCSeqDesc` structure after the `MFXVideoDECODE_DecodeHeader()` function, as the SDK decoder uses the values in the structure for internal memory allocation. Once the application configures the SDK decoder, the rest of the decoding procedure remains unchanged. As shown in the pseudo code below, the application calls the `MFXVideoDECODE_DecodeFrameAsync()` function multiple times to obtain all target views of the current frame picture, one target view at a time. The target view is identified by the FrameID field of the `mfxFrameInfo` structure.

```

1 mfxExtBuffer *eb[2];
2 mfxExtMVCSeqDesc seq_desc;
3 mfxVideoParam init_param;
4
5 init_param.ExtParam=(mfxExtBuffer **) &eb;
6 init_param.NumExtParam=1;
7 eb[0]=(mfxExtBuffer *)&seq_desc;
8 MFXVideoDECODE_DecodeHeader(session, bitstream, &init_param);
9
10 /* select views to decode */
11 mfxExtMVCTargetViews tv;
12 init_param.NumExtParam=2;
13 eb[1]=(mfxExtBuffer *)&tv;
14
15 /* initialize decoder */
16 MFXVideoDECODE_Init(session, &init_param);
17
18 /* perform decoding */
19 for (;;) {
20     MFXVideoDECODE_DecodeFrameAsync(session, bits, work, &disp, &syncp);

```

(continues on next page)

(continued from previous page)

```

21     MFXVideoCORE_SyncOperation(session, syncp, INFINITE);
22 }
23
24 /* close decoder */
25 MFXVideoDECODE_Close(session);

```

## 12.2.5 Encoding Procedures

There are two methods for shared memory allocation and handling in the SDK: external and internal.

### 12.2.5.1 External Memory

The following example shows the pseudo code of the encoding procedure with external memory (legacy mode):

```

1 MFXVideoENCODE_QueryIOSurf(session, &init_param, &request);
2 allocate_pool_of_frame_surfaces(request.NumFrameSuggested);
3 MFXVideoENCODE_Init(session, &init_param);
4 sts=MFX_ERR_MORE_DATA;
5 for (;;) {
6     if (sts==MFX_ERR_MORE_DATA && !end_of_stream()) {
7         find_unlocked_surface_from_the_pool(&surface);
8         fill_content_for_encoding(surface);
9     }
10    surface2=end_of_stream() ?NULL:surface;
11    sts=MFVVideoENCODE_EncodeFrameAsync(session,NULL,surface2,bits,&syncp);
12    if (end_of_stream() && sts==MFX_ERR_MORE_DATA) break;
13    // Skipped other error handling
14    if (sts==MFX_ERR_NONE) {
15        MFXVideoCORE_SyncOperation(session, syncp, INFINITE);
16        do_something_with_encoded_bits(bits);
17    }
18 }
19 MFXVideoENCODE_Close(session);
20 free_pool_of_frame_surfaces();

```

Note the following key points about the example:

- The application uses the `MFVVideoENCODE_QueryIOSurf()` function to obtain the number of working frame surfaces required for reordering input frames.
- The application calls the `MFVVideoENCODE_EncodeFrameAsync()` function for the encoding operation. The input frame must be in an unlocked frame surface from the frame surface pool. If the encoding output is not available, the function returns the status code `mfxStatus::MFX_ERR_MORE_DATA` to request additional input frames.
- Upon successful encoding, the `MFVVideoENCODE_EncodeFrameAsync()` function returns `mfxStatus::MFX_ERR_NONE`. At this point, the encoded bitstream is not yet available because the `MFVVideoENCODE_EncodeFrameAsync()` function is asynchronous. The application must use the `MFXVideoCORE_SyncOperation()` function to synchronize the encoding operation before retrieving the encoded bitstream.
- At the end of the stream, the application continuously calls the `MFVVideoENCODE_EncodeFrameAsync()` function with a NULL surface pointer to drain any remaining bitstreams cached within the SDK encoder, until the function returns `mfxStatus::MFX_ERR_MORE_DATA`.

---

**Note:** It is the application's responsibility to fill pixels outside of the crop window when it is smaller than the frame to be encoded, especially in cases when crops are not aligned to minimum coding block size (16 for AVC and 8 for HEVC and VP9).

---

### 12.2.5.2 Internal Memory

The following example shows the encoding procedure with internal memory:

```

1 MFXVideoENCODE_Init(session, &init_param);
2 sts=MFX_ERR_MORE_DATA;
3 for (;;) {
4     if (sts==MFX_ERR_MORE_DATA && !end_of_stream()) {
5         MFXMemory_GetSurfaceForEncode(session,&surface);
6         fill_content_for_encoding(surface);
7     }
8     surface2=end_of_stream() ?NULL:surface;
9     sts=MFXVideoENCODE_EncodeFrameAsync(session,NULL,surface2,bits,&syncp);
10    if (surface2) surface->FrameInterface->Release(surface2);
11    if (end_of_stream() && sts==MFX_ERR_MORE_DATA) break;
12    // Skipped other error handling
13    if (sts==MFX_ERR_NONE) {
14        MFXVideoCORE_SyncOperation(session, syncp, INFINITE);
15        do_something_with_encoded_bits(bits);
16    }
17 }
18 MFXVideoENCODE_Close(session);

```

There are several key differences in this example, compared to external memory (legacy mode):

- The application doesn't need to call the `MFXVideoENCODE_QueryIOSurf()` function to obtain the number of working frame surfaces since allocation is done by the SDK.
- The application calls the `MFXMemory_GetSurfaceForEncode()` function to get a free surface for the subsequent encode operation.
- The application needs to call the `FrameInterface->(\*Release)` function to decrement the reference counter of the obtained surface after the call to the `MFXVideoENCODE_EncodeFrameAsync()` function.

### 12.2.5.3 Configuration Change

The application changes configuration during encoding by calling the `MFXVideoENCODE_Reset()` function. Depending on the difference in configuration parameters before and after the change, the SDK encoder will either continue the current sequence or start a new one. If the SDK encoder starts a new sequence, it completely resets internal state and begins a new sequence with the IDR frame.

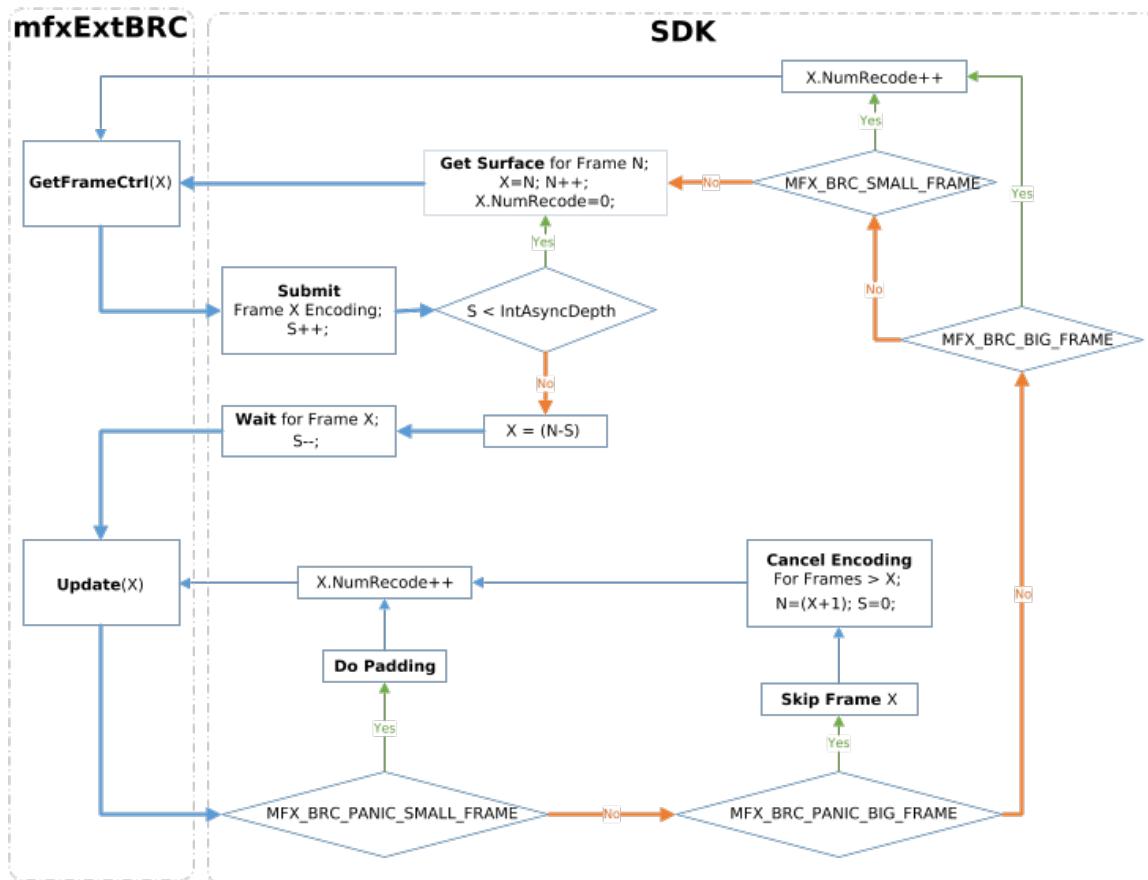
The application controls encoder behavior during parameter change by attaching the `mfxExtEncoderResetOption` structure to the `mfxVideoParam` structure during reset. By using this structure, the application instructs the encoder to start or not start a new sequence after reset. In some cases, the request to continue the current sequence cannot be satisfied and the encoder will fail during reset. To avoid this scenario, the application may query the reset outcome before the actual reset by calling `MFXVideoENCODE_Query()` function with the `mfxExtEncoderResetOption` attached to the `mfxVideoParam` structure.

The application uses the following procedure to change encoding configurations:

1. The application retrieves any cached frames in the SDK encoder by calling the `MFXVideoENCODE_EncodeFrameAsync()` function with a NULL input frame pointer until the function returns `mfxStatus::MFX_ERR_MORE_DATA`.
2. The application calls the `MFXVideoENCODE_Reset()` function with the new configuration:
  - If the function successfully sets the configuration, the application can continue encoding as usual.
  - If the new configuration requires a new memory allocation, the function returns `mfxStatus::MFX_ERR_INCOMPATIBLE_VIDEO_PARAM`. The application must close the SDK encoder and reinitialize the encoding procedure with the new configuration.

#### 12.2.5.4 External Bitrate Control

The application can make the encoder use the external Bitrate Control (BRC) instead of the native bitrate control. To make the encoder use the external BRC, the application should attach the `mfxExtCodingOption2` structure with `ExtBRC = MFX_CODINGOPTION_ON` and the `mfxExtBRC` callback structure to the `mfxVideoParam` structure during encoder initialization. The Init, Reset, and Close callbacks will be invoked inside their corresponding functions: `MFXVideoENCODE_Init()`, `MFXVideoENCODE_Reset()`, and `MFXVideoENCODE_Close()`. The following figure shows asynchronous encoding flow with external BRC (using `GetFrameCtrl` and `Update`):



**Note:** `IntAsyncDepth` is the SDK max internal asynchronous encoding queue size. It is always less than or equal

to `mfxVideoParam::AsyncDepth`.

The following pseudo code shows use of the external BRC:

```

1 #include "mfxvideo.h"
2 #include "mfxbrc.h"
3
4 typedef struct {
5     mfxU32 EncodedOrder;
6     mfxI32 QP;
7     mfxU32 MaxSize;
8     mfxU32 MinSize;
9     mfxU16 Status;
10    mfxU64 StartTime;
11    // ... skipped
12 } MyBrcFrame;
13
14 typedef struct {
15     MyBrcFrame* frame_queue;
16     mfxU32 frame_queue_size;
17     mfxU32 frame_queue_max_size;
18     mfxI32 max_qp[3]; //I,P,B
19     mfxI32 min_qp[3]; //I,P,B
20     // ... skipped
21 } MyBrcContext;
22
23 void* GetExtBuffer(mfxExtBuffer** ExtParam, mfxU16 NumExtParam, mfxU32 bufferID)
24 {
25     int i=0;
26     for(i = 0; i < NumExtParam; i++) {
27         if(ExtParam[i]->BufferId == bufferID) return ExtParam[i];
28     }
29     return NULL;
30 }
31
32 static int IsParametersSupported(mfxVideoParam *par)
33 {
34     // do some checks
35     return 1;
36 }
37
38 static int IsResetPossible(MyBrcContext* ctx, mfxVideoParam *par)
39 {
40     // do some checks
41     return 1;
42 }
43
44 static MyBrcFrame* GetFrame(MyBrcFrame *frame_queue, mfxU32 frame_queue_size,
45     ↵mfxU32 EncodedOrder)
46 {
47     //do some logic
48     if(frame_queue_size) return &frame_queue[0];
49     return NULL;
50 }
51
52 static mfxU32 GetFrameCost(mfxU16 FrameType, mfxU16 PyramidLayer)
53 {

```

(continues on next page)

(continued from previous page)

```

53     // calculate cost
54     return 1;
55 }
56
57 static mfxU32 GetMinSize(MyBrcContext *ctx, mfxU32 cost)
58 {
59     // do some logic
60     return 1;
61 }
62
63 static mfxU32 GetMaxSize(MyBrcContext *ctx, mfxU32 cost)
64 {
65     // do some logic
66     return 1;
67 }
68
69 static mfxI32 GetInitQP(MyBrcContext *ctx, mfxU32 MinSize, mfxU32 MaxSize, mfxU32_
70   ↵cost)
71 {
72     // do some logic
73     return 1;
74 }
75
76 static mfxU64 GetTime()
77 {
78     mfxU64 wallClock;
79     return wallClock;
80 }
81
82 static void UpdateBRCState(mfxU32 CodedFrameSize, MyBrcContext *ctx)
83 {
84     return;
85 }
86
87 static void RemoveFromQueue(MyBrcFrame* frame_queue, mfxU32 frame_queue_size,_
88   ↵MyBrcFrame* frame)
89 {
90     return;
91 }
92
93 static mfxU64 GetMaxFrameEncodingTime(MyBrcContext *ctx)
94 {
95     return 2;
96 }
97
98 mfxStatus MyBrcInit(mfxHDL pthis, mfxVideoParam* par) {
99     MyBrcContext* ctx = (MyBrcContext*)pthis;
100    mfxI32 QpBdOffset;
101    mfxExtCodingOption2* co2;
102    mfxI32 defaultQP;
103
104    if (!pthis || !par)
105        return MFX_ERR_NULL_PTR;
106
107    if (!IsParametersSupported(par))
108        return MFX_ERR_UNSUPPORTED;

```

(continues on next page)

(continued from previous page)

```

108     ctx->frame_queue_max_size = par->AsyncDepth;
109     ctx->frame_queue = (MyBrcFrame*)malloc(sizeof(MyBrcFrame) * ctx->frame_queue_
110     ↪max_size);
111
112     if (!ctx->frame_queue)
113         return MFX_ERR_MEMORY_ALLOC;
114
115     co2 = (mfxExtCodingOption2*)GetExtBuffer(par->ExtParam, par->NumExtParam, MFX_
116     ↪EXTBUFF_CODING_OPTION2);
117     QpBdOffset = (par->mfx.FrameInfo.BitDepthLuma > 8) ? (6 * (par->mfx.FrameInfo.
118     ↪BitDepthLuma - 8)) : 0;
119
120     ctx->max_qp[0] = (co2 && co2->MaxQPI) ? (co2->MaxQPI - QpBdOffset) : defaultQP;
121     ctx->min_qp[0] = (co2 && co2->MinQPI) ? (co2->MinQPI - QpBdOffset) : defaultQP;
122
123     ctx->max_qp[1] = (co2 && co2->MaxQPP) ? (co2->MaxQPP - QpBdOffset) : defaultQP;
124     ctx->min_qp[1] = (co2 && co2->MinQPP) ? (co2->MinQPP - QpBdOffset) : defaultQP;
125
126     ctx->max_qp[2] = (co2 && co2->MaxQPB) ? (co2->MaxQPB - QpBdOffset) : defaultQP;
127     ctx->min_qp[2] = (co2 && co2->MinQPB) ? (co2->MinQPB - QpBdOffset) : defaultQP;
128
129     // skipped initialization of other other BRC parameters
130
131     ctx->frame_queue_size = 0;
132
133     return MFX_ERR_NONE;
134 }
135
136 mfxStatus MyBrcReset(mfxHDL pthis, mfxVideoParam* par) {
137     MyBrcContext* ctx = (MyBrcContext*)pthis;
138
139     if (!pthis || !par)
140         return MFX_ERR_NULL_PTR;
141
142     if (!IsParametersSupported(par))
143         return MFX_ERR_UNSUPPORTED;
144
145     // reset here BRC parameters if required
146
147     return MFX_ERR_NONE;
148 }
149
150 mfxStatus MyBrcClose(mfxHDL pthis) {
151     MyBrcContext* ctx = (MyBrcContext*)pthis;
152
153     if (!pthis)
154         return MFX_ERR_NULL_PTR;
155
156     if (ctx->frame_queue) {
157         free(ctx->frame_queue);
158         ctx->frame_queue = NULL;
159         ctx->frame_queue_max_size = 0;
160         ctx->frame_queue_size = 0;
161     }

```

(continues on next page)

(continued from previous page)

```

162
163     return MFX_ERR_NONE;
164 }
165
166 mfxStatus MyBrcGetFrameCtrl(mfxHDL pthis, mfxBRCFParam* par, mfxBRCFctrl*ctrl) {
167     MyBrcContext* ctx = (MyBrcContext*)pthis;
168     MyBrcFrame* frame = NULL;
169     mfxU32 cost;
170
171     if (!pthis || !par || !ctrl)
172         return MFX_ERR_NULL_PTR;
173
174     if (par->NumRecode > 0)
175         frame = GetFrame(ctx->frame_queue, ctx->frame_queue_size, par->EncodedOrder);
176     else if (ctx->frame_queue_size < ctx->frame_queue_max_size)
177         frame = &ctx->frame_queue[ctx->frame_queue_size++];
178
179     if (!frame)
180         return MFX_ERR_UNDEFINED_BEHAVIOR;
181
182     if (par->NumRecode == 0) {
183         frame->EncodedOrder = par->EncodedOrder;
184         cost = GetFrameCost(par->FrameType, par->PyramidLayer);
185         frame->MinSize = GetMinSize(ctx, cost);
186         frame->MaxSize = GetMaxSize(ctx, cost);
187         frame->QP = GetInitQP(ctx, frame->MinSize, frame->MaxSize, cost); // from QP/size stat
188         frame->StartTime = GetTime();
189     }
190
191     ctrl->QpY = frame->QP;
192
193     return MFX_ERR_NONE;
194 }
195
196 #define DEFAULT_QP_INC 4
197 #define DEFAULT_QP_DEC 4
198
199 mfxStatus MyBrcUpdate(mfxHDL pthis, mfxBRCFParam* par, mfxBRCFctrl*ctrl,mfxBRCFstatus* status) {
200     MyBrcContext* ctx = (MyBrcContext*)pthis;
201     MyBrcFrame* frame = NULL;
202     mfxU32 panic = 0;
203
204     if (!pthis || !par || !ctrl || !status)
205         return MFX_ERR_NULL_PTR;
206
207     frame = GetFrame(ctx->frame_queue, ctx->frame_queue_size, par->EncodedOrder);
208     if (!frame)
209         return MFX_ERR_UNDEFINED_BEHAVIOR;
210
211     // update QP/size stat here
212
213     if ( frame->Status == MFX_BRC_PANIC_BIG_FRAME
214         || frame->Status == MFX_BRC_PANIC_SMALL_FRAME)
215         panic = 1;

```

(continues on next page)

(continued from previous page)

```

216
217     if (panic || (par->CodedFrameSize >= frame->MinSize && par->CodedFrameSize <=
218         →frame->MaxSize)) {
219         UpdateBRCState(par->CodedFrameSize, ctx);
220         RemoveFromQueue(ctx->frame_queue, ctx->frame_queue_size, frame);
221         ctx->frame_queue_size--;
222         status->BRCStatus = MFX_BRC_OK;
223
224         // Here update Min/MaxSize for all queued frames
225
226         return MFX_ERR_NONE;
227     }
228
229     panic = ((GetTime() - frame->StartTime) >= GetMaxFrameEncodingTime(ctx));
230
231     if (par->CodedFrameSize > frame->MaxSize) {
232         if (panic || (frame->QP >= ctx->max_qp[0])) {
233             frame->Status = MFX_BRC_PANIC_BIG_FRAME;
234         } else {
235             frame->Status = MFX_BRC_BIG_FRAME;
236             frame->QP = DEFAULT_QP_INC;
237         }
238     }
239
240     if (par->CodedFrameSize < frame->MinSize) {
241         if (panic || (frame->QP <= ctx->min_qp[0])) {
242             frame->Status = MFX_BRC_PANIC_SMALL_FRAME;
243             status->MinFrameSize = frame->MinSize;
244         } else {
245             frame->Status = MFX_BRC_SMALL_FRAME;
246             frame->QP = DEFAULT_QP_DEC;
247         }
248     }
249
250     status->BRCStatus = frame->Status;
251
252     return MFX_ERR_NONE;
253 }
254
255 void EncoderInit()
256 {
257     //initialize encoder
258     MyBrcContext brc_ctx;
259     mfxExtBRC ext_brc;
260     mfxExtCodingOption2 co2;
261     mfxExtBuffer* ext_buf[2] = {&co2.Header, &ext_brc.Header};
262     mfxVideoParam vpar;
263
264     memset(&brc_ctx, 0, sizeof(MyBrcContext));
265     memset(&ext_brc, 0, sizeof(mfxExtBRC));
266     memset(&co2, 0, sizeof(mfxExtCodingOption2));
267
268     vpar.ExtParam = ext_buf;
269     vpar.NumExtParam = sizeof(ext_buf) / sizeof(ext_buf[0]);
270
271     co2.Header.BufferId = MFX_EXTBUFF_CODING_OPTION2;
272     co2.Header.BufferSz = sizeof(mfxExtCodingOption2);

```

(continues on next page)

(continued from previous page)

```

272     co2.ExtBRC = MFX_CODINGOPTION_ON;
273
274     ext_brc.Header.BufferId = MFX_EXTBUFF_BRC;
275     ext_brc.Header.BufferSz = sizeof(mfxExtBRC);
276     ext_brc.pthis = &brc_ctx;
277     ext_brc.Init = MyBrcInit;
278     ext_brc.Reset = MyBrcReset;
279     ext_brc.Close = MyBrcClose;
280     ext_brc.GetFrameCtrl = MyBrcGetFrameCtrl;
281     ext_brc.Update = MyBrcUpdate;
282
283     sts = MFXVideoENCODE_Query(session, &vpar, &vpar);
284     if (sts == MFX_ERR_UNSUPPORTED || co2.ExtBRC != MFX_CODINGOPTION_ON)
285         // unsupported case
286     sts = sts;
287     else
288         sts = MFXVideoENCODE_Init(session, &vpar);
289 }
```

### 12.2.5.5 JPEG

The application can use the same encoding procedures for JPEG/motion JPEG encoding, as shown in the following pseudo code:

```

// encoder initialization
MFXVideoENCODE_Init (...);
// single frame/picture encoding
MFXVideoENCODE_EncodeFrameAsync (...);
MFXVideoCORE_SyncOperation(...);
// close down
MFXVideoENCODE_Close(...);
```

*ENCODE* supports JPEG baseline profile encoding as follows:

- DCT-based process
- Source image: 8-bit samples within each component
- Sequential
- Huffman coding: 2 AC and 2 DC tables
- 3 loadable quantization matrices
- Interleaved and non-interleaved scans
- Single and multiple scans
  - Chroma sub-sampling ratios:
    - \* Chroma 4:0:0 (gray image)
    - \* Chroma 4:1:1
    - \* Chroma 4:2:0
    - \* Chroma horizontal 4:2:2
    - \* Chroma vertical 4:2:2
    - \* Chroma 4:4:4

- 3 channel images

The application may specify Huffman and quantization tables during encoder initialization by attaching `mfxExtJPEGQuantTables` and `mfxExtJPEGHuffmanTables` buffers to the `mfxVideoParam` structure. If the application does not define tables then the SDK encoder uses tables recommended in ITU-T\* Recommendation T.81. If the application does not define a quantization table it must specify the `Quality` parameter in the `mfxInfoMF` structure. In this case, the SDK encoder scales the default quantization table according to the specified `Quality` parameter.

The application should properly configured chroma sampling format and color format using the `FourCC` and `ChromaFormat` fields in the `mfxFrameInfo` structure. For example, to encode a 4:2:2 vertically sampled YCbCr picture, the application should set `FourCC` to `MFX_FOURCC_YUY2` and `ChromaFormat` to `MFX_CHROMAFORMAT_YUV422V`. To encode a 4:4:4 sampled RGB picture, the application should set `FourCC` to `MFX_FOURCC_RGB4` and `ChromaFormat` to `MFX_CHROMAFORMAT_YUV444`.

The SDK encoder supports different sets of chroma sampling and color formats on different platforms. The application must call the `MFXVideoENCODE_Query()` function to check if the required color format is supported on a given platform and then initialize the encoder with proper values of `FourCC` and `ChromaFormat` in the `mfxFrameInfo` structure.

The application should not define the number of scans and number of components. These numbers are derived by the SDK encoder from the `Interleaved` flag in the `mfxInfoMF` structure and from chroma type. If interleaved coding is specified, then one scan is encoded that contains all image components. Otherwise, the number of scans is equal to number of components. The SDK encoder uses the following component IDs: “1” for luma (Y), “2” for chroma Cb (U), and “3” for chroma Cr (V).

The application should allocate a buffer that is big enough to hold the encoded picture. A rough upper limit may be calculated using the following equation where `Width` and `Height` are weight and height of the picture in pixel and `BytesPerPx` is the number of bytes for one pixel:

```
BufferSizeInKB = 4 + (Width * Height * BytesPerPx + 1023) / 1024;
```

The equation equals 1 for a monochrome picture, 1.5 for NV12 and YV12 color formats, 2 for YUY2 color format, and 3 for RGB32 color format (alpha channel is not encoded).

### 12.2.5.6 Multi-view Video Encoding

Similar to the decoding and video processing initialization procedures, the application attaches the `mfxExtMVCSeqDesc` structure to the `mfxVideoParam` structure for encoding initialization. The `mfxExtMVCSeqDesc` structure configures the SDK MVC encoder to work in three modes:

- **Default dependency mode:** The application specifies `NumView` and all other fields to zero. The SDK encoder creates a single operation point with all views (view identifier 0 : `NumView`-1) as target views. The first view (view identifier 0) is the base view. Other views depend on the base view.
- **Explicit dependency mode:** The application specifies `NumView` and the view dependency array, and sets all other fields to zero. The SDK encoder creates a single operation point with all views (view identifier `View[0 : NumView-1].ViewId`) as target views. The first view (view identifier `View[0].ViewId`) is the base view. View dependencies are defined as `mfxMVCViewDependency` structures.
- **Complete mode:** The application fully specifies the views and their dependencies. The SDK encoder generates a bitstream with corresponding stream structures.

The SDK MVC encoder does not support importing sequence and picture headers via the `mfxExtCodingOptionSPSPPS` structure.

During encoding, the SDK encoding function `MFXVideoENCODE_EncodeFrameAsync()` accumulates input frames until encoding of a picture is possible. The function returns `mfxStatus::MFX_ERR_MORE_DATA` for

more data at input or `mfxStatus::MFX_ERR_NONE` if it successfully accumulated enough data for encoding a picture. The generated bitstream contains the complete picture (multiple views). The application can change this behavior and instruct the encoder to output each view in a separate bitstream buffer. To do so, the application must turn on the `ViewOutput` flag in the `mfxExtCodingOption` structure. In this case, the encoder returns `mfxStatus::MFX_ERR_MORE_BITSTREAM` if it needs more bitstream buffers at output and `mfxStatus::MFX_ERR_NONE` when processing of the picture (multiple views) has been finished. It is recommended that the application provide a new input frame each time the SDK encoder requests new bitstream buffer. The application must submit view data for encoding in the order they are described in the `mfxExtMVCSeqDesc` structure. Particular view data can be submitted for encoding only when all views that it depends upon have already been submitted.

The following pseudo code shows the encoding procedure:

```

1 mfxExtBuffer *eb;
2 mfxExtMVCSeqDesc seq_desc;
3 mfxVideoParam init_param;
4
5 init_param.ExtParam=(mfxExtBuffer **) &eb;
6 init_param.NumExtParam=1;
7 eb=(mfxExtBuffer *) &seq_desc;
8
9 /* init encoder */
10 MFXVideoENCODE_Init(session, &init_param);
11
12 /* perform encoding */
13 for (;;) {
14     MFXVideoENCODE_EncodeFrameAsync(session, NULL, surface2, bits,
15                                     &syncp);
16     MFXVideoCORE_SyncOperation(session, syncp, INFINITE);
17 }
18
19 /* close encoder */
20 MFXVideoENCODE_Close(session);

```

## 12.2.6 Video Processing Procedures

The following example shows the pseudo code of the video processing procedure:

```

1 MFXVideoVPP_QueryIOSurf(session, &init_param, response);
2 allocate_pool_of_surfaces(in_pool, response[0].NumFrameSuggested);
3 allocate_pool_of_surfaces(out_pool, response[1].NumFrameSuggested);
4 MFXVideoVPP_Init(session, &init_param);
5 mfxFrameSurface1 *in=find_unlocked_surface_and_fill_content(in_pool);
6 mfxFrameSurface1 *out=find_unlocked_surface_from_the_pool(out_pool);
7 for (;;) {
8     sts=MFXVideoVPP_RunFrameVPPAsync(session,in,out,NULL,&syncp);
9     if (sts==MFX_ERR_MORE_SURFACE || sts==MFX_ERR_NONE) {
10         MFXVideoCORE_SyncOperation(session, syncp, INFINITE);
11         process_output_frame(out);
12         out=find_unlocked_surface_from_the_pool(out_pool);
13     }
14     if (sts==MFX_ERR_MORE_DATA && in==NULL)
15         break;
16     if (sts==MFX_ERR_NONE || sts==MFX_ERR_MORE_DATA) {
17         in=find_unlocked_surface_from_the_pool(in_pool);
18         fill_content_for_video_processing(in);

```

(continues on next page)

(continued from previous page)

```

19     if (end_of_stream())
20         in=NULL;
21     }
22 }
23 MFXVideoVPP_Close(session);
24 free_pool_of_surfaces(in_pool);
25 free_pool_of_surfaces(out_pool);

```

Note the following key points about the example:

- The application uses the `MFXVideoVPP_QueryIOSurf()` function to obtain the number of frame surfaces needed for input and output. The application must allocate two frame surface pools: one for the input and one for the output.
- The video processing function `MFXVideoVPP_RunFrameVPPAsync()` is asynchronous. The application must use the `MFXVideoCORE_SyncOperation()` function to synchronize to make the output result ready.
- The body of the video processing procedure covers the following three scenarios:
  - If the number of frames consumed at input is equal to the number of frames generated at output, VPP returns `mfxStatus::MFX_ERR_NONE` when an output is ready. The application must process the output frame after synchronization, as the `MFXVideoVPP_RunFrameVPPAsync()` function is asynchronous. The application must provide a NULL input at the end of the sequence to drain any remaining frames.
  - If the number of frames consumed at input is more than the number of frames generated at output, VPP returns `mfxStatus::MFX_ERR_MORE_DATA` for additional input until an output is ready. When the output is ready, VPP returns `mfxStatus::MFX_ERR_NONE`. The application must process the output frame after synchronization and provide a NULL input at the end of the sequence to drain any remaining frames.
  - If the number of frames consumed at input is less than the number of frames generated at output, VPP returns either `mfxStatus::MFX_ERR_MORE_SURFACE` (when more than one output is ready), or `mfxStatus::MFX_ERR_NONE` (when one output is ready and VPP expects new input). In both cases, the application must process the output frame after synchronization and provide a NULL input at the end of the sequence to drain any remaining frames.

### 12.2.6.1 Configuration

The SDK configures the video processing pipeline operation based on the difference between the input and output formats, specified in the `mfxVideoParam` structure. The following list shows several examples:

- When the input color format is `YUY2` and the output color format is `NV12`, the SDK enables color conversion from YUY2 to NV12.
- When the input is interleaved and the output is progressive, the SDK enables deinterlacing.
- When the input is single field and the output is interlaced or progressive, the SDK enables field weaving, optionally with deinterlacing.
- When the input is interlaced and the output is single field, the SDK enables field splitting.

In addition to specifying the input and output formats, the application can provide hints to fine-tune the video processing pipeline operation. The application can disable filters in the pipeline by using the `mfxExtVPPDoNotUse` structure, enable filters by using the `mfxExtVPPDoUse` structure, and configure filters by using dedicated configuration structures. See the *Configurable VPP Filters table* for a complete list of configurable video processing filters, their IDs, and configuration structures. See the *ExtendedBufferID enumerator* for more details.

The SDK ensures that all filters necessary to convert the input format to the output format are included in the pipeline. The SDK may skip some optional filters even if they are explicitly requested by the application, for example, due to limitations of the underlying hardware. To notify the application about skipped optional filters, the SDK returns the `mfxStatus::MFX_WRN_FILTER_SKIPPED` warning. The application can retrieve the list of active filters by attaching the `mfxExtVPDose` structure to the `mfxVideoParam` structure and calling the `MFXVideoVPP_GetVideoParam()` function. The application must allocate enough memory for the filter list.

See the *Configurable VPP Filters table* for a full list of configurable filters.

Table 5: Configurable VPP Filters

Filter ID	Configuration Structure
MFX_EXTBUFF_VPP_DENOISE	<code>mfxExtVPDPnoise</code>
MFX_EXTBUFF_VPP_MCTF	<code>mfxExtVppMctf</code>
MFX_EXTBUFF_VPP_DETAIL	<code>mfxExtVPPDetail</code>
MFX_EXTBUFF_VPP_FRAME_RATE_CONVERSION	<code>mfxExtVPPFrameRateConversion</code>
MFX_EXTBUFF_VPP_IMAGE_STABILIZATION	<code>mfxExtVPPImageStab</code>
MFX_EXTBUFF_VPP_PICSTRUCT_DETECTION	none
MFX_EXTBUFF_VPP_PROCAMP	<code>mfxExtVPPProcAmp</code>
MFX_EXTBUFF_VPP_FIELD_PROCESSING	<code>mfxExtVPPFieldProcessing</code>

The following example shows video processing configuration:

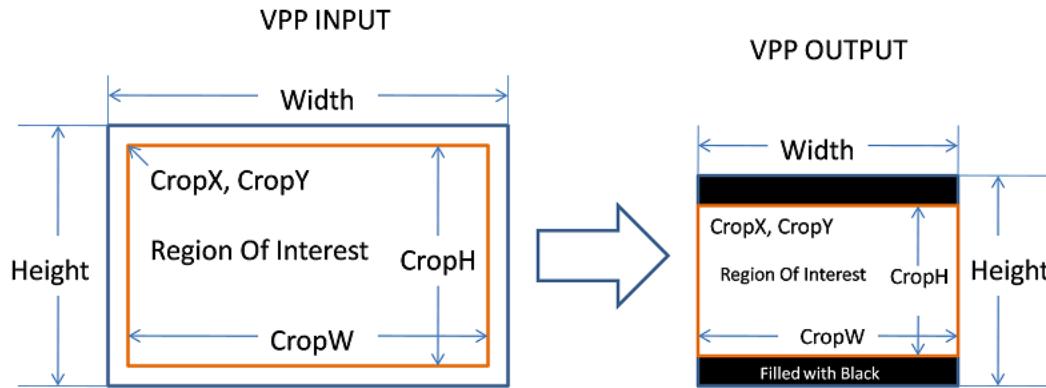
```

1  /* enable image stabilization filter with default settings */
2  mfxExtVPDose du;
3  mfxU32 al=MFX_EXTBUFF_VPP_IMAGE_STABILIZATION;
4
5  du.Header.BufferId=MFX_EXTBUFF_VPP_DOUSE;
6  du.Header.BufferSz=sizeof(mfxExtVPDose);
7  du.NumAlg=1;
8  du.AlgList=&al;
9
10 /* configure the mfxVideoParam structure */
11 mfxVideoParam conf;
12 mfxExtBuffer *eb=(mfxExtBuffer *)&du;
13
14 memset(&conf,0,sizeof(conf));
15 conf.IOPattern=MFX_IOPATTERN_IN_SYSTEM_MEMORY | MFX_IOPATTERN_OUT_SYSTEM_MEMORY;
16 conf.NumExtParam=1;
17 conf.ExtParam=&eb;
18
19 conf.vpp.In.FourCC=MFX_FOURCC_YV12;
20 conf.vpp.Out.FourCC=MFX_FOURCC_NV12;
21 conf.vpp.In.Width=conf.vpp.Out.Width=1920;
22 conf.vpp.In.Height=conf.vpp.Out.Height=1088;
23
24 /* video processing initialization */
25 MFXVideoVPP_Init(session, &conf);

```

### 12.2.6.2 Region of Interest

During video processing operations, the application can specify a region of interest for each frame as shown in the following image:



Specifying a region of interest guides the resizing function to achieve special effects, such as resizing from 16:9 to 4:3, while keeping the aspect ratio intact. Use the `CropX`, `CropY`, `CropW`, and `CropH` parameters in the `mfxVideoParam` structure to specify a region of interest.

The *VPP Region of Interest Operations table* shows examples of VPP operations applied to a region of interest.

Table 6: VPP Region of Interest Operations

Operation	VPP Input Width x Height	VPP Input <code>CropX, CropY, CropW, CropH</code>	VPP Output Width x Height	VPP Output <code>CropX, CropY, CropW, CropH</code>
Cropping	720 x 480	16, 16, 688, 448	720 x 480	16, 16, 688, 448
Resizing	720 x 480	0, 0, 720, 480	1440 x 960	0, 0, 1440, 960
Horizontal stretching	720 x 480	0, 0, 720, 480	640 x 480	0, 0, 640, 480
16:9 4:3 with letter boxing at the top and bottom	1920 x 1088	0, 0, 1920, 1088	720 x 480	0, 36, 720, 408
4:3 16:9 with pillar boxing at the left and right	720 x 480	0, 0, 720, 480	1920 x 1088	144, 0, 1632, 1088

### 12.2.6.3 Multi-view Video Processing

SDK video processing supports processing multiple views. For video processing initialization, the application needs to attach the `mfxExtMVCSeqDesc` structure to the `mfxVideoParam` structure and call the `MFXVideoVPP_Init()` function. The function saves the view identifiers. During video processing, the SDK processes each view individually. The SDK refers to the FrameID field of the `mfxFrameInfo` structure to configure each view according to its processing pipeline. If the video processing source frame is not the output from the SDK MVC decoder, then the application needs to fill the the FrameID field before calling the `MFXVideoVPP_RunFrameVPPAsync()` function. This is shown in the following pseudo code:

```

1 mfxExtBuffer *eb;
2 mfxExtMVCSeqDesc seq_desc;
3 mfxVideoParam init_param;
4

```

(continues on next page)

(continued from previous page)

```

5 init_param.ExtParam = &eb;
6 init_param.NumExtParam=1;
7 eb=(mfxExtBuffer *)&seq_desc;
8
9 /* init VPP */
10 MFXVideoVPP_Init(session, &init_param);
11
12 /* perform processing */
13 for (;;) {
14     MFXVideoVPP_RunFrameVPPAsync(session,in,out,NULL,&syncp);
15     MFXVideoCORE_SyncOperation(session, syncp, INFINITE);
16 }
17
18 /* close VPP */
19 MFXVideoVPP_Close(session);

```

## 12.2.7 Transcoding Procedures

The application can use the SDK encoding, decoding, and video processing functions together for transcoding operations. This section describes the key aspects of connecting two or more SDK functions together.

### 12.2.7.1 Asynchronous Pipeline

The application passes the output of an upstream SDK function to the input of the downstream SDK function to construct an asynchronous pipeline. Pipeline construction is done at runtime and can be dynamically changed, as shown in the following example:

```

1 mfxSyncPoint sp_d, sp_e;
2 MFXVideoDECODE_DecodeFrameAsync(session,bs,work,&vin, &sp_d);
3 if (going_through_vpp) {
4     MFXVideoVPP_RunFrameVPPAsync(session,vin,vout, NULL, &sp_d);
5     MFXVideoENCODE_EncodeFrameAsync(session,NULL,vout,bits2,&sp_e);
6 } else {
7     MFXVideoENCODE_EncodeFrameAsync(session,NULL,vin,bits2,&sp_e);
8 }
9 MFXVideoCORE_SyncOperation(session, sp_e, INFINITE);

```

The SDK simplifies the requirements for asynchronous pipeline synchronization. The application only needs to synchronize after the last SDK function. Explicit synchronization of intermediate results is not required and may slow performance.

The SDK tracks dynamic pipeline construction and verifies dependency on input and output parameters to ensure the execution order of the pipeline function. In the previous example, the SDK will ensure `MFXVideoENCODE_EncodeFrameAsync()` does not begin its operation until `MFXVideoDECODE_DecodeFrameAsync()` or `MFXVideoVPP_RunFrameVPPAsync()` has finished.

During the execution of an asynchronous pipeline, the application must consider the input data as ‘in use’ and must not change it until the execution has completed. The application must also consider output data unavailable until the execution has finished. In addition, for encoders, the application must consider extended and payload buffers as ‘in use’ while the input surface is locked.

The SDK checks dependencies by comparing the input and output parameters of each SDK function in the pipeline. Do not modify the contents of input and output parameters before the previous asynchronous operation finishes. Doing

so will break the dependency check and can result in undefined behavior. An exception occurs when the input and output parameters are structures, in which case overwriting fields in the structures is allowed.

---

**Note:** The dependency check works on the pointers to the structures only.

---

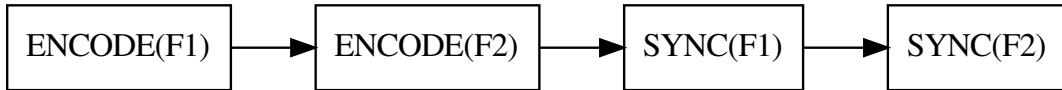
There are two exceptions with respect to intermediate synchronization:

- If the input is from any asynchronous operation, the application must synchronize any input before calling the SDK [MFVideoDECODE\\_DecodeFrameAsync\(\)](#) function.
- When the application calls an asynchronous function to generate an output surface in video memory and passes that surface to a non-SDK component, it must explicitly synchronize the operation before passing the surface to the non-SDK component.

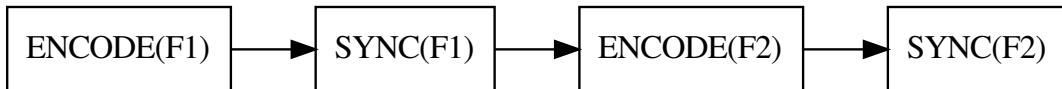
#### 12.2.7.2 Surface Pool Allocation

When connecting SDK function **A** to SDK function **B**, the application must take into account the requirements of both functions to calculate the number of frame surfaces in the surface pool. Typically, the application can use the formula **Na+Nb**, where **Na** is the frame surface requirements for SDK function **A** output, and **Nb** is the frame surface requirements for SDK function **B** input.

For performance considerations, the application must submit multiple operations and delay synchronization as much as possible, which gives the SDK flexibility to organize internal pipelining. For example, the operation sequence:



is recommended, compared with:



In this example, the surface pool needs additional surfaces to take into account multiple asynchronous operations before synchronization. The application can use the `AsyncDepth` parameter of the [mfxVideoParam](#) structure to inform an SDK function of the number of asynchronous operations the application plans to perform before synchronization. The corresponding SDK `QueryIOSurf` function will reflect this number in the `NumFrameSuggested` value. The following example shows a way of calculating the surface needs based on `NumFrameSuggested` values:

```

1 mfxVideoParam init_param_v, init_param_e;
2 mfxFrameAllocRequest response_v[2], response_e;
3
4 // Desired depth
  
```

(continues on next page)

(continued from previous page)

```

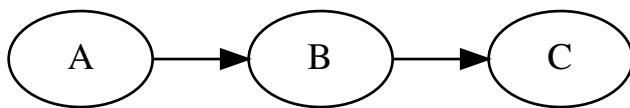
5 mfxU16 async_depth=4;
6
7 init_param_v.AsyncDepth=async_depth;
8 MFXVideoVPP_QueryIOSurf(session, &init_param_v, response_v);
9 init_param_e.AsyncDepth=async_depth;
10 MFXVideoENCODE_QueryIOSurf(session, &init_param_e, &response_e);
11 mfxU32 num_surfaces=    response_v[1].NumFrameSuggested
12         +response_e.NumFrameSuggested
13         -async_depth; /* double counted in ENCODE & VPP */

```

### 12.2.7.3 Pipeline Error Reporting

During asynchronous pipeline construction, each stage SDK function will return a synchronization point (sync point). These synchronization points are useful in tracking errors during the asynchronous pipeline operation.

For example, assume the following pipeline:



The application synchronizes on sync point **C**. If the error occurs in SDK function **C**, then the synchronization returns the exact error code. If the error occurs before SDK function **C**, then the synchronization returns `mfxStatus::MFX_ERR_ABORTED`. The application can then try to synchronize on sync point **B**. Similarly, if the error occurs in SDK function **B**, the synchronization returns the exact error code, or else `mfxStatus::MFX_ERR_ABORTED`. The same logic applies if the error occurs in SDK function **A**.

### 12.2.8 Working with Hardware Acceleration

#### 12.2.8.1 Working with Multiple Media Devices

If your system has multiple graphics adapters you may need hints on which adapter is better suited to process a particular workload. The SDK provides the helper API to select the most suitable adapter for your workload based on the provide workload description. The following example shows workload initialization on a discrete adapter:

```

1 mfxU32 num_adapters_available;
2 mfxIMPL impl;
3
4 // Query number of graphics adapters available on system
5 mfxStatus sts = MFXQueryAdaptersNumber(&num_adapters_available);
6 MSDK_CHECK_STATUS(sts, "MFXQueryAdaptersNumber failed");
7
8 // Allocate memory for response
9 std::vector<mfxAdapterInfo> displays_data(num_adapters_available);
10 mfxAdaptersInfo adapters = { displays_data.data(), mfxU32(displays_data.size()), 0 };
11

```

(continues on next page)

(continued from previous page)

```

12 // Query information about all adapters (mind that first parameter is NULL)
13 sts = MFXQueryAdapters(nullptr, &adapters);
14 MSDK_CHECK_STATUS(sts, "MFXQueryAdapters failed");
15
16 // Find dGfx adapter in list of adapters
17 auto idx_d = std::find_if(adapters.Adapters, adapters.Adapters + adapters.NumActual,
18 [] (const mfxAdapterInfo info)
19 {
20     return info.Platform.MediaAdapterType == mfxMediaAdapterType::MFX_MEDIA_DISCRETE;
21 });
22
23 // No dGfx in list
24 if (idx_d == adapters.Adapters + adapters.NumActual)
25 {
26     printf("Warning: No dGfx detected on machine\n");
27     return -1;
28 }
29
30 mfxU32 idx = static_cast<mfxU32>(std::distance(adapters.Adapters, idx_d));
31
32 // Choose correct implementation for discrete adapter
33 switch (adapters.Adapters[idx].Number)
34 {
35 case 0:
36     impl = MFX_IMPL_HARDWARE;
37     break;
38 case 1:
39     impl = MFX_IMPL_HARDWARE2;
40     break;
41 case 2:
42     impl = MFX_IMPL_HARDWARE3;
43     break;
44 case 3:
45     impl = MFX_IMPL_HARDWARE4;
46     break;
47
48 default:
49     // Try searching on all display adapters
50     impl = MFX_IMPL_HARDWARE_ANY;
51     break;
52 }
53
54 // Initialize mfxSession in regular way with obtained implementation

```

The example shows that after obtaining the adapter list with `MFXQueryAdapters()`, further initialization of `mfxSession` is performed in the regular way. The specific adapter selected using the `MFX_IMPL_HARDWARE`, `MFX_IMPL_HARDWARE2`, `MFX_IMPL_HARDWARE3`, or `MFX_IMPL_HARDWARE4` values of `mfxIMPL`.

The following example shows the use of `MFXQueryAdapters()` for querying the most suitable adapter for a particular encode workload:

```

1 mfxU32 num_adapters_available;
2 mfxIMPL impl;
3 mfxVideoParam Encode_mfxVideoParam;
4
5 // Query number of graphics adapters available on system
6 mfxStatus sts = MFXQueryAdaptersNumber(&num_adapters_available);

```

(continues on next page)

(continued from previous page)

```

7  MSDK_CHECK_STATUS(sts, "MFXQueryAdaptersNumber failed");
8
9  // Allocate memory for response
10 std::vector<mfxAdapterInfo> displays_data(num_adapters_available);
11 mfxAdaptersInfo adapters = { displays_data.data(), mfxU32(displays_data.size()), 0u };
12
13 // Fill description of Encode workload
14 mfxComponentInfo interface_request = { MFX_COMPONENT_ENCODE, Encode_mfxVideoParam };
15
16 // Query information about suitable adapters for Encode workload described by Encode_
17 // mfxVideoParam
18 sts = MFXQueryAdapters(&interface_request, &adapters);
19
20 if (sts == MFX_ERR_NOT_FOUND)
21 {
22     printf("Error: No adapters on machine capable to process desired workload\n");
23     return -1;
24 }
25
26 MSDK_CHECK_STATUS(sts, "MFXQueryAdapters failed");
27
28 // Choose correct implementation for discrete adapter. Mind usage of index 0, this is_
29 // best suitable adapter from MSDK perspective
30 switch (adapters.Adapters[0].Number)
31 {
32 case 0:
33     impl = MFX_IMPL_HARDWARE;
34     break;
35 case 1:
36     impl = MFX_IMPL_HARDWARE2;
37     break;
38 case 2:
39     impl = MFX_IMPL_HARDWARE3;
40     break;
41 case 3:
42     impl = MFX_IMPL_HARDWARE4;
43     break;
44 default:
45     // Try searching on all display adapters
46     impl = MFX_IMPL_HARDWARE_ANY;
47     break;
48 }
49
50 // Initialize mfxSession in regular way with obtained implementation

```

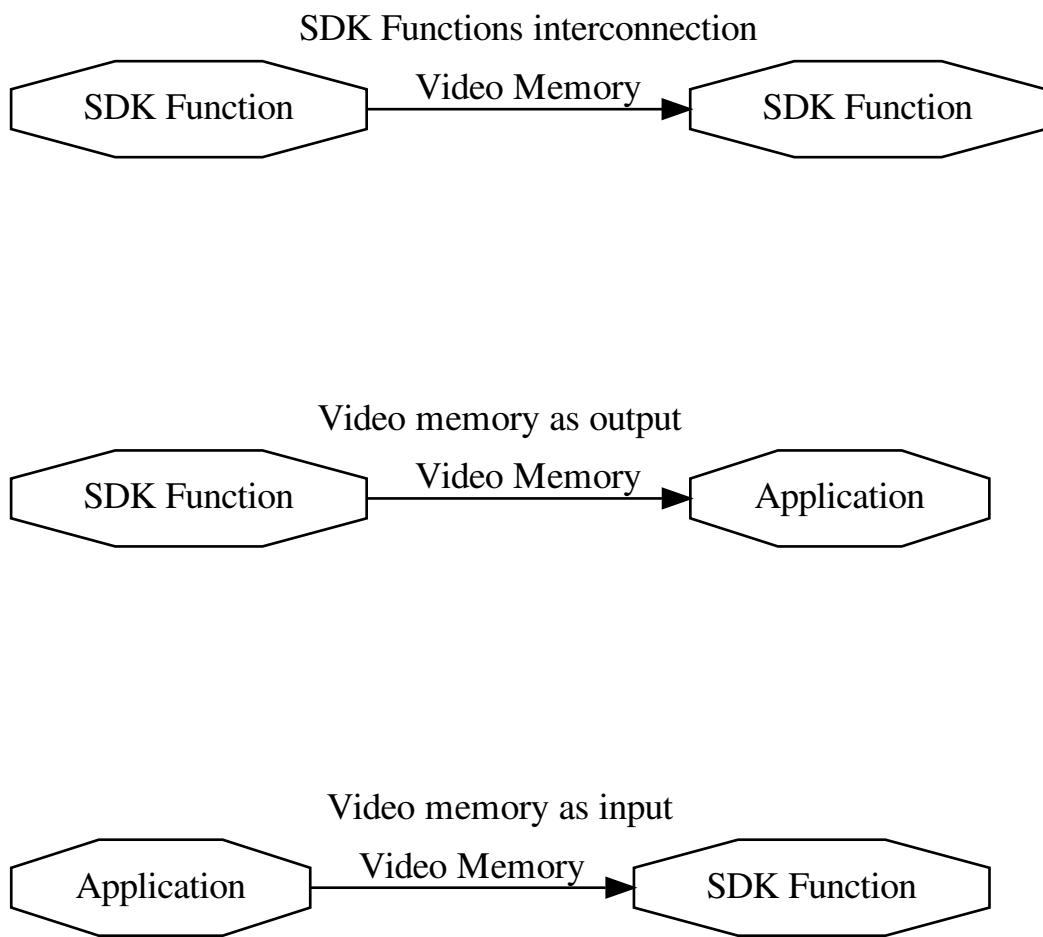
See the [MFXQueryAdapters\(\)](#) description for adapter priority rules.

### 12.2.8.2 Working with Video Memory

To fully utilize the SDK acceleration capability, the application should support OS specific infrastructures. If using Microsoft\* Windows\*, the application should support Microsoft DirectX\*. If using Linux\*, the application should support the VA API for Linux.

The hardware acceleration support in an application consists of video memory support and acceleration device support.

Depending on the usage model, the application can use video memory at different stages in the pipeline. Three major scenarios are shown in the following illustration:



The application must use the `IOPattern` field of the `mfxVideoParam` structure to indicate the I/O access pattern during initialization. Subsequent SDK function calls must follow this access pattern. For example, if an SDK function operates on video memory surfaces at both input and output, the application must specify the access pattern `IOPattern` at initialization in `MFX_IOPATTERN_IN_VIDEO_MEMORY` for input and `MFX_IOPATTERN_OUT_VIDEO_MEMORY` for output. This particular I/O access pattern must not change inside the **Init - Close** sequence.

Initialization of any hardware accelerated SDK component requires the acceleration device handle. This handle is also

used by the SDK component to query hardware capabilities. The application can share its device with the SDK by passing the device handle through the `MFXVideoCORE_SetHandle()` function. It is recommended to share the handle before any actual usage of the SDK.

### 12.2.8.3 Working with Microsoft DirectX\* Applications

The SDK supports two different infrastructures for hardware acceleration on the Microsoft Windows OS: Direct3D\* 9 DXVA2 and Direct3D 11 Video API. If Direct3D 9 DXVA2 is used for hardware acceleration, the application should use the IDirect3DDeviceManager9 interface as the acceleration device handle. If the Direct3D 11 Video API is used for hardware acceleration, the application should use the ID3D11Device interface as the acceleration device handle.

The application should share one of these interfaces with the SDK through the `MFXVideoCORE_SetHandle()` function. If the application does not provide the interface, then the SDK creates its own internal acceleration device. This internal device is not accessible by the application and as a result, the SDK input and output will be limited to system memory only. This will reduce SDK performance. If the SDK fails to create a valid acceleration device, then the SDK cannot proceed with hardware acceleration and returns an error status to the application.

The application must create the Direct3D 9 device with the flag `D3DCREATE_MULTITHREADED`. The flag `D3DCREATE_FPU_PRESERVE` is also recommended. This influences floating-point calculations, including PTS values.

The application must also set multi-threading mode for the Direct3D 11 device. The following example shows how to set multi-threading mode for a Direct3D 11 device:

```

1 ID3D11Device           *pD11Device;
2 ID3D11DeviceContext    *pD11Context;
3 ID3D10Multithread      *pD10Multithread;
4
5 pD11Device->GetImmediateContext(&pD11Context);
6 pD11Context->QueryInterface(IID_ID3D10Multithread, &pD10Multithread);
7 pD10Multithread->SetMultithreadProtected(true);

```

During hardware acceleration, if a Direct3D “device lost” event occurs, the SDK operation terminates with the `mfxStatus::MFX_ERR_DEVICE_LOST` return status. If the application provided the Direct3D device handle, the application must reset the Direct3D device.

When the SDK decoder creates auxiliary devices for hardware acceleration, it must allocate the list of Direct3D surfaces for I/O access, also known as the surface chain, and pass the surface chain as part of the device creation command. In most cases, the surface chain is the frame surface pool mentioned in the *Frame Surface Locking* section.

The application passes the surface chain to the SDK component Init function through an SDK external allocator callback. See the *Memory Allocation and External Allocators* section for details.

Only the decoder Init function requests the external surface chain from the application and uses it for auxiliary device creation. Encoder and VPP Init functions may only request internal surfaces. See the *ExtMemFrameType enumerator* for more details about different memory types.

Depending on configuration parameters, the SDK requires different surface types. It is strongly recommended to call the `MFXVideoENCODE_QueryIOSurf()` function, `MFXVideoDECODE_QueryIOSurf()` function, or `MFXVideoVPP_QueryIOSurf()` function to determine the appropriate type.

See the *SDK Support for Direct3D 9 Surface Types and Color Formats table* for detailed information about Direct3D 9 support.

Table 7: SDK Support for Direct3D 9 Surface Types and Color Formats

Class	Input Surface Type	Input Color Format	Output Surface Type	Output Color Format
DECODE	Not Applicable	Not Applicable	Decoder Render Target	NV12
DECODE (JPEG)			Decoder Render Target	RGB32, YUY2
VPP	Decoder/Processor Render Target	Listed in Color-FourCC	Decoder Render Target	NV12
VPP			Processor Render Target	RGB32
ENCODE	Decoder Render Target	NV12	Not Applicable	Not Applicable
ENCODE (JPEG)	Decoder Render Target	RGB32, YUY2, YV12		

**Note:** In table SDK Support for Direct3D 9 Surface Types and Color Formats, “Decoder Render Target” corresponds to the `DXVA2_VideoDecoderRenderTarget` type and “Processor Render Target” corresponds to `DXVA2_VideoProcessorRenderTarget`.

See the [SDK Support for Direct3D 11 Surface Types and Color Formats table](#) for detailed information about Direct3D 11 support.

Table 8: SDK Support for Direct3D 11 Surface Types and Color Formats

Class	Input Surface Type	Input Color Format	Output Surface Type	Output Color Format
DECODE	Not Applicable	Not Applicable	Decoder Render Target	NV12
DECODE (JPEG)			Decoder/Processor Render Target	RGB32, YUY2
VPP	Decoder/Processor Render Target	Listed in Color-FourCC	Processor Render Target	NV12
VPP			Processor Render Target	RGB32
ENCODE	Decoder/Processor Render Target	NV12	Not Applicable	Not Applicable
ENCODE (JPEG)	Decoder/Processor Render Target	RGB32, YUY2		

**Note:** In table SDK Support for Direct3D 11 Surface Types and Color Formats, “Decoder Render Target” corresponds to the `D3D11_BIND_DECODER` flag and “Processor Render Target” corresponds to `D3D11_BIND_RENDER_TARGET`.

**Note:** Note the following encoding and decoding color format support:

- NV12 is the major encoding and decoding color format.
- The JPEG/MJPEG decoder supports RGB32 and YUY2 output.
- The JPEG/MJPEG encoder supports RGB32 and YUY2 input for Direct3D 9 and Direct3D 11 and YV12 input for Direct3D 9 only.
- VPP supports RGB32 output.

#### 12.2.8.4 Working with VA API Applications

The SDK supports the VA API infrastructure for hardware acceleration on Linux. The application should use the `VADisplay` interface as the acceleration device handle for this infrastructure and share it with the SDK through the `MFXVideoCORE_SetHandle()` function. Because the SDK does not create an internal acceleration device on Linux, the application must always share it with the SDK. This sharing should be done before any actual usage of the SDK, including capability query and component initialization. If the application fails to share the device, the SDK operation will fail.

The following example shows how to obtain the VA display from X Window System:

```

1 Display *x11_display;
2 VADisplay va_display;
3
4 x11_display = XOpenDisplay(current_display);
5 va_display = vaGetDisplay(x11_display);
6
7 MFXVideoCORE_SetHandle(session, MFX_HANDLE_VA_DISPLAY, (mfxHDL) va_display);

```

The following example shows how to obtain the VA display from the Direct Rendering Manager:

```

1 int card;
2 VADisplay va_display;
3
4 card = open("/dev/dri/card0", O_RDWR); /* primary card */
5 va_display = vaGetDisplayDRM(card);
6 vaInitialize(va_display, &major_version, &minor_version);
7
8 MFXVideoCORE_SetHandle(session, MFX_HANDLE_VA_DISPLAY, (mfxHDL) va_display);

```

When the SDK decoder creates a hardware acceleration device, it must allocate the list of video memory surfaces for I/O access, also known as the surface chain, and pass the surface chain as part of the device creation command. The application passes the surface chain to the SDK component Init function through an SDK external allocator callback. See the [Memory Allocation and External Allocators](#) section for details.

Only the decoder Init function requests external surface chain from the application and uses it for device creation. Encoder and VPP Init functions may only request internal surfaces. See the [ExtMemFrameType enumerator](#) for more details about different memory types.

---

**Note:** The VA API does not define any surface types and the application can use either `MFX_MEMTYPE_VIDEO_MEMORY_DECODER_TARGET` or `MFX_MEMTYPE_VIDEO_MEMORY_PROCESSOR_TARGET` to indicate data in video memory.

---

See the [SDK Support for VA API Surface Types and Color Formats table](#) for detailed information about VA API support.

Table 9: SDK Support for VA API Surface Types and Color Formats

SDK Class	SDK Function Input	SDK Function Output
DECODE	Not Applicable	NV12
DECODE (JPEG)		RGB32, YUY2
VPP	Listed in ColorFourCC	NV12, RGB32
ENCODE	NV12	Not Applicable
ENCODE (JPEG)	RGB32, YUY2, YV12	

## 12.2.9 Memory Allocation and External Allocators

There are two models of memory management in the SDK implementation: internal and external.

### 12.2.9.1 External Memory Management

In the external memory model the application must allocate sufficient memory for input and output parameters and buffers, and deallocate it when SDK functions complete their operations. During execution, the SDK functions use callback functions to the application to manage memory for video frames through the external allocator interface `mfxFrameAllocator`.

If an application needs to control the allocation of video frames, it can use callback functions through the `mfxFrameAllocator` interface. If an application does not specify an allocator, an internal allocator is used. However, if an application uses video memory surfaces for input and output, it must specify the hardware acceleration device and an external frame allocator using `mfxFrameAllocator`.

The external frame allocator can allocate different frame types:

- In-system memory.
- In-video memory, as ‘decoder render targets’ or ‘processor render targets.’ See [Working with Hardware Acceleration](#) for additional details.

The external frame allocator responds only to frame allocation requests for the requested memory type and returns `mfxStatus::MFX_ERR_UNSUPPORTED` for all other types. The allocation request uses flags, part of the memory type field, to indicate which SDK class initiated the request, so the external frame allocator can respond accordingly.

Simple external frame allocator:

```

1 #define ALIGN32(X) (((mfxU32)((X)+31)) & (~(mfxU32)31))

2
3 typedef struct {
4     mfxU16 width, height;
5     mfxU8 *base;
6 } mid_struct;
7
8 mfxStatus fa_alloc(mfxHDL pthis, mfxFrameAllocRequest *request, mfxFrameAllocResponse↳*response) {
9     if (! (request->Type&MFX_MEMTYPE_SYSTEM_MEMORY))
10    return MFX_ERR_UNSUPPORTED;
11    if (request->Info.FourCC!=MFX_FOURCC_NV12)
12    return MFX_ERR_UNSUPPORTED;
13    response->NumFrameActual=request->NumFrameMin;
14    for (int i=0;i<request->NumFrameMin;i++) {
15        mid_struct *mmid=(mid_struct *)malloc(sizeof(mid_struct));
16        mmid->width=ALIGN32(request->Info.Width);
17        mmid->height=ALIGN32(request->Info.Height);
18        mmid->base=(mfxU8*)malloc(mmid->width*mmid->height*3/2);
19        response->mids[i]=mmid;
20    }
21    return MFX_ERR_NONE;
22 }

23
24 mfxStatus fa_lock(mfxHDL pthis, mfxMemId mid, mfxFrameData *ptr) {
25     mid_struct *mmid=(mid_struct *)mid;
26     ptr->Pitch=mmid->width;
27     ptr->Y=mmid->base;
28     ptr->U=ptr->Y+mmid->width*mmid->height;

```

(continues on next page)

(continued from previous page)

```

29     ptr->V=ptr->U+1;
30     return MFX_ERR_NONE;
31 }
32
33 mfxStatus fa_unlock(mfxHDL pthis, mfxMemId mid, mfxFrameData *ptr) {
34     if (ptr) ptr->Y=ptr->U=ptr->V=ptr->A=0;
35     return MFX_ERR_NONE;
36 }
37
38 mfxStatus fa_gethdl(mfxHDL pthis, mfxMemId mid, mfxHDL *handle) {
39     return MFX_ERR_UNSUPPORTED;
40 }
41
42 mfxStatus fa_free(mfxHDL pthis, mfxFrameAllocResponse *response) {
43     for (int i=0;i<response->NumFrameActual;i++) {
44         mid_struct *mmid=(mid_struct *)response->mids[i];
45         free(mmid->base); free(mmid);
46     }
47     return MFX_ERR_NONE;
48 }
```

For system memory, it is highly recommended to allocate memory for all planes of the same frame as a single buffer (using one single malloc call).

### 12.2.9.2 Internal Memory Management

In the internal memory management model, the SDK provides interface functions for frames allocation:

- *MFXMemory\_GetSurfaceForVPP ()*
- *MFXMemory\_GetSurfaceForEncode ()*
- *MFXMemory\_GetSurfaceForDecode ()*

These functions are used together with *mfxFrameSurfaceInterface* for surface management. The surface returned by these function is a reference counted object and the application must call *mfxFrameSurfaceInterface::Release* after finishing all operations with the surface. In this model the application doesn't need to create and set the external allocator to the SDK. Another method to obtain an internally allocated surface is to call *MFXVideoDECODE\_DecodeFrameAsync ()* with a working surface equal to NULL (see *Simplified decoding procedure*). In this scenario, the Decoder will allocate a new refcountable *mfxFrameSurface1* and return it to the user. All assumed contracts with the user are similar to the *MFXMemory\_GetSurfaceForXXX* functions.

### 12.2.9.3 mfxFrameSurfaceInterface

oneVPL API version 2.0 introduces *mfxFrameSurfaceInterface*. This interface is a set of callback functions to manage the lifetime of allocated surfaces, get access to pixel data, and obtain native handles and device abstractions (if suitable). Instead of directly accessing *mfxFrameSurface1* structure members, it's recommended to either use the *mfxFrameSurface1::mfxFrameSurfaceInterface*, if present, or call external allocator callback functions, if set.

The following example shows the usage of *mfxFrameSurfaceInterface* for memory sharing:

```

1 // lets decode frame and try to access output in a an optimal way.
2 sts = MFXVideoDECODE_DecodeFrameAsync(session, NULL, NULL, &outsurface, &syncp);
3 if (MFX_ERR_NONE == sts)
```

(continues on next page)

(continued from previous page)

```

4     {
5         outsurface->FrameInterface->GetDeviceHandle(outsurface, &device_handle, &device_
6             ↪type);
7             // if application or component is familiar with mfxHandleType and it's possible to_
8             ↪share memory created by device_handle.
9             if (isDeviceTypeCompatible(device_type) && isPossibleForMemorySharing(device_
10                ↪handle)) {
11                    // get native handle and type
12                    outsurface->FrameInterface->GetNativeHandle(outsurface, &resource, &resource_
13                        ↪type);
14                        if (isResourceTypeCompatible(resource_type)) {
15                            //use memory directly
16                            ProcessNativeMemory(resource);
17                            outsurface->FrameInterface->Release(outsurface);
18                        }
19                    }
20                    // Application or component is not aware about such DeviceHandle or Resource type_
21                    ↪need to map to system memory.
22                    outsurface->FrameInterface->Map(outsurface, MFX_MAP_READ);
23                    ProcessSystemMemory(outsurface);
24                    outsurface->FrameInterface->Unmap(outsurface);
25                    outsurface->FrameInterface->Release(outsurface);
26                }
27            }

```

## 12.2.10 Hardware Device Error Handling

The SDK accelerates decoding, encoding, and video processing through a hardware device. The SDK functions may return errors or warnings if the hardware device encounters errors. See the *Hardware Device Errors and Warnings table* for detailed information about the errors and warnings.

Table 10: Hardware Device Errors and Warnings

Status	Description
<i>mfxStatus::MFX_ERR_DEVICE_FAILED</i>	Hardware device returned unexpected errors. SDK was unable to restore operation.
<i>mfxStatus::MFX_ERR_DEVICE_LOST</i>	Hardware device was lost due to system lock or shutdown.
<i>mfxStatus::MFX_WRN_PARTIAL_ACCELERATION</i>	The hardware does not fully support the specified configuration. The encoding, decoding, or video processing operation may be partially accelerated.
<i>mfxStatus::MFX_WRN_DEVICE_BUSY</i>	Hardware device is currently busy.

SDK `Query`, `QueryIOSurf`, and `Init` functions return *mfxStatus::MFX\_WRN\_PARTIAL\_ACCELERATION* to indicate that the encoding, decoding, or video processing operation can be partially hardware accelerated or not hardware accelerated at all. The application can ignore this warning and proceed with the operation. (Note that SDK functions may return errors or other warnings overwriting *mfxStatus::MFX\_WRN\_PARTIAL\_ACCELERATION*, as it is a lower priority warning.)

SDK functions return *mfxStatus::MFX\_WRN\_DEVICE\_BUSY* to indicate that the hardware device is busy and unable to take commands at this time. Resume the operation by waiting for a few milliseconds and resubmitting the request. The following example shows the decoding pseudo-code:

```

1 mfxStatus sts=MFX_ERR_NONE;
2 for (;;) {
3     // do something

```

(continues on next page)

(continued from previous page)

```

4   sts=MFXVideoDECODE_DecodeFrameAsync(session, bitstream, surface_work, &surface_
5   ↪disp, &syncp);
6   if (sts == MFX_WRN_DEVICE_BUSY) sleep(5);
}

```

The same procedure applies to encoding and video processing.

SDK functions return `mfxStatus::MFX_ERR_DEVICE_LOST` or `mfxStatus::MFX_ERR_DEVICE_FAILED` to indicate that there is a complete failure in hardware acceleration. The application must close and reinitialize the SDK function class. If the application has provided a hardware acceleration device handle to the SDK, the application must reset the device.

## 12.3 Mandatory/Optional APIs and Features

### 12.3.1 Disclaimer

The specification implementer may implement any subset of the oneVPL specification. The specification makes no claim that any codec, encoder, decoder or VPP filter or their underlying features are mandatory for the implementation. The oneVPL API is designed in a way so that the implementation user has several options to discover implementation capabilities:

1. Before session creation user can get implementation capabilities by using `MFXEnumImplementations()` function to discover supported encoders/decoders or VPP filters with their supported color format and memory types.
2. Once session is created, user can use `Query` functions to obtain low level implementation capabilities.

#### 12.3.1.1 Functions must be exposed by any implementation

The *Exported Functions/API Version table* shows the list of functions exported by any implementation with corresponding API version. Implementation of all those functions is mandatory, while majority of them can return `mfxStatus::MFX_ERR_NOT_IMPLEMENTED`.

Table 11: Exported Functions/API Version

Function	API Version
MFXInit	1.0
MFXClose	1.0
MFXQueryIMPL	1.0
MFXQueryVersion	1.0
MFXJoinSession	1.0
MFXDisjoinSession	1.0
MFXCloneSession	1.0
MFXSetPriority	1.0
MFXGetPriority	1.0
MFXVideoCORE_SetFrameAllocator	1.0
MFXVideoCORE_SetHandle	1.0
MFXVideoCORE_GetHandle	1.0
MFXVideoCORE_SyncOperation	1.0
MFXVideoENCODE_Query	1.0
MFXVideoENCODE_QueryIOSurf	1.0

continues on next page

Table 11 – continued from previous page

Function	API Version
MFXVideoENCODE_Init	1.0
MFXVideoENCODE_Reset	1.0
MFXVideoENCODE_Close	1.0
MFXVideoENCODE_GetVideoParam	1.0
MFXVideoENCODE_GetEncodeStat	1.0
MFXVideoENCODE_EncodeFrameAsync	1.0
MFXVideoDECODE_Query	1.0
MFXVideoDECODE_DecodeHeader	1.0
MFXVideoDECODE_QueryIOSurf	1.0
MFXVideoDECODE_Init	1.0
MFXVideoDECODE_Reset	1.0
MFXVideoDECODE_Close	1.0
MFXVideoDECODE_GetVideoParam	1.0
MFXVideoDECODE_GetDecodeStat	1.0
MFXVideoDECODE_SetSkipMode	1.0
MFXVideoDECODE_GetPayload	1.0
MFXVideoDECODE_DecodeFrameAsync	1.0
MFXVideoVPP_Query	1.0
MFXVideoVPP_QueryIOSurf	1.0
MFXVideoVPP_Init	1.0
MFXVideoVPP_Reset	1.0
MFXVideoVPP_Close	1.0
MFXVideoVPP_GetVideoParam	1.0
MFXVideoVPP_GetVPPStat	1.0
MFXVideoVPP_RunFrameVPPAsync	1.0
MFXVideoVPP_RunFrameVPPAsyncEx	1.10
MFXInitEx	1.14
MFXVideoCORE_QueryPlatform	1.19
MFXMemory_GetSurfaceForVPP	2.0
MFXMemory_GetSurfaceForEncode	2.0
MFXMemory_GetSurfaceForDecode	2.0
MFXQueryImplsDescription	2.0
MFXReleaseImplDescription	2.0

### 12.3.1.2 Mandatory API for each implementation

Each implementation must implement following functions:

1. Functions required for the dispatcher to create session: `MFXInitEx()`, `MFXClose()`.
2. Functions required for the dispatcher to return implementation capabilities: `MFXQueryImplsDescription()`, `MFXReleaseImplDescription()`.
3. If implementation implements any encoder, than those functions are mandatory for the implementation: `MFXVideoENCODE_Init()`, `MFXVideoENCODE_Close()`, `MFXVideoENCODE_Query()`, `MFXVideoENCODE_EncodeFrameAsync()`.
4. If implementation implements any decoder, than those functions are mandatory for the implementation: `MFXVideoDECODE_Init()`, `MFXVideoDECODE_Close()`, `MFXVideoDECODE_Query()`, `MFXVideoDECODE_DecodeFrameAsync()`.
5. If implementation implements any VPP filter, than those functions are mandatory for the imple-

mentation: `MFXVideoVPP_Init()`, `MFXVideoVPP_Close()`, `MFXVideoVPP_Query()`, `MFXVideoVPP_RunFrameVPPAsync()`.

6. Function required for the asynchronous operations synchronization `MFXVideoCORE_SyncOperation()` is mandatory for the implementation.

7. Any other functions or extension buffers are optional for the implementation.

If implementation implements either decoder or encoder or VPP filter, its capabilities must be provided by using `mfxImplDescription` structure, which lists mandatory capabilities of the implementation.

---

**Note:** Mandatory function must have implementation and must not return `MFX_ERR_NOT_IMPLEMENTED` status.

---

## 12.4 Appendices

### 12.4.1 oneVPL for Intel® Media Software Development Kit Users

oneVPL is source compatible with Intel® Media Software Development Kit. Applications can use Intel® Media Software Development Kit to target older hardware and oneVPL to target everything else. Some obsolete features of Intel® Media Software Development Kit have been omitted from oneVPL.

#### 12.4.1.1 oneVPL Ease of Use Enhancements

oneVPL provides improved ease of use compared to Intel® Media Software Development Kit. Ease of use enhancements include the following:

- Smart dispatcher with discovery of implementation capabilities. See [SDK Session](#) for more details.
- Simplified decoder initialization. See [Decoding Procedures](#) for more details.
- New memory management and components (session) interoperability. See [Internal Memory Management](#) and [Decoding Procedures](#) for more details.

#### 12.4.1.2 New APIs in oneVPL

oneVPL introduces new functions that are not available in Intel® Media Software Development Kit.

New oneVPL dispatcher functions:

- `MFXLoad()`
- `MFXUnload()`
- `MFXCreateConfig()`
- `MFXSetConfigFilterProperty()`
- `MFXEnumImplementations()`
- `MFXCreateSession()`
- `MFXDispReleaseImplDescription()`

New oneVPL memory management functions:

- `MFXMemory_GetSurfaceForVPP()`

- `MFXMemory_GetSurfaceForEncode()`
- `MFXMemory_GetSurfaceForDecode()`

New oneVPL implementation capabilities retrieval functions:

- `MFXQueryImplsDescription()`
- `MFXReleaseImplDescription()`

#### 12.4.1.3 Intel® Media Software Development Kit Feature Removals

The following Intel® Media Software Development Kit features are considered obsolete and are not included in oneVPL:

- **Audio support.** oneVPL is intended for video processing. Audio APIs that duplicate functionality from other audio libraries such as [Sound Open Firmware](#) have been removed.
- **ENC and PAK interfaces.** Part of the Flexible Encode Infrastructure (FEI) and plugin interfaces which provide additional control over the encoding process for AVC and HEVC encoders. This feature was removed because it is not widely used by customers.
- **User plugins architecture.** oneVPL enables robust video acceleration through API implementations of many different video processing frameworks. Support of a SDK user plugin framework is obsolete.
- **External buffer memory management.** A set of callback functions to replace internal memory allocation is obsolete.
- **Video Processing extended runtime functionality.** Video processing function `MFXVideoVPP_RunFrameVPPAsyncEx` is used for plugins only and is obsolete.
- **External threading.** The new threading model makes the `MFXDoWork` function obsolete.
- **Multi-frame encode.** A set of external buffers to combine several frames into one encoding call. This feature was removed because it is device specific and not commonly used.

#### 12.4.1.4 Intel® Media Software Development Kit API Removals

The following Intel® Media Software Development Kit functions are not included in oneVPL:

- **Audio related functions**
  - `MFXAudioCORE_SyncOperation()`
  - `MFXAudioDECODE_Close()`
  - `MFXAudioDECODE_DecodeFrameAsync()`
  - `MFXAudioDECODE_DecodeHeader()`
  - `MFXAudioDECODE_GetAudioParam()`
  - `MFXAudioDECODE_Init()`
  - `MFXAudioDECODE_Query()`
  - `MFXAudioDECODE_QueryIOSize()`
  - `MFXAudioDECODE_Reset()`
  - `MFXAudioENCODE_Close()`
  - `MFXAudioENCODE_EncodeFrameAsync()`
  - `MFXAudioENCODE_GetAudioParam()`

- MFXAudioENCODE\_Init()
- MFXAudioENCODE\_Query()
- MFXAudioENCODE\_QueryIOSize()
- MFXAudioENCODE\_Reset()
- **Flexible encode infrastructure functions**
  - MFXVideoENC\_Close()
  - MFXVideoENC\_GetVideoParam()
  - MFXVideoENC\_Init()
  - MFXVideoENC\_ProcessFrameAsync()
  - MFXVideoENC\_Query()
  - MFXVideoENC\_QueryIOSurf()
  - MFXVideoENC\_Reset()
  - MFXVideoPAK\_Close()
  - MFXVideoPAK\_GetVideoParam()
  - MFXVideoPAK\_Init()
  - MFXVideoPAK\_ProcessFrameAsync()
  - MFXVideoPAK\_Query()
  - MFXVideoPAK\_QueryIOSurf()
  - MFXVideoPAK\_Reset()
- **User plugin functions**
  - MFXAudioUSER\_ProcessFrameAsync()
  - MFXAudioUSER\_Register()
  - MFXAudioUSER\_Unregister()
  - MFXVideoUSER\_GetPlugin()
  - MFXVideoUSER\_ProcessFrameAsync()
  - MFXVideoUSER\_Register()
  - MFXVideoUSER\_Unregister()
  - MFXVideoUSER\_Load()
  - MFXVideoUSER\_LoadByPath()
  - MFXVideoUSER\_UnLoad()
  - MFXDoWork()
- **Memory functions**
  - MFXVideoCORE\_SetBufferAllocator()
- **Video processing functions**
  - MFXVideoVPP\_RunFrameVPPAsyncEx()

---

**Important:** Corresponding extension buffers are also removed.

---

The following behaviors occur when attempting to use a Intel® Media Software Development Kit API that is not supported by oneVPL:

- Code compiled with the oneVPL API headers will generate a compile and/or link error when attempting to use a removed API.
- Code previously compiled with Intel® Media Software Development Kit and executed using a oneVPL runtime will generate an `MFX_ERR_UNSUPPORTED` error when calling a removed function.

## 12.4.2 Configuration Parameter Constraints

The `mfxFrameInfo` structure is used by both the `mfxVideoParam` structure during SDK class initialization and the `mfxFrameSurface1` structure during the actual SDK class function. The parameter constraints described in the following tables apply.

### 12.4.2.1 DECODE, ENCODE, and VPP Constraints

The *DECODE, ENCODE, and VPP Constraints table* lists parameter constraints common to *DECODE*, *ENCODE*, and *VPP*.

Table 12: DECODE, ENCODE, and VPP Constraints

Pa-ram-eters	During SDK Initialization	During SDK Operation
FourCC	Any valid value	The value must be the same as the initialization value. The only exception is <i>VPP</i> in composition mode, where in some cases it is allowed to mix RGB and NV12 surfaces. See <code>mfxExtVPPComposite</code> for more details.
Chro-maFor-mat	Any valid value	The value must be the same as the initialization value.

### 12.4.2.2 DECODE Constraints

The *DECODE Constraints table* lists *DECODE* parameter constraints.

Table 13: DECODE Constraints

Parameters	During SDK initialization	During SDK operation
Width Height	Aligned frame size	The values must be equal to or larger than the initialization values.
CropX, CropY CropW, CropH	Ignored	<i>DECODE</i> output. The cropping values are per-frame based.
AspectRatioW AspectRatioH	Any valid values or unspecified (zero); if unspecified, values from the input bitstream will be used. See note below the table.	<i>DECODE</i> output.
FrameRate-ExtN FrameRate-ExtD	If unspecified, values from the input bitstream will be used. See note below the table.	<i>DECODE</i> output.
PicStruct	Ignored	<i>DECODE</i> output.

---

**Note:** If the application explicitly sets FrameRateExtN/FrameRateExtD or AspectRatioW/AspectRatioH during initialization, then the decoder will use these values during decoding regardless of the values from bitstream and does not update them on new SPS. If the application sets them to 0, then the decoder uses values from the stream and updates them on each SPS.

---

### 12.4.2.3 ENCODE Constraints

The *ENCODE Constraints table* lists *ENCODE* parameter constraints.

Table 14: ENCODE Constraints

Parameters	During SDK initialization	During SDK operation
Width Height	Encoded frame size	The values must be equal to or larger than the initialization values.
CropX, CropY, CropW, CropH	H.264: Cropped frame size MPEG-2: CropW and CropH Specify the real width and height (maybe unaligned) of the coded frames. CropX and CropY must be zero.	Ignored
AspectRatioW AspectRatioH	Any valid values	Ignored
FrameRateExtN FrameRateExtD	Any valid values	Ignored
PicStruct	<code>MFX_PICSTRUCT_UNKNOWN</code> <code>MFX_PICSTRUCT_PROGRESSIVE</code> <code>MFX_PICSTRUCT_UNKNOWN</code>	The base value must be the same as the initialization value unless <code>MFX_PICSTRUCT_UNKNOWN</code> is specified during initialization. Add other decorative picture structure flags to indicate additional display attributes. Use <code>MFX_PICSTRUCT_UNKNOWN</code> during initialization for field attributes and <code>MFX_PICSTRUCT_PROGRESSIVE</code> for frame attributes. See the <code>PicStruct</code> enumerator for details.

#### 12.4.2.4 VPP Constraints

The *VPP Constraints table* lists *VPP* parameter constraints.

Table 15: VPP Constraints

Parameters	During SDK initialization	During SDK operation
Width, Height	Any valid values	The values must be equal to or larger than the initialization values.
CropX, CropY, CropW, CropH	Ignored	These parameters specify the region of interest from input to output.
AspectRatioW AspectRatioH	Ignored	Aspect ratio values will be passed through from input to output.
FrameRateExtN FrameRateExtD	Any valid values	Frame rate values will be updated with the initialization value at output.
PicStruct	<p><i>MFX_PICSTRUCT_UNKNOWN</i></p> <p><i>MFX_PICSTRUCT_PROGRESSIVE</i></p> <p><i>MFX_PICSTRUCT_FIELD_TFF</i></p> <p><i>MFX_PICSTRUCT_FIELD_BFF</i></p> <p><i>MFX_PICSTRUCT_FIELD_SINGLE</i></p> <p><i>MFX_PICSTRUCT_FIELD_TOP</i></p> <p><i>MFX_PICSTRUCT_FIELD_BOTTOM</i></p>	The base value must be the same as the initialization value unless <i>MFX_PICSTRUCT_UNKNOWN</i> is specified during initialization. Other decorative picture structure flags are passed through or added as needed. See the <i>PicStruct</i> enumerator for details.

#### 12.4.2.5 Specifying Configuration Parameters

The *Configuration Parameters table* summarizes how to specify the configuration parameters during initialization, encoding, decoding, and video processing.

Table 16: Configuration Parameters

Structure (param)	ENCODE Init	ENCODE Encoding	DECODE Init	DECODE Decoding	VPP Init	VPP Pro- cessing
<i>mfxVideoParam</i>						
Protected	R	•	R	•	R	•

continues on next page

Table 16 – continued from previous page

IOPattern	M	•	M	•	M	•
ExtParam	O	•	O	•	O	•
NumExtParam	O	•	O	•	O	•
<i>mfxInfoMFx</i>						
CodecId	M	•	M	•	•	•
CodecProfile	O	•	O/M*	•	•	•
CodecLevel	O	•	O	•	•	•
NumThread	O	•	O	•	•	•
TargetUsage	O	•	•	•	•	•
GopPicSize	O	•	•	•	•	•
GopRefDist	O	•	•	•	•	•
GopOptFlag	O	•	•	•	•	•
IdrInterval	O	•	•	•	•	•
RateControlMethod		•	•	•	•	•
InitialDelayInKBO		•	•	•	•	•
BufferSizeInKB	O	•	•	•	•	•
TargetKbps	M	•	•	•	•	•

continues on next page

Table 16 – continued from previous page

MaxKbps	O	•	•	•	•	•
NumSlice	O	•	•	•	•	•
NumRefFrame	O	•	•	•	•	•
EncodedOrder	M	•	•	•	•	•
<i>mfxFrameInfo</i>						
FourCC	M	M	M	M	M	M
Width	M	M	M	M	M	M
Height	M	M	M	M	M	M
CropX	M	Ign	Ign	U	Ign	M
CropY	M	Ign	Ign	U	Ign	M
CropW	M	Ign	Ign	U	Ign	M
CropH	M	Ign	Ign	U	Ign	M
FrameRateExtN	M	Ign	O	U	M	U
FrameRateExtD	M	Ign	O	U	M	U
AspectRatioW	O	Ign	O	U	Ign	PT
AspectRatioH	O	Ign	O	U	Ign	PT
PicStruct	O	M	Ign	U	M	M/U
ChromaFormat	M	M	M	M	Ign	Ign

Legend for the *Configuration Parameters table*:

Remarks	
Ign	Ignored
PT	Pass Through
•	Does Not Apply
M	Mandated
R	Reserved
O	Optional
U	Updated at output

---

**Note:** *CodecProfile* is mandated for HEVC REXT and SCC profiles and optional for other cases. If the application doesn't explicitly set CodecProfile during initialization, the HEVC decoder will use a profile up to Main10.

---

### 12.4.3 Multiple-segment Encoding

Multiple-segment encoding is useful in video editing applications during production, for example when the encoder encodes multiple video clips according to their time line. In general, one can define multiple-segment encoding as dividing an input sequence of frames into segments and encoding them in different encoding sessions with the same or different parameter sets. For example:

Segment Already Encoded	Segment in Encoding	Segment to be Encoded
0s	200s	500s

---

**Note:** Different encoders can also be used.

---

The application must be able to:

- Extract encoding parameters from the bitstream of previously encoded segment.
- Import these encoding parameters to configure the encoder.

Encoding can then continue on the current segment using either the same or similar encoding parameters.

Extracting the header containing the encoding parameter set from the encoded bitstream is usually the task of a format splitter (de-multiplexer). Alternatively, the SDK [MFXVideoDECODE\\_DecodeHeader\(\)](#) function can export the raw header if the application attaches the *mfxExtCodingOptionSPSPPS* structure as part of the parameters.

The encoder can use the *mfxExtCodingOptionSPSPPS* structure to import the encoding parameters during [MFXVideoENCODE\\_Init\(\)](#). The encoding parameters are in the encoded bitstream format. Upon a successful import of the header parameters, the encoder will generate bitstreams with a compatible (not necessarily bit-exact) header. The [Header Import Functions table](#) shows all functions that can import a header and their error codes if there are unsupported parameters in the header or the encoder is unable to achieve compatibility with the imported header.

Table 17: Header Import Functions

Function Name	Error Code if Import Fails
<a href="#">MFXVideoENCODE_Init()</a>	MFX_ERR_INCOMPATIBLE_VIDEO_PARAM
<a href="#">MFXVideoENCODE_QueryIOSurf()</a>	MFX_ERR_INCOMPATIBLE_VIDEO_PARAM
<a href="#">MFXVideoENCODE_Reset()</a>	MFX_ERR_INCOMPATIBLE_VIDEO_PARAM
<a href="#">MFXVideoENCODE_Query()</a>	MFX_ERR_UNSUPPORTED

The encoder must encode frames to a GOP sequence starting with an IDR frame for H.264 (or I frame for MPEG-2) to ensure that the current segment encoding does not refer to any frames in the previous segment. This ensures that the encoded segment is self-contained, allowing the application to insert the segment anywhere in the final bitstream. After encoding, each encoded segment is HRD compliant. Concatenated segments may not be HRD compliant.

The following example shows the encoder initialization procedure that imports H.264 sequence and picture parameter sets:

```

1 mfxStatus init_encoder() {
2     mfxExtCodingOptionSPSPPS option, *option_array;
3
4     /* configure mfxExtCodingOptionSPSPPS */
5     memset(&option, 0, sizeof(option));
6     option.Header.BufferId=MFX_EXTBUFF_CODING_OPTION_SPSPPS;
7     option.Header.BufferSz=sizeof(option);
8     option.SPSBuffer=sps_buffer;
9     option.SPSBufSize=sps_buffer_length;

```

(continues on next page)

(continued from previous page)

```

10    option.PPSBuffer=pps_buffer;
11    option.PPSBufSize=pps_buffer_length;
12
13    /* configure mfxVideoParam */
14    mfxVideoParam param;
15    //...
16    param.NumExtParam=1;
17    option_array=&option;
18    param.ExtParam=(mfxExtBuffer**)option_array;
19
20    /* encoder initialization */
21    mfxStatus status;
22    status=MFXVideoENCODE_Init(session, &param);
23    if (status==MFX_ERR_INCOMPATIBLE_VIDEO_PARAM) {
24        printf("Initialization failed.\n");
25    } else {
26        printf("Initialized.\n");
27    }
28    return status;
29}

```

## 12.4.4 Streaming and Video Conferencing Features

The following sections address some aspects of additional requirements that streaming or video conferencing applications may use in the encoding or transcoding process. See the *Configuration Change* section for additional information.

### 12.4.4.1 Dynamic Bitrate Change

The SDK encoder supports dynamic bitrate change according to bitrate control mode and HRD conformance requirements. If HRD conformance is required, for example if the application sets the NalHrdConformance option in the *mfxExtCodingOption* structure to ON, the only allowed bitrate control mode is VBR. In this mode, the application can change the TargetKbps and MaxKbps values of the *mfxInfoMFX* structure by calling the *MFXVideoENCODE\_Reset()* function. This sort of change in bitrate usually results in generation of a new keyframe and sequence header. There are some exceptions, such as if HRD information is absent in the stream. In this scenario, the change of TargetKbps does not require a change in the sequence header and as a result the SDK encoder does not insert a keyframe.

If HRD conformance is not required, for example if the application turns off the NalHrdConformance option in the *mfxExtCodingOption* structure, all bitrate control modes are available. In CBR and AVBR modes the application can change TargetKbps. In VBR mode the application can change TargetKbps and MaxKbps values. This sort of change in bitrate will not result in the generation of a new keyframe or sequence header.

The SDK encoder may change some of the initialization parameters provided by the application during initialization. That in turn may lead to incompatibility between the parameters provided by the application during reset and working set of parameters used by the SDK encoder. For this reason, it is strongly recommended to retrieve the actual working parameters using the *MFXVideoENCODE\_GetVideoParam()* function before making any changes to bitrate settings.

In all modes, the SDK encoders will respond to the bitrate changes as quickly as the underlying algorithm allows, without breaking other encoding restrictions such as HRD compliance if it is enabled. How quickly the actual bitrate can catch up with the specified bitrate is implementation dependent.

Alternatively, the application may use the *CQP* encoding mode to perform customized bitrate adjustment on a per-frame base. The application may use any of the encoded or display order modes to use per-frame CQP.

#### 12.4.4.2 Dynamic Resolution Change

The SDK encoder supports dynamic resolution change in all bitrate control modes. The application may change resolution by calling the `MFXVideoENCODE_Reset()` function. The application may decrease or increase resolution up to the size specified during encoder initialization.

Resolution change always results in the insertion of a key IDR frame and a new sequence parameter set in the header. The only exception is the SDK VP9 encoder (see section for *Dynamic reference frame scaling*). The SDK encoder does not guarantee HRD conformance across the resolution change point.

The SDK encoder may change some of the initialization parameters provided by the application during initialization. That in turn may lead to incompatibility of parameters provided by the application during reset and working set of parameters used by the SDK encoder. Due to this potential incompatibility, it is strongly recommended to retrieve the actual working parameters set by `MFXVideoENCODE_GetVideoParam()` function before making any resolution change.

#### 12.4.4.3 Dynamic Reference Frame Scaling

The VP9 standard allows changing the resolution without the insertion of a keyframe. This is possible because the VP9 encoder has the built-in capability to upscale and downscale reference frames to match the resolution of the frame being encoded. By default the SDK VP9 encoder inserts a keyframe when the application does *Dynamic Resolution Change*. In this case, the first frame with a new resolution is encoded using inter prediction from the scaled reference frame of the previous resolution. Dynamic scaling has the following limitations, described in the VP9 specification:

- The resolution of any active reference frame cannot exceed 2x the resolution of the current frame.
- The resolution of any active reference frame cannot be smaller than 1/16 of the current frame resolution.

In the case of dynamic scaling, the SDK VP9 encoder always uses a single active reference frame for the first frame after a resolution change. The SDK VP9 encoder has the following limitations for dynamic resolution change:

- The new resolution shouldn't exceed 16x the resolution of the current frame.
- The new resolution should be less than 1/2 of current frame resolution.

The application may force insertion of a keyframe at the point of resolution change by invoking encoder reset with `mfxExtEncoderResetOption::StartNewSequence` set to `MFX_CODINGOPTION_ON`. If a keyframe is inserted, the dynamic resolution limitations are not enforced.

Note that resolution change with dynamic reference scaling is compatible with multiref (`mfxInfoMFN::NumRefFrame > 1`). For multiref configuration, the SDK VP9 encoder uses multiple references within stream pieces of the same resolution, and uses a single reference at the place of resolution change.

#### 12.4.4.4 Forced Keyframe Generation

The SDK supports forced keyframe generation during encoding. The application can set the FrameType parameter of the `mfxEncodeCtrl` structure to control how the current frame is encoded, as follows:

- If the SDK encoder works in the display order, the application can enforce any current frame to be a keyframe. The application cannot change the frame type of already buffered frames inside the SDK encoder.
- If the SDK encoder works in the encoded order, the application must specify exact frame type for every frame. In this way, the application can enforce the current frame to have any frame type that the particular coding standard allows.

#### 12.4.4.5 Reference List Selection

During streaming or video conferencing, if the application can obtain feedback about how well the client receives certain frames, the application may need to adjust the encoding process to use or not use certain frames as reference. This section describes how to fine-tune the encoding process based on client feedback.

The application can specify the reference window size by specifying the `mfxInfoMFX::NumRefFrame` parameter during encoding initialization. Certain platforms may have limits on the size of the reference window. Use the `MFVideoENCODE_GetVideoParam()` function to retrieve the current working set of parameters.

During encoding, the application can specify the actual reference list lengths by attaching the `mfxExtAVCRefListCtrl` structure to the `MFVideoENCODE_EncodeFrameAsync()` function. `NumRefIdxL0Active` specifies the length of the reference list L0 and `NumRefIdxL1Active` specifies the length of the reference list L1. These two numbers must be less than or equal to the `mfxInfoMFX::NumRefFrame` parameter during encoding initialization.

The application can instruct the SDK encoder to use or not use certain reference frames. To do this, there is a prerequisite that the application uniquely identify each input frame by setting the `mfxFrameData::FrameOrder` parameter. The application then specifies the preferred reference frame list `PreferredRefList` and/or the rejected frame list `RejectedRefList`, and attaches the `mfxExtAVCRefListCtrl` structure to the `MFVideoENCODE_EncodeFrameAsync()` function. The two lists fine-tune how the SDK encoder chooses the reference frames for the current frame. The SDK encoder does not keep `PreferredRefList` and application must send it for each frame if necessary. There are limitations as follows:

- The frames in the lists are ignored if they are out of the reference window.
- If by going through the lists, the SDK encoder cannot find a reference frame for the current frame, the SDK encoder will encode the current frame without using any reference frames.
- If the GOP pattern contains B-frames, the SDK encoder may not be able to follow the `mfxExtAVCRefListCtrl` instructions.

#### 12.4.4.6 Low Latency Encoding and Decoding

The application can set `mfxVideoParam::AsyncDepth` = 1 to disable any decoder buffering of output frames, which is aimed to improve the transcoding throughput. With `mfxVideoParam::AsyncDepth` = 1, the application must synchronize after the decoding or transcoding operation of each frame.

The application can adjust `mfxExtCodingOption::MaxDecFrameBuffering` during encoding initialization to improve decoding latency. It is recommended to set this value equal to the number of reference frames.

#### 12.4.4.7 Reference Picture Marking Repetition SEI Message

The application can request writing the reference picture marking repetition SEI message during encoding initialization by setting `RefPicMarkRep` of the `mfxExtCodingOption` structure. The reference picture marking repetition SEI message repeats certain reference frame information in the output bitstream for robust streaming.

The SDK decoder will respond to the reference picture marking repetition SEI message if the message exists in the bitstream and compare it to the reference list information specified in the sequence/picture headers. The decoder will report any mismatch of the SEI message with the reference list information in the `mfxFrameData::Corrupted` field.

#### 12.4.4.8 Long Term Reference Frame

The application may use long term reference frames to improve coding efficiency or robustness for video conferencing applications. The application controls the long term frame marking process by attaching the `mfxExtAVCRefListCtrl` extended buffer during encoding. The SDK encoder itself never marks a frame as long term.

There are two control lists in the `mfxExtAVCRefListCtrl` extended buffer. The `LongTermRefList` list contains the frame orders (the `FrameOrder` value in the `mfxFrameData` structure) of the frames that should be marked as long term frames. The `RejectedRefList` list contains the frame order of the frames that should be unmarked as long term frames. The application can only mark or unmark the frames that are buffered inside the encoder. Because of this, it is recommended that the application marks a frame when it is submitted for encoding. The application can either explicitly unmark long term reference frames or wait for the IDR frame. When the IDR frame is reached, all long term reference frames will be unmarked.

The SDK encoder puts all long term reference frames at the end of a reference frame list. If the number of active reference frames (the `NumRefIdxL0Active` and `NumRefIdxL1Active` values in the `mfxExtAVCRefListCtrl` extended buffer) is less than than the total reference frame number (the `NumRefFrame` value in the `mfxInfoMFx` structure during the encoding initialization), the SDK encoder may ignore some or all long term reference frames. The application may avoid this by providing a list of preferred reference frames in the `PreferredRefList` list in the `mfxExtAVCRefListCtrl` extended buffer. In this case, the SDK encoder reorders the reference list based on the specified list.

#### 12.4.4.9 Temporal Scalability

The application may specify the temporal hierarchy of frames by using the `mfxExtAvcTemporalLayers` extended buffer during the encoder initialization in the display order encoding mode. The SDK inserts the prefix NAL unit before each slice with a unique temporal and priority ID. The temporal ID starts from zero and the priority ID starts from the `BaseLayerPID` value. The SDK increases the temporal ID and priority ID value by one for each consecutive layer.

If the application needs to specify a unique sequence or picture parameter set ID, the application must use the `mfxExtCodingOptionSPSPPS` extended buffer, with all pointers and sizes set to zero and valid `SPSID` and `PPSID` fields. The same SPS and PPS ID will be used for all temporal layers.

Each temporal layer is a set of frames with the same temporal ID. Each layer is defined by the `Scale` value. The scale for layer N is equal to the ratio between the frame rate of subsequent temporal layers with a temporal ID less than or equal to N and the frame rate of the base temporal layer. The application may skip some temporal layers by specifying the `Scale` value as zero. The application should use an integer ratio of the frame rates for two consecutive temporal layers.

For example, a video sequence with 30 frames/second is typically separated by three temporal layers that can be decoded as 7.5 fps (base layer), 15 fps (base and first temporal layer) and 30 fps (all three layers). In this scenario, `Scale` should have the values {1,2,4,0,0,0,0,0}.

### 12.4.5 Switchable Graphics and Multiple Monitors

The following sections discuss support for switchable graphics and multiple monitor configurations.

#### 12.4.5.1 Switchable Graphics

Switchable Graphics refers to the machine configuration that multiple graphic devices are available (integrated device for power saving and discrete devices for performance.) Usually at one time or instance, one of the graphic devices drives display and becomes the active device, and others become inactive. There are different variations of software or hardware mechanisms to switch between the graphic devices. In one of the switchable graphics variations, it is possible to register an application in an affinity list to certain graphic device so that the launch of the application automatically triggers a switch. The actual techniques to enable such a switch are outside the scope of this document. This section discusses the implication of switchable graphics to the SDK and the SDK applications.

As the SDK performs hardware acceleration through graphic devices, it is critical that the SDK can access to the graphic device in the switchable graphics setting. It is recommended to add the application to the graphic device affinity list. If this is not possible, the application should handle the following cases:

- By design, during the SDK library initialization, the `MFXInit()` function searches for graphic devices. If a SDK implementation is successfully loaded, the `MFXInit()` function returns `mfxStatus::MFX_ERR_NONE` and the `MFXQueryIMPL()` function returns the actual implementation type. If no SDK implementation is loaded, the `MFXInit()` function returns `mfxStatus::MFX_ERR_UNSUPPORTED`. In the switchable graphics environment, if the application is not in the graphic device affinity list, it is possible that the graphic device will not be accessible during the SDK library initialization. The fact that the `MFXInit()` function returns `mfxStatus::MFX_ERR_UNSUPPORTED` does not mean that hardware acceleration is permanently impossible. The user may switch the graphics later and the graphic device will become accessible. It is recommended that the application initialize the SDK library right before the actual decoding, video processing, and encoding operations to determine the hardware acceleration capability.
- During decoding, video processing, and encoding operations, if the application is not in the graphic device affinity list, the previously accessible graphic device may become inaccessible due to a switch event. The SDK functions will return `mfxStatus::MFX_ERR_DEVICE_LOST` or `mfxStatus::MFX_ERR_DEVICE_FAILED`, depending on when the switch occurs and what stage the SDK functions operate. The application should handle these errors and exit gracefully.

#### 12.4.5.2 Multiple Monitors

Multiple monitors refer to the machine configuration that multiple graphic devices are available. Some graphic devices connect to a display and become active and accessible under the Microsoft\* DirectX\* infrastructure. Graphic devices that are not connected to a display are inactive. Using the Microsoft Direct3D\* 9 infrastructure, devices that are not connected to a display are not accessible.

The SDK uses the adapter number to access a specific graphic device. Usually, the graphic device driving the main desktop becomes the primary adapter. Other graphic devices take subsequent adapter numbers after the primary adapter. Under the Microsoft Direct3D 9 infrastructure, only active adapters are accessible and have an adapter number.

The SDK extends the implementation type `mfxIMPL` as shown in the *SDK mfxIMPL Implementation Type Definitions table*:

Table 18: SDK mfxIMPL Implementation Type Definitions

Implementation Type	Definition
<code>MFX_IMPL_HARDWARE</code>	The SDK should initialize on the primary adapter
<code>MFX_IMPL_HARDWARE2</code>	The SDK should initialize on the 2nd graphic adapter
<code>MFX_IMPL_HARDWARE3</code>	The SDK should initialize on the 3rd graphic adapter
<code>MFX_IMPL_HARDWARE4</code>	The SDK should initialize on the 4th graphic adapter
<code>MFX_IMPL_HARDWARE_ANY</code>	The SDK should initialize on any graphic adapter.
<code>MFX_IMPL_AUTO_ANY</code>	The SDK should initialize on any graphic adapter. If not successful, load the software implementation.

The application can use the first four definitions shown in the *SDK mfxIMPL Implementation Type Definitions table* to instruct the SDK library to initializes on a specific graphic device. The application can use the definitions for `MFX_IMPL_HARDWARE_ANY` and `MFX_IMPL_AUTO_ANY` for automatic detection.

If the application uses the Microsoft DirectX surfaces for I/O, it is critical that the application and the SDK works on the same graphic device. It is recommended that the application use the following procedure:

1. The application uses the `MFXInit()` function to initialize the SDK library, with option `MFX_IMPL_HARDWARE_ANY` or `MFX_IMPL_AUTO_ANY`. The `MFXInit()` function returns `mfxStatus::MFX_ERR_NONE` if successful.
2. The application uses the `MFXQueryIMPL()` function to check the actual implementation type. The implementation type `MFX_IMPL_HARDWARE`, `MFX_IMPL_HARDWARE2`, `MFX_IMPL_HARDWARE3`, or `MFX_IMPL_HARDWARE4` indicates the graphic adapter the SDK works on.
3. The application creates the Direct3D device on the respective graphic adapter and passes it to the SDK through the `MFXVideoCORE_SetHandle()` function.

Similar to the switchable graphics cases, interruption may result if the user disconnects monitors from the graphic devices or remaps the primary adapter. If the interruption occurs during the SDK library initialization, the `MFXInit()` function may return `mfxStatus::MFX_ERR_UNSUPPORTED`. This means hardware acceleration is currently not available. It is recommended that the application initialize the SDK library right before the actual decoding, video processing, and encoding operations to determine the hardware acceleration capability.

If the interruption occurs during decoding, video processing, or encoding operations, the SDK functions will return `mfxStatus::MFX_ERR_DEVICE_LOST` or `mfxStatus::MFX_ERR_DEVICE_FAILED`. The application should handle these errors and exit gracefully.

## 12.4.6 Working Directly with VA API for Linux\*

The SDK takes care of all memory and synchronization related operations in VA API. The application may need to extend the SDK functionality by working directly with VA API for Linux\*, for example to implement a customized external allocator. This section describes basic memory management and synchronization techniques.

To create the VA surface pool, the application should call the `vaCreateSurfaces` function:

```

1 VASurfaceAttrib attrib;
2 attrib.type = VASurfaceAttribPixelFormat;
3 attrib.value.type = VAGenericValueTypeInteger;
4 attrib.value.value.i = VA_FOURCC_NV12;
5 attrib.flags = VA_SURFACE_ATTRIB_SETTABLE;
6
7 #define NUM_SURFACES 5;
8 VASurfaceID surfaces[NUMSURFACES];
9
10 vaCreateSurfaces(va_display, VA_RT_FORMAT_YUV420, width, height, surfaces, NUM_
    ↵SURFACES, &attrib, 1);

```

(continues on next page)

(continued from previous page)

To destroy the surface pool, the application should call the `vaDestroySurfaces` function:

```
1 vaDestroySurfaces(va_display, surfaces, NUM_SURFACES);
```

If the application works with hardware acceleration through the SDK then it can access surface data immediately after successful completion of the `MFXVideoCORE_SyncOperation()` call. If the application works with hardware acceleration directly, then it must check surface status before accessing data in video memory. This check can be done asynchronously by calling the `vaQuerySurfaceStatus` function or synchronously by calling the `vaSyncSurface` function.

After successful synchronization, the application can access surface data. Accessing surface data is performed in two steps:

1. Create VAImage from surface.
2. Map image buffer to system memory.

After mapping, the `VAImage.offsets[3]` array holds offsets to each color plain in a mapped buffer and the `VAImage.pitches[3]` array holds color plain pitches in bytes. For packed data formats, only first entries in these arrays are valid. The following example shows how to access data in a NV12 surface:

```
1 VAImage image;
2 unsigned char *buffer, Y, U, V;
3
4 vaDeriveImage(va_display, surface_id, &image);
5 vaMapBuffer(va_display, image.buf, &buffer);
6
7 /* NV12 */
8 Y = buffer + image.offsets[0];
9 U = buffer + image.offsets[1];
10 V = U + 1;
```

After processing data in a VA surface, the application should release resources allocated for the mapped buffer and VAImage object:

```
1 vaUnmapBuffer(va_display, image.buf);
2 vaDestroyImage(va_display, image.image_id);
```

In some cases, in order to retrieve encoded bitstream data from video memory, the application must use the VABuffer to store data. The following example shows how to create, use, and destroy the VABuffer:

```
1 /* create buffer */
2 VABufferID buf_id;
3 vaCreateBuffer(va_display, va_context, VAEncCodedBufferType, buf_size, 1, NULL, &buf_
4 ↩id);
5
6 /* encode frame */
7 // ...
8
9 /* map buffer */
10 VACodedBufferSegment *coded_buffer_segment;
11
12 vaMapBuffer(va_display, buf_id, (void **)(& coded_buffer_segment));
13 size = coded_buffer_segment->size;
```

(continues on next page)

(continued from previous page)

```

14 offset = coded_buffer_segment->bit_offset;
15 buf    = coded_buffer_segment->buf;
16
17 /* retrieve encoded data */
18 // ...
19
20 /* unmap and destroy buffer */
21 vaUnmapBuffer(va_display, buf_id);
22 vaDestroyBuffer(va_display, buf_id);

```

Note that the vaMapBuffer function returns pointers to different objects depending on the mapped buffer type. The VAImage is a plain data buffer and the encoded bitstream is a VACodedBufferSegment structure. The application cannot use VABuffer for synchronization. If encoding, it is recommended to synchronize using the VA surface as described above.

## 12.4.7 CQP HRD Mode Encoding

The application can configure AVC encoder to work in CQP rate control mode with HRD model parameters. SDK will place HRD information to SPS/VUI and choose the appropriate profile/level. It's the responsibility of the application to provide per-frame QP, track HRD conformance, and insert required SEI messages to the bitstream.

The following example shows how to enable CQP HRD mode. The application should set *RateControlMethod* to CQP, *mfxExtCodingOption::VuiNalHrdParameters* to ON, *mfxExtCodingOption::NalHrdConformance* to OFF, and set rate control parameters similar to CBR or VBR modes (instead of QPI, QPP, and QPB). The SDK will choose CBR or VBR HRD mode based on the MaxKbps parameter. If MaxKbps is set to zero, the SDK will use CBR HRD model (write cbr\_flag = 1 to VUI), otherwise the VBR model will be used (and cbr\_flag = 0 is written to VUI).

---

**Note:** For CQP, if implementation doesn't support individual QPI, QPP and QPB parameters, then QPI parameter should be used as a QP parameter across all frames.

---

```

1 mfxExtCodingOption option, *option_array;
2
3 /* configure mfxExtCodingOption */
4 memset(&option, 0, sizeof(option));
5 option.Header.BufferId      = MFX_EXTBUFF_CODING_OPTION;
6 option.Header.BufferSz       = sizeof(option);
7 option.VuiNalHrdParameters  = MFX_CODINGOPTION_ON;
8 option.NalHrdConformance    = MFX_CODINGOPTION_OFF;
9
10 /* configure mfxVideoParam */
11 mfxVideoParam param;
12
13 // ...
14
15 param.mfx.RateControlMethod     = MFX_RATECONTROL_CQP;
16 param.mfx.FrameInfo.FrameRateExtN = valid_non_zero_value;
17 param.mfx.FrameInfo.FrameRateExtD = valid_non_zero_value;
18 param.mfx.BufferSizeInKB        = valid_non_zero_value;
19 param.mfx.InitialDelayInKB     = valid_non_zero_value;
20 param.mfx.TargetKbps           = valid_non_zero_value;
21
22 if (write_cbr_flag == 1)

```

(continues on next page)

(continued from previous page)

```

23     param.mfx.MaxKbps = 0;
24 else /* write_cbr_flag = 0 */
25     param.mfx.MaxKbps = valid_non_zero_value;
26
27 param.NumExtParam = 1;
28 option_array      = &option;
29 param.ExtParam    = (mfxExtBuffer **) &option_array;
30
31 /* encoder initialization */
32 mfxStatus sts;
33 sts = MFXVideoENCODE_Init(session, &param);
34
35 // ...
36
37 /* encoding */
38 mfxEncodeCtrl ctrl;
39 memset(&ctrl, 0, sizeof(ctrl));
40 ctrl.QP = frame_qp;
41
42 sts=MFXVideoENCODE_EncodeFrameAsync(session,&ctrl,surface2,bits,&syncp);

```

## 12.5 Acronyms and Abbreviations

**API** Application Programming Interface

**AVC** Advanced Video Codec (same as H.264 and MPEG-4, part 10)

**BRC** Bit Rate Control

**CQP** Constant Quantization Parameter

**dGPU/dGfx** Discrete graphics

**Direct3D** Microsoft\* Direct3D\* version 9 or 11.1

**Direct3D 9** Microsoft Direct3D version 9

**Direct3D 11** Microsoft Direct3D version 11.1

**DRM** Digital Right Management

**DXVA2** Microsoft DirectX\* Video Acceleration standard 2.0

**H.264** ISO\*/IEC\* 14496-10 and ITU-T\* H.264, MPEG-4 Part 10, Advanced Video Coding, May 2005

**GOP** Group of Picture

**GPB** Generalized P/B picture. B-picture, containing only forward references in both L0 and L1

**HDR** High Dynamic Range

**HRD** Hypothetical Reference Decoder

**I1010** A color format for raw video frames, extends IYUV/I420 for 10 bit

**IDR** Instantaneous decoding fresh picture, a term used in the H.264 specification

**iGPU** Integrated HD graphics

**IYUV** A color format for raw video frames, also known as I420

**LA** Look Ahead. Special encoding mode where encoder performs pre analysis of several frames before actual encoding starts.

**MCTF** Motion Compensated Temporal Filter. Special type of a noise reduction filter which utilizes motion to improve efficiency of video denoising

**MPEG** Motion Picture Expert Group

**MPEG-2** ISO/IEC 13818-2 and ITU-T H.262, MPEG-2 Part 2, Information Technology- Generic Coding of Moving Pictures and Associate Audio Information: Video, 2000

**NAL** Network Abstraction Layer

**NV12** YUV 4:2:0 video format, 12 bits per pixel

**NV16** YUV 4:2:2 video format, 16 bits per pixel

**P010** YUV 4:2:0 video format, extends NV12, 10 bits per pixel

**P210** YUV 4:2:2 video format, 10 bits per pixel

**PPS** Picture Parameter Set

**QP** Quantization Parameter

**RGB32** Thirty-two-bit RGB color format

**RGB4** Thirty-two-bit RGB color format. Also known as RGB32

**SDK** Intel® Media Software Development Kit

**SEI** Supplemental Enhancement Information

**SPS** Sequence Parameter Set

**UYVY** YUV 4:2:2 video format, 16 bits per pixel

**VA API** Video Acceleration API

**VBR** Variable Bit Rate

**VBV** Video Buffering Verifier

**VC-1** SMPTE\* 421M, SMPTE Standard for Television: VC-1 Compressed Video Bitstream Format and Decoding Process, August 2005

**video memory** Memory used by a hardware acceleration device, also known as GPU, to hold frame and other types of video data

**VUI** Video Usability Information

**YUY2** A color format for raw video frames

**YV12** A color format for raw video frames, similar to IYUV with U and V reversed

## 12.6 oneVPL API Versioning

oneVPL is the successor to Intel® Media Software Development Kit. oneVPL API versioning starts from 2.0. There is a correspondent version of Intel® Media Software Development Kit API which is used as a basis for oneVPL and defined as `MFX_LEGACY_VERSION` macro.

Experimental APIs in oneVPL are protected with the following macro:

```
#if (MFX_VERSION >= MFX_VERSION_NEXT)
```

To use the API, define the MFX\_VERSION\_USE\_LATEST macro.

## 12.7 oneVPL API Reference

### 12.7.1 Types

#### 12.7.1.1 Basic Types

**typedef** unsigned char **mfxU8**

Unsigned integer, 8 bit type.

**typedef** char **mfxI8**

Signed integer, 8 bit type.

**typedef** unsigned short **mfxU16**

Unsigned integer, 16 bit type.

**typedef** short **mfxI16**

Signed integer, 16 bit type.

**typedef** unsigned int **mfxU32**

Unsigned integer, 32 bit type.

**typedef** int **mfxI32**

Signed integer, 32 bit type.

**typedef** unsigned int **mfxUL32**

Unsigned integer, 32 bit type.

**typedef** int **mfxL32**

Signed integer, 32 bit type.

**typedef** \_\_UINT64 **mfxU64**

Unsigned integer, 64 bit type.

**typedef** \_\_INT64 **mfxI64**

Signed integer, 64 bit type.

**typedef** float **mfxF32**

Single-precision floating point, 32 bit type.

**typedef** double **mfxF64**

Double-precision floating point, 64 bit type.

**typedef** void \***mfxHDL**

Handle type.

**typedef** *mfxHDL* **mfxMemId**

Memory ID type.

**typedef** void \***mfxThreadTask**

Thread task type.

**typedef** char **mfxChar**

UTF-8 byte.

### 12.7.1.2 Typedefs

```
typedef struct _mfxSession *mfxSession
    SDK session handle.

typedef struct _mfxSyncPoint *mfxSyncPoint
    Synchronization point object handle.

typedef struct _mfxLoader *mfxLoader
    SDK loader handle.

typedef struct _mfxConfig *mfxConfig
    SDK config handle.
```

## 12.7.2 Dispatcher API

### 12.7.2.1 Defines

**MFX\_IMPL\_NAME\_LEN**  
 Maximum allowed length of the implementation name.

**MFX\_STRFIELD\_LEN**  
 Maximum allowed length of the implementation name.

### 12.7.2.2 Enums

#### 12.7.2.3 mfxImplType

**enum mfxImplType**  
 This enum itemizes implementation type.

*Values:*

- enumerator MFX\_IMPL\_TYPE\_SOFTWARE = 0x0001**  
 Pure Software Implementation.
- enumerator MFX\_IMPL\_TYPE\_HARDWARE = 0x0002**  
 Hardware Accelerated Implementation.

#### 12.7.2.4 mfxAccelerationMode

**enum mfxAccelerationMode**  
 This enum itemizes hardware acceleration stack to use.

*Values:*

- enumerator MFX\_ACCEL\_MODE\_NA = 0**  
 Hardware acceleration is not applicable.
- enumerator MFX\_ACCEL\_MODE\_VIA\_D3D9 = 0x0200**  
 Hardware acceleration goes through the Microsoft\* Direct3D9\* infrastructure.
- enumerator MFX\_ACCEL\_MODE\_VIA\_D3D11 = 0x0300**  
 Hardware acceleration goes through the Microsoft\* Direct3D11\* infrastructure.
- enumerator MFX\_ACCEL\_MODE\_VIA\_VAAPI = 0x0400**  
 Hardware acceleration goes through the Linux\* VA-API infrastructure.

## 12.7.2.5 Structures

### 12.7.2.6 mfxVariant

#### **enum mfxVariantType**

The mfxVariantType enumerator data types for mfxVarianf type.

*Values:*

**enumerator MFX\_VARIANT\_TYPE\_UNSET = 0**

Undefined type.

**enumerator MFX\_VARIANT\_TYPE\_U8 = 1**

8-bit unsigned integer.

**enumerator MFX\_VARIANT\_TYPE\_I8**

8-bit signed integer.

**enumerator MFX\_VARIANT\_TYPE\_U16**

16-bit unsigned integer.

**enumerator MFX\_VARIANT\_TYPE\_I16**

16-bit signed integer.

**enumerator MFX\_VARIANT\_TYPE\_U32**

32-bit unsigned integer.

**enumerator MFX\_VARIANT\_TYPE\_I32**

32-bit signed integer.

**enumerator MFX\_VARIANT\_TYPE\_U64**

64-bit unsigned integer.

**enumerator MFX\_VARIANT\_TYPE\_I64**

64-bit signed integer.

**enumerator MFX\_VARIANT\_TYPE\_F32**

32-bit single precision floating point.

**enumerator MFX\_VARIANT\_TYPE\_F64**

64-bit double precision floating point.

**enumerator MFX\_VARIANT\_TYPE\_PTR**

Generic type pointer.

#### **struct mfxVariant**

The mfxVariantType enumerator data types for mfxVarianf type.

#### Public Members

##### *mfxStructVersion* **Version**

Version of the structure.

##### *mfxVariantType* **Type**

Value type.

##### **union mfxVariant::data Data**

Value data member.

##### **union data**

Value data holder.

## Public Members

*mfxU8* **U8**  
mfxU8 data.

*mfxI8* **I8**  
mfxI8 data.

*mfxU16* **U16**  
mfxU16 data.

*mfxI16* **I16**  
mfxI16 data.

*mfxU32* **U32**  
mfxU32 data.

*mfxI32* **I32**  
mfxI32 data.

*mfxU64* **U64**  
mfxU64 data.

*mfxI64* **I64**  
mfxI64 data.

*mfxF32* **F32**  
mfxF32 data.

*mfxF64* **F64**  
mfxF64 data.

*mfxHDL* **Ptr**  
Pointer.

### 12.7.2.7 mfxDecoderDescription

#### struct mfxDecoderDescription

The *mfxDecoderDescription* structure represents the description of a decoder.

## Public Members

*mfxStructVersion* **Version**  
Version of the structure.

*mfxU16* **reserved[7]**  
Reserved for future use.

*mfxU16* **NumCodecs**  
Number of supported decoders.

**struct** *mfxDecoderDescription::decoder* \***Codecs**  
Pointer to the array of decoders.

**struct** **decoder**  
This structure represents the decoder description.

## Public Members

*mfxU32* **CodecID**

Decoder ID in FourCC format.

*mfxU16* **reserved[8]**

Reserved for future use.

*mfxU16* **MaxcodecLevel**

Maximum supported codec level. See the CodecProfile enumerator for possible values.

*mfxU16* **NumProfiles**

Number of supported profiles.

**struct mfxDecoderDescription::decoder::decprofile \*Profiles**

Pointer to the array of profiles supported by the codec.

**struct decprofile**

This structure represents the codec profile description.

## Public Members

*mfxU32* **Profile**

Profile ID. See the CodecProfile enumerator for possible values.

*mfxU16* **reserved[7]**

Reserved for future use.

*mfxU16* **NumMemTypes**

Number of supported memory types.

**struct mfxDecoderDescription::decoder::decprofile::decmemdesc \*MemDesc**

Pointer to the array of memory types.

**struct decmemdesc**

This structure represents the underlying details of the memory type.

## Public Members

*mfx ResourceType* **MemHandleType**

Memory handle type.

*mfxRange32U* **Width**

Range of supported image widths.

*mfxRange32U* **Height**

Range of supported image heights.

*mfxU16* **reserved[7]**

Reserved for future use.

*mfxU16* **NumColorFormats**

Number of supported output color formats.

*mfxU32* **\*ColorFormats**

Pointer to the array of supported output color formats (in FOURCC).

### 12.7.2.8 mfxEncoderDescription

**struct mfxEncoderDescription**

This structure represents an encoder description.

#### Public Members

*mfxStructVersion Version*

Version of the structure.

*mfxU16 reserved[7]*

Reserved for future use.

*mfxU16 NumCodecs*

Number of supported encoders.

**struct mfxEncoderDescription::encoder \*Codecs**

Pointer to the array of encoders.

**struct encoder**

This structure represents encoder description.

#### Public Members

*mfxU32 CodecID*

Encoder ID in FourCC format.

*mfxU16 MaxcodecLevel*

Maximum supported codec level. See the CodecProfile enumerator for possible values.

*mfxU16 BiDirectionalPrediction*

Indicates B-frames support.

*mfxU16 reserved[7]*

Reserved for future use.

*mfxU16 NumProfiles*

Number of supported profiles.

**struct mfxEncoderDescription::encoder::encprofile \*Profiles**

Pointer to the array of profiles supported by the codec.

**struct encprofile**

This structure represents the codec profile description.

#### Public Members

*mfxU32 Profile*

Profile ID. See the CodecProfile enumerator for possible values.

*mfxU16 reserved[7]*

Reserved for future use.

*mfxU16 NumMemTypes*

Number of supported memory types.

**struct mfxEncoderDescription::encoder::encprofile::encmemdesc \*MemDesc**

Pointer to the array of memory types.

**struct encmemdesc**

This structure represents the underlying details of the memory type.

**Public Members*****mfxResourceType MemHandleType***

Memory handle type.

***mfxRange32U Width***

Range of supported image widths.

***mfxRange32U Height***

Range of supported image heights.

***mfxU16 reserved[7]***

Reserved for future use.

***mfxU16 NumColorFormats***

Number of supported input color formats.

***mfxU32 \*ColorFormats***

Pointer to the array of supported input color formats (in FOURCC).

**12.7.2.9 mfxVPPDescription****struct mfxVPPDescription**

This structure represents VPP description.

**Public Members*****mfxStructVersion Version***

Version of the structure.

***mfxU16 reserved[7]***

Reserved for future use.

***mfxU16 NumFilters***

Number of supported VPP filters.

***struct mfxVPPDescription::filter \*Filters***

Pointer to the array of supported filters.

***struct filter***

This structure represents the VPP filters description.

**Public Members*****mfxU32 FilterFourCC***

Filter ID in FourCC format.

***mfxU16 MaxDelayInFrames***

Introduced output delay in frames.

***mfxU16 reserved[7]***

Reserved for future use.

***mfxU16 NumMemTypes***

Number of supported memory types.

***struct mfxVPPDescription::filter::memdesc \*MemDesc***

Pointer to the array of memory types.

***struct memdesc***

This structure represents the underlying details of the memory type.

**Public Members*****mfx ResourceType MemHandleType***

Memory handle type.

***mfxRange32U Width***

Range of supported image widths.

***mfxRange32U Height***

Range of supported image heights.

***mfxU16 reserved[7]***

Reserved for future use.

***mfxU16 NumInFormats***

Number of supported input color formats.

***struct mfxVPPDescription::filter::memdesc::format \*Formats***

Pointer to the array of supported formats.

***struct format***

This structure represents the input color format description.

**Public Members*****mfxU32 InFormat***

Input color in FourCC format.

***mfxU16 reserved[5]***

Reserved for future use.

***mfxU16 NumOutFormat***

Number of supported output color formats.

***mfxU32 \*OutFormats***

Pointer to the array of supported output color formats (in FOURCC).

**12.7.2.10 mfxDeviceDescription*****struct mfxDeviceDescription***

This structure represents device description.

## Public Members

*mfxStructVersion* **Version**

Version of the structure.

*mfxU16* **reserved[7]**

reserved for future use.

*mfxChar* **DeviceID[MFX\_STRFIELD\_LEN]**

Null terminated string with device ID.

*mfxU16* **NumSubDevices**

Number of available uniform sub-devices. Pure software implementation can report 0.

**struct mfxDeviceDescription::subdevices \*SubDevices**

Pointer to the array of available sub-devices.

**struct subdevices**

This structure represents sub-device description.

## Public Members

*mfxU32* **Index**

Index of the sub-device, started from 0 and increased by 1.

*mfxChar* **SubDeviceID[MFX\_STRFIELD\_LEN]**

Null terminated string with unique sub-device ID, mapped to the system ID.

*mfxU32* **reserved[7]**

reserved for future use.

### 12.7.2.11 mfxImplDescription

**struct mfxImplDescription**

This structure represents the implementation description.

## Public Members

*mfxStructVersion* **Version**

Version of the structure.

*mfxImplType* **Impl**

Impl type: software/hardware.

*mfxAccelerationMode* **AccelerationMode**

Hardware acceleration stack to use. OS dependent parameter. Use VA for Linux\* and DX\* for Windows\*.

*mfxVersion* **ApiVersion**

Supported API version.

*mfxChar* **ImplName[MFX\_IMPL\_NAME\_LEN]**

Null-terminated string with implementation name given by vendor.

*mfxChar* **License[MFX\_STRFIELD\_LEN]**

Null-terminated string with license name of the implementation.

*mfxChar* **Keywords**[MFX\_STRFIELD\_LEN]

Null-terminated string with comma-separated list of keywords specific to this implementation that dispatcher can search for.

*mfxU32* **VendorID**

Standard vendor ID 0x8086 - Intel.

*mfxU32* **VendorImplID**

Vendor specific number with given implementation ID.

*mfxDeviceDescription* **Dev**

Supported device.

*mfxDecoderDescription* **Dec**

Decoders configuration.

*mfxEncoderDescription* **Enc**

Encoders configuration.

*mfxVPPDescription* **VPP**

VPP configuration.

*mfxU32* **reserved[16]**

Reserved for future use.

*mfxU32* **NumExtParam**

Number of extension buffers. Reserved for future use. Must be 0.

*mfxExtBuffer* \*\***ExtParam**

Array of extension buffers.

*mfxU64* **Reserved2**

Reserved for future use.

**union mfxImplDescription::[anonymous] ExtParams**

Extension buffers. Reserved for future.

### 12.7.2.12 Functions

*mfxLoader* **MFXLoad()**

This function creates the SDK loader.

**Return** Loader SDK loader handle or NULL if failed.

**void MFXUnload (*mfxLoader* loader)**

This function destroys the SDK dispatcher.

#### Parameters

- [in] loader: SDK loader handle.

*mfxConfig* **MFXCreateConfig (*mfxLoader* loader)**

This function creates dispatcher configuration.

This function creates the dispatcher internal configuration, which is used to filter out available implementations. Then this configuration is used to walk through selected implementations to gather more details and select the appropriate implementation to load. The loader object remembers all created mfxConfig objects and destroys them during the mfxUnload function call.

Multiple configurations per single mfxLoader object are possible.

Usage example:

```
mfxLoader loader = MFXLoad();
mfxConfig cfg = MFXCreateConfig(loader);
MFXCreateSession(loader, 0, &session);
```

**Return** SDK config handle or NULL pointer is failed.

#### Parameters

- [in] loader: SDK loader handle.

*mfxStatus MFXSetConfigFilterProperty (mfxConfig config, const mfxU8 \*name, mfxVariant value)*

This function is used to add additional filter properties (any fields of the *mfxImplDescription* structure) to the configuration of the SDK loader object. One mfxConfig properties can hold only single filter property.

Simple usage example:

```
mfxLoader loader = MFXLoad();
mfxConfig cfg = MFXCreateConfig(loader);
mfxVariant ImplValue;
ImplValue.Type = MFX_VARIANT_TYPE_U32;
ImplValue.Data.U32 = MFX_IMPL_TYPE_HARDWARE;
MFXSetConfigFilterProperty(cfg, "mfxImplDescription.Impl", ImplValue);
MFXCreateSession(loader, 0, &session);
```

**Note** Each new call with the same parameter name will overwrite the previously set value. This may invalidate other properties.

**Note** Each new call with another parameter name will delete the previous property and create a new property based on new name's value.

Usage example with two sessions (multiple loaders):

```
// Create session with software based implementation
mfxLoader loader1 = MFXLoad();
mfxConfig cfg1 = MFXCreateConfig(loader1);
mfxVariant ImplValueSW;
ImplValueSW.Type = MFX_VARIANT_TYPE_U32;
ImplValueSW.Data.U32 = MFX_IMPL_TYPE_SOFTWARE;
MFXSetConfigFilterProperty(cfg1, "mfxImplDescription.Impl", ImplValueSW);
MFXCreateSession(loader1, 0, &sessionSW);

// Create session with hardware based implementation
mfxLoader loader2 = MFXLoad();
mfxConfig cfg2 = MFXCreateConfig(loader2);
mfxVariant ImplValueHW;
ImplValueHW.Type = MFX_VARIANT_TYPE_U32;
ImplValueHW.Data.U32 = MFX_IMPL_TYPE_HARDWARE;
MFXSetConfigFilterProperty(cfg2, "mfxImplDescription.Impl", ImplValueHW);
MFXCreateSession(loader2, 0, &sessionHW);

// use both sessionSW and sessionHW
// ...
// Close everything
MFXClose(sessionSW);
MFXClose(sessionHW);
```

(continues on next page)

(continued from previous page)

```
MFXUnload(loader1); // cfg1 will be destroyed here.  
MFXUnload(loader2); // cfg2 will be destroyed here.
```

Usage example with two decoders (multiple config objects):

```
mfxLoader loader = MFXLoad();  
  
mfxConfig cfg1 = MFXCreateConfig(loader);  
mfxVariant ImplValue;  
val.Type = MFX_VARIANT_TYPE_U32;  
val.Data.U32 = MFX_CODEC_AVC;  
MFXSetConfigFilterProperty(cfg1, "mfxImplDescription.mfxDecoderDescription.decoder.  
CodecID", ImplValue);  
  
mfxConfig cfg2 = MFXCreateConfig(loader);  
mfxVariant ImplValue;  
val.Type = MFX_VARIANT_TYPE_U32;  
val.Data.U32 = MFX_CODEC_HEVC;  
MFXSetConfigFilterProperty(cfg2, "mfxImplDescription.mfxDecoderDescription.decoder.  
CodecID", ImplValue);  
  
MFXCreateSession(loader, 0, &sessionAVC);  
MFXCreateSession(loader, 0, &sessionHEVC);
```

**Return** MFX\_ERR\_NONE The function completed successfully. MFX\_ERR\_NULL\_PTR If config is NULL. MFX\_ERR\_NULL\_PTR If name is NULL.

MFX\_ERR\_NOT\_FOUND If name contains unknown parameter name. MFX\_ERR\_UNSUPPORTED If value data type doesn't equal to the parameter with provided name.

#### Parameters

- [in] config: SDK config handle.
- [in] name: Name of the parameter (see *mfxImplDescription* structure and example).
- [in] value: Value of the parameter.

*mfxStatus MFXEnumImplementations (mfxLoader loader, mfxU32 i, mfxImplCapsDeliveryFormat format,  
mfxHDL \*idesc)*

This function is used to iterate over filtered out implementations to gather their details. This function allocates memory to store *mfxImplDescription* structure instance. Use the MFXDispReleaseImplDescription function to free memory allocated to the *mfxImplDescription* structure.

**Return** MFX\_ERR\_NONE The function completed successfully. The idesc contains valid information. MFX\_ERR\_NULL\_PTR If loader is NULL.

MFX\_ERR\_NULL\_PTR If idesc is NULL.

MFX\_ERR\_NOT\_FOUND Provided index is out of possible range.

MFX\_ERR\_UNSUPPORTED If requested format isn't supported.

#### Parameters

- [in] loader: SDK loader handle.
- [in] i: Index of the implementation.

- [in] format: Format in which capabilities need to be delivered. See the mfxImplCapsDeliveryFormat enumerator for more details.
- [out] idesc: Pointer to the *mfxImplDescription* structure.

*mfxStatus* **MFXCreateSession** (*mfxLoader* loader, *mfxU32* i, *mfxSession* \*session)

This function is used to load and initialize the implementation.

```
mfxLoader loader = MFXLoad();
int i=0;
while(1) {
    mfxImplDescription *idesc;
    MFXEnumImplementations(loader, i, MFX_IMPLCAPS_IMPLDESCSTRUCTURE, (mfxHDL*)&
    ↵idesc);
    if(is_good(idesc)) {
        MFXCreateSession(loader, i,&session);
        // ...
        MFXDispReleaseImplDescription(loader, idesc);
    }
    else
    {
        MFXDispReleaseImplDescription(loader, idesc);
        break;
    }
}
```

**Return** MFX\_ERR\_NONE The function completed successfully. The session contains pointer to the SDK session handle. MFX\_ERR\_NULL\_PTR If loader is NULL.

MFX\_ERR\_NULL\_PTR If session is NULL.

MFX\_ERR\_NOT\_FOUND Provided index is out of possible range.

#### Parameters

- [in] loader: SDK loader handle.
- [in] i: Index of the implementation.
- [out] session: Pointer to the SDK session handle.

*mfxStatus* **MFXDispReleaseImplDescription** (*mfxLoader* loader, *mfxHDL* hdl)

This function destroys handle allocated by MFXQueryImplsCapabilities function.

**Return** MFX\_ERR\_NONE The function completed successfully. MFX\_ERR\_NULL\_PTR If loader is NULL.

MFX\_ERR\_INVALID\_HANDLE Provided hdl handle isn't associated with this loader.

#### Parameters

- [in] loader: SDK loader handle.
- [in] hdl: Handle to destroy. Can be equal to NULL.

## 12.7.3 Enums

### 12.7.3.1 mfxStatus

#### enum mfxStatus

Itemizes status codes returned by SDK functions.

*Values:*

**enumerator MFX\_ERR\_NONE = 0**

No error.

**enumerator MFX\_ERR\_UNKNOWN = -1**

Unknown error.

**enumerator MFX\_ERR\_NULL\_PTR = -2**

Null pointer.

**enumerator MFX\_ERR\_UNSUPPORTED = -3**

Unsupported feature.

**enumerator MFX\_ERR\_MEMORY\_ALLOC = -4**

Failed to allocate memory.

**enumerator MFX\_ERR\_NOT\_ENOUGH\_BUFFER = -5**

Insufficient buffer at input/output.

**enumerator MFX\_ERR\_INVALID\_HANDLE = -6**

Invalid handle.

**enumerator MFX\_ERR\_LOCK\_MEMORY = -7**

Failed to lock the memory block.

**enumerator MFX\_ERR\_NOT\_INITIALIZED = -8**

Member function called before initialization.

**enumerator MFX\_ERR\_NOT\_FOUND = -9**

The specified object is not found.

**enumerator MFX\_ERR\_MORE\_DATA = -10**

Expect more data at input.

**enumerator MFX\_ERR\_MORE\_SURFACE = -11**

Expect more surface at output.

**enumerator MFX\_ERR\_ABORTED = -12**

Operation aborted.

**enumerator MFX\_ERR\_DEVICE\_LOST = -13**

Lose the hardware acceleration device.

**enumerator MFX\_ERR\_INCOMPATIBLE\_VIDEO\_PARAM = -14**

Incompatible video parameters.

**enumerator MFX\_ERR\_INVALID\_VIDEO\_PARAM = -15**

Invalid video parameters.

**enumerator MFX\_ERR\_UNDEFINED\_BEHAVIOR = -16**

Undefined behavior.

**enumerator MFX\_ERR\_DEVICE\_FAILED = -17**

Device operation failure.

---

```
enumerator MFX_ERR_MORE_BITSTREAM = -18
    Expect more bitstream buffers at output.

enumerator MFX_ERR_GPU_HANG = -21
    Device operation failure caused by GPU hang.

enumerator MFX_ERR_REALLOC_SURFACE = -22
    Bigger output surface required.

enumerator MFX_ERR_RESOURCE_MAPPED = -23
    Write access is already acquired and user requested another write access, or read access with
    MFX_MEMORY_NO_WAIT flag.

enumerator MFX_ERR_NOT_IMPLEMENTED = -24
    Feature or function not implemented.

enumerator MFX_WRN_IN_EXECUTION = 1
    The previous asynchronous operation is in execution.

enumerator MFX_WRN_DEVICE_BUSY = 2
    The hardware acceleration device is busy.

enumerator MFX_WRN_VIDEO_PARAM_CHANGED = 3
    The video parameters are changed during decoding.

enumerator MFX_WRN_PARTIAL_ACCELERATION = 4
    Software acceleration is used.

enumerator MFX_WRN_INCOMPATIBLE_VIDEO_PARAM = 5
    Incompatible video parameters.

enumerator MFX_WRN_VALUE_NOT_CHANGED = 6
    The value is saturated based on its valid range.

enumerator MFX_WRN_OUT_OF_RANGE = 7
    The value is out of valid range.

enumerator MFX_WRN_FILTER_SKIPPED = 10
    One of requested filters has been skipped.

enumerator MFX_ERR_NONE_PARTIAL_OUTPUT = 12
    Frame is not ready, but bitstream contains partial output.

enumerator MFX_TASK_DONE = MFX_ERR_NONE
    Task has been completed.

enumerator MFX_TASK_WORKING = 8
    There is some more work to do.

enumerator MFX_TASK_BUSY = 9
    Task is waiting for resources.

enumerator MFX_ERR_MORE_DATA_SUBMIT_TASK = -10000
    Return MFX_ERR_MORE_DATA but submit internal asynchronous task.
```

### 12.7.3.2 mfxIMPL

**typedef mfxI32 mfxIMPL**

This enumerator itemizes SDK implementation types. The implementation type is a bit OR'ed value of the base type and any decorative flags.

**Note** This enumerator is for legacy dispatcher compatibility. New dispatcher doesn't use it.

**enumerator MFX\_IMPL\_AUTO = 0x0000**

Auto Selection/In or Not Supported/Out.

**enumerator MFX\_IMPL\_SOFTWARE = 0x0001**

Pure software implementation.

**enumerator MFX\_IMPL\_HARDWARE = 0x0002**

Hardware accelerated implementation (default device).

**enumerator MFX\_IMPL\_AUTO\_ANY = 0x0003**

Auto selection of any hardware/software implementation.

**enumerator MFX\_IMPL\_HARDWARE\_ANY = 0x0004**

Auto selection of any hardware implementation.

**enumerator MFX\_IMPL\_HARDWARE2 = 0x0005**

Hardware accelerated implementation (2nd device).

**enumerator MFX\_IMPL\_HARDWARE3 = 0x0006**

Hardware accelerated implementation (3rd device).

**enumerator MFX\_IMPL\_HARDWARE4 = 0x0007**

Hardware accelerated implementation (4th device).

**enumerator MFX\_IMPL\_RUNTIME = 0x0008**

This value cannot be used for session initialization. It may be returned by MFXQueryIMPL function to show that session has been initialized in run time mode.

**enumerator MFX\_IMPL\_VIA\_ANY = 0x0100**

Hardware acceleration can go through any supported OS infrastructure. This is default value, it is used by the SDK if none of MFX\_IMPL\_VIA\_xxx flag is specified by application.

**enumerator MFX\_IMPL\_VIA\_D3D9 = 0x0200**

Hardware acceleration goes through the Microsoft\* Direct3D\* 9 infrastructure.

**enumerator MFX\_IMPL\_VIA\_D3D11 = 0x0300**

Hardware acceleration goes through the Microsoft\* Direct3D\* 11 infrastructure.

**enumerator MFX\_IMPL\_VIA\_VAAPI = 0x0400**

Hardware acceleration goes through the Linux\* VA-API infrastructure.

**enumerator MFX\_IMPL\_UNSUPPORTED = 0x0000**

One of the MFXQueryIMPL returns.

**MFX\_IMPL\_BASETYPE (x)**

The application can use the macro MFX\_IMPL\_BASETYPE(x) to obtain the base implementation type.

### 12.7.3.3 mfxImplCapsDeliveryFormat

**enum mfxImplCapsDeliveryFormat**

*Values:*

**enumerator MFX\_IMPLCAPS\_IMPLDESCSTRUCTURE = 1**

Deliver capabilities as *mfxImplDescription* structure.

### 12.7.3.4 mfxPriority

**enum mfxPriority**

The mfxPriority enumerator describes the session priority.

*Values:*

**enumerator MFX\_PRIORITY\_LOW = 0**

Low priority: the session operation halts when high priority tasks are executing and more than 75% of the CPU is being used for normal priority tasks.

**enumerator MFX\_PRIORITY\_NORMAL = 1**

Normal priority: the session operation is halted if there are high priority tasks.

**enumerator MFX\_PRIORITY\_HIGH = 2**

High priority: the session operation blocks other lower priority session operations.

### 12.7.3.5 GPUCopy

**enumerator MFX\_GPUCOPY\_DEFAULT = 0**

Use default mode for the current SDK implementation.

**enumerator MFX\_GPUCOPY\_ON = 1**

Enable GPU accelerated copying.

**enumerator MFX\_GPUCOPY\_OFF = 2**

Disable GPU accelerated copying.

### 12.7.3.6 PlatformCodeName

**enumerator MFX\_PLATFORM\_UNKNOWN = 0**

Unknown platform.

**enumerator MFX\_PLATFORM\_SANDYBRIDGE = 1**

Intel(r) microarchitecture code name Sandy Bridge.

**enumerator MFX\_PLATFORM\_IVYBRIDGE = 2**

Intel(r) microarchitecture code name Ivy Bridge.

**enumerator MFX\_PLATFORM\_HASWELL = 3**

Code name Haswell.

**enumerator MFX\_PLATFORM\_BAYTRAIL = 4**

Code name Bay Trail.

**enumerator MFX\_PLATFORM\_BROADWELL = 5**

Intel(r) microarchitecture code name Broadwell.

**enumerator MFX\_PLATFORM\_CHERRYTRAIL = 6**

Code name Cherry Trail.

---

```

enumerator MFX_PLATFORM_SKYLAKE = 7
    Intel(r) microarchitecture code name Skylake.

enumerator MFX_PLATFORM_APOLLOLAKE = 8
    Code name Apollo Lake.

enumerator MFX_PLATFORM_KABYLAKE = 9
    Code name Kaby Lake.

enumerator MFX_PLATFORM_GEMINILAKE = 10
    Code name Gemini Lake.

enumerator MFX_PLATFORM_COFFEE LAKE = 11
    Code name Coffee Lake.

enumerator MFX_PLATFORM_CANNONLAKE = 20
    Code name Cannon Lake.

enumerator MFX_PLATFORM_ICELAKE = 30
    Code name Ice Lake.

enumerator MFX_PLATFORM_JASPERLAKE = 32
    Code name Jasper Lake.

enumerator MFX_PLATFORM_ELKHARTLAKE = 33
    Code name Elkhart Lake.

enumerator MFX_PLATFORM_TIGERLAKE = 40
    Code name Tiger Lake.

```

### 12.7.3.7 mfxMediaAdapterType

#### **enum mfxMediaAdapterType**

The mfxMediaAdapterType enumerator itemizes types of graphics adapters.

*Values:*

```

enumerator MFX_MEDIA_UNKNOWN = 0xffff
    Unknown type.

enumerator MFX_MEDIA_INTEGRATED = 0
    Integrated graphics adapter.

enumerator MFX_MEDIA_DISCRETE = 1
    Discrete graphics adapter.

```

### 12.7.3.8 mfxMemoryFlags

#### **enum mfxMemoryFlags**

The mfxMemoryFlags enumerator specifies memory access mode.

*Values:*

```

enumerator MFX_MAP_READ = 0x1
    The surface is mapped for reading.

enumerator MFX_MAP_WRITE = 0x2
    The surface is mapped for writing.

enumerator MFX_MAP_READ_WRITE = MFX_MAP_READ | MFX_MAP_WRITE
    The surface is mapped for reading and writing.

```

**enumerator MFX\_MAP\_NOWAIT** = 0x10

The mapping would be done immediately without any implicit synchronizations.

**Attention** This flag is optional.

### 12.7.3.9 mfx ResourceType

**enum mfx ResourceType**

*Values:*

**enumerator MFX\_RESOURCE\_SYSTEM\_SURFACE** = 1

System memory.

**enumerator MFX\_RESOURCE\_VA\_SURFACE** = 2

VA surface.

**enumerator MFX\_RESOURCE\_VA\_BUFFER** = 3

VA buffer.

**enumerator MFX\_RESOURCE\_DX9\_SURFACE** = 4

IDirect3DSurface9.

**enumerator MFX\_RESOURCE\_DX11\_TEXTURE** = 5

ID3D11Texture2D.

**enumerator MFX\_RESOURCE\_DX12\_RESOURCE** = 6

ID3D12Resource.

**enumerator MFX\_RESOURCE\_DMA\_RESOURCE** = 7

DMA resource.

### 12.7.3.10 ColorFourCC

The ColorFourCC enumerator itemizes color formats.

**enumerator MFX\_FOURCC\_NV12** = MFX\_MAKEFOURCC('N', 'V', '1', '2')

NV12 color planes. Native format for 4:2:0/8b Gen hardware implementation.

**enumerator MFX\_FOURCC\_NV21** = MFX\_MAKEFOURCC('N', 'V', '2', '1')

Same as NV12 but with weaved V and U values.

**enumerator MFX\_FOURCC\_YV12** = MFX\_MAKEFOURCC('Y', 'V', '1', '2')

YV12 color planes.

**enumerator MFX\_FOURCC\_IYUV** = MFX\_MAKEFOURCC('I', 'Y', 'U', 'V')

Same as YV12 except that the U and V plane order is reversed.

**enumerator MFX\_FOURCC\_I420** = [MFX\\_FOURCC\\_IYUV](#)

Alias for the IYUV color format.

**enumerator MFX\_FOURCC\_NV16** = MFX\_MAKEFOURCC('N', 'V', '1', '6')

4:2:2 color format with similar to NV12 layout.

**enumerator MFX\_FOURCC\_YUY2** = MFX\_MAKEFOURCC('Y', 'U', 'Y', '2')

YUY2 color planes.

**enumerator MFX\_FOURCC\_RGB565** = MFX\_MAKEFOURCC('R', 'G', 'B', '2')

2 bytes per pixel, uint16 in little-endian format, where 0-4 bits are blue, bits 5-10 are green and bits 11-15 are red.

**enumerator MFX\_FOURCC\_RGBP** = MFX\_MAKEFOURCC('R', 'G', 'B', 'P')

RGB 24 bit planar layout (3 separate channels, 8-bits per sample each). This format should be mapped to D3DFMT\_R8G8B8 or VA\_FOURCC\_RGBP.

**enumerator MFX\_FOURCC\_RGB4** = MFX\_MAKEFOURCC('R', 'G', 'B', '4')

RGB4 (RGB32) color planes. BGRA is the order, 'B' is 8 MSBs, then 8 bits for 'G' channel, then 'R' and 'A' channels.

**enumerator MFX\_FOURCC\_BGRA** = *MFX\_FOURCC\_RGB4*

Alias for the RGB4 color format.

**enumerator MFX\_FOURCC\_P8** = 41

Internal SDK color format. The application should use one of the functions below to create such surface, depending on Direct3D version.

Direct3D\* 9: IDirectXVideoDecoderService::CreateSurface()

Direct3D\* 11: ID3D11Device::CreateBuffer()

**enumerator MFX\_FOURCC\_P8\_TEXTURE** = MFX\_MAKEFOURCC('P', '8', 'M', 'B')

Internal SDK color format. The application should use one of the functions below to create such surface, depending on Direct3D\* version.

Direct3D 9: IDirectXVideoDecoderService::CreateSurface()

Direct3D 11: ID3D11Device::CreateTexture2D()

**enumerator MFX\_FOURCC\_P010** = MFX\_MAKEFOURCC('P', '0', '1', '0')

P010 color format. This is 10 bit per sample format with similar to NV12 layout. This format should be mapped to DXGI\_FORMAT\_P010.

**enumerator MFX\_FOURCC\_I010** = MFX\_MAKEFOURCC('I', '0', '1', '0')

10-bit YUV 4:2:0, each component has its own plane.

**enumerator MFX\_FOURCC\_P016** = MFX\_MAKEFOURCC('P', '0', '1', '6')

P016 color format. This is 16 bit per sample format with similar to NV12 layout. This format should be mapped to DXGI\_FORMAT\_P016.

**enumerator MFX\_FOURCC\_P210** = MFX\_MAKEFOURCC('P', '2', '1', '0')

10 bit per sample 4:2:2 color format with similar to NV12 layout.

**enumerator MFX\_FOURCC\_BGR4** = MFX\_MAKEFOURCC('B', 'G', 'R', '4')

RGBA color format. It is similar to MFX\_FOURCC\_RGB4 but with different order of channels. 'R' is 8 MSBs, then 8 bits for 'G' channel, then 'B' and 'A' channels.

**enumerator MFX\_FOURCC\_A2RGB10** = MFX\_MAKEFOURCC('R', 'G', '1', '0')

10 bits ARGB color format packed in 32 bits. 'A' channel is two MSBs, then 'R', then 'G' and then 'B' channels. This format should be mapped to DXGI\_FORMAT\_R10G10B10A2\_UNORM or D3DFMT\_A2R10G10B10.

**enumerator MFX\_FOURCC\_ARGB16** = MFX\_MAKEFOURCC('R', 'G', '1', '6')

10 bits ARGB color format packed in 64 bits. 'A' channel is 16 MSBs, then 'R', then 'G' and then 'B' channels. This format should be mapped to DXGI\_FORMAT\_R16G16B16A16\_UINT or D3DFMT\_A16B16G16R16 formats.

**enumerator MFX\_FOURCC\_ABGR16** = MFX\_MAKEFOURCC('B', 'G', '1', '6')

10 bits ABGR color format packed in 64 bits. 'A' channel is 16 MSBs, then 'B', then 'G' and then 'R' channels. This format should be mapped to DXGI\_FORMAT\_R16G16B16A16\_UINT or D3DFMT\_A16B16G16R16 formats.

**enumerator MFX\_FOURCC\_R16** = MFX\_MAKEFOURCC('R', '1', '6', 'U')

16 bits single channel color format. This format should be mapped to DXGI\_FORMAT\_R16\_TYPELESS or D3DFMT\_R16F.

---

```

enumerator MFX_FOURCC_AYUV = MFX_MAKEFOURCC('A', 'Y', 'U', 'V')
    YUV 4:4:4, AYUV color format. This format should be mapped to DXGI_FORMAT_AYUV.

enumerator MFX_FOURCC_AYUV_RGB4 = MFX_MAKEFOURCC('A', 'V', 'U', 'Y')
    RGB4 stored in AYUV surface. This format should be mapped to DXGI_FORMAT_AYUV.

enumerator MFX_FOURCC_UYVY = MFX_MAKEFOURCC('U', 'Y', 'V', 'Y')
    UYVY color planes. Same as YUY2 except the byte order is reversed.

enumerator MFX_FOURCC_Y210 = MFX_MAKEFOURCC('Y', '2', '1', '0')
    10 bit per sample 4:2:2 packed color format with similar to YUY2 layout. This format should be mapped to DXGI_FORMAT_Y210.

enumerator MFX_FOURCC_Y410 = MFX_MAKEFOURCC('Y', '4', '1', '0')
    10 bit per sample 4:4:4 packed color format. This format should be mapped to DXGI_FORMAT_Y410.

enumerator MFX_FOURCC_Y216 = MFX_MAKEFOURCC('Y', '2', '1', '6')
    16 bit per sample 4:2:2 packed color format with similar to YUY2 layout. This format should be mapped to DXGI_FORMAT_Y216.

enumerator MFX_FOURCC_Y416 = MFX_MAKEFOURCC('Y', '4', '1', '6')
    16 bit per sample 4:4:4 packed color format. This format should be mapped to DXGI_FORMAT_Y416.

```

### 12.7.3.11 ChromaFormatIdc

The ChromaFormatIdc enumerator itemizes color-sampling formats.

```

enumerator MFX_CHROMAFORMAT_MONOCHROME = 0
    Monochrome.

enumerator MFX_CHROMAFORMAT_YUV420 = 1
    4:2:0 color.

enumerator MFX_CHROMAFORMAT_YUV422 = 2
    4:2:2 color.

enumerator MFX_CHROMAFORMAT_YUV444 = 3
    4:4:4 color.

enumerator MFX_CHROMAFORMAT_YUV400 = MFX\_CHROMAFORMAT\_MONOCHROME
    Equal to monochrome.

enumerator MFX_CHROMAFORMAT_YUV411 = 4
    4:1:1 color.

enumerator MFX_CHROMAFORMAT_YUV422H = MFX\_CHROMAFORMAT\_YUV422
    4:2:2 color, horizontal sub-sampling. It is equal to 4:2:2 color.

enumerator MFX_CHROMAFORMAT_YUV422V = 5
    4:2:2 color, vertical sub-sampling.

enumerator MFX_CHROMAFORMAT_RESERVED1 = 6
    Reserved.

enumerator MFX_CHROMAFORMAT_JPEG_SAMPLING = 6
    Color sampling specified via mfxInfoMFX::SamplingFactorH and SamplingFactorV.

```

### 12.7.3.12 PicStruct

The PicStruct enumerator itemizes picture structure. Use bit-OR'ed values to specify the desired picture type.

**enumerator MFX\_PICSTRUCT\_UNKNOWN** = 0x00

Unspecified or mixed progressive/interlaced/field pictures.

**enumerator MFX\_PICSTRUCT\_PROGRESSIVE** = 0x01

Progressive picture.

**enumerator MFX\_PICSTRUCT\_FIELD\_TFF** = 0x02

Top field in first interlaced picture.

**enumerator MFX\_PICSTRUCT\_FIELD\_BFF** = 0x04

Bottom field in first interlaced picture.

**enumerator MFX\_PICSTRUCT\_FIELD\_REPEATED** = 0x10

First field repeated: pic\_struct=5 or 6 in H.264.

**enumerator MFX\_PICSTRUCT\_FRAME\_DOUBLING** = 0x20

Double the frame for display: pic\_struct=7 in H.264.

**enumerator MFX\_PICSTRUCT\_FRAME\_TRIPLING** = 0x40

Triple the frame for display: pic\_struct=8 in H.264.

**enumerator MFX\_PICSTRUCT\_FIELD\_SINGLE** = 0x100

Single field in a picture.

**enumerator MFX\_PICSTRUCT\_FIELD\_TOP** = *MFX\_PICSTRUCT\_FIELD\_SINGLE | MFX\_PICSTRUCT\_FIELD\_TFF*

Top field in a picture: pic\_struct = 1 in H.265.

**enumerator MFX\_PICSTRUCT\_FIELD\_BOTTOM** = *MFX\_PICSTRUCT\_FIELD\_SINGLE | MFX\_PICSTRUCT\_FIELD\_BFF*

Bottom field in a picture: pic\_struct = 2 in H.265.

**enumerator MFX\_PICSTRUCT\_FIELD\_PAIED\_PREV** = 0x200

Paired with previous field: pic\_struct = 9 or 10 in H.265.

**enumerator MFX\_PICSTRUCT\_FIELD\_PAIED\_NEXT** = 0x400

Paired with next field: pic\_struct = 11 or 12 in H.265

### 12.7.3.13 Frame Data Flags

**enumerator MFX\_TIMESTAMP\_UNKNOWN** = -1

Indicates that time stamp is unknown for this frame/bitstream portion.

**enumerator MFX\_FRAMEORDER\_UNKNOWN** = -1

Unused entry or SDK functions that generate the frame output do not use this frame.

**enumerator MFX\_FRAMEDATA\_ORIGINAL\_TIMESTAMP** = 0x0001

Indicates the time stamp of this frame is not calculated and is a pass-through of the original time stamp.

#### 12.7.3.14 Corruption

The Corruption enumerator itemizes the decoding corruption types. It is a bit-OR'ed value of the following.

**enumerator MFX\_CORRUPTION\_MINOR = 0x0001**

Minor corruption in decoding certain macro-blocks.

**enumerator MFX\_CORRUPTION\_MAJOR = 0x0002**

Major corruption in decoding the frame - incomplete data, for example.

**enumerator MFX\_CORRUPTION\_ABSENT\_TOP\_FIELD = 0x0004**

Top field of frame is absent in bitstream. Only bottom field has been decoded.

**enumerator MFX\_CORRUPTION\_ABSENT\_BOTTOM\_FIELD = 0x0008**

Bottom field of frame is absent in bitstream. Only top field has been decoded.

**enumerator MFX\_CORRUPTION\_REFERENCE\_FRAME = 0x0010**

Decoding used a corrupted reference frame. A corrupted reference frame was used for decoding this frame. For example, if the frame uses a reference frame that was decoded with minor/major corruption flag, then this frame is also marked with a reference corruption flag.

**enumerator MFX\_CORRUPTION\_REFERENCE\_LIST = 0x0020**

The reference list information of this frame does not match what is specified in the Reference Picture Marking Repetition SEI message. (ITU-T H.264 D.1.8 dec\_ref\_pic\_marking\_repetition)

**Note:** Flag MFX\_CORRUPTION\_ABSENT\_TOP\_FIELD/MFX\_CORRUPTION\_ABSENT\_BOTTOM\_FIELD is set by the AVC decoder when it detects that one of fields is not present in the bitstream. Which field is absent depends on value of bottom\_field\_flag (ITU-T\* H.264 7.4.3).

#### 12.7.3.15 TimeStampCalc

The TimeStampCalc enumerator itemizes time-stamp calculation methods.

**enumerator MFX\_TIMESTAMPCALC\_UNKNOWN = 0**

The time stamp calculation is to base on the input frame rate, if time stamp is not explicitly specified.

**enumerator MFX\_TIMESTAMPCALC\_TELECINE = 1**

Adjust time stamp to 29.97fps on 24fps progressively encoded sequences if telecine attributes are available in the bitstream and time stamp is not explicitly specified. The input frame rate must be specified.

#### 12.7.3.16 IOPattern

The IOPattern enumerator itemizes memory access patterns for SDK functions. Use bit-ORed values to specify an input access pattern and an output access pattern.

**enumerator MFX\_IOPATTERN\_IN\_VIDEO\_MEMORY = 0x01**

Input to SDK functions is a video memory surface.

**enumerator MFX\_IOPATTERN\_IN\_SYSTEM\_MEMORY = 0x02**

Input to SDK functions is a linear buffer directly in system memory or in system memory through an external allocator.

**enumerator MFX\_IOPATTERN\_OUT\_VIDEO\_MEMORY = 0x10**

Output to SDK functions is a video memory surface.

---

```
enumerator MFX_IOPATTERN_OUT_SYSTEM_MEMORY = 0x20
```

Output to SDK functions is a linear buffer directly in system memory or in system memory through an external allocator.

### 12.7.3.17 CodecFormatFourCC

The CodecFormatFourCC enumerator itemizes codecs in the FourCC format.

```
enumerator MFX_CODEC_AVC = MFX_MAKEFOURCC('A', 'V', 'C', '')
```

AVC, H.264, or MPEG-4, part 10 codec.

```
enumerator MFX_CODEC_HEVC = MFX_MAKEFOURCC('H', 'E', 'V', 'C')
```

HEVC codec.

```
enumerator MFX_CODEC_MPEG2 = MFX_MAKEFOURCC('M', 'P', 'G', '2')
```

MPEG-2 codec.

```
enumerator MFX_CODEC_VC1 = MFX_MAKEFOURCC('V', 'C', '1', '')
```

VC-1 codec.

```
enumerator MFX_CODEC_VP9 = MFX_MAKEFOURCC('V', 'P', '9', '')
```

VP9 codec.

```
enumerator MFX_CODEC_AV1 = MFX_MAKEFOURCC('A', 'V', '1', '')
```

AV1 codec.

```
enumerator MFX_CODEC_JPEG = MFX_MAKEFOURCC('J', 'P', 'E', 'G')
```

JPEG codec

### 12.7.3.18 CodecProfile

The CodecProfile enumerator itemizes codec profiles for all codecs.

```
enumerator MFX_PROFILE_UNKNOWN = 0
```

Unspecified profile.

H.264 profiles:

```
enumerator MFX_PROFILE_AVC_BASELINE = 66
```

```
enumerator MFX_PROFILE_AVC_MAIN = 77
```

```
enumerator MFX_PROFILE_AVC_EXTENDED = 88
```

```
enumerator MFX_PROFILE_AVC_HIGH = 100
```

```
enumerator MFX_PROFILE_AVC_HIGH10 = 110
```

```
enumerator MFX_PROFILE_AVC_HIGH_422 = 122
```

```
enumerator MFX_PROFILE_AVC_CONSTRAINED_BASELINE = MFX_PROFILE_AVC_BASELINE + MFX_PROFILE_AVC_CONSTRAINED_BASELINE
```

```
enumerator MFX_PROFILE_AVC_CONSTRAINED_HIGH = MFX_PROFILE_AVC_HIGH + MFX_PROFILE_AVC_CONSTRAINED_HIGH
```

Combined with H.264 profile these flags impose additional constraints. See the H.264 specification for the list of constraints.

```
enumerator MFX_PROFILE_AVC_CONSTRAINT_SET0 = (0x100 << 0)
```

```
enumerator MFX_PROFILE_AVC_CONSTRAINT_SET1 = (0x100 << 1)
```

```
enumerator MFX_PROFILE_AVC_CONSTRAINT_SET2 = (0x100 << 2)
```

```
enumerator MFX_PROFILE_AVC_CONSTRAINT_SET3 = (0x100 << 3)
```

```
enumerator MFX_PROFILE_AVC_CONSTRAINT_SET4 = (0x100 << 4)
enumerator MFX_PROFILE_AVC_CONSTRAINT_SET5 = (0x100 << 5)
```

#### 12.7.3.18.1 Multi-view Video Coding Extension Profiles

```
enumerator MFX_PROFILE_AVC_MULTIVIEW_HIGH = 118
    Multi-view high profile.

enumerator MFX_PROFILE_AVC_STEREO_HIGH = 128
    Stereo high profile.
```

#### 12.7.3.18.2 MPEG-2 Profiles

```
enumerator MFX_PROFILE_MPEG2_SIMPLE = 0x50
enumerator MFX_PROFILE_MPEG2_MAIN = 0x40
enumerator MFX_PROFILE_MPEG2_HIGH = 0x10
```

#### 12.7.3.18.3 VC-1 Profiles

```
enumerator MFX_PROFILE_VC1_SIMPLE = (0 + 1)
enumerator MFX_PROFILE_VC1_MAIN = (4 + 1)
enumerator MFX_PROFILE_VC1_ADVANCED = (12 + 1)
```

#### 12.7.3.18.4 HEVC Profiles

```
enumerator MFX_PROFILE_HEVC_MAIN = 1
enumerator MFX_PROFILE_HEVC_MAIN10 = 2
enumerator MFX_PROFILE_HEVC_MAINSP = 3
enumerator MFX_PROFILE_HEVC_REXT = 4
enumerator MFX_PROFILE_HEVC_SCC = 9
```

#### 12.7.3.18.5 VP9 Profiles

```
enumerator MFX_PROFILE_VP8_0 = 0 + 1
enumerator MFX_PROFILE_VP8_1 = 1 + 1
enumerator MFX_PROFILE_VP8_2 = 2 + 1
enumerator MFX_PROFILE_VP8_3 = 3 + 1
```

#### 12.7.3.18.6 VP9 Profiles

```
enumerator MFX_PROFILE_VP9_0 = 1
enumerator MFX_PROFILE_VP9_1 = 2
enumerator MFX_PROFILE_VP9_2 = 3
enumerator MFX_PROFILE_VP9_3 = 4
```

#### 12.7.3.18.7 JPEG Profiles

```
enumerator MFX_PROFILE_JPEG_BASELINE = 1
    Baseline JPEG profile.
```

### 12.7.3.19 CodecLevel

The CodecLevel enumerator itemizes codec levels for all codecs.

```
enumerator MFX_LEVEL_UNKNOWN = 0
    Unspecified level.
```

#### 12.7.3.19.1 H.264 Level 1-1.3

```
enumerator MFX_LEVEL_AVC_1 = 10
enumerator MFX_LEVEL_AVC_1b = 9
enumerator MFX_LEVEL_AVC_11 = 11
enumerator MFX_LEVEL_AVC_12 = 12
enumerator MFX_LEVEL_AVC_13 = 13
```

#### 12.7.3.19.2 H.264 Level 2-2.2

```
enumerator MFX_LEVEL_AVC_2 = 20
enumerator MFX_LEVEL_AVC_21 = 21
enumerator MFX_LEVEL_AVC_22 = 22
```

#### 12.7.3.19.3 H.264 Level 3-3.2

```
enumerator MFX_LEVEL_AVC_3 = 30
enumerator MFX_LEVEL_AVC_31 = 31
enumerator MFX_LEVEL_AVC_32 = 32
```

#### 12.7.3.19.4 H.264 Level 4-4.2

```
enumerator MFX_LEVEL_AVC_4 = 40  
enumerator MFX_LEVEL_AVC_41 = 41  
enumerator MFX_LEVEL_AVC_42 = 42
```

#### 12.7.3.19.5 H.264 Level 5-5.2

```
enumerator MFX_LEVEL_AVC_5 = 50  
enumerator MFX_LEVEL_AVC_51 = 51  
enumerator MFX_LEVEL_AVC_52 = 52
```

#### 12.7.3.19.6 MPEG2 Levels

```
enumerator MFX_LEVEL_MPEG2_LOW = 0xA  
enumerator MFX_LEVEL_MPEG2_MAIN = 0x8  
enumerator MFX_LEVEL_MPEG2_HIGH = 0x4  
enumerator MFX_LEVEL_MPEG2_HIGH1440 = 0x6
```

#### 12.7.3.19.7 VC-1 Level Low (Simple and Main Profiles)

```
enumerator MFX_LEVEL_VC1_LOW = (0 + 1)  
enumerator MFX_LEVEL_VC1_MEDIAN = (2 + 1)  
enumerator MFX_LEVEL_VC1_HIGH = (4 + 1)
```

#### 12.7.3.20 VC-1 Advanced Profile Levels

```
enumerator MFX_LEVEL_VC1_0 = (0x00 + 1)  
enumerator MFX_LEVEL_VC1_1 = (0x01 + 1)  
enumerator MFX_LEVEL_VC1_2 = (0x02 + 1)  
enumerator MFX_LEVEL_VC1_3 = (0x03 + 1)  
enumerator MFX_LEVEL_VC1_4 = (0x04 + 1)
```

#### 12.7.3.20.1 HEVC Levels

```
enumerator MFX_LEVEL_HEVC_1 = 10  
enumerator MFX_LEVEL_HEVC_2 = 20  
enumerator MFX_LEVEL_HEVC_21 = 21  
enumerator MFX_LEVEL_HEVC_3 = 30  
enumerator MFX_LEVEL_HEVC_31 = 31
```

---

```
enumerator MFX_LEVEL_HEVC_4 = 40
enumerator MFX_LEVEL_HEVC_41 = 41
enumerator MFX_LEVEL_HEVC_5 = 50
enumerator MFX_LEVEL_HEVC_51 = 51
enumerator MFX_LEVEL_HEVC_52 = 52
enumerator MFX_LEVEL_HEVC_6 = 60
enumerator MFX_LEVEL_HEVC_61 = 61
enumerator MFX_LEVEL_HEVC_62 = 62
```

### 12.7.3.21 HEVC Tiers

```
enumerator MFX_TIER_HEVC_MAIN = 0
enumerator MFX_TIER_HEVC_HIGH = 0x100
```

### 12.7.3.22 GopOptFlag

The GopOptFlag enumerator itemizes special properties in the GOP (Group of Pictures) sequence.

```
enumerator MFX_GOP_CLOSED = 1
```

The encoder generates closed GOP if this flag is set. Frames in this GOP do not use frames in previous GOP as reference.

The encoder generates open GOP if this flag is not set. In this GOP frames prior to the first frame of GOP in display order may use frames from previous GOP as reference. Frames subsequent to the first frame of GOP in display order do not use frames from previous GOP as reference.

The AVC encoder ignores this flag if IdrInterval in *mfxInfoMFX* structure is set to 0, i.e. if every GOP starts from IDR frame. In this case, GOP is encoded as closed.

This flag does not affect long-term reference frames.

```
enumerator MFX_GOP_STRICT = 2
```

The encoder must strictly follow the given GOP structure as defined by parameter GopPicSize, GopRefDist etc in the *mfxVideoParam* structure. Otherwise, the encoder can adapt the GOP structure for better efficiency, whose range is constrained by parameter GopPicSize and GopRefDist etc. See also description of AdaptiveI and AdaptiveB fields in the *mfxExtCodingOption2* structure.

### 12.7.3.23 TargetUsage

The TargetUsage enumerator itemizes a range of numbers from MFX\_TARGETUSAGE\_1, best quality, to MFX\_TARGETUSAGE\_7, best speed. It indicates trade-offs between quality and speed. The application can use any number in the range. The actual number of supported target usages depends on implementation. If specified target usage is not supported, the SDK encoder will use the closest supported value.

```
enumerator MFX_TARGETUSAGE_1 = 1
```

Best quality

```
enumerator MFX_TARGETUSAGE_2 = 2
```

```
enumerator MFX_TARGETUSAGE_3 = 3
```

---

```

enumerator MFX_TARGETUSAGE_4 = 4
    Balanced quality and speed.

enumerator MFX_TARGETUSAGE_5 = 5

enumerator MFX_TARGETUSAGE_6 = 6

enumerator MFX_TARGETUSAGE_7 = 7
    Best speed

enumerator MFX_TARGETUSAGE_UNKNOWN = 0
    Unspecified target usage.

enumerator MFX_TARGETUSAGE_BEST_QUALITY = MFX_TARGETUSAGE_1
    Best quality.

enumerator MFX_TARGETUSAGE_BALANCED = MFX_TARGETUSAGE_4
    Balanced quality and speed.

enumerator MFX_TARGETUSAGE_BEST_SPEED = MFX_TARGETUSAGE_7
    Best speed.

```

#### 12.7.3.24 RateControlMethod

The RateControlMethod enumerator itemizes bitrate control methods.

```

enumerator MFX_RATECONTROL_CBR = 1
    Use the constant bitrate control algorithm.

enumerator MFX_RATECONTROL_VBR = 2
    Use the variable bitrate control algorithm.

enumerator MFX_RATECONTROL_CQP = 3
    Use the constant quantization parameter algorithm.

enumerator MFX_RATECONTROL_AVBR = 4
    Use the average variable bitrate control algorithm.

enumerator MFX_RATECONTROL_LA = 8
    Use the VBR algorithm with look ahead. It is a special bitrate control mode in the SDK AVC encoder that has been designed to improve encoding quality. It works by performing extensive analysis of several dozen frames before the actual encoding and as a side effect significantly increases encoding delay and memory consumption.

    The only available rate control parameter in this mode is mfxInfoMFX::TargetKbps. Two other parameters, MaxKbps and InitialDelayInKB, are ignored. To control LA depth the application can use mfxExtCodingOption2::LookAheadDepth parameter.

    This method is not HRD compliant.

enumerator MFX_RATECONTROL_ICQ = 9
    Use the Intelligent Constant Quality algorithm. This algorithm improves subjective video quality of encoded stream. Depending on content, it may or may not decrease objective video quality. Only one control parameter is used - quality factor, specified by mfxInfoMFX::ICQQuality.

enumerator MFX_RATECONTROL_VCM = 10
    Use the Video Conferencing Mode algorithm. This algorithm is similar to the VBR and uses the same set of parameters mfxInfoMFX::InitialDelayInKB, TargetKbps and MaxKbps. It is tuned for IPPP GOP pattern and streams with strong temporal correlation between frames. It produces better objective and subjective video quality in these conditions than other bitrate control algorithms. It does not support interlaced content, B frames and produced stream is not HRD compliant.

```

**enumerator MFX\_RATECONTROL\_LA\_ICQ = 11**

Use Intelligent Constant Quality algorithm with look ahead. Quality factor is specified by *mfxInfoMFX::ICQQuality*. To control LA depth the application can use *mfxExtCodingOption2::LookAheadDepth* parameter.

This method is not HRD compliant.

**enumerator MFX\_RATECONTROL\_LA\_HRD = 13**

MFX\_RATECONTROL\_LA\_EXT has been removed

Use HRD compliant look ahead rate control algorithm.

**enumerator MFX\_RATECONTROL\_QVBR = 14**

Use the variable bitrate control algorithm with constant quality. This algorithm trying to achieve the target subjective quality with the minimum number of bits, while the bitrate constraint and HRD compliance are satisfied. It uses the same set of parameters as VBR and quality factor specified by *mfxExtCodingOption3::QVBRQuality*.

### 12.7.3.25 TrellisControl

The TrellisControl enumerator is used to control trellis quantization in AVC encoder. The application can turn it on or off for any combination of I, P, and B frames by combining different enumerator values. For example, MFX\_TRELLIS\_I | MFX\_TRELLIS\_B turns it on for I and B frames.

**enumerator MFX\_TRELLIS\_UNKNOWN = 0**

Default value, it is up to the SDK encoder to turn trellis quantization on or off.

**enumerator MFX\_TRELLIS\_OFF = 0x01**

Turn trellis quantization off for all frame types.

**enumerator MFX\_TRELLIS\_I = 0x02**

Turn trellis quantization on for I frames.

**enumerator MFX\_TRELLIS\_P = 0x04**

Turn trellis quantization on for P frames.

**enumerator MFX\_TRELLIS\_B = 0x08**

Turn trellis quantization on for B frames.

### 12.7.3.26 BRefControl

The BRefControl enumerator is used to control usage of B frames as reference in AVC encoder.

**enumerator MFX\_B\_REF\_UNKNOWN = 0**

Default value, it is up to the SDK encoder to use B frames as reference.

**enumerator MFX\_B\_REF\_OFF = 1**

Do not use B frames as reference.

**enumerator MFX\_B\_REF\_PYRAMID = 2**

Arrange B frames in so-called “B pyramid” reference structure.

### 12.7.3.27 LookAheadDownSampling

The LookAheadDownSampling enumerator is used to control down sampling in look ahead bitrate control mode in AVC encoder.

**enumerator MFX\_LOOKAHEAD\_DS\_UNKNOWN = 0**

Default value, it is up to the SDK encoder what down sampling value to use.

**enumerator MFX\_LOOKAHEAD\_DS\_OFF = 1**

Do not use down sampling, perform estimation on original size frames. This is the slowest setting that produces the best quality.

**enumerator MFX\_LOOKAHEAD\_DS\_2x = 2**

Down sample frames two times before estimation.

**enumerator MFX\_LOOKAHEAD\_DS\_4x = 3**

Down sample frames four times before estimation. This option may significantly degrade quality.

### 12.7.3.28 BPSEIControl

The BPSEIControl enumerator is used to control insertion of buffering period SEI in the encoded bitstream.

**enumerator MFX\_BPSEI\_DEFAULT = 0x00**

encoder decides when to insert BP SEI.

**enumerator MFX\_BPSEI\_IFRAME = 0x01**

BP SEI should be inserted with every I frame

### 12.7.3.29 SkipFrame

The SkipFrame enumerator is used to define usage of `mfxEncodeCtrl::SkipFrame` parameter.

**enumerator MFX\_SKIPFRAME\_NO\_SKIP = 0**

Frame skipping is disabled, `mfxEncodeCtrl::SkipFrame` is ignored.

**enumerator MFX\_SKIPFRAME\_INSERT\_DUMMY = 1**

Skipping is allowed, when `mfxEncodeCtrl::SkipFrame` is set encoder inserts into bitstream frame where all macroblocks are encoded as skipped. Only non-reference P and B frames can be skipped. If GopRefDist = 1 and `mfxEncodeCtrl::SkipFrame` is set for reference P frame, it will be encoded as non-reference.

**enumerator MFX\_SKIPFRAME\_INSERT NOTHING = 2**

Similar to MFX\_SKIPFRAME\_INSERT\_DUMMY, but when `mfxEncodeCtrl::SkipFrame` is set encoder inserts nothing into bitstream.

**enumerator MFX\_SKIPFRAME\_BRC\_ONLY = 3**

`mfxEncodeCtrl::SkipFrame` indicates number of missed frames before the current frame. Affects only BRC, current frame will be encoded as usual.

### 12.7.3.30 IntraRefreshTypes

The IntraRefreshTypes enumerator itemizes types of intra refresh.

**enumerator MFX\_REFRESH\_NO = 0**

Encode without refresh.

**enumerator MFX\_REFRESH\_VERTICAL = 1**

Vertical refresh, by column of MBs.

**enumerator MFX\_REFRESH\_HORIZONTAL = 2**

Horizontal refresh, by rows of MBs.

**enumerator MFX\_REFRESH\_SLICE = 3**

Horizontal refresh by slices without overlapping.

### 12.7.3.31 WeightedPred

The WeightedPred enumerator itemizes weighted prediction modes.

**enumerator MFX\_WEIGHTED\_PRED\_UNKNOWN = 0**

Allow encoder to decide.

**enumerator MFX\_WEIGHTED\_PRED\_DEFAULT = 1**

Use default weighted prediction.

**enumerator MFX\_WEIGHTED\_PRED\_EXPLICIT = 2**

Use explicit weighted prediction.

**enumerator MFX\_WEIGHTED\_PRED\_IMPLICIT = 3**

Use implicit weighted prediction (for B-frames only).

### 12.7.3.32 PRefType

The PRefType enumerator itemizes models of reference list construction and DPB management when GopRefDist=1.

**enumerator MFX\_P\_REF\_DEFAULT = 0**

Allow encoder to decide.

**enumerator MFX\_P\_REF\_SIMPLE = 1**

Regular sliding window used for DPB removal process.

**enumerator MFX\_P\_REF\_PYRAMID = 2**

Let N be the max reference list's size. Encoder treat each N's frame as ‘strong’ reference and the others as “weak” references. Encoder uses ‘weak’ reference only for prediction of the next frame and removes it from DPB right after. ‘Strong’ references removed from DPB by sliding window.

### 12.7.3.33 ScenarioInfo

The ScenarioInfo enumerator itemizes scenarios for the encoding session.

```
enumerator MFX_SCENARIO_UNKNOWN = 0
enumerator MFX_SCENARIO_DISPLAY_Remoting = 1
enumerator MFX_SCENARIO_VIDEO_CONFERENCE = 2
enumerator MFX_SCENARIO_ARCHIVE = 3
enumerator MFX_SCENARIO_LIVE_STREAMING = 4
enumerator MFX_SCENARIO_CAMERA_CAPTURE = 5
enumerator MFX_SCENARIO_VIDEO_SURVEILLANCE = 6
enumerator MFX_SCENARIO_GAME_STREAMING = 7
enumerator MFX_SCENARIO_REMOTE_GAMING = 8
```

### 12.7.3.34 ContentInfo

The ContentInfo enumerator itemizes content types for the encoding session.

```
enumerator MFX_CONTENT_UNKNOWN = 0
enumerator MFX_CONTENT_FULL_SCREEN_VIDEO = 1
enumerator MFX_CONTENT_NON_VIDEO_SCREEN = 2
```

### 12.7.3.35 IntraPredBlockSize/InterPredBlockSize

IntraPredBlockSize/InterPredBlockSize specifies minimum block size of inter-prediction.

```
enumerator MFX_BLOCKSIZE_UNKNOWN = 0
    Unspecified.
enumerator MFX_BLOCKSIZE_MIN_16x16 = 1
    16x16
enumerator MFX_BLOCKSIZE_MIN_8x8 = 2
    16x16, 8x8
enumerator MFX_BLOCKSIZE_MIN_4x4 = 3
    16x16, 8x8, 4x4
```

### 12.7.3.36 MVPrecision

The MVPrecision enumerator specifies the motion estimation precision

```
enumerator MFX_MVPRECISION_UNKNOWN = 0
enumerator MFX_MVPRECISION_INTEGER = (1 << 0)
enumerator MFX_MVPRECISION_HALFPEL = (1 << 1)
enumerator MFX_MVPRECISION_QUARTERPEL = (1 << 2)
```

### 12.7.3.37 CodingOptionValue

The CodingOptionValue enumerator defines a three-state coding option setting.

<b>enumerator MFX_CODINGOPTION_UNKNOWN = 0</b>	Unspecified.
<b>enumerator MFX_CODINGOPTION_ON = 0x10</b>	Coding option set.
<b>enumerator MFX_CODINGOPTION_OFF = 0x20</b>	Coding option not set.
<b>enumerator MFX_CODINGOPTION_ADAPTIVE = 0x30</b>	Reserved

### 12.7.3.38 BitstreamDataFlag

The BitstreamDataFlag enumerator uses bit-ORed values to itemize additional information about the bitstream buffer.

<b>enumerator MFX_BITSTREAM_COMPLETE_FRAME = 0x0001</b>	
	The bitstream buffer contains a complete frame or complementary field pair of data for the bitstream. For decoding, this means that the decoder can proceed with this buffer without waiting for the start of the next frame, which effectively reduces decoding latency. If this flag is set, but the bitstream buffer contains incomplete frame or pair of field, then decoder will produce corrupted output.
<b>enumerator MFX_BITSTREAM_EOS = 0x0002</b>	

The bitstream buffer contains the end of the stream. For decoding, this means that the application does not have any additional bitstream data to send to decoder.

### 12.7.3.39 ExtendedBufferID

The ExtendedBufferID enumerator itemizes and defines identifiers (BufferId) for extended buffers or video processing algorithm identifiers.

<b>enumerator MFX_EXTBUFF_THREADS_PARAM = MFX_MAKEFOURCC('T', 'H', 'D', 'P')</b>	<i>mfxExtThreadsParam</i> buffer ID
<b>enumerator MFX_EXTBUFF_CODING_OPTION = MFX_MAKEFOURCC('C', 'D', 'O', 'P')</b>	This extended buffer defines additional encoding controls. See the <i>mfxExtCodingOption</i> structure for details. The application can attach this buffer to the structure for encoding initialization.
<b>enumerator MFX_EXTBUFF_CODING_OPTION_SPSPPS = MFX_MAKEFOURCC('C', 'O', 'S', 'P')</b>	This extended buffer defines sequence header and picture header for encoders and decoders. See the <i>mfxExtCodingOptionsSPSPPS</i> structure for details. The application can attach this buffer to the <i>mfxVideoParam</i> structure for encoding initialization, and for obtaining raw headers from the decoders and encoders.
<b>enumerator MFX_EXTBUFF_VPP_DONOTUSE = MFX_MAKEFOURCC('N', 'U', 'S', 'E')</b>	This extended buffer defines a list of VPP algorithms that applications should not use. See the <i>mfxExtVPP-DoNotUse</i> structure for details. The application can attach this buffer to the <i>mfxVideoParam</i> structure for video processing initialization.
<b>enumerator MFX_EXTBUFF_VPP_AUXDATA = MFX_MAKEFOURCC('A', 'U', 'X', 'D')</b>	This extended buffer defines auxiliary information at the VPP output. See the <i>mfxExtVppAuxData</i> structure for details. The application can attach this buffer to the <i>mfxEncodeCtrl</i> structure for per-frame encoding control.

---

**enumerator MFX\_EXTBUFF\_VPP\_DENOISE = MFX\_MAKEFOURCC('D', 'N', 'I', 'S')**

The extended buffer defines control parameters for the VPP denoise filter algorithm. See the *mfxExtVPPDenoise* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for video processing initialization.

**enumerator MFX\_EXTBUFF\_VPP\_SCENE\_ANALYSIS = MFX\_MAKEFOURCC('S', 'C', 'L', 'Y')**

**enumerator MFX\_EXTBUFF\_VPP\_PROCAMP = MFX\_MAKEFOURCC('P', 'A', 'M', 'P')**

The extended buffer defines control parameters for the VPP ProcAmp filter algorithm. See the *mfxExtVPPProcAmp* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for video processing initialization or to the *mfxFrameData* structure in the *mfxFrameSurface1* structure of output surface for per-frame processing configuration.

**enumerator MFX\_EXTBUFF\_VPP\_DETAIL = MFX\_MAKEFOURCC('D', 'E', 'T', 'I')**

The extended buffer defines control parameters for the VPP detail filter algorithm. See the *mfxExtVPPDetail* structure for details. The application can attach this buffer to the structure for video processing initialization.

**enumerator MFX\_EXTBUFF\_VIDEO\_SIGNAL\_INFO = MFX\_MAKEFOURCC('V', 'S', 'T', 'N')**

This extended buffer defines video signal type. See the *mfxExtVideoSignalInfo* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding initialization, and for retrieving such information from the decoders.

**enumerator MFX\_EXTBUFF\_VPP\_DOUSE = MFX\_MAKEFOURCC('D', 'U', 'S', 'E')**

This extended buffer defines a list of VPP algorithms that applications should use. See the *mfxExtVPPDoUse* structure for details. The application can attach this buffer to the structure for video processing initialization.

**enumerator MFX\_EXTBUFF\_AVC\_REFLIST\_CTRL = MFX\_MAKEFOURCC('R', 'L', 'S', 'T')**

This extended buffer defines additional encoding controls for reference list. See the *mfxExtAVCRefListCtrl* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding & decoding initialization, or the *mfxEncodeCtrl* structure for per-frame encoding configuration.

**enumerator MFX\_EXTBUFF\_VPP\_FRAME\_RATE\_CONVERSION = MFX\_MAKEFOURCC('F', 'R', 'C', 'I')**

This extended buffer defines control parameters for the VPP frame rate conversion algorithm. See the *mfxExtVPPFrameRateConversion* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for video processing initialization.

**enumerator MFX\_EXTBUFF\_PICTURE\_TIMING\_SEI = MFX\_MAKEFOURCC('P', 'T', 'S', 'E')**

This extended buffer configures the H.264 picture timing SEI message. See the *mfxExtPictureTimingSEI* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding initialization, or the *mfxEncodeCtrl* structure for per-frame encoding configuration.

**enumerator MFX\_EXTBUFF\_AVC\_TEMPORAL\_LAYERS = MFX\_MAKEFOURCC('A', 'T', 'M', 'L')**

This extended buffer configures the structure of temporal layers inside the encoded H.264 bitstream. See the *mfxExtAvcTemporalLayers* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding initialization.

**enumerator MFX\_EXTBUFF\_CODING\_OPTION2 = MFX\_MAKEFOURCC('C', 'D', 'O', '2')**

This extended buffer defines additional encoding controls. See the *mfxExtCodingOption2* structure for details. The application can attach this buffer to the structure for encoding initialization.

**enumerator MFX\_EXTBUFF\_VPP\_IMAGE\_STABILIZATION = MFX\_MAKEFOURCC('T', 'S', 'T', 'B')**

This extended buffer defines control parameters for the VPP image stabilization filter algorithm. See the *mfxExtVPPImageStab* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for video processing initialization.

**enumerator MFX\_EXTBUFF\_ENCODER\_CAPABILITY = MFX\_MAKEFOURCC('E', 'N', 'C', 'P')**

This extended buffer is used to retrieve SDK encoder capability. See the *mfxExtEncoderCapability* structure for details. The application can attach this buffer to the *mfxVideoParam* structure before calling MFXVideoENCODE\_Query function.

---

**enumerator MFX\_EXTBUFF\_ENCODER\_RESET\_OPTION** = MFX\_MAKEFOURCC('E', 'N', 'R', 'O')

This extended buffer is used to control encoder reset behavior and also to query possible encoder reset outcome. See the *mfxExtEncoderResetOption* structure for details. The application can attach this buffer to the *mfxVideoParam* structure before calling MFXVideoENCODE\_Query or MFXVideoENCODE\_Reset functions.

**enumerator MFX\_EXTBUFF\_ENCODED\_FRAME\_INFO** = MFX\_MAKEFOURCC('E', 'N', 'F', 'T')

This extended buffer is used by the SDK encoder to report additional information about encoded picture. See the *mfxExtAVCEncodedFrameInfo* structure for details. The application can attach this buffer to the *mfxBitstream* structure before calling MFXVideoENCODE\_EncodeFrameAsync function.

**enumerator MFX\_EXTBUFF\_VPP\_COMPOSITE** = MFX\_MAKEFOURCC('V', 'C', 'M', 'P')

This extended buffer is used to control composition of several input surfaces in the one output. In this mode, the VPP skips any other filters. The VPP returns error if any mandatory filter is specified and filter skipped warning for optional filter. The only supported filters are deinterlacing and interlaced scaling.

**enumerator MFX\_EXTBUFF\_VPP\_VIDEO\_SIGNAL\_INFO** = MFX\_MAKEFOURCC('V', 'V', 'S', 'T')

This extended buffer is used to control transfer matrix and nominal range of YUV frames. The application should provide it during initialization.

**enumerator MFX\_EXTBUFF\_ENCODER\_ROI** = MFX\_MAKEFOURCC('E', 'R', 'O', 'T')

This extended buffer is used by the application to specify different Region Of Interests during encoding. The application should provide it at initialization or at runtime.

**enumerator MFX\_EXTBUFF\_VPP\_DEINTERLACING** = MFX\_MAKEFOURCC('V', 'P', 'D', 'T')

This extended buffer is used by the application to specify different deinterlacing algorithms.

**enumerator MFX\_EXTBUFF\_AVC\_REFLISTS** = MFX\_MAKEFOURCC('R', 'L', 'T', 'S')

This extended buffer specifies reference lists for the SDK encoder.

**enumerator MFX\_EXTBUFF\_DEC\_VIDEO\_PROCESSING** = MFX\_MAKEFOURCC('D', 'E', 'C', 'V')

See the *mfxExtDecVideoProcessing* structure for details.

**enumerator MFX\_EXTBUFF\_VPP\_FIELD\_PROCESSING** = MFX\_MAKEFOURCC('F', 'P', 'R', 'O')

The extended buffer defines control parameters for the VPP field-processing algorithm. See the *mfxExtVPP-FieldProcessing* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for video processing initialization or to the *mfxFrameData* structure during runtime.

**enumerator MFX\_EXTBUFF\_CODING\_OPTION3** = MFX\_MAKEFOURCC('C', 'D', 'O', '3')

This extended buffer defines additional encoding controls. See the *mfxExtCodingOption3* structure for details.

The application can attach this buffer to the structure for encoding initialization.

**enumerator MFX\_EXTBUFF\_CHROMA\_LOC\_INFO** = MFX\_MAKEFOURCC('C', 'L', 'T', 'N')

This extended buffer defines chroma samples location information. See the *mfxExtChromaLocInfo* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding initialization.

**enumerator MFX\_EXTBUFF\_MBQP** = MFX\_MAKEFOURCC('M', 'B', 'Q', 'P')

This extended buffer defines per-macroblock QP. See the *mfxExtMBQP* structure for details. The application can attach this buffer to the *mfxEncodeCtrl* structure for per-frame encoding configuration.

**enumerator MFX\_EXTBUFF\_MB\_FORCE\_INTRA** = MFX\_MAKEFOURCC('M', 'B', 'F', 'I')

This extended buffer defines per-macroblock force intra flag. See the *mfxExtMBForceIntra* structure for details.

The application can attach this buffer to the *mfxEncodeCtrl* structure for per-frame encoding configuration.

**enumerator MFX\_EXTBUFF\_HEVC\_TILES** = MFX\_MAKEFOURCC('2', '6', '5', 'T')

This extended buffer defines additional encoding controls for HEVC tiles. See the *mfxExtHEVCTiles* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding initialization.

**enumerator MFX\_EXTBUFF\_MB\_DISABLE\_SKIP\_MAP** = MFX\_MAKEFOURCC('M', 'D', 'S', 'M')

This extended buffer defines macroblock map for current frame which forces specified macroblocks to be non skip. See the *mfxExtMBDisableSkipMap* structure for details. The application can attach this buffer to the *mfxEncodeCtrl* structure for per-frame encoding configuration.

**enumerator MFX\_EXTBUFF\_HEVC\_PARAM** = MFX\_MAKEFOURCC('2', '6', '5', 'P')

See the *mfxExtHEVCPParam* structure for details.

**enumerator MFX\_EXTBUFF\_DECODED\_FRAME\_INFO** = MFX\_MAKEFOURCC('D', 'E', 'F', 'T')

This extended buffer is used by SDK decoders to report additional information about decoded frame. See the *mfxExtDecodedFrameInfo* structure for more details.

**enumerator MFX\_EXTBUFF\_TIME\_CODE** = MFX\_MAKEFOURCC('T', 'M', 'C', 'D')

See the *mfxExtTimeCode* structure for more details.

**enumerator MFX\_EXTBUFF\_HEVC\_REGION** = MFX\_MAKEFOURCC('2', '6', '5', 'R')

This extended buffer specifies the region to encode. The application can attach this buffer to the *mfxVideoParam* structure during HEVC encoder initialization.

**enumerator MFX\_EXTBUFF\_PRED\_WEIGHT\_TABLE** = MFX\_MAKEFOURCC('E', 'P', 'W', 'T')

See the *mfxExtPredWeightTable* structure for details.

**enumerator MFX\_EXTBUFF\_DIRTY\_RECTANGLES** = MFX\_MAKEFOURCC('D', 'R', 'O', 'T')

See the *mfxExtDirtryRect* structure for details.

**enumerator MFX\_EXTBUFF\_MOVING\_RECTANGLES** = MFX\_MAKEFOURCC('M', 'R', 'O', 'T')

See the *mfxExtMoveRect* structure for details.

**enumerator MFX\_EXTBUFF\_CODING\_OPTION\_VPS** = MFX\_MAKEFOURCC('C', 'O', 'V', 'P')

See the *mfxExtCodingOptionVPS* structure for details.

**enumerator MFX\_EXTBUFF\_VPP\_ROTATION** = MFX\_MAKEFOURCC('R', 'O', 'T', '')

See the *mfxExtVPPRotation* structure for details.

**enumerator MFX\_EXTBUFF\_ENCODED\_SLICES\_INFO** = MFX\_MAKEFOURCC('E', 'N', 'S', 'T')

See the *mfxExtEncodedSlicesInfo* structure for details.

**enumerator MFX\_EXTBUFF\_VPP\_SCALING** = MFX\_MAKEFOURCC('V', 'S', 'C', 'L')

See the *mfxExtVPPScaling* structure for details.

**enumerator MFX\_EXTBUFF\_HEVC\_REFLIST\_CTRL** = *MFX\_EXTBUFF\_AVC\_REFLIST\_CTRL*

This extended buffer defines additional encoding controls for reference list. See the *mfxExtAVCRefListCtrl* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding & decoding initialization, or the *mfxEncodeCtrl* structure for per-frame encoding configuration.

**enumerator MFX\_EXTBUFF\_HEVC\_REFLISTS** = *MFX\_EXTBUFF\_AVC\_REFLISTS*

This extended buffer specifies reference lists for the SDK encoder.

**enumerator MFX\_EXTBUFF\_HEVC\_TEMPORAL\_LAYERS** = *MFX\_EXTBUFF\_AVC\_TEMPORAL\_LAYERS*

This extended buffer configures the structure of temporal layers inside the encoded H.264 bitstream. See the *mfxExtAvcTemporalLayers* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding initialization.

**enumerator MFX\_EXTBUFF\_VPP\_MIRRORING** = MFX\_MAKEFOURCC('M', 'T', 'R', 'R')

See the *mfxExtVPPMirroring* structure for details.

**enumerator MFX\_EXTBUFF\_MV\_OVER\_PIC\_BOUNDARIES** = MFX\_MAKEFOURCC('M', 'V', 'P', 'B')

See the *mfxExtMVOverPicBoundaries* structure for details.

**enumerator MFX\_EXTBUFF\_VPP\_COLORFILL** = MFX\_MAKEFOURCC('V', 'C', 'L', 'F')

See the *mfxExtVPPColorFill* structure for details.

**enumerator MFX\_EXTBUFF\_DECODE\_ERROR\_REPORT** = MFX\_MAKEFOURCC('D', 'E', 'R', 'R')

This extended buffer is used by SDK decoders to report error information before frames get decoded. See the *mfxExtDecodeErrorReport* structure for more details.

**enumerator MFX\_EXTBUFF\_VPP\_COLOR\_CONVERSION** = MFX\_MAKEFOURCC('V', 'C', 'S', 'C')

See the *mfxExtColorConversion* structure for details.

**enumerator MFX\_EXTBUFF\_CONTENT\_LIGHT\_LEVEL\_INFO** = MFX\_MAKEFOURCC('L', 'L', 'T', 'S')  
This extended buffer configures HDR SEI message. See the *mfxExtContentLightLevelInfo* structure for details.

**enumerator MFX\_EXTBUFF\_MASTERING\_DISPLAY\_COLOUR\_VOLUME** = MFX\_MAKEFOURCC('D', 'C', 'V', 'S')  
This extended buffer configures HDR SEI message. See the *mfxExtMasteringDisplayColourVolume* structure for details.

**enumerator MFX\_EXTBUFF\_MULTI\_FRAME\_PARAM** = MFX\_MAKEFOURCC('M', 'F', 'R', 'P')  
This extended buffer allow to specify multi-frame submission parameters.

**enumerator MFX\_EXTBUFF\_MULTI\_FRAME\_CONTROL** = MFX\_MAKEFOURCC('M', 'F', 'R', 'C')  
This extended buffer allow to manage multi-frame submission in runtime.

**enumerator MFX\_EXTBUFF\_ENCODED\_UNITS\_INFO** = MFX\_MAKEFOURCC('E', 'N', 'U', 'T')  
See the *mfxExtEncodedUnitsInfo* structure for details.

**enumerator MFX\_EXTBUFF\_VPP\_MCTF** = MFX\_MAKEFOURCC('M', 'C', 'T', 'F')  
This video processing algorithm identifier is used to enable MCTF via *mfxExtVPDoUse* and together with *mfxExtVppMctf*

**enumerator MFX\_EXTBUFF\_VP9\_SEGMENTATION** = MFX\_MAKEFOURCC('9', 'S', 'E', 'G')  
Extends *mfxVideoParam* structure with VP9 segmentation parameters. See the *mfxExtVP9Segmentation* structure for details.

**enumerator MFX\_EXTBUFF\_VP9\_TEMPORAL\_LAYERS** = MFX\_MAKEFOURCC('9', 'T', 'M', 'L')  
Extends *mfxVideoParam* structure with parameters for VP9 temporal scalability. See the *mfxExtVP9TemporalLayers* structure for details.

**enumerator MFX\_EXTBUFF\_VP9\_PARAM** = MFX\_MAKEFOURCC('9', 'P', 'A', 'R')  
Extends *mfxVideoParam* structure with VP9-specific parameters. See the *mfxExtVP9Param* structure for details.

**enumerator MFX\_EXTBUFF\_AVC\_ROUNDING\_OFFSET** = MFX\_MAKEFOURCC('R', 'N', 'D', 'O')  
See the *mfxExtAVCRoundingOffset* structure for details.

**enumerator MFX\_EXTBUFF\_PARTIAL\_BITSTREAM\_PARAM** = MFX\_MAKEFOURCC('P', 'B', 'O', 'P')  
See the *mfxExtPartialBitstreamParam* structure for details.

**enumerator MFX\_EXTBUFF\_BRC** = MFX\_MAKEFOURCC('E', 'B', 'R', 'C')

**enumerator MFX\_EXTBUFF\_VP8\_CODING\_OPTION** = MFX\_MAKEFOURCC('V', 'P', '8', 'E')  
This extended buffer describes VP8 encoder configuration parameters. See the *mfxExtVP8CodingOption* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding initialization.

**enumerator MFX\_EXTBUFF\_JPEG\_QT** = MFX\_MAKEFOURCC('J', 'P', 'G', 'Q')  
This extended buffer defines quantization tables for JPEG encoder.

**enumerator MFX\_EXTBUFF\_JPEG\_HUFFMAN** = MFX\_MAKEFOURCC('J', 'P', 'G', 'H')  
This extended buffer defines Huffman tables for JPEG encoder.

**enumerator MFX\_EXTBUFF\_ENCODER\_IPCM\_AREA** = MFX\_MAKEFOURCC('P', 'C', 'M', 'R')  
See the *mfxExtEncoderIPCMArea* structure for details.

**enumerator MFX\_EXTBUFF\_INSERT\_HEADERS** = MFX\_MAKEFOURCC('S', 'P', 'R', 'E')  
See the *mfxExtInsertHeaders* structure for details.

**enumerator MFX\_EXTBUFF\_MVC\_SEQ\_DESC** = MFX\_MAKEFOURCC('M', 'V', 'C', 'D')  
This extended buffer describes the MVC stream information of view dependencies, view identifiers, and operation points. See the ITU\*-T H.264 specification chapter H.7.3.2.1.4 for details.

**enumerator MFX\_EXTBUFF\_MVC\_TARGET\_VIEWS** = MFX\_MAKEFOURCC('M', 'V', 'C', 'T')  
This extended buffer defines target views at the decoder output.

---

```
enumerator MFX_EXTBUFF_CENC_PARAM = MFX_MAKEFOURCC('C', 'E', 'N', 'P')
```

This structure is used to pass decryption status report index for Common Encryption usage model. See the mfxExtCencParam structure for more details.

#### 12.7.3.40 PayloadCtrlFlags

The PayloadCtrlFlags enumerator itemizes additional payload properties.

```
enumerator MFX_PAYLOAD_CTRL_SUFFIX = 0x00000001
```

Insert this payload into HEVC Suffix SEI NAL-unit.

#### 12.7.3.41 ExtMemBufferType

```
enumerator MFX_MEMTYPE_PERSISTENT_MEMORY = 0x0002
```

Memory page for persistent use.

#### 12.7.3.42 ExtMemFrameType

The ExtMemFrameType enumerator specifies the memory type of frame. It is a bit-ORed value of one of the following. For information on working with video memory surfaces, see the section Working with hardware acceleration.

```
enumerator MFX_MEMTYPE_DXVA2_DECODER_TARGET = 0x0010
```

Frames are in video memory and belong to video decoder render targets.

```
enumerator MFX_MEMTYPE_DXVA2_PROCESSOR_TARGET = 0x0020
```

Frames are in video memory and belong to video processor render targets.

```
enumerator MFX_MEMTYPE_VIDEO_MEMORY_DECODER_TARGET = MFX_MEMTYPE_DXVA2_DECODER_TARGET
```

Frames are in video memory and belong to video decoder render targets.

```
enumerator MFX_MEMTYPE_VIDEO_MEMORY_PROCESSOR_TARGET = MFX_MEMTYPE_DXVA2_PROCESSOR_TARGET
```

Frames are in video memory and belong to video processor render targets.

```
enumerator MFX_MEMTYPE_SYSTEM_MEMORY = 0x0040
```

The frames are in system memory.

```
enumerator MFX_MEMTYPE_RESERVED1 = 0x0080
```

```
enumerator MFX_MEMTYPE_FROM_ENCODE = 0x0100
```

Allocation request comes from an ENCODE function

```
enumerator MFX_MEMTYPE_FROM_DECODE = 0x0200
```

Allocation request comes from a DECODE function

```
enumerator MFX_MEMTYPE_FROM_VPPIN = 0x0400
```

Allocation request comes from a VPP function for input frame allocation

```
enumerator MFX_MEMTYPE_FROM_VPPOUT = 0x0800
```

Allocation request comes from a VPP function for output frame allocation

```
enumerator MFX_MEMTYPE_FROM_ENC = 0x2000
```

Allocation request comes from an ENC function

```
enumerator MFX_MEMTYPE_INTERNAL_FRAME = 0x0001
```

Allocation request for internal frames

```
enumerator MFX_MEMTYPE_EXTERNAL_FRAME = 0x0002
```

Allocation request for I/O frames

**enumerator MFX\_MEMTYPE\_EXPORT\_FRAME** = 0x0008

Application requests frame handle export to some associated object. For Linux frame handle can be considered to be exported to DRM Prime FD, DRM FLink or DRM FrameBuffer Handle. Specifics of export types and export procedure depends on external frame allocator implementation

**enumerator MFX\_MEMTYPE\_SHARED\_RESOURCE** = *MFX\_MEMTYPE\_EXPORT\_FRAME*

For DX11 allocation use shared resource bind flag.

**enumerator MFX\_MEMTYPE\_VIDEO\_MEMORY\_ENCODER\_TARGET** = 0x1000

Frames are in video memory and belong to video encoder render targets.

#### 12.7.3.43 FrameType

The FrameType enumerator itemizes frame types. Use bit-ORed values to specify all that apply.

**enumerator MFX\_FRAMETYPE\_UNKNOWN** = 0x0000

Frame type is unspecified.

**enumerator MFX\_FRAMETYPE\_I** = 0x0001

This frame or the first field is encoded as an I frame/field.

**enumerator MFX\_FRAMETYPE\_P** = 0x0002

This frame or the first field is encoded as an P frame/field.

**enumerator MFX\_FRAMETYPE\_B** = 0x0004

This frame or the first field is encoded as an B frame/field.

**enumerator MFX\_FRAMETYPE\_S** = 0x0008

This frame or the first field is either an SI- or SP-frame/field.

**enumerator MFX\_FRAMETYPE\_REF** = 0x0040

This frame or the first field is encoded as a reference.

**enumerator MFX\_FRAMETYPE\_IDR** = 0x0080

This frame or the first field is encoded as an IDR.

**enumerator MFX\_FRAMETYPE\_xI** = 0x0100

The second field is encoded as an I-field.

**enumerator MFX\_FRAMETYPE\_xP** = 0x0200

The second field is encoded as an P-field.

**enumerator MFX\_FRAMETYPE\_xB** = 0x0400

The second field is encoded as an S-field.

**enumerator MFX\_FRAMETYPE\_xS** = 0x0800

The second field is an SI- or SP-field.

**enumerator MFX\_FRAMETYPE\_xREF** = 0x4000

The second field is encoded as a reference.

**enumerator MFX\_FRAMETYPE\_xIDR** = 0x8000

The second field is encoded as an IDR.

#### 12.7.3.44 MfxNalUnitType

The MfxNalUnitType enumerator specifies NAL unit types supported by the SDK HEVC encoder.

**enumerator MFX\_HEVC\_NALU\_TYPE\_UNKNOWN = 0**

The SDK encoder will decide what NAL unit type to use.

**enumerator MFX\_HEVC\_NALU\_TYPE\_TRAIL\_N = (0 + 1)**

See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

**enumerator MFX\_HEVC\_NALU\_TYPE\_TRAIL\_R = (1 + 1)**

See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

**enumerator MFX\_HEVC\_NALU\_TYPE\_RADL\_N = (6 + 1)**

See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

**enumerator MFX\_HEVC\_NALU\_TYPE\_RADL\_R = (7 + 1)**

See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

**enumerator MFX\_HEVC\_NALU\_TYPE\_RASL\_N = (8 + 1)**

See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

**enumerator MFX\_HEVC\_NALU\_TYPE\_RASL\_R = (9 + 1)**

See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

**enumerator MFX\_HEVC\_NALU\_TYPE\_IDR\_W\_RADL = (19 + 1)**

See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

**enumerator MFX\_HEVC\_NALU\_TYPE\_IDR\_N\_LP = (20 + 1)**

See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

**enumerator MFX\_HEVC\_NALU\_TYPE\_CRA\_NUT = (21 + 1)**

See Table 7-1 of the ITU-T H.265 specification for the definition of these type.

#### 12.7.3.45 mfxHandleType

**enum mfxHandleType**

The mfxHandleType enumerator itemizes system handle types that SDK implementations might use.

*Values:*

**enumerator MFX\_HANDLE\_DIRECT3D\_DEVICE\_MANAGER9 = 1**

Pointer to the IDirect3DDeviceManager9 interface. See Working with Microsoft\* DirectX\* Applications for more details on how to use this handle.

**enumerator MFX\_HANDLE\_D3D9\_DEVICE\_MANAGER = MFX\_HANDLE\_DIRECT3D\_DEVICE\_MANAGER9**

Pointer to the IDirect3DDeviceManager9 interface. See Working with Microsoft\* DirectX\* Applications for more details on how to use this handle.

**enumerator MFX\_HANDLE\_RESERVED1 = 2**

**enumerator MFX\_HANDLE\_D3D11\_DEVICE = 3**

Pointer to the ID3D11Device interface. See Working with Microsoft\* DirectX\* Applications for more details on how to use this handle.

**enumerator MFX\_HANDLE\_VA\_DISPLAY = 4**

Pointer to VADisplay interface. See Working with VA-API Applications for more details on how to use this handle.

**enumerator MFX\_HANDLE\_RESERVED3 = 5**

---

```
enumerator MFX_HANDLE_VA_CONFIG_ID = 6
    Pointer to VAConfigID interface. It represents external VA config for Common Encryption usage model.

enumerator MFX_HANDLE_VA_CONTEXT_ID = 7
    Pointer to VAContextID interface. It represents external VA context for Common Encryption usage model.
```

#### 12.7.3.46 mfxSkipMode

##### **enum mfxSkipMode**

The mfxSkipMode enumerator describes the decoder skip-mode options.

*Values:*

```
enumerator MFX_SKIPMODE_NOSKIP = 0
enumerator MFX_SKIPMODE_MORE = 1
    Do not skip any frames.

enumerator MFX_SKIPMODE_LESS = 2
    Skip more frames.
```

#### 12.7.3.47 FrcAlgm

The FrcAlgm enumerator itemizes frame rate conversion algorithms. See description of mfxExtVPPFrameRateConversion structure for more details.

##### **enumerator MFX\_FRCALGM\_PRESERVE\_TIMESTAMP = 0x0001**

Frame dropping/repetition based frame rate conversion algorithm with preserved original time stamps. Any inserted frames will carry MFX\_TIMESTAMP\_UNKNOWN.

##### **enumerator MFX\_FRCALGM\_DISTRIBUTED\_TIMESTAMP = 0x0002**

Frame dropping/repetition based frame rate conversion algorithm with distributed time stamps. The algorithm distributes output time stamps evenly according to the output frame rate.

##### **enumerator MFX\_FRCALGM\_FRAME\_INTERPOLATION = 0x0004**

Frame rate conversion algorithm based on frame interpolation. This flag may be combined with MFX\_FRCALGM\_PRESERVE\_TIMESTAMP or MFX\_FRCALGM\_DISTRIBUTED\_TIMESTAMP flags.

#### 12.7.3.48 ImageStabMode

The ImageStabMode enumerator itemizes image stabilization modes. See description of mfxExtVPPIImageStab structure for more details.

##### **enumerator MFX\_IMAGESTAB\_MODE\_UPSCALE = 0x0001**

Upscale mode.

##### **enumerator MFX\_IMAGESTAB\_MODE\_BOXING = 0x0002**

Boxing mode.

### 12.7.3.49 InsertHDRPayload

The InsertHDRPayload enumerator itemizes HDR payloads insertion rules.

**enumerator MFX\_PAYLOAD\_OFF = 0**  
Don't insert payload.

**enumerator MFX\_PAYLOAD\_IDR = 1**  
Insert payload on IDR frames.

### 12.7.3.50 LongTermIdx

The LongTermIdx specifies long term index of picture control

**enumerator MFX\_LONGTERM\_IDX\_NO\_IDX = 0xFFFF**  
Long term index of picture is undefined.

### 12.7.3.51 TransferMatrix

The TransferMatrix enumerator itemizes color transfer matrices.

**enumerator MFX\_TRANSFERMATRIX\_UNKNOWN = 0**  
Transfer matrix isn't specified

**enumerator MFX\_TRANSFERMATRIX\_BT709 = 1**  
Transfer matrix from ITU-R BT.709 standard.

**enumerator MFX\_TRANSFERMATRIX\_BT601 = 2**  
Transfer matrix from ITU-R BT.601 standard.

### 12.7.3.52 NominalRange

The NominalRange enumerator itemizes pixel's value nominal range.

**enumerator MFX\_NOMINALRANGE\_UNKNOWN = 0**  
Range isn't defined.

**enumerator MFX\_NOMINALRANGE\_0\_255 = 1**  
Range is [0,255].

**enumerator MFX\_NOMINALRANGE\_16\_235 = 2**  
Range is [16,235].

### 12.7.3.53 ROImode

The ROImode enumerator itemizes QP adjustment mode for ROIs.

**enumerator MFX\_ROI\_MODE\_PRIORITY = 0**  
Priority mode.

**enumerator MFX\_ROI\_MODE\_QP\_DELTA = 1**  
QP mode

**enumerator MFX\_ROI\_MODE\_QP\_VALUE = 2**  
Absolute QP

### 12.7.3.54 DeinterlacingMode

The DeinterlacingMode enumerator itemizes VPP deinterlacing modes.

```
enumerator MFX_DEINTERLACING_BOB = 1
    BOB deinterlacing mode.

enumerator MFX_DEINTERLACING_ADVANCED = 2
    Advanced deinterlacing mode.

enumerator MFX_DEINTERLACING_AUTO_DOUBLE = 3
    Auto mode with deinterlacing double frame rate output.

enumerator MFX_DEINTERLACING_AUTO_SINGLE = 4
    Auto mode with deinterlacing single frame rate output.

enumerator MFX_DEINTERLACING_FULL_FR_OUT = 5
    Deinterlace only mode with full frame rate output.

enumerator MFX_DEINTERLACING_HALF_FR_OUT = 6
    Deinterlace only Mode with half frame rate output.

enumerator MFX_DEINTERLACING_24FPS_OUT = 7
    24 fps fixed output mode.

enumerator MFX_DEINTERLACING_FIXED_TELECINE_PATTERN = 8
    Fixed telecine pattern removal mode.

enumerator MFX_DEINTERLACING_30FPS_OUT = 9
    30 fps fixed output mode.

enumerator MFX_DEINTERLACING_DETECT_INTERLACE = 10
    Only interlace detection.

enumerator MFX_DEINTERLACING_ADVANCED_NOREF = 11
    Advanced deinterlacing mode without using of reference frames.

enumerator MFX_DEINTERLACING_ADVANCED_SCD = 12
    Advanced deinterlacing mode with scene change detection.

enumerator MFX_DEINTERLACING_FIELD_WEAVERSING = 13
    Field weaving.
```

### 12.7.3.55 TelecinePattern

The TelecinePattern enumerator itemizes telecine patterns.

```
enumerator MFX_TELECINE_PATTERN_32 = 0
    3:2 telecine.

enumerator MFX_TELECINE_PATTERN_2332 = 1
    2:3:3:2 telecine.

enumerator MFX_TELECINE_PATTERN_FRAME_REPEAT = 2
    One frame repeat telecine.

enumerator MFX_TELECINE_PATTERN_41 = 3
    4:1 telecine.

enumerator MFX_TELECINE_POSITION_PROVIDED = 4
    User must provide position inside a sequence of 5 frames where the artifacts start.
```

### 12.7.3.56 VPPFieldProcessingMode

The VPPFieldProcessingMode enumerator is used to control VPP field processing algorithm.

**enumerator MFX\_VPP\_COPY\_FRAME** = 0x01

Copy the whole frame.

**enumerator MFX\_VPP\_COPY\_FIELD** = 0x02

Copy only one field.

**enumerator MFX\_VPP\_SWAP\_FIELDS** = 0x03

Swap top and bottom fields.

### 12.7.3.57 PicType

The PicType enumerator itemizes picture type.

**enumerator MFX\_PICTYPE\_UNKNOWN** = 0x00

Picture type is unknown.

**enumerator MFX\_PICTYPE\_FRAME** = 0x01

Picture is a frame.

**enumerator MFX\_PICTYPE\_TOPFIELD** = 0x02

Picture is a top field.

**enumerator MFX\_PICTYPE\_BOTTOMFIELD** = 0x04

Picture is a bottom field.

### 12.7.3.58 MBQPMode

The MBQPMode enumerator itemizes QP update modes.

**enumerator MFX\_MBQP\_MODE\_QP\_VALUE** = 0

QP array contains QP values.

**enumerator MFX\_MBQP\_MODE\_QP\_DELTA** = 1

QP array contains deltas for QP.

**enumerator MFX\_MBQP\_MODE\_QP\_ADAPTIVE** = 2

QP array contains deltas for QP or absolute QP values.

### 12.7.3.59 GeneralConstraintFlags

The GeneralConstraintFlags enumerator uses bit-ORed values to itemize HEVC bitstream indications for specific profiles. Each value indicates for format range extensions profiles.

**enumerator MFX\_HEVC\_CONSTR\_REXT\_MAX\_12BIT** = (1 << 0)

**enumerator MFX\_HEVC\_CONSTR\_REXT\_MAX\_10BIT** = (1 << 1)

**enumerator MFX\_HEVC\_CONSTR\_REXT\_MAX\_8BIT** = (1 << 2)

**enumerator MFX\_HEVC\_CONSTR\_REXT\_MAX\_422CHROMA** = (1 << 3)

**enumerator MFX\_HEVC\_CONSTR\_REXT\_MAX\_420CHROMA** = (1 << 4)

**enumerator MFX\_HEVC\_CONSTR\_REXT\_MAX\_MONOCHROME** = (1 << 5)

**enumerator MFX\_HEVC\_CONSTR\_REXT\_INTRA** = (1 << 6)

---

```
enumerator MFX_HEVC_CONSTR_REXT_ONE_PICTURE_ONLY = (1 << 7)
enumerator MFX_HEVC_CONSTR_REXT_LOWER_BIT_RATE = (1 << 8)
```

### 12.7.3.60 SampleAdaptiveOffset

The SampleAdaptiveOffset enumerator uses bit-ORed values to itemize corresponding HEVC encoding feature.

```
enumerator MFX_SAO_UNKNOWN = 0x00
    Use default value for platform/TargetUsage.
```

```
enumerator MFX_SAO_DISABLE = 0x01
```

Disable SAO. If set during Init leads to SPS sample\_adaptive\_offset\_enabled\_flag = 0. If set during Runtime, leads to slice\_sao\_luma\_flag = 0 and slice\_sao\_chroma\_flag = 0 for current frame.

```
enumerator MFX_SAO_ENABLE_LUMA = 0x02
    Enable SAO for luma (slice_sao_luma_flag = 1).
```

```
enumerator MFX_SAO_ENABLE_CHROMA = 0x04
    Enable SAO for chroma (slice_sao_chroma_flag = 1).
```

### 12.7.3.61 ErrorTypes

The ErrorTypes enumerator uses bit-ORed values to itemize bitstream error types.

```
enumerator MFX_ERROR_PPS = (1 << 0)
    Invalid/corrupted PPS.
```

```
enumerator MFX_ERROR_SPS = (1 << 1)
    Invalid/corrupted SPS.
```

```
enumerator MFX_ERROR_SLICEHEADER = (1 << 2)
    Invalid/corrupted slice header.
```

```
enumerator MFX_ERROR_SLICEDATA = (1 << 3)
    Invalid/corrupted slice data.
```

```
enumerator MFX_ERROR_FRAME_GAP = (1 << 4)
    Missed frames.
```

### 12.7.3.62 HEVCRegionType

The HEVCRegionType enumerator itemizes type of HEVC region.

```
enumerator MFX_HEVC_REGION_SLICE = 0
    Slice type.
```

### 12.7.3.63 HEVCRegionEncoding

The HEVCRegionEncoding enumerator itemizes HEVC region's encoding.

```
enumerator MFX_HEVC_REGION_ENCODING_ON = 0
enumerator MFX_HEVC_REGION_ENCODING_OFF = 1
```

### 12.7.3.64 Angle

The Angle enumerator itemizes valid rotation angles.

```
enumerator MFX_ANGLE_0 = 0
    0 degrees.

enumerator MFX_ANGLE_90 = 90
    90 degrees.

enumerator MFX_ANGLE_180 = 180
    180 degrees.

enumerator MFX_ANGLE_270 = 270
    270 degrees.
```

### 12.7.3.65 ScalingMode

The ScalingMode enumerator itemizes variants of scaling filter implementation.

```
enumerator MFX_SCALING_MODE_DEFAULT = 0
    Default scaling mode. SDK selects the most appropriate scaling method.

enumerator MFX_SCALING_MODE_LOWPOWER = 1
    Low power scaling mode which is applicable for platform SDK implementations. The exact scaling algorithm
    is defined by the SDK.

enumerator MFX_SCALING_MODE_QUALITY = 2
    The best quality scaling mode
```

### 12.7.3.66 InterpolationMode

The InterpolationMode enumerator specifies type of interpolation method used by VPP scaling filter.

```
enumerator MFX_INTERPOLATION_DEFAULT = 0
    Default interpolation mode for scaling. SDK selects the most appropriate scaling method.

enumerator MFX_INTERPOLATION_NEAREST_NEIGHBOR = 1
    Nearest neighbor interpolation method

enumerator MFX_INTERPOLATION_BILINEAR = 2
    Bilinear interpolation method

enumerator MFX_INTERPOLATION_ADVANCED = 3
    Advanced interpolation method is defined by each SDK and usually gives best quality
```

### 12.7.3.67 MirroringType

The MirroringType enumerator itemizes mirroring types.

```
enumerator MFX_MIRRORING_DISABLED = 0
enumerator MFX_MIRRORING_HORIZONTAL = 1
enumerator MFX_MIRRORING_VERTICAL = 2
```

### 12.7.3.68 ChromaSiting

The ChromaSiting enumerator defines chroma location. Use bit-OR'ed values to specify the desired location.

```
enumerator MFX_CHROMA_SITING_UNKNOWN = 0x0000
    Unspecified.

enumerator MFX_CHROMA_SITING_VERTICAL_TOP = 0x0001
    Chroma samples are co-sited vertically on the top with the luma samples.

enumerator MFX_CHROMA_SITING_VERTICAL_CENTER = 0x0002
    Chroma samples are not co-sited vertically with the luma samples.

enumerator MFX_CHROMA_SITING_VERTICAL_BOTTOM = 0x0004
    Chroma samples are co-sited vertically on the bottom with the luma samples.

enumerator MFX_CHROMA_SITING_HORIZONTAL_LEFT = 0x0010
    Chroma samples are co-sited horizontally on the left with the luma samples.

enumerator MFX_CHROMA_SITING_HORIZONTAL_CENTER = 0x0020
    Chroma samples are not co-sited horizontally with the luma samples.
```

### 12.7.3.69 VP9ReferenceFrame

The VP9ReferenceFrame enumerator itemizes reference frame type by the mfxVP9SegmentParam::ReferenceFrame parameter.

```
enumerator MFX_VP9_REF_INTRA = 0
    Intra.

enumerator MFX_VP9_REF_LAST = 1
    Last.

enumerator MFX_VP9_REF_GOLDEN = 2
    Golden.

enumerator MFX_VP9_REF_ALTREF = 3
    Alternative reference.
```

### 12.7.3.70 SegmentIdBlockSize

The SegmentIdBlockSize enumerator indicates the block size represented by each segment\_id in segmentation map. These values are used with the mfxExtVP9Segmentation::SegmentIdBlockSize parameter.

```
enumerator MFX_VP9_SEGMENT_ID_BLOCK_SIZE_UNKNOWN = 0
    Unspecified block size.

enumerator MFX_VP9_SEGMENT_ID_BLOCK_SIZE_8x8 = 8
    8x8 block size.

enumerator MFX_VP9_SEGMENT_ID_BLOCK_SIZE_16x16 = 16
    16x16 block size.

enumerator MFX_VP9_SEGMENT_ID_BLOCK_SIZE_32x32 = 32
    32x32 block size.

enumerator MFX_VP9_SEGMENT_ID_BLOCK_SIZE_64x64 = 64
    64x64 block size.
```

### 12.7.3.71 SegmentFeature

The SegmentFeature enumerator indicates features enabled for the segment. These values are used with the mfxVP9SegmentParam::FeatureEnabled parameter.

```
enumerator MFX_VP9_SEGMENT_FEATURE_QINDEX = 0x0001
    Quantization index delta.

enumerator MFX_VP9_SEGMENT_FEATURE_LOOP_FILTER = 0x0002
    Loop filter level delta.

enumerator MFX_VP9_SEGMENT_FEATURE_REFERENCE = 0x0004
    Reference frame.

enumerator MFX_VP9_SEGMENT_FEATURE_SKIP = 0x0008
    Skip.
```

### 12.7.3.72 mfxComponentType

#### enum mfxComponentType

The mfxComponentType enumerator describes type of workload passed to MFXQueryAdapters.

*Values:*

```
enumerator MFX_COMPONENT_ENCODE = 1
    Encode workload.

enumerator MFX_COMPONENT_DECODE = 2
    Decode workload.

enumerator MFX_COMPONENT_VPP = 3
    VPP workload.
```

### 12.7.3.73 PartialBitstreamOutput

The PartialBitstreamOutput enumerator indicates flags of partial bitstream output type.

<b>enumerator MFX_PARTIAL_BITSTREAM_NONE = 0</b>	Don't use partial output
<b>enumerator MFX_PARTIAL_BITSTREAM_SLICE = 1</b>	Partial bitstream output will be aligned to slice granularity
<b>enumerator MFX_PARTIAL_BITSTREAM_BLOCK = 2</b>	Partial bitstream output will be aligned to user-defined block size granularity
<b>enumerator MFX_PARTIAL_BITSTREAM_ANY = 3</b>	Partial bitstream output will be return any coded data available at the end of SyncOperation timeout

### 12.7.3.74 BRCStatus

The BRCStatus enumerator itemizes instructions to the SDK encoder by `mfxExtBrc::Update`.

<b>enumerator MFX_BRC_OK = 0</b>	CodedFrameSize is acceptable, no further recoding/padding/skip required, proceed to next frame.
<b>enumerator MFX_BRC_BIG_FRAME = 1</b>	Coded frame is too big, recoding required.
<b>enumerator MFX_BRC_SMALL_FRAME = 2</b>	Coded frame is too small, recoding required.
<b>enumerator MFX_BRC_PANIC_BIG_FRAME = 3</b>	Coded frame is too big, no further recoding possible - skip frame.
<b>enumerator MFX_BRC_PANIC_SMALL_FRAME = 4</b>	Coded frame is too small, no further recoding possible - required padding to <code>mfxBRCFrameStatus::MinFrameSize</code> .

### 12.7.3.75 Rotation

The Rotation enumerator itemizes the JPEG rotation options.

<b>enumerator MFX_ROTATION_0 = 0</b>	No rotation.
<b>enumerator MFX_ROTATION_90 = 1</b>	90 degree rotation.
<b>enumerator MFX_ROTATION_180 = 2</b>	180 degree rotation.
<b>enumerator MFX_ROTATION_270 = 3</b>	270 degree rotation.

### 12.7.3.76 JPEGColorFormat

The JPEGColorFormat enumerator itemizes the JPEG color format options.

**enumerator MFX\_JPEG\_COLORFORMAT\_UNKNOWN = 0**

**enumerator MFX\_JPEG\_COLORFORMAT\_YCbCr = 1**

Unknown color format. The SDK decoder tries to determine color format from available in bitstream information. If such information is not present, then MFX\_JPEG\_COLORFORMAT\_YCbCr color format is assumed.

**enumerator MFX\_JPEG\_COLORFORMAT\_RGB = 2**

Bitstream contains Y, Cb and Cr components.

### 12.7.3.77 JPEGScanType

The JPEGScanType enumerator itemizes the JPEG scan types.

**enumerator MFX\_SCANTYPE\_UNKNOWN = 0**

Unknown scan type.

**enumerator MFX\_SCANTYPE\_INTERLEAVED = 1**

Interleaved scan.

**enumerator MFX\_SCANTYPE\_NONINTERLEAVED = 2**

Non-interleaved scan.

### 12.7.3.78 Protected

The Protected enumerator describes the protection schemes.

**enumerator MFX\_PROTECTION\_CENC\_WV\_CLASSIC = 0x0004**

The protection scheme is based on the Widevine\* DRM from Google\*.

**enumerator MFX\_PROTECTION\_CENC\_WV\_GOOGLE\_DASH = 0x0005**

The protection scheme is based on the Widevine\* Modular DRM\* from Google\*.

## 12.7.4 Structs

### 12.7.4.1 mfxRange32U

**struct mfxRange32U**

This structure represents a range of unsigned values

#### Public Members

*mfxU32 Min*

Minimal value of the range.

*mfxU32 Max*

Maximal value of the range.

*mfxU32 Step*

Value incrementation step.

#### 12.7.4.2 mfxI16Pair

**struct mfxI16Pair**

Thus structure represents a pair of numbers of type mfxI16.

##### Public Members

*mfxI16 x*

First number.

*mfxI16 y*

Second number.

#### 12.7.4.3 mfxHDLPair

**struct mfxHDLPair**

Thus structure represents pair of handles of type mfxHDL.

##### Public Members

*mfxHDL first*

First handle.

*mfxHDL second*

Second handle.

#### 12.7.4.4 mfxVersion

**union mfxVersion**

#include <mfxcommon.h> The *mfxVersion* union describes the version of the SDK implementation.

##### Public Members

*mfxU16 Minor*

Minor number of the SDK implementation.

*mfxU16 Major*

Major number of the SDK implementation.

**struct mfxVersion::[anonymous] [anonymous]**

*mfxU32 Version*

SDK implementation version number.

#### 12.7.4.5 mfxStructVersion

##### **union mfxStructVersion**

#include <mfxdefs.h> Introduce field Version for any structures. Minor number is incremented when reserved fields are used, major number is incremented when size of structure is increased. Assumed that any structure changes are backward binary compatible. *mfxStructVersion* starts from {1,0} for any new API structures, if *mfxStructVersion* is added to the existent legacy structure (replacing reserved fields) it starts from {1, 1}.

##### **Public Members**

###### *mfxU8 Minor*

Minor number of the correspondent structure.

###### *mfxU8 Major*

Major number of the correspondent structure.

###### **struct mfxStructVersion::[anonymous] [anonymous]**

###### *mfxU16 Version*

Structure version number

#### 12.7.4.6 mfxPlatform

##### **struct mfxPlatform**

The *mfxPlatform* structure contains information about hardware platform.

##### **Public Members**

###### *mfxU16 CodeName*

Microarchitecture code name. See the PlatformCodeName enumerator for a list of possible values.

###### *mfxU16 DeviceId*

Unique identifier of graphics device.

###### *mfxU16 MediaAdapterType*

Description of graphics adapter type. See the mfxMediaAdapterType enumerator for a list of possible values.

###### *mfxU16 reserved[13]*

Reserved for future use.

#### 12.7.4.7 mfxInitParam

##### **struct mfxInitParam**

This structure specifies advanced initialization parameters. A zero value in any of the fields indicates that the corresponding field is not explicitly specified.

## Public Members

### *mfxIMPL Implementation*

*mfxIMPL* enumerator that indicates the desired SDK implementation.

### *mfxVersion Version*

Structure which specifies minimum library version or zero, if not specified.

### *mfxU16 ExternalThreads*

Desired threading mode. Value 0 means internal threading, 1 – external.

### *mfxExtBuffer \*\*ExtParam*

Points to an array of pointers to the extra configuration structures; see the ExtendedBufferID enumerator for a list of extended configurations.

### *mfxU16 NumExtParam*

The number of extra configuration structures attached to this structure.

### **union *mfxInitParam*::[anonymous] [anonymous]**

### *mfxU16 GPUCopy*

Enables or disables GPU accelerated copying between video and system memory in the SDK components.

See the GPUCopy enumerator for a list of valid values.

## 12.7.4.8 *mfxInfoMFx*

### **struct *mfxInfoMFx***

The *mfxInfoMFx* structure specifies configurations for decoding, encoding and transcoding processes. A zero value in any of these fields indicates that the field is not explicitly specified.

## Public Members

### *mfxU32 reserved[7]*

Reserved for future use.

### *mfxU16 LowPower*

For encoders set this flag to ON to reduce power consumption and GPU usage. See the CodingOptionValue enumerator for values of this option. Use Query function to check if this feature is supported.

### *mfxU16 BRCParamMultiplier*

Specifies a multiplier for bitrate control parameters. Affects next four variables InitialDelayInKB, BufferSizeInKB, TargetKbps, MaxKbps. If this value is not equal to zero encoder calculates BRC parameters as value \* BRCParamMultiplier.

### *mfxFrameInfo FrameInfo*

*mfxFrameInfo* structure that specifies frame parameters

### *mfxU32 CodecId*

Specifies the codec format identifier in the FourCC code; see the CodecFormatFourCC enumerator for details. This is a mandated input parameter for QueryIOSurf and Init functions.

### *mfxU16 CodecProfile*

Specifies the codec profile; see the CodecProfile enumerator for details. Specify the codec profile explicitly or the SDK functions will determine the correct profile from other sources, such as resolution and bitrate.

### *mfxU16 CodecLevel*

Codec level; see the CodecLevel enumerator for details. Specify the codec level explicitly or the SDK functions will determine the correct level from other sources, such as resolution and bitrate.

**mfxU16 TargetUsage**

Target usage model that guides the encoding process; see the TargetUsage enumerator for details.

**mfxU16 GopPicSize**

Number of pictures within the current GOP (Group of Pictures); if GopPicSize = 0, then the GOP size is unspecified. If GopPicSize = 1, only I-frames are used. Pseudo-code that demonstrates how SDK uses this parameter.

```
mfxU16 get_gop_sequence (...) {
    pos=display_frame_order;
    if (pos == 0)
        return MFX_FRAMETYPE_I | MFX_FRAMETYPE_IDR | MFX_FRAMETYPE_REF;

    If (GopPicSize == 1) // Only I-frames
        return MFX_FRAMETYPE_I | MFX_FRAMETYPE_REF;

    if (GopPicSize == 0)
        frameInGOP = pos;      //Unlimited GOP
    else
        frameInGOP = pos%GopPicSize;

    if (frameInGOP == 0)
        return MFX_FRAMETYPE_I | MFX_FRAMETYPE_REF;

    if (GopRefDist == 1 || GopRefDist == 0)      // Only I,P frames
        return MFX_FRAMETYPE_P | MFX_FRAMETYPE_REF;

    frameInPattern = (frameInGOP-1)%GopRefDist;
    if (frameInPattern == GopRefDist - 1)
        return MFX_FRAMETYPE_P | MFX_FRAMETYPE_REF;

    return MFX_FRAMETYPE_B;
}
```

**mfxU16 GopRefDist**

Distance between I- or P (or GPB) - key frames; if it is zero, the GOP structure is unspecified. Note: If GopRefDist = 1, there are no regular B-frames used (only P or GPB); if *mfxExtCodingOption3::GPB* is ON, GPB frames (B without backward references) are used instead of P.

**mfxU16 GopOptFlag**

ORs of the GopOptFlag enumerator indicate the additional flags for the GOP specification.

**mfxU16 IdrInterval**

For H.264, IdrInterval specifies IDR-frame interval in terms of I-frames; if IdrInterval = 0, then every I-frame is an IDR-frame. If IdrInterval = 1, then every other I-frame is an IDR-frame, etc.

For HEVC, if IdrInterval = 0, then only first I-frame is an IDR-frame. If IdrInterval = 1, then every I-frame is an IDR-frame. If IdrInterval = 2, then every other I-frame is an IDR-frame, etc.

For MPEG2, IdrInterval defines sequence header interval in terms of I-frames. If IdrInterval = N, SDK inserts the sequence header before every Nth I-frame. If IdrInterval = 0 (default), SDK inserts the sequence header once at the beginning of the stream.

If GopPicSize or GopRefDist is zero, IdrInterval is undefined.

**mfxU16 InitialDelayInKB**

Initial size of the Video Buffering Verifier (VBV) buffer.

**Note** In this context, KB is 1000 bytes and Kbps is 1000 bps.

***mfxU16 QPI***

Quantization Parameter (QP) for I frames for constant QP mode (CQP). Zero QP is not valid and means that default value is assigned by the library. Non-zero QPI might be clipped to supported QPI range.

**Note** Default QPI value is implementation dependent and subject to change without additional notice in this document.

***mfxU16 Accuracy***

Specifies accuracy range in the unit of tenth of percent.

***mfxU16 BufferSizeInKB***

BufferSizeInKB represents the maximum possible size of any compressed frames.

***mfxU16 TargetKbps***

Constant bitrate TargetKbps. Used to estimate the targeted frame size by dividing the frame rate by the bitrate.

***mfxU16 QPP***

Quantization Parameter (QP) for P frames for constant QP mode (CQP). Zero QP is not valid and means that default value is assigned by the library. Non-zero QPP might be clipped to supported QPI range.

**Note** Default QPP value is implementation dependent and subject to change without additional notice in this document.

***mfxU16 ICQQuality***

This parameter is for Intelligent Constant Quality (ICQ) bitrate control algorithm. It is value in the 1...51 range, where 1 corresponds the best quality.

***mfxU16 MaxKbps***

the maximum bitrate at which the encoded data enters the Video Buffering Verifier (VBV) buffer.

***mfxU16 QPB***

Quantization Parameter (QP) for B frames for constant QP mode (CQP). Zero QP is not valid and means that default value is assigned by the library. Non-zero QPI might be clipped to supported QPB range.

**Note** Default QPB value is implementation dependent and subject to change without additional notice in this document.

***mfxU16 Convergence***

Convergence period in the unit of 100 frames.

***mfxU16 NumSlice***

Number of slices in each video frame; each slice contains one or more macro-block rows. If NumSlice equals zero, the encoder may choose any slice partitioning allowed by the codec standard. See also [\*mfxExtCodingOption2::NumMbPerSlice\*](#).

***mfxU16 NumRefFrame***

Max number of all available reference frames (for AVC/HEVC NumRefFrame defines DPB size); if NumRefFrame = 0, this parameter is not specified. See also [\*mfxExtCodingOption3::NumRefActiveP\*](#), NumRefActiveBL0 and NumRefActiveBL1 which set a number of active references.

***mfxU16 EncodedOrder***

If not zero, EncodedOrder specifies that ENCODE takes the input surfaces in the encoded order and uses explicit frame type control. Application still must provide GopRefDist and [\*mfxExtCodingOption2::BRefType\*](#) so SDK can pack headers and build reference lists correctly.

***mfxU16 DecodedOrder***

For AVC and HEVC, used to instruct the decoder to return output frames in the decoded order. Must be zero for all other decoders. When enabled, correctness of [\*mfxFrameData::TimeStamp\*](#) and FrameOrder for output surface is not guaranteed, the application should ignore them.

***mfxU16 ExtendedPicStruct***

Instructs DECODE to output extended picture structure values for additional display attributes. See the PicStruct description for details.

***mfxU16 TimeStampCalc***

Time stamp calculation method; see the TimeStampCalc description for details.

***mfxU16 SliceGroupsPresent***

Nonzero value indicates that slice groups are present in the bitstream. Only AVC decoder uses this field.

***mfxU16 MaxDecFrameBuffering***

Nonzero value specifies the maximum required size of the decoded picture buffer in frames for AVC and HEVC decoders.

***mfxU16 EnableReallocRequest***

For decoders supporting dynamic resolution change (VP9), set this option to ON to allow MFXVideoDECODE\_DecodeFrameAsync return MFX\_ERR\_REALLOC\_SURFACE. See the CodingOptionValue enumerator for values of this option. Use Query function to check if this feature is supported.

***mfxU16 JPEGChromaFormat***

Specify the chroma sampling format that has been used to encode JPEG picture. See the ChromaFormat enumerator.

***mfxU16 Rotation***

Rotation option of the output JPEG picture; see the Rotation enumerator for details.

***mfxU16 JPEGColorFormat***

Specify the color format that has been used to encode JPEG picture. See the JPEGColorFormat enumerator for details.

***mfxU16 InterleavedDec***

Specify JPEG scan type for decoder. See the JPEGScanType enumerator for details.

***mfxU8 SamplingFactorH[4]***

Horizontal sampling factor.

***mfxU8 SamplingFactorV[4]***

Vertical sampling factor.

***mfxU16 Interleaved***

Non-interleaved or interleaved scans. If it is equal to MFX\_SCANTYPE\_INTERLEAVED then the image is encoded as interleaved, all components are encoded in one scan. See the JPEG Scan Type enumerator for details.

***mfxU16 Quality***

Specifies the image quality if the application does not specified quantization table. This is the value from 1 to 100 inclusive. “100” is the best quality.

***mfxU16 RestartInterval***

Specifies the number of MCU in the restart interval. “0” means no restart interval.

---

**Note:** The *mfxInfoMFx::InitialDelayInKB*, *mfxInfoMFx::TargetKbps*, *mfxInfoMFx::MaxKbps* parameters are for the constant bitrate (CBR), variable bitrate control (VBR), and CQP HRD algorithms.

The SDK encoders follow the Hypothetical Reference Decoding (HRD) model. The HRD model assumes that data flows into a buffer of the fixed size BufferSizeInKB with a constant bitrate TargetKbps. (Estimate the targeted frame size by dividing the frame rate by the bitrate.)

The decoder starts decoding after the buffer reaches the initial size InitialDelayInKB, which is equivalent to reaching an initial delay of  $\text{InitialDelayInKB} \times 8000 / \text{TargetKbps}$ . In this context, KB is 1000 bytes and Kbps is 1000 bps.

If InitialDelayInKB or BufferSizeInKB is equal to zero, the value is calculated using bitrate, frame rate, profile, level, and so on.

TargetKbps must be specified for encoding initialization.

For variable bitrate control, the MaxKbps parameter specifies the maximum bitrate at which the encoded data enters the Video Buffering Verifier (VBV) buffer. If MaxKbps is equal to zero, the value is calculated from bitrate, frame rate, profile, and level.

**Note:** The `mfxInfoMFx::TargetKbps`, `mfxInfoMFx::Accuracy`, `mfxInfoMFx::Convergence` parameters are for the average variable bitrate control (AVBR) algorithm. The algorithm focuses on overall encoding quality while meeting the specified bitrate, TargetKbps, within the accuracy range, Accuracy, after a Convergence period. This method does not follow HRD and the instant bitrate is not capped or padded.

#### 12.7.4.9 mfxFrameId

##### **struct mfxFrameId**

The `mfxFrameId` structure describes the view and layer of a frame picture.

##### **Public Members**

###### `mfxU16 TemporalId`

The temporal identifier as defined in the annex H of the ITU\*-T H.264 specification.

###### `mfxU16 PriorityId`

Reserved and must be zero.

###### `mfxU16 DependencyId`

Reserved for future use.

###### `mfxU16 QualityId`

Reserved for future use.

###### `mfxU16 ViewId`

The view identifier as defined in the annex H of the ITU-T H.264 specification.

#### 12.7.4.10 mfxFrameInfo

##### **struct mfxFrameInfo**

The `mfxFrameInfo` structure specifies properties of video frames. See also “Configuration Parameter Constraints” chapter.

## FrameRate

Specify the frame rate by the formula: FrameRateExtN / FrameRateExtD.

For encoding, frame rate must be specified. For decoding, frame rate may be unspecified (FrameRateExtN and FrameRateExtD are all zeros.) In this case, the frame rate is default to 30 frames per second.

*mfxU32 FrameRateExtN*

Numerator.

*mfxU32 FrameRateExtD*

Denominator.

## AspectRatio

These parameters specify the sample aspect ratio. If sample aspect ratio is explicitly defined by the standards (see Table 6-3 in the MPEG-2 specification or Table E-1 in the H.264 specification), AspectRatioW and AspectRatioH should be the defined values. Otherwise, the sample aspect ratio can be derived as follows:

AspectRatioW=display\_aspect\_ratio\_width\*display\_height;

AspectRatioH=display\_aspect\_ratio\_height\*display\_width;

For MPEG-2, the above display aspect ratio must be one of the defined values in Table 6-3. For H.264, there is no restriction on display aspect ratio values.

If both parameters are zero, the encoder uses default value of sample aspect ratio.

*mfxU16 AspectRatioW*

Ratio for width.

*mfxU16 AspectRatioH*

Ratio for height.

## ROI

Display the region of interest of the frame; specify the display width and height in *mfxVideoParam*.

*mfxU16 CropX*

X coordinate.

*mfxU16 CropY*

Y coordinate.

*mfxU16 CropW*

Width in pixels.

*mfxU16 CropH*

Height in pixels.

## Public Members

*mfxU32 reserved[4]*

Reserved for future use.

*mfxU16 reserved4*

Reserved for future use.

*mfxU16 BitDepthLuma*

Number of bits used to represent luma samples.

**Note** Not all codecs and SDK implementations support this value. Use Query function to check if this feature is supported.

*mfxU16 BitDepthChroma*

Number of bits used to represent chroma samples.

**Note** Not all codecs and SDK implementations support this value. Use Query function to check if this feature is supported.

*mfxU16 Shift*

When not zero indicates that values of luma and chroma samples are shifted. Use BitDepthLuma and BitDepthChroma to calculate shift size. Use zero value to indicate absence of shift.

**Note** Not all codecs and SDK implementations support this value. Use Query function to check if this feature is supported.

*mfxFrameId FrameId*

Frame ID. Describes the view and layer of a frame picture.

*mfxU32 FourCC*

FourCC code of the color format; see the ColorFourCC enumerator for details.

*mfxU16 Width*

Width of the video frame in pixels. Must be a multiple of 16.

*mfxU16 Height*

Height of the video frame in pixels. Must be a multiple of 16 for progressive frame sequence and a multiple of 32 otherwise.

*mfxU64 BufferSize*

Size of frame buffer in bytes. Valid only for plain formats (when FourCC is P8); Width, Height and crops in this case are invalid.

*mfxU16 PicStruct*

Picture type as specified in the PicStruct enumerator.

*mfxU16 ChromaFormat*

Color sampling method; the value of ChromaFormat is the same as that of ChromaFormatIdc. ChromaFormat is not defined if FourCC is zero.

---

**Note:** Data alignment for Shift = 0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Value	0	0	0	0	0	0	Valid data										

Data alignment for Shift != 0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	Valid data											0	0	0	0	0

#### 12.7.4.11 mfxVideoParam

##### struct mfxVideoParam

The *mfxVideoParam* structure contains configuration parameters for encoding, decoding, transcoding and video processing.

##### Public Members

###### *mfxU32 AllocId*

Unique component ID that will be passed by SDK to *mfxFrameAllocRequest*. Useful in pipelines where several components of the same type share the same allocator.

###### *mfxU16 AsyncDepth*

Specifies how many asynchronous operations an application performs before the application explicitly synchronizes the result. If zero, the value is not specified.

###### *mfxInfoMFX mfx*

Configurations related to encoding, decoding and transcoding; see the definition of the *mfxInfoMFX* structure for details.

###### *mfxInfoVPP vpp*

Configurations related to video processing; see the definition of the *mfxInfoVPP* structure for details.

###### *mfxU16 Protected*

Specifies the content protection mechanism; see the Protected enumerator for a list of supported protection schemes.

###### *mfxU16 IOPattern*

Input and output memory access types for SDK functions; see the enumerator IOPattern for details. The

Query functions return the natively supported IOPattern if the Query input argument is NULL. This parameter is a mandated input for QueryIOSurf and Init functions. For DECODE, the output pattern must be specified; for ENCODE, the input pattern must be specified; and for VPP, both input and output pattern must be specified.

*mfxExtBuffer* \*\***ExtParam**

The number of extra configuration structures attached to this structure.

*mfxU16* **NumExtParam**

Points to an array of pointers to the extra configuration structures; see the ExtendedBufferID enumerator for a list of extended configurations. The list of extended buffers should not contain duplicated entries, i.e. entries of the same type. If *mfxVideoParam* structure is used to query the SDK capability, then list of extended buffers attached to input and output *mfxVideoParam* structure should be equal, i.e. should contain the same number of extended buffers of the same type.

#### 12.7.4.12 mfxFrameData

**struct mfxY410**

The *mfxY410* structure specifies “pixel” in Y410 color format

##### Public Members

*mfxU32* **U**

U component.

*mfxU32* **Y**

Y component.

*mfxU32* **V**

V component.

*mfxU32* **A**

A component.

**struct mfxA2RGB10**

The *mfxA2RGB10* structure specifies “pixel” in A2RGB10 color format

##### Public Members

*mfxU32* **B**

B component.

*mfxU32* **G**

G component.

*mfxU32* **R**

R component.

*mfxU32* **A**

A component.

**struct mfxFrameData**

The *mfxFrameData* structure describes frame buffer pointers.

## Extension Buffers

### *mfxU16 NumExtParam*

The number of extra configuration structures attached to this structure.

## General members

### *mfxU16 reserved[9]*

Reserved for future use

### *mfxU16 MemType*

Allocated memory type; see the ExtMemFrameType enumerator for details. Used for better integration of 3rd party plugins into SDK pipeline.

### *mfxU16 PitchHigh*

Distance in bytes between the start of two consecutive rows in a frame.

### *mfxU64 TimeStamp*

Time stamp of the video frame in units of 90KHz (divide TimeStamp by 90,000 (90 KHz) to obtain the time in seconds). A value of MFX\_TIMESTAMP\_UNKNOWN indicates that there is no time stamp.

### *mfxU32 FrameOrder*

Current frame counter for the top field of the current frame; an invalid value of MFX\_FRAMEORDER\_UNKNOWN indicates that SDK functions that generate the frame output do not use this frame.

### *mfxU16 Locked*

Counter flag for the application; if Locked is greater than zero then the application locks the frame or field pair. Do not move, alter or delete the frame.

## Color Planes

Data pointers to corresponding color channels (planes). The frame buffer pointers must be 16-byte aligned. The application has to specify pointers to all color channels even for packed formats. For example, for YUY2 format the application has to specify Y, U and V pointers. For RGB32 – R, G, B and A pointers.

### *mfxU8 \*A*

A channel.

### *mfxMemId MemId*

Memory ID of the data buffers; if any of the preceding data pointers is non-zero then the SDK ignores MemId.

## Additional Flags

### *mfxU16 Corrupted*

Some part of the frame or field pair is corrupted. See the Corruption enumerator for details.

### *mfxU16 DataFlag*

Additional flags to indicate frame data properties. See the FrameDataFlag enumerator for details.

## Public Members

*mfxExtBuffer* **\*\*ExtParam**

Points to an array of pointers to the extra configuration structures; see the ExtendedBufferID enumerator for a list of extended configurations.

*mfxU16* **PitchLow**

Distance in bytes between the start of two consecutive rows in a frame.

*mfxU8* **\*Y**

Y channel.

*mfxU16* **\*Y16**

Y16 channel.

*mfxU8* **\*R**

R channel.

*mfxU8* **\*UV**

UV channel for UV merged formats.

*mfxU8* **\*VU**

YU channel for VU merged formats.

*mfxU8* **\*CbCr**

CbCr channel for CbCr merged formats.

*mfxU8* **\*CrCb**

CrCb channel for CrCb merged formats.

*mfxU8* **\*Cb**

Cb channel.

*mfxU8* **\*U**

U channel.

*mfxU16* **\*U16**

U16 channel.

*mfxU8* **\*G**

G channel.

*mfxY410* **\*Y410**

T410 channel for Y410 format (merged AVYU).

*mfxU8* **\*Cr**

Cr channel.

*mfxU8* **\*V**

V channel.

*mfxU16* **\*V16**

V16 channel.

*mfxU8* **\*B**

B channel.

*mfxA2RGB10* **\*A2RGB10**

A2RGB10 channel for A2RGB10 format (merged ARGB).

### 12.7.4.13 mfxFrameSurfaceInterface

`struct mfxFrameSurfaceInterface`

#### Public Members

##### *mfxHDL Context*

This context of memory interface. User should not touch (change, set, null) this pointer.

##### *mfxStructVersion Version*

The version of the structure.

##### *mfxStatus (\*AddRef)(mfxFrameSurface1 \*surface)*

This function increments the internal reference counter of the surface, so user is going to keep the surface. The surface cannot be destroyed until user wouldn't call (\*Release). It's expected that users would call (\*AddRef)() each time when they create new links (copy structure, etc) to the surface for proper management.

**Return** MFX\_ERR\_NONE if no error. MFX\_ERR\_NULL\_PTR if surface is NULL.

MFX\_ERR\_INVALID\_HANDLE if mfxFrameSurfaceInterface->Context is invalid (for example NULL).

MFX\_ERR\_UNKNOWN in case of any internal error.

#### Parameters

- [in] surface: valid surface.

##### *mfxStatus (\*Release)(mfxFrameSurface1 \*surface)*

This function decrements the internal reference counter of the surface, users have to care about calling of (\*Release) after (\*AddRef) or when it's required according to the allocation logic. For instance, users have to call (\*Release) to release a surface obtained with GetSurfaceForXXX function.

**Return** MFX\_ERR\_NONE if no error. MFX\_ERR\_NULL\_PTR if surface is NULL.

MFX\_ERR\_INVALID\_HANDLE if mfxFrameSurfaceInterface->Context is invalid (for example NULL).

MFX\_ERR\_UNDEFINED\_BEHAVIOR if Reference Counter of surface is zero before call.

MFX\_ERR\_UNKNOWN in case of any internal error.

#### Parameters

- [in] surface: valid surface.

##### *mfxStatus (\*GetRefCounter)(mfxFrameSurface1 \*surface, mfxU32 \*counter)*

This function returns current reference counter of *mfxFrameSurface1* structure.

**Return** MFX\_ERR\_NONE if no error. MFX\_ERR\_NULL\_PTR if surface or counter is NULL.

MFX\_ERR\_INVALID\_HANDLE if mfxFrameSurfaceInterface->Context is invalid (for example NULL).

MFX\_ERR\_UNKNOWN in case of any internal error.

#### Parameters

- [in] surface: valid surface.

- [out] counter: sets counter to the current reference counter value.

**mfxStatus (\*Map) (mfxFrameSurface1 \*surface, mfxU32 flags)**

This function set pointers of surface->Info.Data to actual pixel data, providing read-write access. In case of video memory, actual surface with data in video memory becomes mapped to system memory. An application can map a surface for read with any value of mfxFrameSurface1::Data.Locked, but for write only when mfxFrameSurface1::Data.Locked equals to 0. Note: surface allows shared read access, but exclusive write access. Let consider the following cases: -Map with Write or ReadWrite flags. Request during active another read or write access returns MFX\_ERR\_LOCK\_MEMORY error immediately, without waiting. MFX\_MAP\_NOWAIT doesn't impact behavior. Such request doesn't lead to any implicit synchronizations. -Map with Read flag. Request during active write access will wait for resource to become free, or exits immediately with error if MFX\_MAP\_NOWAIT flag was set. This request may lead to the implicit synchronization (with same logic as Synchronize call) waiting for surface to become ready to use (all dependencies should be resolved and upstream components finished writing to this surface). It is guaranteed that read access will be acquired right after synchronization without allowing other thread to acquire this surface for writing. If MFX\_MAP\_NOWAIT was set and surface isn't ready yet (has some unresolved data dependencies or active processing) read access request exits immediately with error. Read-write access with MFX\_MAP\_READ\_WRITE provides exclusive simultaneous reading and writing access.

**Return** MFX\_ERR\_NONE if no error. MFX\_ERR\_NULL\_PTR if surface is NULL.

MFX\_ERR\_INVALID\_HANDLE if mfxFrameSurfaceInterface->Context is invalid (for example NULL).

MFX\_ERR\_UNSUPPORTED if flags are invalid.

MFX\_ERR\_LOCK\_MEMORY if user wants to map the surface for write and surface->Data.Locked doesn't equal to 0.

MFX\_ERR\_UNKNOWN in case of any internal error.

#### Parameters

- [in] surface: valid surface.
- [out] flags: to specify mapping mode.
- [out] surface->Info.Data: - pointers set to actual pixel data.

**mfxStatus (\*Unmap) (mfxFrameSurface1 \*surface)**

This function invalidates pointers of surface->Info.Data and sets them to NULL. In case of video memory, actual surface with data in video memory becomes unmapped.

**Return** MFX\_ERR\_NONE if no error. MFX\_ERR\_NULL\_PTR if surface is NULL.

MFX\_ERR\_INVALID\_HANDLE if mfxFrameSurfaceInterface->Context is invalid (for example NULL).

MFX\_ERR\_UNSUPPORTED if surface is already unmapped.

MFX\_ERR\_UNKNOWN in case of any internal error.

#### Parameters

- [in] surface: valid surface
- [out] surface->Info.Data: - pointers set to NULL

**mfxStatus (\*GetNativeHandle) (mfxFrameSurface1 \*surface, mfxHDL \*resource, mfxResourceType \*resource\_type)**

This function returns a native resource's handle and type. The handle is returned *as-is* that means the

reference counter of base resources is not incremented. Native resource is not detached from surface, the library still owns the resource. User must not anyhow destroy native resource or rely that this resource will be alive after (\*Release).

**Return** MFX\_ERR\_NONE if no error. MFX\_ERR\_NULL\_PTR if any of surface, resource or resource\_type is NULL.

MFX\_ERR\_INVALID\_HANDLE if any of surface, resource or resource\_type is not valid object (no native resource was allocated).

MFX\_ERR\_UNSUPPORTED if surface is in system memory.

MFX\_ERR\_UNKNOWN in case of any internal error.

#### Parameters

- [in] surface: valid surface.
- [out] resource: - pointer is set to the native handle of the resource.
- [out] resource\_type: - type of native resource (see mfxResourceType enumeration).

`mfxStatus (*GetDeviceHandle)(mfxFrameSurface1 *surface, mfxHDL *device_handle, mfxHandleType *device_type)`

This function returns a device abstraction which was used to create that resource. The handle is returned *as-is* that means the reference counter for device abstraction is not incremented. Native resource is not detached from surface, the library still has a reference to the resource. User must not anyhow destroy device or rely that this device will be alive after (\*Release).

**Return** MFX\_ERR\_NONE if no error. MFX\_ERR\_NULL\_PTR if any of surface, devic\_handle or device\_type is NULL.

MFX\_ERR\_INVALID\_HANDLE if any of surface, resource or resource\_type is not valid object (no native resource was allocated).

MFX\_ERR\_UNSUPPORTED if surface is in system memory.

MFX\_ERR\_UNKNOWN in case of any internal error.

#### Parameters

- [in] surface: valid surface.
- [out] device\_handle: - pointer is set to the device which created the resource
- [out] device\_type: - type of device (see mfxHandleType enumeration).

`mfxStatus (*Synchronize)(mfxFrameSurface1 *surface, mfxU32 wait)`

This function guarantees readiness both of the data (pixels) and any frame's meta information (e.g. corruption flags) after function complete. Instead of MFXVideoCORE\_SyncOperation users may directly call (\*Synchronize) after correspondent Decode/VPP function calls (MFXVideoDECODE\_DecodeFrameAsync or MFXVideoVPP\_RunFrameVPPAsync). The prerequisites to call the functions are: main processing functions returned MFX\_ERR\_NONE and valid *mfxFrameSurface1* object.

**Return** MFX\_ERR\_NONE if no error. MFX\_ERR\_NULL\_PTR if surface is NULL.

MFX\_ERR\_INVALID\_HANDLE if any of surface is not valid object .

MFX\_WRN\_IN\_EXECUTION if the given timeout is expired and the surface is not ready.

MFX\_ERR\_ABORTED if the specified asynchronous function aborted due to data dependency on a previous asynchronous function that did not complete.

MFX\_ERR\_UNKNOWN in case of any internal error.

#### Parameters

- [in] surface: - valid surface.
- [out] wait: - wait time in milliseconds.

### 12.7.4.14 mfxFrameSurface1

#### **struct mfxFrameSurface1**

The *mfxFrameSurface1* structure defines the uncompressed frames surface information and data buffers. The frame surface is in the frame or complementary field pairs of pixels up to four color-channels, in two parts: *mfxFrameInfo* and *mfxFrameData*.

#### Public Members

##### **struct mfxFrameSurfaceInterface \*FrameInterface**

*mfxFrameSurfaceInterface* specifies interface to work with surface.

##### **mfxFrameInfo Info**

*mfxFrameInfo* structure specifies surface properties.

##### **mfxFrameData Data**

*mfxFrameData* structure describes the actual frame buffer.

### 12.7.4.15 mfxBitstream

#### **struct mfxBitstream**

The *mfxBitstream* structure defines the buffer that holds compressed video data.

#### Public Members

##### **mfxEncryptedData \*EncryptedData**

Reserved and must be zero.

##### **mfxExtBuffer \*\*ExtParam**

Array of extended buffers for additional bitstream configuration. See the ExtendedBufferID enumerator for a complete list of extended buffers.

##### **mfxU16 NumExtParam**

The number of extended buffers attached to this structure.

##### **mfxI64 DecodeTimeStamp**

Decode time stamp of the compressed bitstream in units of 90KHz. A value of MFX\_TIMESTAMP\_UNKNOWN indicates that there is no time stamp. This value is calculated by the SDK encoder from the presentation time stamp provided by the application in *mfxFrameSurface1* structure and from the frame rate provided by the application during the SDK encoder initialization.

##### **mfxU64 TimeStamp**

Time stamp of the compressed bitstream in units of 90KHz. A value of MFX\_TIMESTAMP\_UNKNOWN indicates that there is no time stamp.

##### **mfxU8 \*Data**

Bitstream buffer pointer, 32-bytes aligned.

***mfxU32 DataOffset***

Next reading or writing position in the bitstream buffer.

***mfxU32 DataLength***

Size of the actual bitstream data in bytes.

***mfxU32 MaxLength***

Allocated bitstream buffer size in bytes.

***mfxU16 PicStruct***

Type of the picture in the bitstream; this is an output parameter.

***mfxU16 FrameType***

Frame type of the picture in the bitstream; this is an output parameter.

***mfxU16 DataFlag***

Indicates additional bitstream properties; see the BitstreamDataFlag enumerator for details.

***mfxU16 reserved2***

Reserved for future use.

**12.7.4.16 mfxEncodeStat****struct mfxEncodeStat**

The *mfxEncodeStat* structure returns statistics collected during encoding.

**Public Members*****mfxU32 NumFrame***

Number of encoded frames.

***mfxU64 NumBit***

Number of bits for all encoded frames.

***mfxU32 NumCachedFrame***

Number of internally cached frames.

**12.7.4.17 mfxDecodeStat****struct mfxDecodeStat**

The *mfxDecodeStat* structure returns statistics collected during decoding.

**Public Members*****mfxU32 NumFrame***

Number of total decoded frames.

***mfxU32 NumSkippedFrame***

Number of skipped frames.

***mfxU32 NumError***

Number of errors recovered.

***mfxU32 NumCachedFrame***

Number of internally cached frames.

### 12.7.4.18 mfxPayload

#### **struct mfxPayload**

The *mfxPayload* structure describes user data payload in MPEG-2 or SEI message payload in H.264. For encoding, these payloads can be inserted into the bitstream. The payload buffer must contain a valid formatted payload. For H.264, this is the sei\_message() as specified in the section 7.3.2.3.1 ‘Supplemental enhancement information message syntax’ of the ISO/IEC 14496-10 specification. For MPEG-2, this is the section 6.2.2.2.2 ‘User data’ of the ISO/IEC 13818-2 specification, excluding the user data start\_code. For decoding, these payloads can be retrieved as the decoder parses the bitstream and caches them in an internal buffer.

#### Public Members

##### *mfxU32 CtrlFlags*

Additional payload properties. See the PayloadCtrlFlags enumerator for details.

##### *mfxU8 \*Data*

Pointer to the actual payload data buffer.

##### *mfxU32 NumBit*

Number of bits in the payload data

##### *mfxU16 Type*

MPEG-2 user data start code or H.264 SEI message type.

##### *mfxU16 BufSize*

Payload buffer size in bytes.

Code	Supported Types
MPEG	0x01B2 //User Data
AVC	02 //pan_scan_rect 03 //filler_payload 04 //user_data_registered_itu_t_t35 05 //user_data_unregistered 06 //recovery_point 09 //scene_info 13 //full_frame_freeze 14 //full_frame_freeze_release 15 //full_frame_snapshot 16 //progressive_refinement_segment_start 17 //progressive_refinement_segment_end 19 //film_grain_characteristics 20 //deblocking_filter_display_preference 21 //stereo_video_info 45 //frame_packing_arrangement
HEVC	All

### 12.7.4.19 mfxEncodeCtrl

**struct mfxEncodeCtrl**

The *mfxEncodeCtrl* structure contains parameters for per-frame based encoding control.

#### Public Members

***mfxExtBuffer* Header**

Extension buffer header.

***mfxU16* MfxNalUnitType**

Type of NAL unit that contains encoding frame. All supported values are defined by MfxNalUnitType enumerator. Other values defined in ITU-T H.265 specification are not supported.

The SDK encoder uses this field only if application sets *mfxExtCodingOption3::EnableNalUnitType* option to ON during encoder initialization.

**Note** Only encoded order is supported. If application specifies this value in display order or uses value inappropriate for current frame or invalid value, then SDK encoder silently ignores it.

***mfxU16* SkipFrame**

Indicates that current frame should be skipped or number of missed frames before the current frame. See the *mfxExtCodingOption2::SkipFrame* for details.

***mfxU16* QP**

If nonzero, this value overwrites the global QP value for the current frame in the constant QP mode.

***mfxU16* FrameType**

Encoding frame type; see the FrameType enumerator for details. If the encoder works in the encoded order, the application must specify the frame type. If the encoder works in the display order, only key frames are enforceable.

***mfxU16* NumExtParam**

Number of extra control buffers.

***mfxU16* NumPayload**

Number of payload records to insert into the bitstream.

***mfxExtBuffer* \*\*ExtParam**

Pointer to an array of pointers to external buffers that provide additional information or control to the encoder for this frame or field pair; a typical usage is to pass the VPP auxiliary data generated by the video processing pipeline to the encoder. See the ExtendedBufferID for the list of extended buffers.

***mfxPayload* \*\*Payload**

Pointer to an array of pointers to user data (MPEG-2) or SEI messages (H.264) for insertion into the bitstream; for field pictures, odd payloads are associated with the first field and even payloads are associated with the second field. See the *mfxPayload* structure for payload definitions.

#### 12.7.4.20 mfxFrameAllocRequest

**struct mfxFrameAllocRequest**

The *mfxFrameAllocRequest* structure describes multiple frame allocations when initializing encoders, decoders and video preprocessors. A range specifies the number of video frames. Applications are free to allocate additional frames. In any case, the minimum number of frames must be at least NumFrameMin or the called function will return an error.

##### Public Members

*mfxU32 AllocId*

Unique (within the session) ID of component requested the allocation.

*mfxFrameInfo Info*

Describes the properties of allocated frames.

*mfxU16 Type*

Allocated memory type; see the ExtMemFrameType enumerator for details.

*mfxU16 NumFrameMin*

Minimum number of allocated frames.

*mfxU16 NumFrameSuggested*

Suggested number of allocated frames.

#### 12.7.4.21 mfxFrameAllocResponse

**struct mfxFrameAllocResponse**

The *mfxFrameAllocResponse* structure describes the response to multiple frame allocations. The calling function returns the number of video frames actually allocated and pointers to their memory IDs.

##### Public Members

*mfxU32 AllocId*

Unique (within the session) ID of component requested the allocation.

*mfxMemId \*mids*

Pointer to the array of the returned memory IDs; the application allocates or frees this array.

*mfxU16 NumFrameActual*

Number of frames actually allocated.

#### 12.7.4.22 mfxFrameAllocator

**struct mfxFrameAllocator**

The *mfxFrameAllocator* structure describes the callback functions Alloc, Lock, Unlock, GetHDL and Free that the SDK implementation might use for allocating internal frames. Applications that operate on OS-specific video surfaces must implement these callback functions.

Using the default allocator implies that frame data passes in or out of SDK functions through pointers, as opposed to using memory IDs.

The SDK behavior is undefined when using an incompletely defined external allocator. See the section Memory Allocation and External Allocators for additional information.

## Public Members

### `mfxHDL pthis`

Pointer to the allocator object.

### `mfxStatus (*Alloc)(mfxHDL pthis, mfxFrameAllocRequest *request, mfxFrameAllocResponse *response)`

This function allocates surface frames. For decoders, MFXVideoDECODE\_Init calls Alloc only once. That call includes all frame allocation requests. For encoders, MFXVideoENCODE\_Init calls Alloc twice: once for the input surfaces and again for the internal reconstructed surfaces.

If two SDK components must share DirectX\* surfaces, this function should pass the pre-allocated surface chain to SDK instead of allocating new DirectX surfaces. See the Surface Pool Allocation section for additional information.

**Return** MFX\_ERR\_NONE The function successfully allocated the memory block.  
MFX\_ERR\_MEMORY\_ALLOC The function failed to allocate the video frames.

MFX\_ERR\_UNSUPPORTED The function does not support allocating the specified type of memory.

### Parameters

- [in] `pthis`: Pointer to the allocator object.
- [in] `request`: Pointer to the `mfxFrameAllocRequest` structure that specifies the type and number of required frames.
- [out] `response`: Pointer to the `mfxFrameAllocResponse` structure that retrieves frames actually allocated.

### `mfxStatus (*Lock)(mfxHDL pthis, mfxMemId mid, mfxFrameData *ptr)`

This function locks a frame and returns its pointer.

**Return** MFX\_ERR\_NONE The function successfully locked the memory block.  
MFX\_ERR\_LOCK\_MEMORY This function failed to lock the frame.

### Parameters

- [in] `pthis`: Pointer to the allocator object.
- [in] `mid`: Memory block ID.
- [out] `ptr`: Pointer to the returned frame structure.

### `mfxStatus (*Unlock)(mfxHDL pthis, mfxMemId mid, mfxFrameData *ptr)`

This function unlocks a frame and invalidates the specified frame structure.

**Return** MFX\_ERR\_NONE The function successfully locked the memory block.

### Parameters

- [in] `pthis`: Pointer to the allocator object.
- [in] `mid`: Memory block ID.
- [out] `ptr`: Pointer to the frame structure; This pointer can be NULL.

### `mfxStatus (*GetHDL)(mfxHDL pthis, mfxMemId mid, mfxHDL *handle)`

This function returns the OS-specific handle associated with a video frame. If the handle is a COM interface, the reference counter must increase. The SDK will release the interface afterward.

**Return** MFX\_ERR\_NONE The function successfully returned the OS-specific handle.  
MFX\_ERR\_UNSUPPORTED The function does not support obtaining OS-specific handle..

**Parameters**

- [in] pthis: Pointer to the allocator object.
- [in] mid: Memory block ID.
- [out] handle: Pointer to the returned OS-specific handle.

*mfxStatus (\*Free)(mfxHDL pthis, mfxFrameAllocResponse \*response)*

This function de-allocates all allocated frames.

**Return** MFX\_ERR\_NONE The function successfully de-allocated the memory block.

**Parameters**

- [in] pthis: Pointer to the allocator object.
- [in] response: Pointer to the *mfxFrameAllocResponse* structure returned by the Alloc function.

**12.7.4.23 mfxComponentInfo****struct mfxComponentInfo**

The *mfxComponentInfo* structure contains workload description, which is accepted by MFXQueryAdapters function.

**Public Members***mfxComponentType* **Type**

Type of workload: Encode, Decode, VPP. See *mfxComponentType* enumerator for possible values.

*mfxVideoParam* **Requirements**

Detailed description of workload, see *mfxVideoParam* for details.

**12.7.4.24 mfxAdapterInfo****struct mfxAdapterInfo**

The *mfxAdapterInfo* structure contains a description of the graphics adapter.

**Public Members***mfxPlatform* **Platform**

Platform type description, see *mfxPlatform* for details.

*mfxU32* **Number**

Value which uniquely characterizes media adapter. On Windows\* this number can be used for initialization through DXVA interface (see [example](#)).

#### 12.7.4.25 mfxAdaptersInfo

**struct mfxAdaptersInfo**

The *mfxAdaptersInfo* structure contains description of all graphics adapters available on the current system.

##### Public Members

*mfxAdapterInfo* \***Adapters**

Pointer to array of *mfxAdapterInfo* structs allocated by user.

*mfxU32* **NumAlloc**

Length of Adapters array.

*mfxU32* **NumActual**

Number of Adapters entries filled by MFXQueryAdapters.

#### 12.7.4.26 mfxQPandMode

**struct mfxQPandMode**

The *mfxQPandMode* structure specifies per-MB or per-CU mode and QP or deltaQP value depending on the mode type.

##### Public Members

*mfxU8* **QP**

QP for MB or CU. Valid when Mode = MFX\_MBQP\_MODE\_QP\_VALUE. For AVC valid range is 1..51. For HEVC valid range is 1..51. Application's provided QP values should be valid; otherwise invalid QP values may cause undefined behavior. MBQP map should be aligned for 16x16 block size. (align rule is (width +15 /16) && (height +15 /16)) For MPEG2 QP corresponds to quantizer\_scale of the ISO\*VIEC\* 13818-2 specification and have valid range 1..112.

*mfxI8* **DeltaQP**

Pointer to a list of per-macroblock QP deltas in raster scan order. For block i: QP[i] = BrQP[i] + DeltaQP[i]. Valid when Mode = MFX\_MBQP\_MODE\_QP\_DELTA.

*mfxU16* **Mode**

Defines QP update mode. Can be equal to MFX\_MBQP\_MODE\_QP\_VALUE or MFX\_MBQP\_MODE\_QP\_DELTA.

#### 12.7.4.27 VPP Structures

##### 12.7.4.27.1 mfxInfoVPP

**struct mfxInfoVPP**

## Public Members

### *mfxFrameInfo* In

Input format for video processing.

### *mfxFrameInfo* Out

Output format for video processing.

## 12.7.4.27.2 mfxVPPStat

### struct mfxVPPStat

The *mfxVPPStat* structure returns statistics collected during video processing.

## Public Members

### *mfxU32 NumFrame*

Total number of frames processed.

### *mfxU32 NumCachedFrame*

Number of internally cached frames.

## 12.7.4.28 Extension buffers structures

### 12.7.4.28.1 mfxExtBuffer

### struct mfxExtBuffer

This structure is the common header definition for external buffers and video processing hints.

## Public Members

### *mfxU32 BufferId*

Identifier of the buffer content. See the ExtendedBufferID enumerator for a complete list of extended buffers.

### *mfxU32 BufferSz*

Size of the buffer.

### 12.7.4.28.2 mfxExtCodingOption

### struct mfxExtCodingOption

The *mfxExtCodingOption* structure specifies additional options for encoding.

The application can attach this extended buffer to the *mfxVideoParam* structure to configure initialization.

## Public Members

### *mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_CODING\_OPTION.

### *mfxU16* **RateDistortionOpt**

Set this flag if rate distortion optimization is needed. See the CodingOptionValue enumerator for values of this option.

### *mfxU16* **MECostType**

Motion estimation cost type; this value is reserved and must be zero.

### *mfxU16* **MESearchType**

Motion estimation search algorithm; this value is reserved and must be zero.

### *mfxI16Pair* **MVSearchWindow**

Rectangular size of the search window for motion estimation; this parameter is reserved and must be (0, 0).

### *mfxU16* **FramePicture**

Set this flag to encode interlaced fields as interlaced frames; this flag does not affect progressive input frames. See the CodingOptionValue enumerator for values of this option.

### *mfxU16* **CAVLC**

If set, CAVLC is used; if unset, CABAC is used for encoding. See the CodingOptionValue enumerator for values of this option.

### *mfxU16* **RecoveryPointSEI**

Set this flag to insert the recovery point SEI message at the beginning of every intra refresh cycle. See the description of IntRefType in *mfxExtCodingOption2* structure for details on how to enable and configure intra refresh.

If intra refresh is not enabled then this flag is ignored.

See the CodingOptionValue enumerator for values of this option.

### *mfxU16* **ViewOutput**

Set this flag to instruct the MVC encoder to output each view in separate bitstream buffer. See the CodingOptionValue enumerator for values of this option and SDK Reference Manual for Multi-View Video Coding for more details about usage of this flag.

### *mfxU16* **NalHrdConformance**

If this option is turned ON, then AVC encoder produces HRD conformant bitstream. If it is turned OFF, then AVC encoder may, but not necessary does, violate HRD conformance. I.e. this option can force encoder to produce HRD conformant stream, but cannot force it to produce non-conformant stream.

See the CodingOptionValue enumerator for values of this option.

### *mfxU16* **SingleSeiNalUnit**

If set, encoder puts all SEI messages in the singe NAL unit. It includes both kinds of messages, provided by application and created by encoder. It is three states option, see CodingOptionValue enumerator for values of this option:

UNKNOWN - put each SEI in its own NAL unit,

ON - put all SEI messages in the same NAL unit,

OFF - the same as unknown

### *mfxU16* **VuiVclHrdParameters**

If set and VBR rate control method is used then VCL HRD parameters are written in bitstream with

identical to NAL HRD parameters content. See the CodingOptionValue enumerator for values of this option.

***mfxU16 RefPicListReordering***

Set this flag to activate reference picture list reordering; this value is reserved and must be zero.

***mfxU16 ResetRefList***

Set this flag to reset the reference list to non-IDR I-frames of a GOP sequence. See the CodingOptionValue enumerator for values of this option.

***mfxU16 RefPicMarkRep***

Set this flag to write the reference picture marking repetition SEI message into the output bitstream. See the CodingOptionValue enumerator for values of this option.

***mfxU16 FieldOutput***

Set this flag to instruct the AVC encoder to output bitstreams immediately after the encoder encodes a field, in the field-encoding mode. See the CodingOptionValue enumerator for values of this option.

***mfxU16 IntraPredBlockSize***

Minimum block size of intra-prediction; This value is reserved and must be zero.

***mfxU16 InterPredBlockSize***

Minimum block size of inter-prediction; This value is reserved and must be zero.

***mfxU16 Mvprecision***

Specify the motion estimation precision; this parameter is reserved and must be zero.

***mfxU16 MaxDecFrameBuffering***

Specifies the maximum number of frames buffered in a DPB. A value of zero means unspecified.

***mfxU16 AUDelimiter***

Set this flag to insert the Access Unit Delimiter NAL. See the CodingOptionValue enumerator for values of this option.

***mfxU16 PicTimingSEI***

Set this flag to insert the picture timing SEI with pic\_struct syntax element. See sub-clauses D.1.2 and D.2.2 of the ISO/IEC 14496-10 specification for the definition of this syntax element. See the CodingOptionValue enumerator for values of this option. The default value is ON.

***mfxU16 VuiNalHrdParameters***

Set this flag to insert NAL HRD parameters in the VUI header. See the CodingOptionValue enumerator for values of this option.

#### 12.7.4.28.3 ***mfxExtCodingOption2***

**`struct mfxExtCodingOption2`**

The *mfxExtCodingOption2* structure together with *mfxExtCodingOption* structure specifies additional options for encoding.

The application can attach this extended buffer to the *mfxVideoParam* structure to configure initialization and to the *mfxEncodeCtrl* during runtime.

## Public Members

### *mfxExtBuffer* Header

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_CODING\_OPTION2.

### *mfxU16* IntRefType

Specifies intra refresh type. See the IntraRefreshTypes. The major goal of intra refresh is improvement of error resilience without significant impact on encoded bitstream size caused by I frames. The SDK encoder achieves this by encoding part of each frame in refresh cycle using intra MBs. MFX\_REFRESH\_NO means no refresh. MFX\_REFRESH\_VERTICAL means vertical refresh, by column of MBs. MFX\_REFRESH\_HORIZONTAL means horizontal refresh, by rows of MBs. MFX\_REFRESH\_SLICE means horizontal refresh by slices without overlapping. In case of MFX\_REFRESH\_SLICE SDK ignores IntRefCycleSize (size of refresh cycle equals number slices). This parameter is valid during initialization and runtime. When used with temporal scalability, intra refresh applied only to base layer.

### *mfxU16* IntRefCycleSize

Specifies number of pictures within refresh cycle starting from 2. 0 and 1 are invalid values. This parameter is valid only during initialization.

### *mfxI16* IntRefQPDelta

Specifies QP difference for inserted intra MBs. This is signed value in [-51, 51] range. This parameter is valid during initialization and runtime.

### *mfxU32* MaxFrameSize

Specify maximum encoded frame size in byte. This parameter is used in VBR based bitrate control modes and ignored in others. The SDK encoder tries to keep frame size below specified limit but minor overshoots are possible to preserve visual quality. This parameter is valid during initialization and runtime. It is recommended to set MaxFrameSize to 5x-10x target frame size ((TargetKbps\*1000)/(8\* FrameRate-ExtN/FrameRateExtD)) for I frames and 2x-4x target frame size for P/B frames.

### *mfxU32* MaxSliceSize

Specify maximum slice size in bytes. If this parameter is specified other controls over number of slices are ignored.

**Note** Not all codecs and SDK implementations support this value. Use Query function to check if this feature is supported.

### *mfxU16* BitrateLimit

Modifies bitrate to be in the range imposed by the SDK encoder. Setting this flag off may lead to violation of HRD conformance. Mind that specifying bitrate below the SDK encoder range might significantly affect quality. If on this option takes effect in non CQP modes: if TargetKbps is not in the range imposed by the SDK encoder, it will be changed to be in the range. See the CodingOptionValue enumerator for values of this option. The default value is ON, i.e. bitrate is limited. This parameter is valid only during initialization. Flag works with MFX\_CODEC\_AVC only, it is ignored with other codecs.

### *mfxU16* MBBRC

Setting this flag enables macroblock level bitrate control that generally improves subjective visual quality. Enabling this flag may have negative impact on performance and objective visual quality metric. See the CodingOptionValue enumerator for values of this option. The default value depends on target usage settings.

### *mfxU16* ExtBRC

Turn ON this option to enable external BRC. See the CodingOptionValue enumerator for values of this option. Use Query function to check if this feature is supported.

### *mfxU16* LookAheadDepth

Specifies the depth of look ahead rate control algorithm. It is the number of frames that SDK encoder

analyzes before encoding. Valid value range is from 10 to 100 inclusive. To instruct the SDK encoder to use the default value the application should zero this field.

#### *mfxU16* **Trellis**

This option is used to control trellis quantization in AVC encoder. See TrellisControl enumerator for possible values of this option. This parameter is valid only during initialization.

#### *mfxU16* **RepeatPPS**

This flag controls picture parameter set repetition in AVC encoder. Turn ON this flag to repeat PPS with each frame. See the CodingOptionValue enumerator for values of this option. The default value is ON. This parameter is valid only during initialization.

#### *mfxU16* **BRefType**

This option controls usage of B frames as reference. See BRefControl enumerator for possible values of this option. This parameter is valid only during initialization.

#### *mfxU16* **AdaptiveI**

This flag controls insertion of I frames by the SDK encoder. Turn ON this flag to allow changing of frame type from P and B to I. This option is ignored if GopOptFlag in *mfxInfoMFX* structure is equal to MFX\_GOP\_STRICT. See the CodingOptionValue enumerator for values of this option. This parameter is valid only during initialization.

#### *mfxU16* **AdaptiveB**

This flag controls changing of frame type from B to P. Turn ON this flag to allow such changing. This option is ignored if GopOptFlag in *mfxInfoMFX* structure is equal to MFX\_GOP\_STRICT. See the CodingOptionValue enumerator for values of this option. This parameter is valid only during initialization.

#### *mfxU16* **LookAheadDS**

This option controls down sampling in look ahead bitrate control mode. See LookAheadDownSampling enumerator for possible values of this option. This parameter is valid only during initialization.

#### *mfxU16* **NumMbPerSlice**

This option specifies suggested slice size in number of macroblocks. The SDK can adjust this number based on platform capability. If this option is specified, i.e. if it is not equal to zero, the SDK ignores *mfxInfoMFX::NumSlice* parameter.

#### *mfxU16* **SkipFrame**

This option enables usage of mfxEncodeCtrl::SkipFrameparameter. See the SkipFrame enumerator for values of this option.

**Note** Not all codecs and SDK implementations support this value. Use Query function to check if this feature is supported.

#### *mfxU8* **MinQPI**

Minimum allowed QP value for I frame types. Valid range is 1..51 inclusive. Zero means default value, i.e.no limitations on QP.

**Note** Not all codecs and SDK implementations support this value. Use Query function to check if this feature is supported.

#### *mfxU8* **MaxQPI**

Maximum allowed QP value for I frame types. Valid range is 1..51 inclusive. Zero means default value, i.e.no limitations on QP.

**Note** Not all codecs and SDK implementations support this value. Use Query function to check if this feature is supported.

#### *mfxU8* **MinQPP**

Minimum allowed QP value for P frame types. Valid range is 1..51 inclusive. Zero means default value, i.e.no limitations on QP.

**Note** Not all codecs and SDK implementations support this value. Use Query function to check if this feature is supported.

#### *mfxU8 MaxQPP*

Maximum allowed QP value for P frame types. Valid range is 1..51 inclusive. Zero means default value, i.e.no limitations on QP.

**Note** Not all codecs and SDK implementations support this value. Use Query function to check if this feature is supported.

#### *mfxU8 MinQPB*

Minimum allowed QP value for B frame types. Valid range is 1..51 inclusive. Zero means default value, i.e.no limitations on QP.

**Note** Not all codecs and SDK implementations support this value. Use Query function to check if this feature is supported.

#### *mfxU8 MaxQPB*

Maximum allowed QP value for B frame types. Valid range is 1..51 inclusive. Zero means default value, i.e.no limitations on QP.

**Note** Not all codecs and SDK implementations support this value. Use Query function to check if this feature is supported.

#### *mfxU16 FixedFrameRate*

This option sets fixed\_frame\_rate\_flag in VUI.

**Note** Not all codecs and SDK implementations support this value. Use Query function to check if this feature is supported.

#### *mfxU16 DisableDeblockingIdc*

This option disables deblocking.

**Note** Not all codecs and SDK implementations support this value. Use Query function to check if this feature is supported.

#### *mfxU16 DisableVUI*

This option completely disables VUI in output bitstream.

**Note** Not all codecs and SDK implementations support this value. Use Query function to check if this feature is supported.

#### *mfxU16 BufferingPeriodSEI*

This option controls insertion of buffering period SEI in the encoded bitstream. It should be one of the following values:

MFX\_BPSEI\_DEFAULT – encoder decides when to insert BP SEI,

MFX\_BPSEI\_IFRAME – BP SEI should be inserted with every I frame.

#### *mfxU16 EnableMAD*

Turn ON this flag to enable per-frame reporting of Mean Absolute Difference. This parameter is valid only during initialization.

#### *mfxU16 UseRawRef*

Turn ON this flag to use raw frames for reference instead of reconstructed frames. This parameter is valid during initialization and runtime (only if was turned ON during initialization).

**Note** Not all codecs and SDK implementations support this value. Use Query function to check if this feature is supported.

#### 12.7.4.28.4 mfxExtCodingOption3

##### **struct mfxExtCodingOption3**

The *mfxExtCodingOption3* structure together with *mfxExtCodingOption* and *mfxExtCodingOption2* structures specifies additional options for encoding. The application can attach this extended buffer to the *mfxVideoParam* structure to configure initialization and to the *mfxEncodeCtrl* during runtime.

##### **Public Members**

###### *mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_CODING\_OPTION3.

###### *mfxU16* **NumSliceI**

The number of slices for I frames.

**Note** Not all codecs and SDK implementations support these values. Use Query function to check if this feature is supported

###### *mfxU16* **NumSliceP**

The number of slices for P frames.

**Note** Not all codecs and SDK implementations support these values. Use Query function to check if this feature is supported

###### *mfxU16* **NumSliceB**

The number of slices for B frames.

**Note** Not all codecs and SDK implementations support these values. Use Query function to check if this feature is supported

###### *mfxU16* **WinBRCMaxAvgKbps**

When rate control method is MFX\_RATECONTROL\_VBR, MFX\_RATECONTROL\_LA, MFX\_RATECONTROL\_LA\_HRD or MFX\_RATECONTROL\_QVBR this parameter specifies the maximum bitrate averaged over a sliding window specified by WinBRCSize. For MFX\_RATECONTROL\_CBR this parameter is ignored and equals TargetKbps.

###### *mfxU16* **WinBRCSize**

When rate control method is MFX\_RATECONTROL\_CBR, MFX\_RATECONTROL\_VBR, MFX\_RATECONTROL\_LA, MFX\_RATECONTROL\_LA\_HRD or MFX\_RATECONTROL\_QVBR this parameter specifies sliding window size in frames. Set this parameter to zero to disable sliding window.

###### *mfxU16* **QVBRQuality**

When rate control method is MFX\_RATECONTROL\_QVBR this parameter specifies quality factor. It is a value in the 1,...,51 range, where 1 corresponds to the best quality.

###### *mfxU16* **EnableMBQP**

Turn ON this option to enable per-macroblock QP control, rate control method must be MFX\_RATECONTROL\_CQP. See the CodingOptionValue enumerator for values of this option. This parameter is valid only during initialization.

###### *mfxU16* **IntRefCycleDist**

Distance between the beginnings of the intra-refresh cycles in frames. Zero means no distance between cycles.

###### *mfxU16* **DirectBiasAdjustment**

Turn ON this option to enable the ENC mode decision algorithm to bias to fewer B Direct/Skip types. Applies only to B frames, all other frames will ignore this setting. See the CodingOptionValue enumerator for values of this option.

***mfxU16 GlobalMotionBiasAdjustment***

Enables global motion bias. See the CodingOptionValue enumerator for values of this option.

***mfxU16 MVCostScalingFactor***

Values are:

- 0: set MV cost to be 0
- 1: scale MV cost to be 1/2 of the default value
- 2: scale MV cost to be 1/4 of the default value
- 3: scale MV cost to be 1/8 of the default value

***mfxU16 MBDisableSkipMap***

Turn ON this option to enable usage of *mfxExtMBDisableSkipMap*. See the CodingOptionValue enumerator for values of this option. This parameter is valid only during initialization.

***mfxU16 WeightedPred***

Weighted prediction mode. See the WeightedPred enumerator for values of these options.

***mfxU16 WeightedBiPred***

Weighted prediction mode. See the WeightedPred enumerator for values of these options.

***mfxU16 AspectRatioInfoPresent***

Instructs encoder whether aspect ratio info should present in VUI parameters. See the CodingOptionValue enumerator for values of this option.

***mfxU16 OverscanInfoPresent***

Instructs encoder whether overscan info should present in VUI parameters. See the CodingOptionValue enumerator for values of this option.

***mfxU16 OverscanAppropriate***

ON indicates that the cropped decoded pictures output are suitable for display using overscan. OFF indicates that the cropped decoded pictures output contain visually important information in the entire region out to the edges of the cropping rectangle of the picture. See the CodingOptionValue enumerator for values of this option.

***mfxU16 TimingInfoPresent***

Instructs encoder whether frame rate info should present in VUI parameters. See the CodingOptionValue enumerator for values of this option.

***mfxU16 BitstreamRestriction***

Instructs encoder whether bitstream restriction info should present in VUI parameters. See the CodingOptionValue enumerator for values of this option.

***mfxU16 LowDelayHrd***

Corresponds to AVC syntax element low\_delay\_hrd\_flag (VUI). See the CodingOptionValue enumerator for values of this option.

***mfxU16 MotionVectorsOverPicBoundaries***

When set to OFF, no sample outside the picture boundaries and no sample at a fractional sample position for which the sample value is derived using one or more samples outside the picture boundaries is used for inter prediction of any sample.

When set to ON, one or more samples outside picture boundaries may be used in inter prediction.

See the CodingOptionValue enumerator for values of this option.

***mfxU16 reserved1[2]***

***mfxU16 ScenarioInfo***

Provides a hint to encoder about the scenario for the encoding session. See the ScenarioInfo enumerator for values of this option.

***mfxU16 ContentInfo***

Provides a hint to encoder about the content for the encoding session. See the ContentInfo enumerator for values of this option.

***mfxU16 PRefType***

When GopRefDist=1, specifies the model of reference list construction and DPB management. See the PRefType enumerator for values of this option.

***mfxU16 FadeDetection***

Instructs encoder whether internal fade detection algorithm should be used for calculation of weigh/offset values for pred\_weight\_table unless application provided *mfxExtPredWeightTable* for this frame. See the CodingOptionValue enumerator for values of this option.

***mfxU16 reserved2[2]******mfxU16 GPB***

Turn this option OFF to make HEVC encoder use regular P-frames instead of GPB. See the CodingOptionValue enumerator for values of this option.

***mfxU32 MaxFrameSizeI***

Same as *mfxExtCodingOption2::MaxFrameSize* but affects only I-frames. MaxFrameSizeI must be set if MaxFrameSizeP is set. If MaxFrameSizeI is not specified or greater than spec limitation, spec limitation will be applied to the sizes of I-frames.

***mfxU32 MaxFrameSizeP***

Same as *mfxExtCodingOption2::MaxFrameSize* but affects only P/B-frames. If MaxFrameSizeP equals 0, the SDK sets MaxFrameSizeP equal to MaxFrameSizeI. If MaxFrameSizeP is not specified or greater than spec limitation, spec limitation will be applied to the sizes of P/B-frames.

***mfxU32 reserved3[3]******mfxU16 EnableQPOffset***

Enables QPOffset control. See the CodingOptionValue enumerator for values of this option.

***mfxI16 QPOffset[8]***

When EnableQPOffset set to ON and RateControlMethod is CQP specifies QP offset per pyramid layer. For B-pyramid, B-frame QP = QPB + QPOffset[layer]. For P-pyramid, P-frame QP = QPP + QPOff-set[layer].

***mfxU16 NumRefActiveP[8]***

< Max number of active references for P and B frames in reference picture lists 0 and 1 correspondingly. Array index is pyramid layer. Max number of active references for P frames. Array index is pyramid layer.

***mfxU16 NumRefActiveBL0[8]***

Max number of active references for B frames in reference picture list 0. Array index is pyramid layer.

***mfxU16 NumRefActiveBL1[8]***

Max number of active references for B frames in reference picture list 1. Array index is pyramid layer.

***mfxU16 reserved6******mfxU16 TransformSkip***

For HEVC if this option turned ON, transform\_skip\_enabled\_flag will be set to 1 in PPS, OFF specifies that transform\_skip\_enabled\_flag will be set to 0.

***mfxU16 TargetChromaFormatPlus1***

Minus 1 specifies target encoding chroma format (see ChromaFormatIdc enumerator). May differ from source one. TargetChromaFormatPlus1 = 0 mean default target chroma format which is equal to source

(mfxVideoParam::mfx::FrameInfo::ChromaFormat + 1), except RGB4 source format. In case of RGB4 source format default target chroma format is 4:2:0 (instead of 4:4:4) for the purpose of backward compatibility.

#### *mfxU16 TargetBitDepthLuma*

Target encoding bit-depth for luma samples. May differ from source one. 0 mean default target bit-depth which is equal to source (mfxVideoParam::mfx::FrameInfo::BitDepthLuma).

#### *mfxU16 TargetBitDepthChroma*

Target encoding bit-depth for chroma samples. May differ from source one. 0 mean default target bit-depth which is equal to source (mfxVideoParam::mfx::FrameInfo::BitDepthChroma).

#### *mfxU16 BRCPanicMode*

Controls panic mode in AVC and MPEG2 encoders.

#### *mfxU16 LowDelayBRC*

When rate control method is MFX\_RATECONTROL\_VBR, MFX\_RATECONTROL\_QVBR or MFX\_RATECONTROL\_VCM this parameter specifies frame size tolerance. Set this parameter to MFX\_CODINGOPTION\_ON to allow strictly obey average frame size set by MaxKbps, e.g. cases when `MaxFrameSize == (MaxKbps*1000)/(8* FrameRateExtN/FrameRateExtD)`. Also `MaxFrameSizeI` and `MaxFrameSizeP` can be set separately.

#### *mfxU16 EnableMBForceIntra*

Turn ON this option to enable usage of *mfxExtMBForceIntra* for AVC encoder. See the CodingOptionValue enumerator for values of this option. This parameter is valid only during initialization.

#### *mfxU16 AdaptiveMaxFrameSize*

If this option is ON, BRC may decide a larger P or B frame size than what `MaxFrameSizeP` dictates when the scene change is detected. It may benefit the video quality. `AdaptiveMaxFrameSize` feature is not supported with `LowPower` ON or if the value of `MaxFrameSizeP = 0`.

#### *mfxU16 RepartitionCheckEnable*

Controls AVC encoder attempts to predict from small partitions. Default value allows encoder to choose preferred mode, `MFX_CODINGOPTION_ON` forces encoder to favor quality, `MFX_CODINGOPTION_OFF` forces encoder to favor performance.

#### *mfxU16 reserved5[3]*

#### *mfxU16 EncodedUnitsInfo*

Turn this option ON to make encoded units info available in *mfxExtEncodedUnitsInfo*.

#### *mfxU16 EnableNalUnitType*

If this option is turned ON, then HEVC encoder uses NAL unit type provided by application in *mfxEncoderCtrl::MfxNalUnitType* field.

**Note** This parameter is valid only during initialization.

**Note** Not all codecs and SDK implementations support this value. Use Query function to check if this feature is supported.

#### *mfxU16 ExtBrcAdaptiveLTR*

Turn OFF to prevent Adaptive marking of Long Term Reference Frames when using ExtBRC. When ON and using ExtBRC, encoders will mark, modify, or remove LTR frames based on encoding parameters and content properties. The application must set each input frame's *mfxFrameData::FrameOrder* for correct operation of LTR.

#### *mfxU16 reserved[163]*

#### 12.7.4.28.5 mfxExtCodingOptionSPSPPS

##### **struct mfxExtCodingOptionSPSPPS**

Attach this structure as part of the extended buffers to configure the SDK encoder during MFXVideoENCODE\_Init. The sequence or picture parameters specified by this structure overwrite any such parameters specified by the structure or any other extended buffers attached therein.

For H.264, SPSBuffer and PPSBuffer must point to valid bitstreams that contain the sequence parameter set and picture parameter set, respectively. For MPEG-2, SPSBuffer must point to valid bitstreams that contain the sequence header followed by any sequence header extension. The PPSBuffer pointer is ignored. The SDK encoder imports parameters from these buffers. If the encoder does not support the specified parameters, the encoder does not initialize and returns the status code MFX\_ERR\_INCOMPATIBLE\_VIDEO\_PARAM.

Check with the MFXVideoENCODE\_Query function for the support of this multiple segment encoding feature. If this feature is not supported, the query returns MFX\_ERR\_UNSUPPORTED.

#### Public Members

##### *mfxExtBuffer Header*

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_CODING\_OPTION\_SPSPPS.

##### *mfxU8 \*SPSBuffer*

Pointer to a valid bitstream that contains the SPS (sequence parameter set for H.264 or sequence header followed by any sequence header extension for MPEG-2) buffer; can be NULL to skip specifying the SPS.

##### *mfxU8 \*PPSBuffer*

Pointer to a valid bitstream that contains the PPS (picture parameter set for H.264 or picture header followed by any picture header extension for MPEG-2) buffer; can be NULL to skip specifying the PPS.

##### *mfxU16 SPSBufSize*

Size of the SPS in bytes

##### *mfxU16 PPSBufSize*

Size of the PPS in bytes

##### *mfxU16 SPSId*

SPS identifier; the value is reserved and must be zero.

##### *mfxU16 PPSId*

PPS identifier; the value is reserved and must be zero.

#### 12.7.4.28.6 mfxExtInsertHeaders

##### **struct mfxExtInsertHeaders**

Runtime ctrl buffer for SPS/PPS insertion with current encoding frame

## Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_INSERT\_HEADERS.

*mfxU16* **SPS**

tri-state option to insert SPS

*mfxU16* **PPS**

tri-state option to insert PPS

*mfxU16* **reserved[8]**

### 12.7.4.28.7 mfxExtCodingOptionVPS

**struct mfxExtCodingOptionVPS**

Attach this structure as part of the extended buffers to configure the SDK encoder during MFXVideoENCODE\_Init. The sequence or picture parameters specified by this structure overwrite any such parameters specified by the structure or any other extended buffers attached therein.

If the encoder does not support the specified parameters, the encoder does not initialize and returns the status code MFX\_ERR\_INCOMPATIBLE\_VIDEO\_PARAM.

Check with the MFXVideoENCODE\_Query function for the support of this multiple segment encoding feature. If this feature is not supported, the query returns MFX\_ERR\_UNSUPPORTED.

## Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_CODING\_OPTION\_VPS.

*mfxU8* \***VPSBuffer**

Pointer to a valid bitstream that contains the VPS (video parameter set for HEVC) buffer.

*mfxU16* **VPSBufSize**

Size of the VPS in bytes

*mfxU16* **VPSId**

VPS identifier; the value is reserved and must be zero.

### 12.7.4.28.8 mfxExtThreadsParam

**struct mfxExtThreadsParam**

Attached to the *mfxInitParam* structure during the SDK session initialization, *mfxExtThreadsParam* structure specifies options for threads created by this session.

## Public Members

***mfxExtBuffer Header***  
 Must be MFX\_EXTBUFF\_THREADS\_PARAM

***mfxU16 NumThread***  
 The number of threads.

***mfxI32 SchedulingType***  
 Scheduling policy for all threads.

***mfxI32 Priority***  
 Priority for all threads.

***mfxU16 reserved[55]***  
 Reserved for future use

### 12.7.4.28.9 mfxExtVideoSignalInfo

#### struct mfxExtVideoSignalInfo

The *mfxExtVideoSignalInfo* structure defines the video signal information.

For H.264, see Annex E of the ISO/IEC 14496-10 specification for the definition of these parameters.

For MPEG-2, see section 6.3.6 of the ITU\* H.262 specification for the definition of these parameters. The field VideoFullRange is ignored.

For VC-1, see section 6.1.14.5 of the SMPTE\* 421M specification. The fields VideoFormat and VideoFullRange are ignored.

**Note** If ColourDescriptionPresent is zero, the color description information (including ColourPrimaries, TransferCharacteristics, and MatrixCoefficients) will/does not present in the bitstream.

## Public Members

***mfxExtBuffer Header***  
 Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_VIDEO\_SIGNAL\_INFO.

***mfxU16 VideoFormat***

***mfxU16 VideoFullRange***

***mfxU16 ColourDescriptionPresent***

***mfxU16 ColourPrimaries***

***mfxU16 TransferCharacteristics***

***mfxU16 MatrixCoefficients***

### 12.7.4.28.10 mfxExtAVCRefListCtrl

**struct mfxExtAVCRefListCtrl**

The *mfxExtAVCRefListCtrl* structure configures reference frame options for the H.264 encoder. See Reference List Selection and Long-term Reference frame chapters for more details.

**mfxExtAVCRefListCtrl::PreferredRefList** Specify list of frames that should be used to predict the current frame.

**Note** Not all implementations of the SDK encoder support LongTermIdx and ApplyLongTermIdx fields in this structure. The application has to use query mode 1 to determine if such functionality is supported. To do so, the application has to attach this extended buffer to *mfxVideoParam* structure and call MFXVideoENCODE\_Query function. If function returns MFX\_ERR\_NONE and these fields were set to one, then such functionality is supported. If function fails or sets fields to zero then this functionality is not supported.

**mfxExtAVCRefListCtrl::RejectedRefList** Specify list of frames that should not be used for prediction.

**mfxExtAVCRefListCtrl::LongTermRefList** Specify list of frames that should be marked as long-term reference frame.

#### Public Members

##### *mfxExtBuffer* Header

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_AVC\_REFLIST\_CTRL.

##### *mfxU16* NumRefIdxL0Active

Specify the number of reference frames in the active reference list L0. This number should be less or equal to the NumRefFrame parameter from encoding initialization.

##### *mfxU16* NumRefIdxL1Active

Specify the number of reference frames in the active reference list L1. This number should be less or equal to the NumRefFrame parameter from encoding initialization.

##### *mfxU32* FrameOrder

Together FrameOrder and PicStruct fields are used to identify reference picture. Use FrameOrder = MFX\_FRAMEORDER\_UNKNOWN to mark unused entry.

##### *mfxU16* PicStruct

Together FrameOrder and PicStruct fields are used to identify reference picture. Use FrameOrder = MFX\_FRAMEORDER\_UNKNOWN to mark unused entry.

##### *mfxU16* ViewId

Reserved and must be zero.

##### *mfxU16* LongTermIdx

Index that should be used by the SDK encoder to mark long-term reference frame.

##### *mfxU16* reserved[3]

Reserved

**struct mfxExtAVCRefListCtrl::[anonymous] PreferredRefList[32]**

**struct mfxExtAVCRefListCtrl::[anonymous] RejectedRefList[16]**

**struct mfxExtAVCRefListCtrl::[anonymous] LongTermRefList[16]**

##### *mfxU16* ApplyLongTermIdx

If it is equal to zero, the SDK encoder assigns long-term index according to internal algorithm. If it is equal to one, the SDK encoder uses LongTermIdx value as long-term index.

#### 12.7.4.28.11 mfxExtMasteringDisplayColourVolume

**struct mfxExtMasteringDisplayColourVolume**

The *mfxExtMasteringDisplayColourVolume* configures the HDR SEI message. If application attaches this structure to the *mfxEncodeCtrl* at runtime, the encoder inserts the HDR SEI message for current frame and ignores InsertPayloadToggle. If application attaches this structure to the *mfxVideoParam* during initialization or reset, the encoder inserts HDR SEI message based on InsertPayloadToggle. Fields semantic defined in ITU-T\* H.265 Annex D.

##### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_MASTERING\_DISPLAY\_COLOUR\_VOLUME.

*mfxU16* **InsertPayloadToggle**

InsertHDRPayload enumerator value.

*mfxU16* **DisplayPrimariesX[3]**

Color primaries for a video source in increments of 0.00002. Consist of RGB x coordinates and define how to convert colors from RGB color space to CIE XYZ color space. These fields belong to the [0..50000] range.

*mfxU16* **DisplayPrimariesY[3]**

Color primaries for a video source in increments of 0.00002. Consist of RGB y coordinates and define how to convert colors from RGB color space to CIE XYZ color space. These fields belong to the [0..50000] range.

*mfxU16* **WhitePointX**

White point X coordinate.

*mfxU16* **WhitePointY**

White point Y coordinate.

*mfxU32* **MaxDisplayMasteringLuminance**

Specify maximum luminance of the display on which the content was authored in units of 0.00001 candelas per square meter. These fields belong to the [1..65535] range.

*mfxU32* **MinDisplayMasteringLuminance**

Specify minimum luminance of the display on which the content was authored in units of 0.00001 candelas per square meter. These fields belong to the [1..65535] range.

#### 12.7.4.28.12 mfxExtContentLightLevelInfo

**struct mfxExtContentLightLevelInfo**

The *mfxExtContentLightLevelInfo* structure configures the HDR SEI message. If application attaches this structure to the *mfxEncodeCtrl* structure at runtime, the encoder inserts the HDR SEI message for current frame and ignores InsertPayloadToggle. If application attaches this structure to the *mfxVideoParam* structure during initialization or reset, the encoder inserts HDR SEI message based on InsertPayloadToggle. Fields semantic defined in ITU-T\* H.265 Annex D.

## Public Members

*mfxExtBuffer Header*

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_CONTENT\_LIGHT\_LEVEL\_INFO.

*mfxU16 InsertPayloadToggle*

InsertHDRPayload enumerator value.

*mfxU16 MaxContentLightLevel*

Maximum luminance level of the content. The field belongs to the [1..65535] range.

*mfxU16 MaxPicAverageLightLevel*

Maximum average per-frame luminance level of the content. The field belongs to the [1..65535] range.

### 12.7.4.28.13 mfxExtPictureTimingSEI

**struct mfxExtPictureTimingSEI**

The *mfxExtPictureTimingSEI* structure configures the H.264 picture timing SEI message. The encoder ignores it if HRD information in stream is absent and PicTimingSEI option in *mfxExtCodingOption* structure is turned off. See *mfxExtCodingOption* for details.

If the application attaches this structure to the *mfxVideoParam* structure during initialization, the encoder inserts the picture timing SEI message based on provided template in every access unit of coded bitstream.

If application attaches this structure to the *mfxEncodeCtrl* structure at runtime, the encoder inserts the picture timing SEI message based on provided template in access unit that represents current frame.

These parameters define the picture timing information. An invalid value of 0xFFFF indicates that application does not set the value and encoder must calculate it.

See Annex D of the ISO\*VIEC\* 14496-10 specification for the definition of these parameters.

## Public Members

*mfxExtBuffer Header*

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_PICTURE\_TIMING\_SEI.

*mfxU32 reserved[14]*

*mfxU16 ClockTimestampFlag*

*mfxU16 CtType*

*mfxU16 NuitFieldBasedFlag*

*mfxU16 CountingType*

*mfxU16 FullTimestampFlag*

*mfxU16 DiscontinuityFlag*

*mfxU16 CntDroppedFlag*

*mfxU16 NFrames*

*mfxU16 SecondsFlag*

*mfxU16 MinutesFlag*

*mfxU16 HoursFlag*

```

mfxU16 SecondsValue
mfxU16 MinutesValue
mfxU16 HoursValue
mfxU32 TimeOffset
struct mfxExtPictureTimingSEI::[anonymous] TimeStamp[3]

```

#### 12.7.4.28.14 mfxExtAvcTemporalLayers

##### **struct mfxExtAvcTemporalLayers**

The *mfxExtAvcTemporalLayers* structure configures the H.264 temporal layers hierarchy. If application attaches it to the *mfxVideoParam* structure during initialization, the SDK encoder generates the temporal layers and inserts the prefix NAL unit before each slice to indicate the temporal and priority IDs of the layer.

This structure can be used with the display-order encoding mode only.

##### Public Members

###### *mfxExtBuffer* Header

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_AVC\_TEMPORAL\_LAYERS.

###### *mfxU16* BaseLayerPID

The priority ID of the base layer; the SDK encoder increases the ID for each temporal layer and writes to the prefix NAL unit.

###### *mfxU16* Scale

The ratio between the frame rates of the current temporal layer and the base layer.

#### 12.7.4.28.15 mfxExtEncoderCapability

##### **struct mfxExtEncoderCapability**

The *mfxExtEncoderCapability* structure is used to retrieve SDK encoder capability. See description of mode 4 of the MFXVideoENCODE\_Query function for details how to use this structure.

**Note** Not all implementations of the SDK encoder support this extended buffer. The application has to use query mode 1 to determine if such functionality is supported. To do so, the application has to attach this extended buffer to *mfxVideoParam* structure and call MFXVideoENCODE\_Query function. If function returns MFX\_ERR\_NONE then such functionality is supported.

##### Public Members

###### *mfxExtBuffer* Header

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_ENCODER\_CAPABILITY.

###### *mfxU32* MBPerSec

Specify the maximum processing rate in macro blocks per second.

#### 12.7.4.28.16 mfxExtEncoderResetOption

**struct mfxExtEncoderResetOption**

The *mfxExtEncoderResetOption* structure is used to control the SDK encoder behavior during reset. By using this structure, the application instructs the SDK encoder to start new coded sequence after reset or continue encoding of current sequence.

This structure is also used in mode 3 of MFXVideoENCODE\_Query function to check for reset outcome before actual reset. The application should set StartNewSequence to required behavior and call query function. If query fails, see status codes below, then such reset is not possible in current encoder state. If the application sets StartNewSequence to MFX\_CODINGOPTION\_UNKNOWN then query function replaces it by actual reset type: MFX\_CODINGOPTION\_ON if the SDK encoder will begin new sequence after reset or MFX\_CODINGOPTION\_OFF if the SDK encoder will continue current sequence.

Using this structure may cause next status codes from MFXVideoENCODE\_Reset and MFXVideoENCODE\_Query functions:

- MFX\_ERR\_INVALID\_VIDEO\_PARAM - if such reset is not possible. For example, the application sets StartNewSequence to off and requests resolution change.
- MFX\_ERR\_INCOMPATIBLE\_VIDEO\_PARAM - if the application requests change that leads to memory allocation. For example, the application set StartNewSequence to on and requests resolution change to bigger than initialization value.
- MFX\_ERR\_NONE - if such reset is possible.

There is limited list of parameters that can be changed without starting a new coded sequence:

- Bitrate parameters, TargetKbps and MaxKbps in the *mfxInfoMFX* structure.
- Number of slices, NumSlice in the *mfxInfoMFX* structure. Number of slices should be equal or less than number of slices during initialization.
- Number of temporal layers in *mfxExtAvcTemporalLayers* structure. Reset should be called immediately before encoding of frame from base layer and number of reference frames should be big enough for new temporal layers structure.
- Quantization parameters, QPI, QPP and QPB in the *mfxInfoMFX* structure.

As it is described in Configuration Change chapter, the application should retrieve all cached frames before calling reset. When query function checks for reset outcome, it expects that this requirement be satisfied. If it is not true and there are some cached frames inside the SDK encoder, then query result may differ from reset one, because the SDK encoder may insert IDR frame to produce valid coded sequence.

See also Appendix ‘Streaming and Video Conferencing Features’.

**Note** Not all implementations of the SDK encoder support this extended buffer. The application has to use query mode 1 to determine if such functionality is supported. To do so, the application has to attach this extended buffer to *mfxVideoParam* structure and call MFXVideoENCODE\_Query function. If function returns MFX\_ERR\_NONE then such functionality is supported.

## Public Members

### *mfxExtBuffer* Header

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_ENCODER\_RESET\_OPTION.

### *mfxU16* StartNewSequence

Instructs encoder to start new sequence after reset. It is one of the CodingOptionValue options:

MFX\_CODINGOPTION\_ON – the SDK encoder completely reset internal state and begins new coded sequence after reset, including insertion of IDR frame, sequence and picture headers.

MFX\_CODINGOPTION\_OFF – the SDK encoder continues encoding of current coded sequence after reset, without insertion of IDR frame.

MFX\_CODINGOPTION\_UNKNOWN – depending on the current encoder state and changes in configuration parameters the SDK encoder may or may not start new coded sequence. This value is also used to query reset outcome.

## 12.7.4.28.17 *mfxExtAVCEncodedFrameInfo*

### **struct mfxExtAVCEncodedFrameInfo**

The *mfxExtAVCEncodedFrameInfo* is used by the SDK encoder to report additional information about encoded picture. The application can attach this buffer to the *mfxBitstream* structure before calling MFXVideoENCODE\_EncodeFrameAsync function. For interlaced content the SDK encoder requires two such structures. They correspond to fields in encoded order.

**Note** Not all implementations of the SDK encoder support this extended buffer. The application has to use query mode 1 to determine if such functionality is supported. To do so, the application has to attach this extended buffer to *mfxVideoParam* structure and call MFXVideoENCODE\_Query function. If function returns MFX\_ERR\_NONE then such functionality is supported.

## Public Members

### *mfxExtBuffer* Header

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_ENCODED\_FRAME\_INFO.

### *mfxU32* FrameOrder

Frame order of encoded picture.

Frame order of reference picture.

### *mfxU16* PicStruct

Picture structure of encoded picture.

Picture structure of reference picture.

### *mfxU16* LongTermIdx

Long term index of encoded picture if applicable.

Long term index of reference picture if applicable.

### *mfxU32* MAD

Mean Absolute Difference between original pixels of the frame and motion compensated (for inter macroblocks) or spatially predicted (for intra macroblocks) pixels. Only luma component, Y plane, is used in calculation.

***mfxU16 BRCPanicMode***

Bitrate control was not able to allocate enough bits for this frame. Frame quality may be unacceptably low.

***mfxU16 QP***

Luma QP.

***mfxU32 SecondFieldOffset***

Offset to second field. Second field starts at *mfxBitstream::Data* + *mfxBitstream::DataOffset* + *mfxExtAVCEncodedFrameInfo::SecondFieldOffset*.

***struct mfxExtAVCEncodedFrameInfo::[anonymous] UsedRefListL1[32]***

Reference lists that have been used to encode picture.

#### 12.7.4.28.18 ***mfxExtEncoderROI***

***struct mfxExtEncoderROI***

The *mfxExtEncoderROI* structure is used by the application to specify different Region Of Interests during encoding. It may be used at initialization or at runtime.

##### Public Members

***mfxExtBuffer Header***

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_ENCODER\_ROI.

***mfxU16 NumROI***

Number of ROI descriptions in array. The Query function mode 2 returns maximum supported value (set it to 256 and Query will update it to maximum supported value).

***mfxU16 ROIMode***

QP adjustment mode for ROIs. Defines if Priority or DeltaQP is used during encoding.

***mfxU32 Left***

Left ROI's coordinate.

***mfxU32 Top***

Top ROI's coordinate.

***mfxU32 Right***

Right ROI's coordinate.

***mfxU32 Bottom***

Bottom ROI's coordinate.

***mfxI16 DeltaQP***

Delta QP of ROI. Used if ROI Mode = MFX\_ROI\_MODE\_QP\_DELTA. This is absolute value in the -51...51 range, which will be added to the MB QP. Lesser value produces better quality.

***struct mfxExtEncoderROI::[anonymous] ROI[256]***

ROI location rectangle. ROI rectangle definition is using end-point exclusive notation. In other words, the pixel with (Right, Bottom) coordinates lies immediately outside of the ROI. Left, Top, Right, Bottom should be aligned by codec-specific block boundaries (should be dividable by 16 for AVC, or by 32 for HEVC). Every ROI with unaligned coordinates will be expanded by SDK to minimal-area block-aligned ROI, enclosing the original one. For example (5, 5, 15, 31) ROI will be expanded to (0, 0, 16, 32) for AVC encoder, or to (0, 0, 32, 32) for HEVC. Array of ROIs. Different ROI may overlap each other. If macroblock belongs to several ROI, Priority from ROI with lowest index is used.

### 12.7.4.28.19 mfxExtEncoderIPCMArea

```
struct mfxExtEncoderIPCMArea
```

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_ENCODER\_IPCM\_AREA.

*mfxU32* **Left**

Left Area's coordinate.

*mfxU32* **Top**

Top Area's coordinate.

*mfxU32* **Right**

Right Area's coordinate.

*mfxU32* **Bottom**

Bottom Area's coordinate.

**struct mfxExtEncoderIPCMArea::[anonymous] Area[64]**

Number of Area's

### 12.7.4.28.20 mfxExtAVCRefLists

```
struct mfxExtAVCRefLists
```

The *mfxExtAVCRefLists* structure specifies reference lists for the SDK encoder. It may be used together with the *mfxExtAVCRefListCtrl* structure to create customized reference lists. If both structures are used together, then the SDK encoder takes reference lists from *mfxExtAVCRefLists* structure and modifies them according to the *mfxExtAVCRefListCtrl* instructions. In case of interlaced coding, the first *mfxExtAVCRefLists* structure affects TOP field and the second – BOTTOM field.

**Note** Not all implementations of the SDK encoder support this structure. The application has to use query function to determine if it is supported

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_AVC\_REFLISTS.

*mfxU16* **NumRefIdxL0Active**

Specify the number of reference frames in the active reference list L0. This number should be less or equal to the NumRefFrame parameter from encoding initialization.

*mfxU16* **NumRefIdxL1Active**

Specify the number of reference frames in the active reference list L1. This number should be less or equal to the NumRefFrame parameter from encoding initialization.

**struct mfxExtAVCRefLists::mfxRefPic RefPicList1[32]**

Specify L0 and L1 reference lists.

**struct mfxRefPic**

## Public Members

### *mfxU32* **FrameOrder**

Together these fields are used to identify reference picture. Use FrameOrder = MFX\_FRAMEORDER\_UNKNOWN to mark unused entry. Use PicStruct = MFX\_PICSTRUCT\_FIELD\_TFF for TOP field, PicStruct = MFX\_PICSTRUCT\_FIELD\_BFF for BOTTOM field.

### *mfxU16* **PicStruct**

Together these fields are used to identify reference picture. Use FrameOrder = MFX\_FRAMEORDER\_UNKNOWN to mark unused entry. Use PicStruct = MFX\_PICSTRUCT\_FIELD\_TFF for TOP field, PicStruct = MFX\_PICSTRUCT\_FIELD\_BFF for BOTTOM field.

## 12.7.4.28.21 mfxExtChromaLocInfo

### **struct mfxExtChromaLocInfo**

The *mfxExtChromaLocInfo* structure defines the location of chroma samples information.

Members of this structure define the location of chroma samples information.

See Annex E of the ISO\*VIEC\* 14496-10 specification for the definition of these parameters.

## Public Members

### *mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_CHROMA\_LOC\_INFO.

### *mfxU16* **ChromaLocInfoPresentFlag**

### *mfxU16* **ChromaSampleLocTypeTopField**

### *mfxU16* **ChromaSampleLocTypeBottomField**

### *mfxU16* **reserved[9]**

## 12.7.4.28.22 mfxExtMBForceIntra

### **struct mfxExtMBForceIntra**

The *mfxExtMBForceIntra* structure specifies macroblock map for current frame which forces specified macroblocks to be encoded as Intra if *mfxExtCodingOption3::EnableMBForceIntra* was turned ON during encoder initialization. The application can attach this extended buffer to the *mfxEncodeCtrl* during runtime.

## Public Members

### *mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_MB\_FORCE\_INTRAS.

### *mfxU32* **MapSize**

Macroblock map size.

### *mfxU8* \***Map**

Pointer to a list of force intra macroblock flags in raster scan order. Each flag is one byte in map. Set flag to 1 to force corresponding macroblock to be encoded as intra. In case of interlaced encoding, the first half of map affects top field and the second – bottom field.

### 12.7.4.28.23 mfxExtMBQP

**struct mfxExtMBQP**

The *mfxExtMBQP* structure specifies per-macroblock QP for current frame if *mfxExtCodingOption3::EnableMBQP* was turned ON during encoder initialization. The application can attach this extended buffer to the *mfxEncodeCtrl* during runtime.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_MBQP.

*mfxU16* **Mode**

Defines QP update mode. See MBQPMode enumerator for more details.

*mfxU16* **BlockSize**

QP block size, valid for HEVC only during Init and Runtime.

*mfxU32* **NumQPAalloc**

Size of allocated by application QP or DeltaQP array.

*mfxU8* \***QP**

Pointer to a list of per-macroblock QP in raster scan order. In case of interlaced encoding the first half of QP array affects top field and the second – bottom field. Valid when Mode = MFX\_MBQP\_MODE\_QP\_VALUE

For AVC valid range is 1..51.

For HEVC valid range is 1..51. Application's provided QP values should be valid; otherwise invalid QP values may cause undefined behavior. MBQP map should be aligned for 16x16 block size. (align rule is (width +15 /16) && (height +15 /16))

For MPEG2 QP corresponds to quantizer\_scale of the ISO\*VIEC\* 13818-2 specification and have valid range 1..112.

*mfxI8* \***DeltaQP**

Pointer to a list of per-macroblock QP deltas in raster scan order. For block i: QP[i] = BrcQP[i] + DeltaQP[i]. Valid when Mode = MFX\_MBQP\_MODE\_QP\_DELTA.

*mfxQPandMode* \***QPmode**

Block-granularity modes when MFX\_MBQP\_MODE\_QP\_ADAPTIVE is set.

### 12.7.4.28.24 mfxExtHEVCTiles

**struct mfxExtHEVCTiles**

The *mfxExtHEVCTiles* structure configures tiles options for the HEVC encoder. The application can attach this extended buffer to the *mfxVideoParam* structure to configure initialization.

## Public Members

### *mfxExtBuffer Header*

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_HEVC\_TILES.

### *mfxU16 NumTileRows*

Number of tile rows.

### *mfxU16 NumTileColumns*

Number of tile columns.

## 12.7.4.28.25 mfxExtMBDisableSkipMap

### **struct mfxExtMBDisableSkipMap**

The *mfxExtMBDisableSkipMap* structure specifies macroblock map for current frame which forces specified macroblocks to be non skip if *mfxExtCodingOption3::MBDisableSkipMap* was turned ON during encoder initialization. The application can attach this extended buffer to the *mfxEncodeCtrl* during runtime.

## Public Members

### *mfxExtBuffer Header*

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_MB\_DISABLE\_SKIP\_MAP.

### *mfxU32 MapSize*

Macroblock map size.

### *mfxU8 \*Map*

Pointer to a list of non-skip macroblock flags in raster scan order. Each flag is one byte in map. Set flag to 1 to force corresponding macroblock to be non-skip. In case of interlaced encoding the first half of map affects top field and the second – bottom field.

## 12.7.4.28.26 mfxExtHEVCPParam

### **struct mfxExtHEVCPParam**

## Public Members

### *mfxExtBuffer Header*

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_HEVC\_PARAM.

### *mfxU16 PicWidthInLumaSamples*

Specifies the width of each coded picture in units of luma samples.

### *mfxU16 PicHeightInLumaSamples*

Specifies the height of each coded picture in units of luma samples.

### *mfxU64 GeneralConstraintFlags*

Additional flags to specify exact profile/constraints. See the GeneralConstraintFlags enumerator for values of this field.

### *mfxU16 SampleAdaptiveOffset*

Controls SampleAdaptiveOffset encoding feature. See enum SampleAdaptiveOffset for supported values (bit-ORed). Valid during encoder Init and Runtime.

***mfxU16 LCUSize***

Specifies largest coding unit size (max luma coding block). Valid during encoder Init.

**12.7.4.28.27 mfxExtDecodeErrorReport****struct mfxExtDecodeErrorReport**

This structure is used by the SDK decoders to report bitstream error information right after DecodeHeader or DecodeFrameAsync. The application can attach this extended buffer to the *mfxBitstream* structure at runtime.

**Public Members*****mfxExtBuffer Header***

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_DECODE\_ERROR\_REPORT.

***mfxU32 ErrorTypes***

Bitstream error types (bit-ORed values). See ErrorTypes enumerator for the list of possible types.

**12.7.4.28.28 mfxExtDecodedFrameInfo****struct mfxExtDecodedFrameInfo**

This structure is used by the SDK decoders to report additional information about decoded frame. The application can attach this extended buffer to the mfxFrameSurface1::mfxFrameData structure at runtime.

**Public Members*****mfxExtBuffer Header***

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_DECODED\_FRAME\_INFO.

***mfxU16 FrameType***

Frame type. See FrameType enumerator for the list of possible types.

**12.7.4.28.29 mfxExtTimeCode****struct mfxExtTimeCode**

This structure is used by the SDK to pass MPEG 2 specific timing information.

See ISO/IEC 13818-2 and ITU-T H.262, MPEG-2 Part 2 for the definition of these parameters.

**Public Members*****mfxExtBuffer Header***

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_TIME\_CODE.

***mfxU16 DropFrameFlag***

Indicated dropped frame.

***mfxU16 TimeCodeHours***

Hours.

*mfxU16 TimeCodeMinutes*

Minutes.

*mfxU16 TimeCodeSeconds*

Seconds.

*mfxU16 TimeCodePictures*

Pictures.

#### 12.7.4.28.30 mfxExtHEVCRegion

**struct mfxExtHEVCRegion**

Attached to the *mfxVideoParam* structure during HEVC encoder initialization, specifies the region to encode.

##### Public Members

*mfxExtBuffer Header*

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_HEVC\_REGION.

*mfxU32 RegionId*

ID of region.

*mfxU16 RegionType*

Type of region. See HEVCRegionType enumerator for the list of possible types.

*mfxU16 RegionEncoding*

Set to MFX\_HEVC\_REGION\_ENCODING\_ON to encode only specified region.

#### 12.7.4.28.31 mfxExtPredWeightTable

**struct mfxExtPredWeightTable**

When *mfxExtCodingOption3::WeightedPred* was set to explicit during encoder Init or Reset and the current frame is P-frame or *mfxExtCodingOption3::WeightedBiPred* was set to explicit during encoder Init or Reset and the current frame is B-frame, attached to *mfxEncodeCtrl*, this structure specifies weighted prediction table for current frame.

##### Public Members

*mfxExtBuffer Header*

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_PRED\_WEIGHT\_TABLE.

*mfxU16 LumaLog2WeightDenom*

Base 2 logarithm of the denominator for all luma weighting factors. Value shall be in the range of 0 to 7, inclusive.

*mfxU16 ChromaLog2WeightDenom*

Base 2 logarithm of the denominator for all chroma weighting factors. Value shall be in the range of 0 to 7, inclusive.

*mfxU16 LumaWeightFlag[2][32]*

LumaWeightFlag[L][R] equal to 1 specifies that the weighting factors for the luma component are specified for R's entry of RefPicList L.

***mfxU16 ChromaWeightFlag[2][32]***

ChromaWeightFlag[L][R] equal to 1 specifies that the weighting factors for the chroma component are specified for R's entry of RefPicList L.

***mfxI16 Weights[2][32][3][2]***

The values of the weights and offsets used in the encoding processing. The value of Weights[i][j][k][m] is interpreted as: i refers to reference picture list 0 or 1; j refers to reference list entry 0-31; k refers to data for the luma component when it is 0, the Cb chroma component when it is 1 and the Cr chroma component when it is 2; m refers to weight when it is 0 and offset when it is 1

#### 12.7.4.28.32 **mfxExtAVCRoundingOffset**

**struct mfxExtAVCRoundingOffset**

This structure is used by the SDK encoders to set rounding offset parameters for quantization. It is per-frame based encoding control, and can be attached to some frames and skipped for others. When the extension buffer is set the application can attach it to the *mfxEncodeCtrl* during runtime.

**Public Members**
***mfxExtBuffer Header***

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_AVC\_ROUNDING\_OFFSET.

***mfxU16 EnableRoundingIntra***

Enable rounding offset for intra blocks. See the CodingOptionValue enumerator for values of this option.

***mfxU16 RoundingOffsetIntra***

Intra rounding offset. Value shall be in the range of 0 to 7, inclusive.

***mfxU16 EnableRoundingInter***

Enable rounding offset for inter blocks. See the CodingOptionValue enumerator for values of this option.

***mfxU16 RoundingOffsetInter***

Inter rounding offset. Value shall be in the range of 0 to 7, inclusive.

#### 12.7.4.28.33 **mfxExtDirtyRect**

**struct mfxExtDirtyRect**

Used by the application to specify dirty regions within a frame during encoding. It may be used at initialization or at runtime.

Dirty rectangle definition is using end-point exclusive notation. In other words, the pixel with (Right, Bottom) coordinates lies immediately outside of the Dirty rectangle. Left, Top, Right, Bottom should be aligned by codec-specific block boundaries (should be dividable by 16 for AVC, or by block size (8, 16, 32 or 64, depends on platform) for HEVC). Every Dirty rectangle with unaligned coordinates will be expanded by SDK to minimal-area block-aligned Dirty rectangle, enclosing the original one. For example (5, 5, 15, 31) Dirty rectangle will be expanded to (0, 0, 16, 32) for AVC encoder, or to (0, 0, 32, 32) for HEVC, if block size is 32. Dirty rectangle (0, 0, 0, 0) is a valid dirty rectangle and means that frame is not changed.

## Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_DIRTY\_RECTANGLES.

*mfxU16* **NumRect**

Number of dirty rectangles.

*mfxU32* **Left**

Dirty region coordinate.

*mfxU32* **Top**

Dirty region coordinate.

*mfxU32* **Right**

Dirty region coordinate.

*mfxU32* **Bottom**

Dirty region coordinate.

**struct** *mfxExtDirtyRect*::[anonymous] **Rect**[256]

Array of dirty rectangles.

### 12.7.4.28.34 mfxExtMoveRect

**struct** *mfxExtMoveRect*

Used by the application to specify moving regions within a frame during encoding.

Destination rectangle location should be aligned to MB boundaries (should be dividable by 16). If not, the SDK encoder truncates it to MB boundaries, for example, both 17 and 31 will be truncated to 16.

## Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_MOVING\_RECTANGLE.

*mfxU16* **NumRect**

Number of moving rectangles.

*mfxU32* **DestLeft**

Destination rectangle location.

*mfxU32* **DestTop**

Destination rectangle location.

*mfxU32* **DestRight**

Destination rectangle location.

*mfxU32* **DestBottom**

Destination rectangle location.

*mfxU32* **SourceLeft**

Source rectangle location.

*mfxU32* **SourceTop**

Source rectangle location.

**struct** *mfxExtMoveRect*::[anonymous] **Rect**[256]

Array of moving rectangles.

#### 12.7.4.28.35 mfxExtMVOverPicBoundaries

**struct mfxExtMVOverPicBoundaries**

Attached to the [mfxVideoParam](#) structure instructs encoder to use or not use samples over specified picture border for inter prediction.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_MV\_OVER\_PIC\_BOUNDARIES.

*mfxU16* **StickTop**

When set to OFF, one or more samples outside corresponding picture boundary may be used in inter prediction. See the CodingOptionValue enumerator for values of this option.

*mfxU16* **StickBottom**

When set to OFF, one or more samples outside corresponding picture boundary may be used in inter prediction. See the CodingOptionValue enumerator for values of this option.

*mfxU16* **StickLeft**

When set to OFF, one or more samples outside corresponding picture boundary may be used in inter prediction. See the CodingOptionValue enumerator for values of this option.

*mfxU16* **StickRight**

When set to OFF, one or more samples outside corresponding picture boundary may be used in inter prediction. See the CodingOptionValue enumerator for values of this option.

#### 12.7.4.28.36 mfxVP9SegmentParam

**struct mfxVP9SegmentParam**

The [mfxVP9SegmentParam](#) structure contains features and parameters for the segment.

#### Public Members

*mfxU16* **FeatureEnabled**

Indicates which features are enabled for the segment. See SegmentFeature enumerator for values for this option. Values from the enumerator can be bit-OR'ed. Support of particular feature depends on underlying HW platform. Application can check which features are supported by calling Query.

*mfxI16* **QIndexDelta**

Quantization index delta for the segment. Ignored if MFX\_VP9\_SEGMENT\_FEATURE\_QINDEX isn't set in FeatureEnabled. Valid range for this parameter is [-255, 255]. If QIndexDelta is out of this range, it will be ignored. If QIndexDelta is within valid range, but sum of base quantization index and QIndexDelta is out of [0, 255], QIndexDelta will be clamped.

*mfxI16* **LoopFilterLevelDelta**

Loop filter level delta for the segment. Ignored if MFX\_VP9\_SEGMENT\_FEATURE\_LOOP\_FILTER isn't set in FeatureEnabled. Valid range for this parameter is [-63, 63]. If LoopFilterLevelDelta is out of this range, it will be ignored. If LoopFilterLevelDelta is within valid range, but sum of base loop filter level and LoopFilterLevelDelta is out of [0, 63], LoopFilterLevelDelta will be clamped.

*mfxU16* **ReferenceFrame**

Reference frame for the segment. See VP9ReferenceFrame enumerator for values for this option. Ignored if MFX\_VP9\_SEGMENT\_FEATURE\_REFERENCE isn't set in FeatureEnabled.

#### 12.7.4.28.37 mfxExtVP9Segmentation

##### **struct mfxExtVP9Segmentation**

In VP9 encoder it's possible to divide a frame to up to 8 segments and apply particular features (like delta for quantization index or for loop filter level) on segment basis. “Uncompressed header” of every frame indicates if segmentation is enabled for current frame, and (if segmentation enabled) contains full information about features applied to every segment. Every “Mode info block” of coded frame has segment\_id in the range [0, 7].

To enable Segmentation *mfxExtVP9Segmentation* structure with correct settings should be passed to the encoder. It can be attached to the *mfxVideoParam* structure during initialization or MFXVideoENCODE\_Reset call (static configuration). If *mfxExtVP9Segmentation* buffer isn't attached during initialization, segmentation is disabled for static configuration. If the buffer isn't attached for Reset call, encoder continues to use static configuration for segmentation which was actual before this Reset call. If *mfxExtVP9Segmentation* buffer with NumSegments=0 is provided during initialization or Reset call, segmentation becomes disabled for static configuration.

Also the buffer can be attached to the *mfxEncodeCtrl* structure during runtime (dynamic configuration). Dynamic configuration is applied to current frame only (after encoding of current frame SDK Encoder will switch to next dynamic configuration, or to static configuration if dynamic isn't provided for next frame).

The SegmentIdBlockSize, NumSegmentIdAlloc, SegmentId parameters represent segmentation map. Here, segmentation map is array of segment\_ids (one byte per segment\_id) for blocks of size NxN in raster scan order. Size NxN is specified by application and is constant for whole frame. If *mfxExtVP9Segmentation* is attached during initialization and/or during runtime, all three parameters should be set to proper values not conflicting with each other and with NumSegments. If any of them not set, or any conflict/error in these parameters detected by SDK, segmentation map discarded.

#### Public Members

##### *mfxExtBuffer* Header

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_VP9\_SEGMENTATION.

##### *mfxU16* NumSegments

Number of segments for frame. Value 0 means that segmentation is disabled. Sending of 0 for particular frame will disable segmentation for this frame only. Sending of 0 to Reset function will disable segmentation permanently (can be enabled again by subsequent Reset call).

##### *mfxVP9SegmentParam* Segment[8]

Array of structures *mfxVP9SegmentParam* containing features and parameters for every segment. Entries with indexes bigger than NumSegments-1 are ignored. See the *mfxVP9SegmentParam* structure for definitions of segment features and their parameters.

##### *mfxU16* SegmentIdBlockSize

Size of block (NxN) for segmentation map. See SegmentIdBlockSize enumerator for values for this option. Encoded block which is bigger than SegmentIdBlockSize uses segment\_id taken from its top-left sub-block from segmentation map. Application can check if particular block size is supported by calling of Query.

##### *mfxU32* NumSegmentIdAlloc

Size of buffer allocated for segmentation map (in bytes). Application must assure that NumSegmentIdAlloc is enough to cover frame resolution with blocks of size SegmentIdBlockSize. Otherwise segmentation map will be discarded.

##### *mfxU8* \*SegmentId

Pointer to segmentation map buffer which holds array of segment\_ids in raster scan order. Application is responsible for allocation and release of this memory. Buffer pointed by SegmentId provided during initialization or Reset call should be considered in use until another SegmentId is provided via Reset call (if any), or until call of MFXVideoENCODE\_Close. Buffer pointed by SegmentId provided with

*mfxEncodeCtrl* should be considered in use while input surface is locked by SDK. Every segment\_id in the map should be in the range of [0, NumSegments-1]. If some segment\_id is out of valid range, segmentation map cannot be applied. If buffer *mfxExtVP9Segmentation* is attached to *mfxEncodeCtrl* in runtime, SegmentId can be zero. In this case segmentation map from static configuration will be used.

#### 12.7.4.28.38 mfxVP9TemporalLayer

##### **struct mfxVP9TemporalLayer**

The *mfxVP9TemporalLayer* structure specifies temporal layer.

##### **Public Members**

###### *mfxU16 FrameRateScale*

The ratio between the frame rates of the current temporal layer and the base layer. The SDK treats particular temporal layer as “defined” if it has FrameRateScale > 0. If base layer defined, it must have FrameRateScale equal to 1. FrameRateScale of each next layer (if defined) must be multiple of and greater than FrameRateScale of previous layer.

###### *mfxU16 TargetKbps*

Target bitrate for current temporal layer (ignored if RateControlMethod is CQP). If RateControlMethod is not CQP, application must provide TargetKbps for every defined temporal layer. TargetKbps of each next layer (if defined) must be greater than TargetKbps of previous layer.

#### 12.7.4.28.39 mfxExtVP9TemporalLayers

##### **struct mfxExtVP9TemporalLayers**

The SDK allows to encode VP9 bitstream that contains several subset bitstreams that differ in frame rates also called “temporal layers”. On decoder side each temporal layer can be extracted from coded stream and decoded separately. The *mfxExtVP9TemporalLayers* structure configures the temporal layers for SDK VP9 encoder. It can be attached to the *mfxVideoParam* structure during initialization or MFXVideoENCODE\_Reset call. If *mfxExtVP9TemporalLayers* buffer isn’t attached during initialization, temporal scalability is disabled. If the buffer isn’t attached for Reset call, encoder continues to use temporal scalability configuration which was actual before this Reset call. In SDK API temporal layers are ordered by their frame rates in ascending order. Temporal layer 0 (having lowest frame rate) is called base layer. Each next temporal layer includes all previous layers. Temporal scalability feature has requirements for minimum number of allocated reference frames (controlled by SDK API parameter NumRefFrame). If NumRefFrame set by application isn’t enough to build reference structure for requested number of temporal layers, the SDK corrects NumRefFrame. Temporal layer structure is reset (re-started) after key-frames.

##### **Public Members**

###### *mfxExtBuffer Header*

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_VP9\_TEMPORAL\_LAYERS.

###### *mfxVP9TemporalLayer Layer[8]*

The array of temporal layers. Layer[0] specifies base layer. The SDK reads layers from the array while they are defined (have FrameRateScale>0). All layers starting from first layer with FrameRateScale=0 are ignored. Last layer which is not ignored is “highest layer”. Highest layer has frame rate specified in *mfxVideoParam*. Frame rates of lower layers are calculated using their FrameRateScale. TargetKbps of highest layer should be equal to TargetKbps specified in *mfxVideoParam*. If it’s not true, TargetKbps of highest temporal layers has priority. If there are no defined layers in Layer array, temporal scalability

feature is disabled. E.g. to disable temporal scalability in runtime, application should pass to Reset call `mfxExtVP9TemporalLayers` buffer with all FrameRateScale set to 0.

#### 12.7.4.28.40 mfxExtVP9Param

##### **struct mfxExtVP9Param**

Attached to the `mfxVideoParam` structure extends it with VP9-specific parameters. Used by both decoder and encoder.

##### **Public Members**

###### **`mfxExtBuffer` Header**

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_VP9\_PARAM.

###### **`mfxU16` FrameWidth**

Width of the coded frame in pixels.

###### **`mfxU16` FrameHeight**

Height of the coded frame in pixels.

###### **`mfxU16` WriteIVFHeaders**

Turn this option ON to make encoder insert IVF container headers to output stream. NumFrame field of IVF sequence header will be zero, it's responsibility of application to update it with correct value. See the CodingOptionValue enumerator for values of this option.

###### **`mfxI16` QIndexDeltaLumaDC**

Specifies an offset for a particular quantization parameter.

###### **`mfxI16` QIndexDeltaChromaAC**

Specifies an offset for a particular quantization parameter.

###### **`mfxI16` QIndexDeltaChromaDC**

Specifies an offset for a particular quantization parameter.

###### **`mfxU16` NumTileRows**

Number of tile rows. Should be power of two. Maximum number of tile rows is 4 (per VP9 specification). In addition maximum supported number of tile rows may depend on underlying hardware platform. Use Query function to check if particular pair of values (NumTileRows, NumTileColumns) is supported. In VP9 tile rows have dependencies and cannot be encoded/decoded in parallel. So tile rows are always encoded by the SDK in serial mode (one-by-one).

###### **`mfxU16` NumTileColumns**

Number of tile columns. Should be power of two. Restricted with maximum and minimum tile width in luma pixels defined in VP9 specification (4096 and 256 respectively). In addition maximum supported number of tile columns may depend on underlying hardware platform. Use Query function to check if particular pair of values (NumTileRows, NumTileColumns) is supported. In VP9 tile columns don't have dependencies and can be encoded/decoded in parallel. So tile columns can be encoded by the SDK in both parallel and serial modes. Parallel mode is automatically utilized by the SDK when NumTileColumns exceeds 1 and doesn't exceed number of tile coding engines on the platform. In other cases serial mode is used. Parallel mode is capable to encode more than 1 tile row (within limitations provided by VP9 specification and particular platform). Serial mode supports only tile grids 1xN and Nx1.

#### 12.7.4.28.41 mfxEncodedUnitInfo

**struct mfxEncodedUnitInfo**

The structure *mfxEncodedUnitInfo* is used to report encoded unit info.

##### Public Members

*mfxU16 Type*

Codec-dependent coding unit type (NALU type for AVC/HEVC, start\_code for MPEG2 etc).

*mfxU32 Offset*

Offset relatively to associated *mfxBitstream::DataOffset*.

*mfxU32 Size*

Unit size including delimiter.

#### 12.7.4.28.42 mfxExtEncodedUnitsInfo

**struct mfxExtEncodedUnitsInfo**

If *mfxExtCodingOption3::EncodedUnitsInfo* was set to MFX\_CODINGOPTION\_ON during encoder initialization, structure *mfxExtEncodedUnitsInfo* attached to the *mfxBitstream* structure during encoding is used to report information about coding units in the resulting bitstream.

The number of filled items in UnitInfo is min(NumUnitsEncoded, NumUnitsAlloc).

For counting a minimal amount of encoded units you can use algorithm:

```
nSEI = amountOfApplicationDefinedSEI;
if (CodingOption3.NumSlice[IPB] != 0 || mfxVideoParam.mfx.NumSlice != 0)
    ExpectedAmount = 10 + nSEI + Max(CodingOption3.NumSlice[IPB], mfxVideoParam.mfx.
    ↵NumSlice);
else if (CodingOption2.NumMBPerSlice != 0)
    ExpectedAmount = 10 + nSEI + (FrameWidth * FrameHeight) / (256 * CodingOption2.
    ↵NumMBPerSlice);
else if (CodingOption2.MaxSliceSize != 0)
    ExpectedAmount = 10 + nSEI + Round(MaxBitrate / (FrameRate*CodingOption2.
    ↵MaxSliceSize));
else
    ExpectedAmount = 10 + nSEI;

if (mfxFrameInfo.PictStruct != MFX_PICSTRUCT_PROGRESSIVE)
    ExpectedAmount = ExpectedAmount * 2;

if (temporalScaleabilityEnabled)
    ExpectedAmount = ExpectedAmount * 2;
```

**Note** Only AVC encoder supports it.

## Public Members

### *mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_ENCODED\_UNITS\_INFO.

### *mfxEncodedUnitInfo* \***UnitInfo**

Pointer to an array of structures *mfxEncodedUnitsInfo* of size equal to or greater than NumUnitsAlloc.

### *mfxU16* **NumUnitsAlloc**

UnitInfo array size.

### *mfxU16* **NumUnitsEncoded**

Output field. Number of coding units to report. If NumUnitsEncoded is greater than NumUnitsAlloc, UnitInfo array will contain information only for the first NumUnitsAlloc units; user may consider to reallocate UnitInfo array to avoid this for consequent frames.

## 12.7.4.28.43 **mfxExtPartialBitstreamParam**

### **struct mfxExtPartialBitstreamParam**

This structure is used by an encoder to output parts of bitstream as soon as they ready. The application can attach this extended buffer to the *mfxVideoParam* structure at init time. If this option is turned ON (Granularity != MFX\_PARTIAL\_BITSTREAM\_NONE), then encoder can output bitstream by part based with required granularity.

This parameter is valid only during initialization and reset. Absence of this buffer means default or previously configured bitstream output behavior.

**Note** Not all codecs and SDK implementations support this feature. Use Query function to check if this feature is supported.

## Public Members

### *mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_PARTIAL\_BITSTREAM\_PARAM.

### *mfxU32* **BlockSize**

Output block granularity for PartialBitstreamGranularity, valid only for MFX\_PARTIAL\_BITSTREAM\_BLOCK.

### *mfxU16* **Granularity**

Granularity of the partial bitstream: slice/block/any, all types of granularity state in PartialBitstreamOutput enum.

## 12.7.4.29 VPP Extension Buffers

### 12.7.4.29.1 **mfxExtVPPDoNotUse**

### **struct mfxExtVPPDoNotUse**

The *mfxExtVPPDoNotUse* structure tells the VPP not to use certain filters in pipeline. See “Configurable VPP filters” table for complete list of configurable filters. The user can attach this structure to the *mfxVideoParam* structure when initializing video processing.

## Public Members

### *mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_VPP\_DONOTUSE.

### *mfxU32* **NumAlg**

Number of filters (algorithms) not to use

### *mfxU32* \***AlgList**

Pointer to a list of filters (algorithms) not to use

## 12.7.4.29.2 *mfxExtVPPDoUse*

### **struct mfxExtVPPDoUse**

The *mfxExtVPPDoUse* structure tells the VPP to include certain filters in pipeline.

Each filter may be included in pipeline by two different ways. First one, by adding filter ID to this structure. In this case, default filter parameters are used. Second one, by attaching filter configuration structure directly to the *mfxVideoParam* structure. In this case, adding filter ID to *mfxExtVPPDoUse* structure is optional. See Table “Configurable VPP filters” for complete list of configurable filters, their IDs and configuration structures.

The user can attach this structure to the *mfxVideoParam* structure when initializing video processing.

**Note** MFX\_EXTBUFF\_VPP\_COMPOSITE cannot be enabled using *mfxExtVPPDoUse* because default parameters are undefined for this filter. Application must attach appropriate filter configuration structure directly to the *mfxVideoParam* structure to enable it.

## Public Members

### *mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_VPP\_DOUSE.

### *mfxU32* **NumAlg**

Number of filters (algorithms) to use

### *mfxU32* \***AlgList**

Pointer to a list of filters (algorithms) to use

## 12.7.4.29.3 *mfxExtVPPDenoise*

### **struct mfxExtVPPDenoise**

The *mfxExtVPPDenoise* structure is a hint structure that configures the VPP denoise filter algorithm.

## Public Members

### *mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_VPP\_DENOISE.

### *mfxU16* **DenoiseFactor**

Value of 0-100 (inclusive) indicates the level of noise to remove.

#### 12.7.4.29.4 mfxExtVPPDetail

**struct mfxExtVPPDetail**

The *mfxExtVPPDetail* structure is a hint structure that configures the VPP detail/edge enhancement filter algorithm.

#### Public Members

*mfxExtBuffer Header*

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_VPP\_DETAIL.

*mfxU16 DetailFactor*

0-100 value (inclusive) to indicate the level of details to be enhanced.

#### 12.7.4.29.5 mfxExtVPPProcAmp

**struct mfxExtVPPProcAmp**

The *mfxExtVPPProcAmp* structure is a hint structure that configures the VPP ProcAmp filter algorithm. The structure parameters will be clipped to their corresponding range and rounded by their corresponding increment.

**Note** There are no default values for fields in this structure, all settings must be explicitly specified every time this buffer is submitted for processing.

#### Public Members

*mfxExtBuffer Header*

Extension buffer header. Header.BufferId must be equal to FX\_EXTBUFF\_VPP\_PROCAMP.

*mfxF64 Brightness*

The brightness parameter is in the range of -100.0F to 100.0F, in increments of 0.1F. Setting this field to 0.0F will disable brightness adjustment.

*mfxF64 Contrast*

The contrast parameter in the range of 0.0F to 10.0F, in increments of 0.01F, is used for manual contrast adjustment. Setting this field to 1.0F will disable contrast adjustment. If the parameter is negative, contrast will be adjusted automatically.

*mfxF64 Hue*

The hue parameter is in the range of -180F to 180F, in increments of 0.1F. Setting this field to 0.0F will disable hue adjustment.

*mfxF64 Saturation*

The saturation parameter is in the range of 0.0F to 10.0F, in increments of 0.01F. Setting this field to 1.0F will disable saturation adjustment.

#### 12.7.4.29.6 mfxExtVPPDeinterlacing

**struct mfxExtVPPDeinterlacing**

This structure is used by the application to specify different deinterlacing algorithms

##### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_VPP\_DEINTERLACING.

*mfxU16* **Mode**

Deinterlacing algorithm. See the DeinterlacingMode enumerator for details.

*mfxU16* **TelecinePattern**

Specifies telecine pattern when Mode = MFX\_DEINTERLACING\_FIXED\_TELECINE\_PATTERN. See the TelecinePattern enumerator for details.

*mfxU16* **TelecineLocation**

Specifies position inside a sequence of 5 frames where the artifacts start when TelecinePattern = MFX\_TELECINE\_POSITION\_PROVIDED

*mfxU16* **reserved[9]**

Reserved for the future use

#### 12.7.4.29.7 mfxExtEncodedSlicesInfo

**struct mfxExtEncodedSlicesInfo**

The *mfxExtEncodedSlicesInfo* is used by the SDK encoder to report additional information about encoded slices. The application can attach this buffer to the *mfxBitstream* structure before calling MFXVideoENCODE\_EncodeFrameAsync function.

**Note** Not all implementations of the SDK encoder support this extended buffer. The application has to use query mode 1 to determine if such functionality is supported. To do so, the application has to attach this extended buffer to *mfxVideoParam* structure and call MFXVideoENCODE\_Query function. If function returns MFX\_ERR\_NONE then such functionality is supported.

##### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_ENCODED\_SLICES\_INFO.

*mfxU16* **SliceSizeOverflow**

When *mfxExtCodingOption2::MaxSliceSize* is used, indicates the requested slice size was not met for one or more generated slices.

*mfxU16* **NumSliceNonCompliant**

When *mfxExtCodingOption2::MaxSliceSize* is used, indicates the number of generated slices exceeds specification limits.

*mfxU16* **NumEncodedSlice**

Number of encoded slices.

*mfxU16* **NumSliceSizeAlloc**

SliceSize array allocation size. Must be specified by application.

***mfxU16 \*SliceSize***

Slice size in bytes. Array must be allocated by application.

#### 12.7.4.29.8 ***mfxExtVppAuxData***

**`struct mfxExtVppAuxData`**

The *mfxExtVppAuxData* structure returns auxiliary data generated by the video processing pipeline. The encoding process may use the auxiliary data by attaching this structure to the *mfxEncodeCtrl* structure.

#### Public Members

***mfxExtBuffer Header***

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_VPP\_AUXDATA.

***mfxU16 PicStruct***

Detected picture structure - top field first, bottom field first, progressive or unknown if video processor cannot detect picture structure. See the PicStruct enumerator for definition of these values.

By default, detection is turned off and the application should explicitly enable it by using *mfxExtVPP-DoUse* buffer and MFX\_EXTBUFF\_VPP\_PICSTRUCT\_DETECTION algorithm.

#### 12.7.4.29.9 ***mfxExtVPPFrameRateConversion***

**`struct mfxExtVPPFrameRateConversion`**

The *mfxExtVPPFrameRateConversion* structure configures the VPP frame rate conversion filter. The user can attach this structure to the *mfxVideoParam* structure when initializing video processing, resetting it or query its capability.

On some platforms advanced frame rate conversion algorithm, algorithm based on frame interpolation, is not supported. To query its support the application should add MFX\_FRCALGM\_FRAME\_INTERPOLATION flag to Algorithm value in *mfxExtVPPFrameRateConversion* structure, attach it to structure and call MFXVideoVPP\_Query function. If filter is supported the function returns MFX\_ERR\_NONE status and copies content of input structure to output one. If advanced filter is not supported then simple filter will be used and function returns MFX\_WRN\_INCOMPATIBLE\_VIDEO\_PARAM, copies content of input structure to output one and corrects Algorithm value.

If advanced FRC algorithm is not supported both MFXVideoVPP\_Init and MFXVideoVPP\_Reset functions returns MFX\_WRN\_INCOMPATIBLE\_VIDEO\_PARAM status.

#### Public Members

***mfxExtBuffer Header***

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_VPP\_FRAME\_RATE\_CONVERSION.

***mfxU16 Algorithm***

See the FrcAlgm enumerator for a list of frame rate conversion algorithms.

#### 12.7.4.29.10 `mfxExtVPPImageStab`

**struct mfxExtVPPImageStab**

The `mfxExtVPPImageStab` structure is a hint structure that configures the VPP image stabilization filter.

On some platforms this filter is not supported. To query its support, the application should use the same approach that it uses to configure VPP filters - by adding filter ID to `mfxExtVPPDoUse` structure or by attaching `mfxExtVPPImageStab` structure directly to the `mfxVideoParam` structure and calling `MFXVideoVPP_Query` function. If this filter is supported function returns `MFX_ERR_NONE` status and copies content of input structure to output one. If filter is not supported function returns `MFX_WRN_FILTER_SKIPPED`, removes filter from `mfxExtVPPDoUse` structure and zeroes `mfxExtVPPImageStab` structure.

If image stabilization filter is not supported, both `MFXVideoVPP_Init` and `MFXVideoVPP_Reset` functions returns `MFX_WRN_FILTER_SKIPPED` status.

The application can retrieve list of active filters by attaching `mfxExtVPPDoUse` structure to `mfxVideoParam` structure and calling `MFXVideoVPP_GetVideoParam` function. The application must allocate enough memory for filter list.

#### Public Members

**`mfxExtBuffer` Header**

Extension buffer header. Header.BufferId must be equal to `MFX_EXTBUFF_VPP_IMAGE_STABILIZATION`.

**`mfxU16` Mode**

Image stabilization mode. See `ImageStabMode` enumerator for possible values.

#### 12.7.4.29.11 `mfxVPPCompInputStream`

**struct mfxVPPCompInputStream**

The `mfxVPPCompInputStream` structure is used to specify input stream details for composition of several input surfaces in the one output.

#### Public Members

**`mfxU32` DstX**

X coordinate of location of input stream in output surface.

**`mfxU32` DstY**

Y coordinate of location of input stream in output surface.

**`mfxU32` DstW**

Width of location of input stream in output surface.

**`mfxU32` DstH**

Height of location of input stream in output surface.

**`mfxU16` LumaKeyEnable**

None zero value enables luma keying for the input stream. Luma keying is used to mark some of the areas of the frame with specified luma values as transparent. It may be used for closed captioning, for example.

**`mfxU16` LumaKeyMin**

Minimum value of luma key, inclusive. Pixels whose luma values fit in this range are rendered transparent.

***mfxU16 LumaKeyMax***

Maximum value of luma key, inclusive. Pixels whose luma values fit in this range are rendered transparent.

***mfxU16 GlobalAlphaEnable***

None zero value enables global alpha blending for this input stream.

***mfxU16 GlobalAlpha***

Alpha value for this stream in [0..255] range. 0 – transparent, 255 – opaque.

***mfxU16 PixelAlphaEnable***

None zero value enables per pixel alpha blending for this input stream. The stream should have RGB color format.

***mfxU16 TileId***

Specify the tile this video stream assigned to. Should be in range [0..NumTiles). Valid only if NumTiles > 0.

#### 12.7.4.29.12 ***mfxExtVPPComposite***

**`struct mfxExtVPPComposite`**

The *mfxExtVPPComposite* structure is used to control composition of several input surfaces in the one output. In this mode, the VPP skips any other filters. The VPP returns error if any mandatory filter is specified and filter skipped warning for optional filter. The only supported filters are deinterlacing and interlaced scaling. The only supported combinations of input and output color formats are:

- RGB to RGB,
- NV12 to NV12,
- RGB and NV12 to NV12, for per pixel alpha blending use case.

The VPP returns MFX\_ERR\_MORE\_DATA for additional input until an output is ready. When the output is ready, VPP returns MFX\_ERR\_NONE. The application must process the output frame after synchronization.

Composition process is controlled by:

- *mfxFrameInfo::CropXYWH* in input surface- defines location of picture in the input frame,
- *InputStream[i].DstXYWH* defines location of the cropped input picture in the output frame,
- *mfxFrameInfo::CropXYWH* in output surface - defines actual part of output frame. All pixels in output frame outside this region will be filled by specified color.

If the application uses composition process on video streams with different frame sizes, the application should provide maximum frame size in *mfxVideoParam* during initialization, reset or query operations.

If the application uses composition process, MFXVideoVPP\_QueryIOSurf function returns cumulative number of input surfaces, i.e. number required to process all input video streams. The function sets frame size in the *mfxFrameAllocRequest* equal to the size provided by application in the *mfxVideoParam*.

Composition process supports all types of surfaces

All input surfaces should have the same type and color format, except per pixel alpha blending case, where it is allowed to mix NV12 and RGB surfaces.

There are three different blending use cases:

- Luma keying. In this case, all input surfaces should have NV12 color format specified during VPP initialization. Part of each surface, including first one, may be rendered transparent by using LumaKeyEnable, LumaKeyMin and LumaKeyMax values.
- Global alpha blending. In this case, all input surfaces should have the same color format specified during VPP initialization. It should be either NV12 or RGB. Each input surface, including first one, can be blended with underling surfaces by using GlobalAlphaEnable and GlobalAlpha values.
- Per pixel alpha blending. In this case, it is allowed to mix NV12 and RGB input surfaces. Each RGB input surface, including first one, can be blended with underling surfaces by using PixelAlphaEnable value.

It is not allowed to mix different blending use cases in the same function call.

In special case where destination region of the output surface defined by output crops is fully covered with destination sub-regions of the surfaces, the fast compositing mode can be enabled. The main use case for this mode is a video-wall scenario with fixed destination surface partition into sub-regions of potentially different size.

In order to trigger this mode, application must cluster input surfaces into tiles, defining at least one tile by setting the NumTiles field to be greater then 0 and assigning surfaces to the corresponding tiles setting TileId field to the value within [0..NumTiles) range per input surface. Tiles should also satisfy following additional constraints:

- each tile should not have more than 8 surfaces assigned to it;
- tile bounding boxes, as defined by the enclosing rectangles of a union of a surfaces assigned to this tile, should not intersect;

Background color may be changed dynamically through Reset. No default value. YUV black is (0;128;128) or (16;128;128) depending on the sample range. The SDK uses YUV or RGB triple depending on output color format.

## Public Members

### *mfxExtBuffer* Header

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_VPP\_COMPOSITE.

### *mfxU16* **Y**

Y value of the background color.

### *mfxU16* **R**

R value of the background color.

### *mfxU16* **U**

U value of the background color.

### *mfxU16* **G**

G value of the background color.

### *mfxU16* **V**

V value of the background color.

### *mfxU16* **B**

B value of the background color.

### *mfxU16* **NumTiles**

Number of input surface clusters grouped together to enable fast compositing. May be changed dynamically at runtime through Reset.

***mfxU16 NumInputStream***

Number of input surfaces to compose one output. May be changed dynamically at runtime through Reset. Number of surfaces can be decreased or increased, but should not exceed number specified during initialization. Query mode 2 should be used to find maximum supported number.

***mfxVPPCompInputStream \*InputStream***

This array of *mfxVPPCompInputStream* structures describes composition of input video streams. It should consist of exactly NumInputStream elements.

**12.7.4.29.13 mfxExtVPPVideoSignalInfo****struct mfxExtVPPVideoSignalInfo**

The *mfxExtVPPVideoSignalInfo* structure is used to control transfer matrix and nominal range of YUV frames. The application should provide it during initialization. It is supported for all kinds of conversion YUV->YUV, YUV->RGB, RGB->YUV.

**Note** This structure is used by VPP only and is not compatible with *mfxExtVideoSignalInfo*.

**Public Members*****mfxExtBuffer Header***

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_VPP\_VIDEO\_SIGNAL\_INFO.

***mfxU16 TransferMatrix***

Transfer matrix.

***mfxU16 NominalRange***

Nominal range.

**12.7.4.29.14 mfxExtVPPFieldProcessing****struct mfxExtVPPFieldProcessing**

The *mfxExtVPPFieldProcessing* structure configures the VPP field processing algorithm. The application can attach this extended buffer to the *mfxVideoParam* structure to configure initialization and/or to the *mfxFrameData* during runtime, runtime configuration has priority over initialization configuration. If field processing algorithm was activated via *mfxExtVPPDoUse* structure and *mfxExtVPPFieldProcessing* extended buffer was not provided during initialization, this buffer must be attached to *mfxFrameData* of each input surface.

**Public Members*****mfxExtBuffer Header***

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_VPP\_FIELD\_PROCESSING.

***mfxU16 Mode***

Specifies the mode of field processing algorithm. See the VPPFieldProcessingMode enumerator for values of this option.

***mfxU16 InField***

When Mode is MFX\_VPP\_COPY\_FIELD specifies input field. See the PicType enumerator for values of this parameter.

***mfxU16 OutField***

When Mode is MFX\_VPP\_COPY\_FIELD specifies output field. See the PicType enumerator for values of this parameter.

#### 12.7.4.29.15 mfxExtDecVideoProcessing

**struct mfxExtDecVideoProcessing**

If attached to the *mfxVideoParam* structure during the Init stage this buffer will instruct decoder to resize output frames via fixed function resize engine (if supported by HW) utilizing direct pipe connection bypassing intermediate memory operations. Main benefits of this mode of pipeline operation are offloading resize operation to dedicated engine reducing power consumption and memory traffic.

#### Public Members

***mfxExtBuffer Header***

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_DEC\_VIDEO\_PROCESSING.

***struct mfxExtDecVideoProcessing::mfxIn In***

Input surface description.

***struct mfxExtDecVideoProcessing::mfxOut Out***

Output surface description.

***struct mfxIn***

Input surface description.

#### Public Members

***mfxU16 CropX***

X coordinate of region of interest of the input surface.

***mfxU16 CropY***

Y coordinate of region of interest of the input surface.

***mfxU16 CropW***

Width coordinate of region of interest of the input surface.

***mfxU16 CropH***

Height coordinate of region of interest of the input surface.

***struct mfxOut***

Output surface description.

#### Public Members

***mfxU32 FourCC***

FourCC of output surface Note: Should be MFX\_FOURCC\_NV12.

***mfxU16 ChromaFormat***

Chroma Format of output surface.

**Note** Should be MFX\_CHROMAFORMAT\_YUV420

***mfxU16 Width***

Width of output surface

***mfxU16 Height***

Height of output surface

***mfxU16 CropX***

X coordinate of region of interest of the output surface.

***mfxU16 CropY***

Y coordinate of region of interest of the output surface.

***mfxU16 CropW***

Width coordinate of region of interest of the output surface.

***mfxU16 CropH***

Height coordinate of region of interest of the output surface.

#### 12.7.4.29.16 mfxExtVPPRotation

**struct mfxExtVPPRotation**

The *mfxExtVPPRotation* structure configures the VPP Rotation filter algorithm.

**Public Members**
***mfxExtBuffer Header***

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_VPP\_ROTATION.

***mfxU16 Angle***

Rotation angle. See Angle enumerator for supported values.

#### 12.7.4.29.17 mfxExtVPPScaling

**struct mfxExtVPPScaling**

The *mfxExtVPPScaling* structure configures the VPP Scaling filter algorithm. Not all combinations of ScalingMode and InterpolationMethod are supported in the SDK. The application has to use query function to determine if a combination is supported.

**Public Members**
***mfxExtBuffer Header***

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_VPP\_SCALING.

***mfxU16 ScalingMode***

Scaling mode. See ScalingMode for possible values.

***mfxU16 InterpolationMethod***

Interpolation mode for scaling algorithm. See InterpolationMode for possible values.

### 12.7.4.29.18 mfxExtVPPMirroring

**struct mfxExtVPPMirroring**

The *mfxExtVPPMirroring* structure configures the VPP Mirroring filter algorithm.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_VPP\_MIRRORING.

*mfxU16* **Type**

Mirroring type. See MirroringType for possible values.

### 12.7.4.29.19 mfxExtVPPColorFill

**struct mfxExtVPPColorFill**

The *mfxExtVPPColorFill* structure configures the VPP ColorFill filter algorithm.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_VPP\_COLORFILL.

*mfxU16* **Enable**

Set to ON makes VPP fill the area between Width/Height and Crop borders. See the CodingOptionValue enumerator for values of this option.

### 12.7.4.29.20 mfxExtColorConversion

**struct mfxExtColorConversion**

The *mfxExtColorConversion* structure is a hint structure that tunes the VPP Color Conversion algorithm, when attached to the *mfxVideoParam* structure during VPP Init.

#### Public Members

*mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_VPP\_COLOR\_CONVERSION.

*mfxU16* **ChromaSiting**

See ChromaSiting enumerator for details.

ChromaSiting is applied on input or output surface depending on the scenario:

VPP Input	VPP Output	ChromaSiting indicates
MFX_CHROMAFORMAT_YUV420 MFX_CHROMAFORMAT_YUV422	MFX_CHROMAFORMAT_YUV444	The input chroma location.
MFX_CHROMAFORMAT_YUV444	MFX_CHROMAFORMAT_YUV420 MFX_CHROMAFORMAT_YUV422	The output chroma location.
MFX_CHROMAFORMAT_YUV420	MFX_CHROMAFORMAT_YUV420	Chroma location for both input and output.
MFX_CHROMAFORMAT_YUV420	MFX_CHROMAFORMAT_YUV422	Horizontal location for both input and output. Vertical location for input.

#### 12.7.4.29.21 mfxExtVppMctf

##### struct mfxExtVppMctf

The *mfxExtVppMctf* structure allows to setup Motion-Compensated Temporal Filter (MCTF) during the VPP initialization and to control parameters at runtime. By default, MCTF is off; an application may enable it by adding MFX\_EXTBUFF\_VPP\_MCTF to *mfxExtVPPDoUse* buffer or by attaching *mfxExtVppMctf* to *mfxVideoParam* during initialization or reset.

##### Public Members

###### *mfxExtBuffer* Header

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_VPP\_MCTF.

###### *mfxU16* FilterStrength

0..20 value (inclusive) to indicate the filter-strength of MCTF. A strength of MCTF process controls degree of possible changes of pixel values eligible for MCTF; the bigger the strength the larger the change is; it is a dimensionless quantity, values 1..20 inclusively imply strength; value 0 stands for AUTO mode and is valid during initialization or reset only; if invalid value is given, it is fixed to default value which is 0. If this field is 1..20 inclusive, MCTF operates in fixed-strength mode with the given strength of MCTF process. At runtime, value 0 and values greater than 20 are ignored.

#### 12.7.4.30 Bitrate Control Extension Buffers

##### 12.7.4.30.1 mfxBRCFrameParam

##### struct mfxBRCFrameParam

The *mfxBRCFrameParam* structure describes frame parameters required for external BRC functions.

##### Public Members

###### *mfxU16* SceneChange

Frame belongs to a new scene if non zero.

###### *mfxU16* LongTerm

Frame is a Long Term Reference frame if non zero.

###### *mfxU32* FrameCmplx

Frame Complexity Frame spatial complexity if non zero. Zero if complexity is not available.

***mfxU32 EncodedOrder***

The frame number in a sequence of reordered frames starting from encoder Init.

***mfxU32 DisplayOrder***

The frame number in a sequence of frames in display order starting from last IDR.

***mfxU32 CodedFrameSize***

Size of the frame in bytes after encoding.

***mfxU16 FrameType***

See FrameType enumerator

***mfxU16 PyramidLayer***

B-pyramid or P-pyramid layer that the frame belongs to.

***mfxU16 NumRecode***

Number of recodings performed for this frame.

***mfxU16 NumExtParam***

Reserved for future use.

***mfxExtBuffer \*\*\*ExtParam***

Reserved for future use.

Frame spatial complexity is calculated according to the following formula:

$$R = \frac{16}{WH} \sum_{k=0}^{\frac{W}{4}-1} \sum_{l=0}^{\frac{H}{4}-1} \left[ \frac{\sum_{i=0}^3 \sum_{j=0}^3 |P[k * 4 + i][l * 4 + j] - P[k * 4 + i - 1][l * 4 + j]|}{16} \right]$$

$$C = \frac{16}{WH} \sum_{k=0}^{\frac{W}{4}-1} \sum_{l=0}^{\frac{H}{4}-1} \left[ \frac{\sum_{i=0}^3 \sum_{j=0}^3 |P[k * 4 + i][l * 4 + j] - P[k * 4 + i][l * 4 + j - 1]|}{16} \right]$$

$$FrameCmplx = \sqrt{R^2 + C^2}$$

#### 12.7.4.30.2 mfxBRCFrameCtrl

**struct mfxBRCFrameCtrl**

The *mfxBRCFrameCtrl* structure specifies controls for next frame encoding provided by external BRC functions.

**Public Members**
***mfxI32 QpY***

Frame-level Luma QP.

***mfxU32 InitialCpbRemovalDelay***

See `initial_cpb_removal_delay` in codec standard. Ignored if no HRD control: `mfxExtCodingOption::VuiNalHrdParameters` = `MFX_CODINGOPTION_OFF`. Calculated by encoder if `initial_cpb_removal_delay==0 && initial_cpb_removal_offset == 0 && HRD control is switched on`.

***mfxU32 InitialCpbRemovalOffset***

See `initial_cpb_removal_offset` in codec standard. Ignored if no HRD control: `mfxExtCodingOption::VuiNalHrdParameters` = `MFX_CODINGOPTION_OFF`. Calculated by encoder if `initial_cpb_removal_delay==0 && initial_cpb_removal_offset == 0 && HRD control is switched on`.

***mfxU32 MaxFrameSize***

Max frame size in bytes. Option for repack feature. Driver calls PAK until current frame size is less than or equal to maxFrameSize, or number of repacking for this frame is equal to maxNumRePak. Repack is available if there is driver support, MaxFrameSize !=0, MaxNumRePak != 0. Ignored if maxNumRePak == 0.

***mfxU8 DeltaQP[8]***

Option for repack feature. Ignored if maxNumRePak == 0 or maxNumRePak==0. If current frame size > maxFrameSize and or number of repacking (nRepack) for this frame <= maxNumRePak, PAK is called with QP = *mfxBRCFrameCtrl::QpY* + Sum(DeltaQP[i]), where i = [0,nRepack]. Non zero DeltaQP[nRepack] are ignored if nRepack > maxNumRePak. If repacking feature is on ( maxFrameSize & maxNumRePak are not zero), it is calculated by encoder.

***mfxU16 MaxNumRePak***

Number of possible repacks in driver if current frame size > maxFrameSize. Ignored if maxFrameSize==0. See maxFrameSize description. Possible values are [0,8].

***mfxU16 NumExtParam***

Reserved for future use.

***mfxExtBuffer \*\*ExtParam***

Reserved for future use.

#### 12.7.4.30.3 ***mfxBRCFrameStatus***

***struct mfxBRCFrameStatus***

The *mfxBRCFrameStatus* structure specifies instructions for the SDK encoder provided by external BRC after each frame encoding. See the BRCStatus enumerator for details.

**Public Members**
***mfxU32 MinFrameSize***

Size in bytes, coded frame must be padded to when Status = MFX\_BRC\_PANIC\_SMALL\_FRAME.

***mfxU16 BRCStatus***

See BRCStatus enumerator.

#### 12.7.4.30.4 ***mfxExtBRC***

***struct mfxExtBRC***

The *mfxExtBRC* structure contains a set of callbacks to perform external bitrate control. Can be attached to *mfxVideoParam* structure during encoder initialization. Turn *mfxExtCodingOption2::ExtBRC* option ON to make the encoder use the external BRC instead of the native one.

## Public Members

### *mfxExtBuffer* Header

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_BRC.

### *mfxHDL* pthis

Pointer to the BRC object.

### *mfxStatus* (\***Init**) (*mfxHDL* pthis, *mfxVideoParam* \*par)

This function initializes the BRC session according to parameters from input *mfxVideoParam* and attached structures. It does not modify the input *mfxVideoParam* and attached structures. Invoked during MFXVideoENCODE\_Init.

**Return** MFX\_ERR\_NONE The function completed successfully. MFX\_ERR\_UNSUPPORTED The function detected unsupported video parameters.

#### Parameters

- [in] pthis: Pointer to the BRC object.
- [in] par: Pointer to the *mfxVideoParam* structure that was used for the encoder initialization.

### *mfxStatus* (\***Reset**) (*mfxHDL* pthis, *mfxVideoParam* \*par)

This function resets BRC session according to new parameters. It does not modify the input *mfxVideoParam* and attached structures. Invoked during MFXVideoENCODE\_Reset.

**Return** MFX\_ERR\_NONE The function completed successfully. MFX\_ERR\_UNSUPPORTED The function detected unsupported video parameters.

MFX\_ERR\_INCOMPATIBLE\_VIDEO\_PARAM The function detected that the video parameters provided by the application are incompatible with initialization parameters. Reset requires additional memory allocation and cannot be executed.

#### Parameters

- [in] pthis: Pointer to the BRC object.
- [in] par: Pointer to the *mfxVideoParam* structure that was used for the encoder initialization.

### *mfxStatus* (\***Close**) (*mfxHDL* pthis)

This function de-allocates any internal resources acquired in Init for this BRC session. Invoked during MFXVideoENCODE\_Close.

**Return** MFX\_ERR\_NONE The function completed successfully.

#### Parameters

- [in] pthis: Pointer to the BRC object.

### *mfxStatus* (\***GetFrameCtrl**) (*mfxHDL* pthis, *mfxBRCFrameParam* \*par, *mfxBRCFrameCtrl* \*ctrl)

This function returns controls (ctrl) to encode next frame based on info from input *mfxBRCFrameParam* structure (par) and internal BRC state. Invoked asynchronously before each frame encoding or recoding.

**Return** MFX\_ERR\_NONE The function completed successfully.

#### Parameters

- [in] pthis: Pointer to the BRC object.
- [in] par: Pointer to the *mfxVideoParam* structure that was used for the encoder initialization.

- [out] ctrl: Pointer to the output *mfxBRCFrameCtrl* structure.

*mfxStatus* (\***Update**) (*mfxHDL* pthis, *mfxBRCFrameParam* \*par, *mfxBRCFrameCtrl* \*ctrl, *mfxBRCFrameStatus* \*status)

This function updates internal BRC state and returns status to instruct encoder whether it should recode the previous frame, skip it, do padding, or proceed to next frame based on info from input *mfxBRCFrameParam* and *mfxBRCFrameCtrl* structures. Invoked asynchronously after each frame encoding or recoding.

**Return** MFX\_ERR\_NONE The function completed successfully.

#### Parameters

- [in] pthis: Pointer to the BRC object.
- [in] par: Pointer to the *mfxVideoParam* structure that was used for the encoder initialization.
- [in] ctrl: Pointer to the output *mfxBRCFrameCtrl* structure.
- [in] status: Pointer to the output *mfxBRCFrameStatus* structure.

### 12.7.4.31 VP8 Extension Buffers

#### 12.7.4.31.1 *mfxExtVP8CodingOption*

**struct mfxExtVP8CodingOption**

##### Public Members

###### *mfxExtBuffer* Header

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_VP8\_CODING\_OPTION.

###### *mfxU16* Version

Determines the bitstream version. Corresponds to the same VP8 syntax element in frame\_tag.

###### *mfxU16* EnableMultipleSegments

Set this option to ON, to enable segmentation. This is tri-state option. See the CodingOptionValue enumerator for values of this option.

###### *mfxU16* LoopFilterType

Selecting the type of filter (normal or simple). Corresponds to VP8 syntax element filter\_type.

###### *mfxU16* LoopFilterLevel[4]

Controls the filter strength. Corresponds to VP8 syntax element loop\_filter\_level.

###### *mfxU16* SharpnessLevel

Controls the filter sensitivity. Corresponds to VP8 syntax element sharpness\_level.

###### *mfxU16* NumTokenPartitions

Specifies number of token partitions in the coded frame.

###### *mfxI16* LoopFilterRefTypeDelta[4]

Loop filter level delta for reference type (intra, last, golden, altref).

###### *mfxI16* LoopFilterMbModeDelta[4]

Loop filter level delta for MB modes.

###### *mfxI16* SegmentQPDelta[4]

QP delta for segment.

***mfxI16 CoeffTypeQPDelta[5]***

QP delta for coefficient type (YDC, Y2AC, Y2DC, UVAC, UVDC).

***mfxU16 WriteIVFHeaders***

Set this option to ON, to enable insertion of IVF container headers into bitstream. This is tri-state option. See the CodingOptionValue enumerator for values of this option

***mfxU32 NumFramesForIVFHeader***

Specifies number of frames for IVF header when WriteIVFHeaders is ON.

### 12.7.4.32 JPEG Extension Buffers

#### 12.7.4.32.1 mfxExtJPEGQuantTables

**struct mfxExtJPEGQuantTables**

The *mfxExtJPEGQuantTables* structure specifies quantization tables. The application may specify up to 4 quantization tables. The SDK encoder assigns ID to each table. That ID is equal to table index in Qm array. Table “0” is used for encoding of Y component, table “1” for U component and table “2” for V component. The application may specify fewer tables than number of components in the image. If two tables are specified, then table “1” is used for both U and V components. If only one table is specified then it is used for all components in the image. Table below illustrate this behavior.

#### Public Members

***mfxExtBuffer Header***

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_JPEG\_QT.

***mfxU16 NumTable***

Number of quantization tables defined in Qmarray.

***mfxU16 Qm[4][64]***

Quantization table values.

Table ID	0	1	2
Number of tables			
0	Y, U, V		
1	Y	U, V	
2	Y	U	V

#### 12.7.4.32.2 mfxExtJPEGHuffmanTables

**struct mfxExtJPEGHuffmanTables**

The *mfxExtJPEGHuffmanTables* structure specifies Huffman tables. The application may specify up to 2 quantization table pairs for baseline process. The SDK encoder assigns ID to each table. That ID is equal to table index in DCTables and ACTables arrays. Table “0” is used for encoding of Y component, table “1” for U and V component. The application may specify only one table in this case it will be used for all components in the image. Table below illustrate this behavior.

## Public Members

### *mfxExtBuffer* **Header**

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_JPEG\_HUFFMAN.

### *mfxU16* **NumDCTable**

Number of DC quantization table in DCTables array.

### *mfxU16* **NumACTable**

Number of AC quantization table in ACTables array.

### *mfxU8* **Bits[16]**

Number of codes for each code length.

### *mfxU8* **Values[12]**

List of the 8-bit symbol values.

Array of AC tables.

### **struct** *mfxExtJPEGHuffmanTables*::[**anonymous**] **DCTables[4]**

Array of DC tables.

### **struct** *mfxExtJPEGHuffmanTables*::[**anonymous**] **ACTables[4]**

List of the 8-bit symbol values.

Table ID	0	1
Number of tables		
0	Y, U, V	
1	Y	U, V

## 12.7.4.33 MVC Extension Buffers

### 12.7.4.33.1 *mfxMVCViewDependency*

#### **struct** *mfxMVCViewDependency*

This structure describes MVC view dependencies.

## Public Members

### *mfxU16* **ViewId**

View identifier of this dependency structure.

### *mfxU16* **NumAnchorRefsL0**

Number of view components for inter-view prediction in the initial reference picture list RefPicList0 for anchor view components.

### *mfxU16* **NumAnchorRefsL1**

Number of view components for inter-view prediction in the initial reference picture list RefPicList1 for anchor view components.

### *mfxU16* **AnchorRefL0[16]**

View identifiers of the view components for inter-view prediction in the initial reference picture list RefPicList0 for anchor view components.

### *mfxU16* **AnchorRefL1[16]**

View identifiers of the view components for inter-view prediction in the initial reference picture list RefPicList1 for anchor view components.

***mfxU16 NumNonAnchorRefsL0***

Number of view components for inter-view prediction in the initial reference picture list RefPicList0 for non-anchor view components.

***mfxU16 NumNonAnchorRefsL1***

Number of view components for inter-view prediction in the initial reference picture list RefPicList1 for non-anchor view components.

***mfxU16 NonAnchorRefL0[16]***

View identifiers of the view components for inter-view prediction in the initial reference picture list RefPicList0 for non-anchor view components.

#### 12.7.4.33.2 mfxMVCOperationPoint

**struct mfxMVCOperationPoint**

The *mfxMVCOperationPoint* structure describes the MVC operation point.

##### Public Members

***mfxU16 TemporalId***

Temporal identifier of the operation point.

***mfxU16 LevelIdc***

Level value signaled for the operation point.

***mfxU16 NumViews***

Number of views required for decoding the target output views corresponding to the operation point.

***mfxU16 NumTargetViews***

Number of target output views for the operation point.

***mfxU16 \*TargetViewId***

View identifiers of the target output views for operation point.

#### 12.7.4.33.3 mfxExtMVCSeqDesc

**struct mfxExtMVCSeqDesc**

The *mfxExtMVCSeqDesc* structure describes the MVC stream information of view dependencies, view identifiers, and operation points. See the ITU\*-T H.264 specification chapter H.7.3.2.1.4 for details.

##### Public Members

***mfxExtBuffer Header***

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_MVC\_SEQUENCE\_DESCRIPTION.

***mfxU32 NumView***

Number of views.

***mfxU32 NumViewAlloc***

The allocated view dependency array size.

***mfxMVCViewDependency \*View***

Pointer to a list of the *mfxMVCViewDependency*.

***mfxU32 NumViewId***

Number of view identifiers.

***mfxU32 NumViewIdAlloc***

The allocated view identifier array size.

***mfxU16 \*ViewId***

Pointer to the list of view identifier.

***mfxU32 NumOP***

Number of operation points.

***mfxU32 NumOPAlloc***

The allocated operation point array size.

***mfxMVCOperationPoint \*OP***

Pointer to a list of the *mfxMVCOperationPoint* structure.

***mfxU16 NumRefsTotal***

Total number of reference frames in all views required to decode the stream. This value is returned from the MFXVideoDECODE\_Decodeheader function. Do not modify this value.

#### 12.7.4.33.4 mfxExtMVCTargetViews

**struct mfxExtMVCTargetViews**

The mfxExtMvcTargetViews structure configures views for the decoding output.

**Public Members**
***mfxExtBuffer Header***

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_MVC\_TARGET\_VIEWS.

***mfxU16 TemporalId***

The temporal identifier to be decoded.

***mfxU32 NumView***

The number of views to be decoded.

***mfxU16 ViewId[1024]***

List of view identifiers to be decoded.

#### 12.7.4.34 PCP Extension Buffers

**struct \_mfxExtCencParam**

This structure is used to pass decryption status report index for Common Encryption usage model. The application can attach this extended buffer to the *mfxBitstream* structure at runtime.

## Public Members

### `mfxExtBuffer` Header

Extension buffer header. Header.BufferId must be equal to MFX\_EXTBUFF\_CENC\_PARAM.

### `mfxU32` StatusReportIndex

Decryption status report index.

## 12.7.5 Defines

`MFX_VERSION_MAJOR`

`MFX_VERSION_MINOR`

`MFX_VERSION_NEXT`

`MFX_VERSION`

`MFX_LEGACY_VERSION`

This is correspondent version of MediaSDK Legacy API which is used as a basis for the current oneVPL API.

`MFX_STRUCT_VERSION` (*MAJOR, MINOR*)

`MFX_VARIANT_VERSION`

`MFX_ENCODERDESCRIPTION_VERSION`

`MFX_DECODERDESCRIPTION_VERSION`

`MFX_VPPDESCRIPTION_VERSION`

`MFX_DEVICEDESCRIPTION_VERSION`

`MFX_IMPLDESCRIPTION_VERSION`

`MFX_FRAMESURFACEINTERFACE_VERSION`

`MFX_FRAMESURFACE1_VERSION`

## 12.7.6 Functions

### 12.7.6.1 Implementation Capabilities

`mfxHDL *MFXQueryImplsDescription` (`mfxImplCapsDeliveryFormat` *format*, `mfxU32` \**num\_impls*)

This function delivers implementation capabilities in the requested format according to the format value.

**Return** Array of handles to the capability report or NULL in case of unsupported format or NULL *num\_impls* pointer. Length of array equals to *num\_impls*.

#### Parameters

- [in] *format*: Format in which capabilities must be delivered. See `mfxImplCapsDeliveryFormat` for more details.
- [out] *num\_impls*: Number of the implementations.

---

**Important:** Function `MFXQueryImplsDescription ()` is mandatory for any implementation.

---

***mfxStatus MFXReleaseImplDescription (mfxHDL hdl)***

This function destroys the handle allocated by the MFXQueryImplsCapabilities function. Implementation must remember which handles are released. Once the last handle is released, this function must release memory allocated for the array of handles.

**Return** MFX\_ERR\_NONE The function completed successfully.

**Parameters**

- [in] hdl: Handle to destroy. Can be equal to NULL.

**Important:** Function *MFXReleaseImplDescription()* is mandatory for any implementation.

**12.7.6.2 Session Management*****mfxStatus MFXInit (mfxIMPL impl, mfxVersion \*ver, mfxSession \*session)***

This function creates and initializes an SDK session. Call this function before calling any other SDK function. If the desired implementation specified by impl is MFX\_IMPL\_AUTO, the function will search for the platform-specific SDK implementation. If the function cannot find it, it will use the software implementation.

The ver argument indicates the desired version of the library implementation. The loaded SDK will have an API version compatible to the specified version (equal in the major version number, and no less in the minor version number.) If the desired version is not specified, the default is to use the API version from the SDK release, with which an application is built.

We recommend that production applications always specify the minimum API version that meets their functional requirements. For example, if an application uses only H.264 decoding as described in API v1.0, have the application initialize the library with API v1.0. This ensures backward compatibility.

**Return** MFX\_ERR\_NONE The function completed successfully. The output parameter contains the handle of the session. MFX\_ERR\_UNSUPPORTED The function cannot find the desired SDK implementation or version.

**Parameters**

- [in] impl: mfxIMPL enumerator that indicates the desired SDK implementation.
- [in] ver: Pointer to the minimum library version or zero, if not specified.
- [out] session: Pointer to the SDK session handle.

***mfxStatus MFXInitEx (mfxInitParam par, mfxSession \*session)***

This function creates and initializes an SDK session. Call this function before calling any other SDK functions. If the desired implementation specified by par. Implementation is MFX\_IMPL\_AUTO, the function will search for the platform-specific SDK implementation. If the function cannot find it, it will use the software implementation.

The argument par.Version indicates the desired version of the library implementation. The loaded SDK will have an API version compatible to the specified version (equal in the major version number, and no less in the minor version number.) If the desired version is not specified, the default is to use the API version from the SDK release, with which an application is built.

We recommend that production applications always specify the minimum API version that meets their functional requirements. For example, if an application uses only H.264 decoding as described in API v1.0, have the application initialize the library with API v1.0. This ensures backward compatibility.

The argument `par.ExternalThreads` specifies threading mode. Value 0 means that SDK should internally create and handle work threads (this essentially equivalent of regular `MFXInit`). I

**Return** `MFX_ERR_NONE` The function completed successfully. The output parameter contains the handle of the session. `MFX_ERR_UNSUPPORTED` The function cannot find the desired SDK implementation or version.

#### Parameters

- [in] `par: mfxInitParam` structure that indicates the desired SDK implementation, minimum library version and desired threading mode.
- [out] `session`: Pointer to the SDK session handle.

---

**Important:** Function `MFXInitEx()` is mandatory for any implementation.

---

`mfxStatus MFXClose (mfxSession session)`

This function completes and deinitializes an SDK session. Any active tasks in execution or in queue are aborted. The application cannot call any SDK function after this function.

All child sessions must be disjoined before closing a parent session.

**Return** `MFX_ERR_NONE` The function completed successfully.

#### Parameters

- [in] `session`: SDK session handle.

---

**Important:** Function `MFXClose()` is mandatory for any implementation.

---

`mfxStatus MFXQueryIMPL (mfxSession session, mfxIMPL *impl)`

This function returns the implementation type of a given session.

**Return** `MFX_ERR_NONE` The function completed successfully.

#### Parameters

- [in] `session`: SDK session handle.
- [out] `impl`: Pointer to the implementation type

`mfxStatus MFXQueryVersion (mfxSession session, mfxVersion *version)`

This function returns the SDK implementation version.

**Return** `MFX_ERR_NONE` The function completed successfully.

#### Parameters

- [in] `session`: SDK session handle.
- [out] `version`: Pointer to the returned implementation version.

`mfxStatus MFXJoinSession (mfxSession session, mfxSession child)`

This function joins the child session to the current session.

After joining, the two sessions share thread and resource scheduling for asynchronous operations. However, each session still maintains its own device manager and buffer/frame allocator. Therefore, the application must use a compatible device manager and buffer/frame allocator to share data between two joined sessions.

The application can join multiple sessions by calling this function multiple times. When joining the first two sessions, the current session becomes the parent responsible for thread and resource scheduling of any later joined sessions.

Joining of two parent sessions is not supported.

**Return** MFX\_ERR\_NONE The function completed successfully. MFX\_WRN\_IN\_EXECUTION Active tasks are executing or in queue in one of the sessions. Call this function again after all tasks are completed.

MFX\_ERR\_UNSUPPORTED The child session cannot be joined with the current session.

#### Parameters

- [inout] session: The current session handle.
- [in] child: The child session handle to be joined

*mfxStatus MFXJoinSession (mfxSession session)*

This function removes the joined state of the current session. After disjoining, the current session becomes independent. The application must ensure there is no active task running in the session before calling this function.

**Return** MFX\_ERR\_NONE The function completed successfully. MFX\_WRN\_IN\_EXECUTION Active tasks are executing or in queue in one of the sessions. Call this function again after all tasks are completed.

MFX\_ERR\_UNDEFINED\_BEHAVIOR The session is independent, or this session is the parent of all joined sessions.

#### Parameters

- [inout] session: The current session handle.

*mfxStatus MFXCloneSession (mfxSession session, mfxSession \*clone)*

This function creates a clean copy of the current session. The cloned session is an independent session. It does not inherit any user-defined buffer, frame allocator, or device manager handles from the current session. This function is a light-weight equivalent of MFXJoinSession after MFXInit.

**Return** MFX\_ERR\_NONE The function completed successfully.

#### Parameters

- [in] session: The current session handle.
- [out] clone: Pointer to the cloned session handle.

*mfxStatus MFXSetPriority (mfxSession session, mfxPriority priority)*

This function sets the current session priority.

**Return** MFX\_ERR\_NONE The function completed successfully.

#### Parameters

- [in] session: The current session handle.
- [in] priority: Priority value.

*mfxStatus MFXGetPriority (mfxSession session, mfxPriority \*priority)*

This function returns the current session priority.

**Return** MFX\_ERR\_NONE The function completed successfully.

**Parameters**

- [in] session: The current session handle.
- [out] priority: Pointer to the priority value.

**12.7.6.3 VideoCORE**

*mfxStatus MFXVideoCORE\_SetFrameAllocator (mfxSession session, mfxFrameAllocator \*allocator)*

This function sets the external allocator callback structure for frame allocation. If the allocator argument is NULL, the SDK uses the default allocator, which allocates frames from system memory or hardware devices. The behavior of the SDK is undefined if it uses this function while the previous allocator is in use. A general guideline is to set the allocator immediately after initializing the session.

**Return** MFX\_ERR\_NONE The function completed successfully.

**Parameters**

- [in] session: SDK session handle.
- [in] allocator: Pointer to the *mfxFrameAllocator* structure

*mfxStatus MFXVideoCORE\_SetHandle (mfxSession session, mfxHandleType type, mfxHDL hdl)*

This function sets any essential system handle that SDK might use. If the specified system handle is a COM interface, the reference counter of the COM interface will increase. The counter will decrease when the SDK session closes.

**Return** MFX\_ERR\_NONE The function completed successfully. MFX\_ERR\_UNDEFINED\_BEHAVIOR

The same handle is redefined. For example, the function has been called twice with the same handle type or internal handle has been created by the SDK before this function call.

**Parameters**

- [in] session: SDK session handle.
- [in] type: Handle type
- [in] hdl: Handle to be set

*mfxStatus MFXVideoCORE\_GetHandle (mfxSession session, mfxHandleType type, mfxHDL \*hdl)*

This function obtains system handles previously set by the MFXVideoCORE\_SetHandle function. If the handler is a COM interface, the reference counter of the interface increases. The calling application must release the COM interface.

**Return** MFX\_ERR\_NONE The function completed successfully. MFX\_ERR\_UNDEFINED\_BEHAVIOR

Specified handle type not found.

**Parameters**

- [in] session: SDK session handle.
- [in] type: Handle type
- [in] hdl: Pointer to the handle to be set

*mfxStatus MFXVideoCORE\_QueryPlatform (mfxSession session, mfxPlatform \*platform)*

This function returns information about current hardware platform.

**Return** MFX\_ERR\_NONE The function completed successfully.

## Parameters

- [in] session: SDK session handle.
- [out] platform: Pointer to the *mfxPlatform* structure

*mfxStatus* **MFXVideoCORE\_SyncOperation** (*mfxSession* session, *mfxSyncPoint* syncp, *mfxU32* wait)

This function initiates execution of an asynchronous function not already started and returns the status code after the specified asynchronous operation completes. If wait is zero, the function returns immediately.

**Return** MFX\_ERR\_NONE The function completed successfully. MFX\_ERR\_NONE\_PARTIAL\_OUTPUT

The function completed successfully, bitstream contains a portion of the encoded frame according to required granularity.

MFX\_WRN\_IN\_EXECUTION The specified asynchronous function is in execution.

MFX\_ERR\_ABORTED The specified asynchronous function aborted due to data dependency on a previous asynchronous function that did not complete.

## Parameters

- [in] session: SDK session handle.
- [in] syncp: Sync point
- [in] wait: wait time in milliseconds

---

**Important:** Function *MFXVideoCORE\_SyncOperation ()* is mandatory for any implementation.

---

### 12.7.6.4 Memory

*mfxStatus* **MFXMemory\_GetSurfaceForVPP** (*mfxSession* session, *mfxFrameSurface1* \*\*surface)

This function returns surface which can be used as input for VPP. VPP should be initialized before this call. Surface should be released with *mfxFrameSurface1::FrameInterface.Release(...)* after usage. Value of *mfxFrameSurface1::Data.Locked* for returned surface is 0.

**Return** MFX\_ERR\_NONE The function completed successfully. MFX\_ERR\_NULL\_PTR If surface is NULL.

MFX\_ERR\_INVALID\_HANDLE If session was not initialized.

MFX\_ERR\_NOT\_INITIALIZED If VPP wasn't initialized (allocator needs to know surface size from somewhere).

MFX\_ERR\_MEMORY\_ALLOC In case of any other internal allocation error.

## Parameters

- [in] session: SDK session handle.
- [out] surface: Pointer is set to valid *mfxFrameSurface1* object.

*mfxStatus* **MFXMemory\_GetSurfaceForEncode** (*mfxSession* session, *mfxFrameSurface1* \*\*surface)

This function returns surface which can be used as input for Encoder. Encoder should be initialized before this call. Surface should be released with *mfxFrameSurface1::FrameInterface.Release(...)* after usage. Value of *mfxFrameSurface1::Data.Locked* for returned surface is 0.

**Return** MFX\_ERR\_NONE The function completed successfully. MFX\_ERR\_NULL\_PTR If surface is NULL.

MFX\_ERR\_INVALID\_HANDLE If session was not initialized.

MFX\_ERR\_NOT\_INITIALIZED If Encoder wasn't initialized (allocator needs to know surface size from somewhere).

MFX\_ERR\_MEMORY\_ALLOC In case of any other internal allocation error.

#### Parameters

- [in] session: SDK session handle.
- [out] surface: Pointer is set to valid *mfxFrameSurface1* object.

#### *mfxStatus MFXMemory\_GetSurfaceForDecode (mfxSession session, mfxFrameSurface1 \*\*surface)*

This function returns surface which can be used as input for Decoder. Decoder should be initialized before this call. Surface should be released with *mfxFrameSurface1::FrameInterface.Release(...)* after usage. Value of *mfxFrameSurface1::Data.Locked* for returned surface is 0.' Note: this function was added to simplify transition from legacy surface management to proposed internal allocation approach. Previously, user allocated surfaces for working pool and fed decoder with them in *DecodeFrameAsync* calls. With *MFXMemory\_GetSurfaceForDecode* it is possible to change the existing pipeline just changing source of work surfaces. Newly developed applications should prefer direct usage of *DecodeFrameAsync* with internal allocation.'

**Return** MFX\_ERR\_NONE The function completed successfully. MFX\_ERR\_NULL\_PTR If surface is NULL.

MFX\_ERR\_INVALID\_HANDLE If session was not initialized.

MFX\_ERR\_NOT\_INITIALIZED If Decoder wasn't initialized (allocator needs to know surface size from somewhere).

MFX\_ERR\_MEMORY\_ALLOC In case of any other internal allocation error.

#### Parameters

- [in] session: SDK session handle.
- [out] surface: Pointer is set to valid *mfxFrameSurface1* object.

### 12.7.6.5 VideoENCODE

#### *mfxStatus MFXVideoENCODE\_Query (mfxSession session, mfxVideoParam \*in, mfxVideoParam \*out)*

This function works in either of four modes:

If the in pointer is zero, the function returns the class configurability in the output structure. A non-zero value in each field of the output structure that the SDK implementation can configure the field with Init.

If the in parameter is non-zero, the function checks the validity of the fields in the input structure. Then the function returns the corrected values in the output structure. If there is insufficient information to determine the validity or correction is impossible, the function zeroes the fields. This feature can verify whether the SDK implementation supports certain profiles, levels or bitrates.

If the in parameter is non-zero and *mfxExtEncoderResetOption* structure is attached to it, then the function queries for the outcome of the *MFXVideoENCODE\_Reset* function and returns it in the *mfxExtEncoderResetOption* structure attached to out. The query function succeeds if such reset is possible and returns error otherwise. Unlike other modes that are independent of the SDK encoder state, this one checks if reset is possible in the present SDK encoder state. This mode also requires completely defined *mfxVideoParam* structure, unlike other modes that support partially defined configurations. See *mfxExtEncoderResetOption* description for more details.

If the in parameter is non-zero and *mfxExtEncoderCapability* structure is attached to it, then the function returns encoder capability in *mfxExtEncoderCapability* structure attached to out. It is recommended to fill in *mfxVideoParam* structure and set hardware acceleration device handle before calling the function in this mode.

The application can call this function before or after it initializes the encoder. The CodecId field of the output structure is a mandated field (to be filled by the application) to identify the coding standard.

**Return** MFX\_ERR\_NONE The function completed successfully. MFX\_ERR\_UNSUPPORTED The function failed to identify a specific implementation for the required features.

MFX\_WRN\_PARTIAL\_ACCELERATION The underlying hardware does not fully support the specified video parameters. The encoding may be partially accelerated. Only SDK hardware implementations may return this status code.

MFX\_WRN\_INCOMPATIBLE\_VIDEO\_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

#### Parameters

- [in] session: SDK session handle.
- [in] in: Pointer to the *mfxVideoParam* structure as input
- [out] out: Pointer to the *mfxVideoParam* structure as output

---

**Important:** The *MFXVideoENCODE\_Query()* function is mandatory when implementing an encoder.

---

*mfxStatus MFXVideoENCODE\_QueryIOSurf(mfxSession session, mfxVideoParam \*par, mfxFrameAllocRequest \*request)*

This function returns minimum and suggested numbers of the input frame surfaces required for encoding initialization and their type. Init will call the external allocator for the required frames with the same set of numbers. The use of this function is recommended. For more information, see the section Working with hardware acceleration. This function does not validate I/O parameters except those used in calculating the number of input surfaces.

**Return** MFX\_ERR\_NONE The function completed successfully. MFX\_ERR\_INVALID\_VIDEO\_PARAM The function detected invalid video parameters. These parameters may be out of the valid range, or the combination of them resulted in incompatibility. Incompatibility not resolved.

MFX\_WRN\_PARTIAL\_ACCELERATION The underlying hardware does not fully support the specified video parameters. The encoding may be partially accelerated. Only SDK hardware implementations may return this status code.

MFX\_WRN\_INCOMPATIBLE\_VIDEO\_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

#### Parameters

- [in] session: SDK session handle.
- [in] par: Pointer to the *mfxVideoParam* structure as input
- [in] request: Pointer to the *mfxFrameAllocRequest* structure as output

*mfxStatus MFXVideoENCODE\_Init(mfxSession session, mfxVideoParam \*par)*

This function allocates memory and prepares tables and necessary structures for encoding. This function also does extensive validation to ensure if the configuration, as specified in the input parameters, is supported.

**Return** MFX\_ERR\_NONE The function completed successfully. MFX\_ERR\_INVALID\_VIDEO\_PARAM The function detected invalid video parameters. These parameters may be out of the valid range, or the combination of them resulted in incompatibility. Incompatibility not resolved.

MFX\_WRN\_PARTIAL\_ACCELERATION The underlying hardware does not fully support the specified video parameters. The encoding may be partially accelerated. Only SDK hardware implementations may return this status code.

MFX\_WRN\_INCOMPATIBLE\_VIDEO\_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

MFX\_ERR\_UNDEFINED\_BEHAVIOR The function is called twice without a close;

#### Parameters

- [in] session: SDK session handle.
- [in] par: Pointer to the *mfxVideoParam* structure

**Important:** The *MFXVideoEncode\_Init()* function is mandatory when implementing an encoder.

*mfxStatus MFXVideoEncode\_Reset (mfxSession session, mfxVideoParam \*par)*

This function stops the current encoding operation and restores internal structures or parameters for a new encoding operation, possibly with new parameters.

**Return** MFX\_ERR\_NONE The function completed successfully. MFX\_ERR\_INVALID\_VIDEO\_PARAM The function detected invalid video parameters. These parameters may be out of the valid range, or the combination of them resulted in incompatibility. Incompatibility not resolved.

MFX\_ERR\_INCOMPATIBLE\_VIDEO\_PARAM The function detected that provided by the application video parameters are incompatible with initialization parameters. Reset requires additional memory allocation and cannot be executed. The application should close the SDK component and then reinitialize it.

MFX\_WRN\_INCOMPATIBLE\_VIDEO\_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

#### Parameters

- [in] session: SDK session handle.
- [in] par: Pointer to the *mfxVideoParam* structure

*mfxStatus MFXVideoEncode\_Close (mfxSession session)*

This function terminates the current encoding operation and de-allocates any internal tables or structures.

**Return** MFX\_ERR\_NONE The function completed successfully.

#### Parameters

- [in] session: SDK session handle.

**Important:** The *MFXVideoEncode\_Close()* function is mandatory when implementing an encoder.

*mfxStatus MFXVideoEncode\_GetVideoParam (mfxSession session, mfxVideoParam \*par)*

This function retrieves current working parameters to the specified output structure. If extended buffers are to be returned, the application must allocate those extended buffers and attach them as part of the output structure. The

application can retrieve a copy of the bitstream header, by attaching the *mfxExtCodingOptionSPSPPS* structure to the *mfxVideoParam* structure.

**Return** MFX\_ERR\_NONE The function completed successfully.

#### Parameters

- [in] session: SDK session handle.
- [in] par: Pointer to the corresponding parameter structure

*mfxStatus* **MFXVideoENCODE\_GetEncodeStat** (*mfxSession* session, *mfxEncodeStat* \*stat)

This function obtains statistics collected during encoding.

**Return** MFX\_ERR\_NONE The function completed successfully.

#### Parameters

- [in] session: SDK session handle.
- [in] stat: Pointer to the *mfxEncodeStat* structure

*mfxStatus* **MFXVideoENCODE\_EncodeFrameAsync** (*mfxSession* session, *mfxEncodeCtrl* \*ctrl,  
*mfxFrameSurface1* \*surface, *mfxBitstream* \*bs,  
*mfxSyncPoint* \*syncp)

This function takes a single input frame in either encoded or display order and generates its output bitstream. In the case of encoded ordering the *mfxEncodeCtrl* structure must specify the explicit frame type. In the case of display ordering, this function handles frame order shuffling according to the GOP structure parameters specified during initialization.

Since encoding may process frames differently from the input order, not every call of the function generates output and the function returns MFX\_ERR\_MORE\_DATA. If the encoder needs to cache the frame, the function locks the frame. The application should not alter the frame until the encoder unlocks the frame. If there is output (with return status MFX\_ERR\_NONE), the return is a frame worth of bitstream.

It is the calling application's responsibility to ensure that there is sufficient space in the output buffer. The value BufferSizeInKB in the *mfxVideoParam* structure at encoding initialization specifies the maximum possible size for any compressed frames. This value can also be obtained from MFXVideoENCODE\_GetVideoParam after encoding initialization.

To mark the end of the encoding sequence, call this function with a NULL surface pointer. Repeat the call to drain any remaining internally cached bitstreams (one frame at a time) until MFX\_ERR\_MORE\_DATA is returned.

This function is asynchronous.

**Return** MFX\_ERR\_NONE The function completed successfully. MFX\_ERR\_NOT\_ENOUGH\_BUFFER The bitstream buffer size is insufficient.

MFX\_ERR\_MORE\_DATA The function requires more data to generate any output.

MFX\_ERR\_DEVICE\_LOST Hardware device was lost; See Working with Microsoft® DirectX® Applications section for further information.

MFX\_WRN\_DEVICE\_BUSY Hardware device is currently busy. Call this function again in a few milliseconds.

MFX\_ERR\_INCOMPATIBLE\_VIDEO\_PARAM Inconsistent parameters detected not conforming to Appendix A.

#### Parameters

- [in] session: SDK session handle.
- [in] ctrl: Pointer to the *mfxEncodeCtrl* structure for per-frame encoding control; this parameter is optional(it can be NULL) if the encoder works in the display order mode.
- [in] surface: Pointer to the frame surface structure
- [out] bs: Pointer to the output bitstream
- [out] syncp: Pointer to the returned sync point associated with this operation

---

**Important:** The *MFXVideoENCODE\_EncodeFrameAsync()* function is mandatory when implementing an encoder.

---

### 12.7.6.6 VideoDECODE

*mfxStatus MFXVideoDECODE\_Query (mfxSession session, mfxVideoParam \*in, mfxVideoParam \*out)*

This function works in one of two modes:

- 1.If the in pointer is zero, the function returns the class configurability in the output structure. A non-zero value in each field of the output structure indicates that the field is configurable by the SDK implementation with the *MFXVideoDECODE\_Init* function).
- 2.If the in parameter is non-zero, the function checks the validity of the fields in the input structure. Then the function returns the corrected values to the output structure. If there is insufficient information to determine the validity or correction is impossible, the function zeros the fields. This feature can verify whether the SDK implementation supports certain profiles, levels or bitrates.

The application can call this function before or after it initializes the decoder. The CodecId field of the output structure is a mandated field (to be filled by the application) to identify the coding standard.

**Return** MFX\_ERR\_NONE The function completed successfully. MFX\_ERR\_UNSUPPORTED The function failed to identify a specific implementation for the required features.

MFX\_WRN\_PARTIAL\_ACCELERATION The underlying hardware does not fully support the specified video parameters. The decoding may be partially accelerated. Only SDK hardware implementations may return this status code.

MFX\_WRN\_INCOMPATIBLE\_VIDEO\_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

#### Parameters

- [in] session: SDK session handle.
- [in] in: Pointer to the *mfxVideoParam* structure as input
- [out] out: Pointer to the *mfxVideoParam* structure as output

---

**Important:** The *MFXVideoDECODE\_Query()* is mandatory when implementing a decoder.

---

*mfxStatus MFXVideoDECODE\_DecodeHeader (mfxSession session, mfxBitstream \*bs, mfxVideoParam \*par)*

This function parses the input bitstream and fills the *mfxVideoParam* structure with appropriate values, such as resolution and frame rate, for the Init function. The application can then pass the resulting structure to the *MFXVideoDECODE\_Init* function for decoder initialization.

An application can call this function at any time before or after decoder initialization. If the SDK finds a sequence header in the bitstream, the function moves the bitstream pointer to the first bit of the sequence header. Otherwise, the function moves the bitstream pointer close to the end of the bitstream buffer but leaves enough data in the buffer to avoid possible loss of start code.

The CodecId field of the *mfxVideoParam* structure is a mandated field (to be filled by the application) to identify the coding standard.

The application can retrieve a copy of the bitstream header, by attaching the *mfxExtCodingOptionSPSPPS* structure to the *mfxVideoParam* structure.

**Return** MFX\_ERR\_NONE The function successfully filled structure. It does not mean that the stream can be decoded by SDK. The application should call MFXVideoDECODE\_Query function to check if decoding of the stream is supported. MFX\_ERR\_MORE\_DATA The function requires more bitstream data

MFX\_ERR\_UNSUPPORTED CodecId field of the *mfxVideoParam* structure indicates some unsupported codec. MFX\_ERR\_INVALID\_HANDLE session is not initialized

MFX\_ERR\_NULL\_PTR bs or par pointer is NULL.

#### Parameters

- [in] session: SDK session handle.
- [in] bs: Pointer to the bitstream
- [in] par: Pointer to the *mfxVideoParam* structure

*mfxStatus* **MFXVideoDECODE\_QueryIOSurf** (*mfxSession* session, *mfxVideoParam* \*par, *mfxFrameAllocRequest* \*request)

This function returns minimum and suggested numbers of the output frame surfaces required for decoding initialization and their type. Init will call the external allocator for the required frames with the same set of numbers. The use of this function is recommended. For more information, see the section Working with hardware acceleration. The CodecId field of the *mfxVideoParam* structure is a mandated field (to be filled by the application) to identify the coding standard. This function does not validate I/O parameters except those used in calculating the number of output surfaces.

**Return** MFX\_ERR\_NONE The function completed successfully. MFX\_ERR\_INVALID\_VIDEO\_PARAM The function detected invalid video parameters. These parameters may be out of the valid range, or the combination of them resulted in incompatibility. Incompatibility not resolved.

MFX\_WRN\_PARTIAL\_ACCELERATION The underlying hardware does not fully support the specified video parameters. The encoding may be partially accelerated. Only SDK hardware implementations may return this status code.

MFX\_WRN\_INCOMPATIBLE\_VIDEO\_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

#### Parameters

- [in] session: SDK session handle.
- [in] par: Pointer to the *mfxVideoParam* structure as input
- [in] request: Pointer to the *mfxFrameAllocRequest* structure as output

*mfxStatus* **MFXVideoDECODE\_Init** (*mfxSession* session, *mfxVideoParam* \*par)

This function allocates memory and prepares tables and necessary structures for encoding. This function also does extensive validation to ensure if the configuration, as specified in the input parameters, is supported.

**Return** MFX\_ERR\_NONE The function completed successfully. MFX\_ERR\_INVALID\_VIDEO\_PARAM The function detected invalid video parameters. These parameters may be out of the valid range, or the combination of them resulted in incompatibility. Incompatibility not resolved.

MFX\_WRN\_PARTIAL\_ACCELERATION The underlying hardware does not fully support the specified video parameters. The encoding may be partially accelerated. Only SDK hardware implementations may return this status code.

MFX\_WRN\_INCOMPATIBLE\_VIDEO\_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

MFX\_ERR\_UNDEFINED\_BEHAVIOR The function is called twice without a close;

#### Parameters

- [in] session: SDK session handle.
- [in] par: Pointer to the *mfxVideoParam* structure

---

**Important:** The *MFXVideoDECODE\_Init()* is mandatory when implementing a decoder.

---

*mfxStatus MFXVideoDECODE\_Reset (mfxSession session, mfxVideoParam \*par)*

This function stops the current decoding operation and restores internal structures or parameters for a new decoding operation. Reset serves two purposes: It recovers the decoder from errors. It restarts decoding from a new position. The function resets the old sequence header (sequence parameter set in H.264, or sequence header in MPEG-2 and VC-1). The decoder will expect a new sequence header before it decodes the next frame and will skip any bitstream before encountering the new sequence header.

**Return** MFX\_ERR\_NONE The function completed successfully. MFX\_ERR\_INVALID\_VIDEO\_PARAM The function detected that video parameters are wrong or they conflict with initialization parameters. Reset is impossible.

MFX\_ERR\_INCOMPATIBLE\_VIDEO\_PARAM The function detected that provided by the application video parameters are incompatible with initialization parameters. Reset requires additional memory allocation and cannot be executed. The application should close the SDK component and then reinitialize it.

MFX\_WRN\_INCOMPATIBLE\_VIDEO\_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

#### Parameters

- [in] session: SDK session handle.
- [in] par: Pointer to the *mfxVideoParam* structure

*mfxStatus MFXVideoDECODE\_Close (mfxSession session)*

This function terminates the current decoding operation and de-allocates any internal tables or structures.

**Return** MFX\_ERR\_NONE The function completed successfully.

#### Parameters

- [in] session: SDK session handle.

---

**Important:** The *MFXVideoDECODE\_Close()* is mandatory when implementing a decoder.

---

`mfxStatus MFXVideoDECODE_GetVideoParam (mfxSession session, mfxVideoParam *par)`

This function retrieves current working parameters to the specified output structure. If extended buffers are to be returned, the application must allocate those extended buffers and attach them as part of the output structure. The application can retrieve a copy of the bitstream header, by attaching the `mfxExtCodingOptionSPSPPS` structure to the `mfxVideoParam` structure.

**Return** MFX\_ERR\_NONE The function completed successfully.

#### Parameters

- [in] session: SDK session handle.
- [in] par: Pointer to the corresponding parameter structure

`mfxStatus MFXVideoDECODE_GetDecodeStat (mfxSession session, mfxDecodeStat *stat)`

This function obtains statistics collected during decoding.

**Return** MFX\_ERR\_NONE The function completed successfully.

#### Parameters

- [in] session: SDK session handle.
- [in] stat: Pointer to the `mfxDecodeStat` structure

`mfxStatus MFXVideoDECODE_SetSkipMode (mfxSession session, mfxSkipMode mode)`

This function sets the decoder skip mode. The application may use it to increase decoding performance by sacrificing output quality. The rising of skip level firstly results in skipping of some decoding operations like deblocking and then leads to frame skipping; firstly, B then P. Particular details are platform dependent.

**Return** MFX\_ERR\_NONE The function completed successfully and the output surface is ready for decoding  
MFX\_WRN\_VALUE\_NOT\_CHANGED The skip mode is not affected as the maximum or minimum skip range is reached.

#### Parameters

- [in] session: SDK session handle.
- [in] mode: Decoder skip mode. See the `mfxSkipMode` enumerator for details.

`mfxStatus MFXVideoDECODE_GetPayload (mfxSession session, mfxU64 *ts, mfxPayload *payload)`

This function extracts user data (MPEG-2) or SEI (H.264) messages from the bitstream. Internally, the decoder implementation stores encountered user data or SEI messages. The application may call this function multiple times to retrieve the user data or SEI messages, one at a time.

If there is no payload available, the function returns with `payload->NumBit=0`.

**Return** MFX\_ERR\_NONE The function completed successfully and the output buffer is ready for decoding  
MFX\_ERR\_NOT\_ENOUGH\_BUFFER The payload buffer size is insufficient.

#### Parameters

- [in] session: SDK session handle.
- [in] ts: Pointer to the user data time stamp in units of 90 KHz; divide ts by 90,000 (90 KHz) to obtain the time in seconds; the time stamp matches the payload with a specific decoded frame.
- [in] payload: Pointer to the `mfxPayload` structure; the payload contains user data in MPEG-2 or SEI messages in H.264.

*mfxStatus MFXVideoDECODE\_DecodeFrameAsync (mfxSession session, mfxBitstream \*bs, mfxFrameSurface1 \*surface\_work, mfxFrameSurface1 \*\*surface\_out, mfxSyncPoint \*syncp)*

This function decodes the input bitstream to a single output frame.

The surface\_work parameter provides a working frame buffer for the decoder. The application should allocate the working frame buffer, which stores decoded frames. If the function requires caching frames after decoding, the function locks the frames and the application must provide a new frame buffer in the next call.

If, and only if, the function returns MFX\_ERR\_NONE, the pointer surface\_out points to the output frame in the display order. If there are no further frames, the function will reset the pointer to zero and return the appropriate status code.

Before decoding the first frame, a sequence header(sequence parameter set in H.264 or sequence header in MPEG-2 and VC-1) must be present. The function skips any bitstreams before it encounters the new sequence header.

The input bitstream bs can be of any size. If there are not enough bits to decode a frame, the function returns MFX\_ERR\_MORE\_DATA, and consumes all input bits except if a partial start code or sequence header is at the end of the buffer. In this case, the function leaves the last few bytes in the bitstream buffer. If there is more incoming bitstream, the application should append the incoming bitstream to the bitstream buffer. Otherwise, the application should ignore the remaining bytes in the bitstream buffer and apply the end of stream procedure described below.

The application must set bs to NULL to signal end of stream. The application may need to call this function several times to drain any internally cached frames until the function returns MFX\_ERR\_MORE\_DATA.

If more than one frame is in the bitstream buffer, the function decodes until the buffer is consumed. The decoding process can be interrupted for events such as if the decoder needs additional working buffers, is readying a frame for retrieval, or encountering a new header. In these cases, the function returns appropriate status code and moves the bitstream pointer to the remaining data.

The decoder may return MFX\_ERR\_NONE without taking any data from the input bitstream buffer. If the application appends additional data to the bitstream buffer, it is possible that the bitstream buffer may contain more than 1 frame. It is recommended that the application invoke the function repeatedly until the function returns MFX\_ERR\_MORE\_DATA, before appending any more data to the bitstream buffer. This function is asynchronous.

**Return** MFX\_ERR\_NONE The function completed successfully and the output surface is ready for decoding  
MFX\_ERR\_MORE\_DATA The function requires more bitstream at input before decoding can proceed.

MFX\_ERR\_MORE\_SURFACE The function requires more frame surface at output before decoding can proceed.

MFX\_ERR\_DEVICE\_LOST Hardware device was lost; See the Working with Microsoft® DirectX® Applications section for further information.

MFX\_WRN\_DEVICE\_BUSY Hardware device is currently busy. Call this function again in a few milliseconds.

MFX\_WRN\_VIDEO\_PARAM\_CHANGED The decoder detected a new sequence header in the bitstream. Video parameters may have changed.

MFX\_ERR\_INCOMPATIBLE\_VIDEO\_PARAM The decoder detected incompatible video parameters in the bitstream and failed to follow them.

MFX\_ERR\_REALLOC\_SURFACE Bigger surface\_work required. May be returned only if *mfxInfoMFX::EnableReallocRequest* was set to ON during initialization.

## Parameters

- [in] session: SDK session handle.

- [in] bs: Pointer to the input bitstream
- [in] surface\_work: Pointer to the working frame buffer for the decoder
- [out] surface\_out: Pointer to the output frame in the display order
- [out] syncp: Pointer to the sync point associated with this operation

---

**Important:** The `MFXVideoDECODE_DecodeFrameAsync()` is mandatory when implementing a decoder.

---

### 12.7.6.7 VideoVPP

`mfxStatus MFXVideoVPP_Query(mfxSession session, mfxVideoParam *in, mfxVideoParam *out)`

This function works in one of two modes:

- 1.If the in pointer is zero, the function returns the class configurability in the output structure. A non-zero value in a field indicates that the SDK implementation can configure it with Init.
- 2.If the in parameter is non-zero, the function checks the validity of the fields in the input structure. Then the function returns the corrected values to the output structure. If there is insufficient information to determine the validity or correction is impossible, the function zeroes the fields.

The application can call this function before or after it initializes the preprocessor.

**Return** MFX\_ERR\_NONE The function completed successfully. MFX\_ERR\_UNSUPPORTED The SDK implementation does not support the specified configuration.

MFX\_WRN\_PARTIAL\_ACCELERATION The underlying hardware does not fully support the specified video parameters. The video processing may be partially accelerated. Only SDK hardware implementations may return this status code.

MFX\_WRN\_INCOMPATIBLE\_VIDEO\_PARAM The function detected some video parameters were incompatible with others; incompatibility resolved.

#### Parameters

- [in] session: SDK session handle.
- [in] in: Pointer to the `mfxVideoParam` structure as input
- [out] out: Pointer to the `mfxVideoParam` structure as output

---

**Important:** The `MFXVideoVPP_Query()` function is mandatory when implementing a VPP filter.

---

`mfxStatus MFXVideoVPP_QueryIOSurf(mfxSession session, mfxVideoParam *par, mfxFrameAllocRequest request[2])`

This function returns minimum and suggested numbers of the input frame surfaces required for video processing initialization and their type. The parameter `request[0]` refers to the input requirements; `request[1]` refers to output requirements. Init will call the external allocator for the required frames with the same set of numbers. The use of this function is recommended. For more information, see the section Working with hardware acceleration. This function does not validate I/O parameters except those used in calculating the number of input surfaces.

**Return** MFX\_ERR\_NONE The function completed successfully. MFX\_ERR\_INVALID\_VIDEO\_PARAM The function detected invalid video parameters. These parameters may be out of the valid range, or the combination of them resulted in incompatibility. Incompatibility not resolved.

**MFX\_WRN\_PARTIAL\_ACCELERATION** The underlying hardware does not fully support the specified video parameters. The video processing may be partially accelerated. Only SDK hardware implementations may return this status code.

**MFX\_WRN\_INCOMPATIBLE\_VIDEO\_PARAM** The function detected some video parameters were incompatible with others; incompatibility resolved.

#### Parameters

- [in] session: SDK session handle.
- [in] par: Pointer to the *mfxVideoParam* structure as input
- [in] request: Pointer to the *mfxFrameAllocRequest* structure; use request[0] for input requirements and request[1] for output requirements for video processing.

*mfxStatus MFXVideoVPP\_Init (mfxSession session, mfxVideoParam \*par)*

This function allocates memory and prepares tables and necessary structures for video processing. This function also does extensive validation to ensure if the configuration, as specified in the input parameters, is supported.

**Return** **MFX\_ERR\_NONE** The function completed successfully. **MFX\_ERR\_INVALID\_VIDEO\_PARAM** The function detected invalid video parameters. These parameters may be out of the valid range, or the combination of them resulted in incompatibility. Incompatibility not resolved.

**MFX\_WRN\_PARTIAL\_ACCELERATION** The underlying hardware does not fully support the specified video parameters. The video processing may be partially accelerated. Only SDK hardware implementations may return this status code.

**MFX\_WRN\_INCOMPATIBLE\_VIDEO\_PARAM** The function detected some video parameters were incompatible with others; incompatibility resolved.

**MFX\_ERR\_UNDEFINED\_BEHAVIOR** The function is called twice without a close.

**MFX\_WRN\_FILTER\_SKIPPED** The VPP skipped one or more filters requested by the application.

#### Parameters

- [in] session: SDK session handle.
- [in] par: Pointer to the *mfxVideoParam* structure

---

**Important:** The *MFXVideoVPP\_Init()* function is mandatory when implementing a VPP filter.

---

*mfxStatus MFXVideoVPP\_Reset (mfxSession session, mfxVideoParam \*par)*

This function stops the current video processing operation and restores internal structures or parameters for a new operation.

**Return** **MFX\_ERR\_NONE** The function completed successfully. **MFX\_ERR\_INVALID\_VIDEO\_PARAM** The function detected that video parameters are wrong or they conflict with initialization parameters. Reset is impossible.

**MFX\_ERR\_INCOMPATIBLE\_VIDEO\_PARAM** The function detected that provided by the application video parameters are incompatible with initialization parameters. Reset requires additional memory allocation and cannot be executed. The application should close the SDK component and then reinitialize it.

**MFX\_WRN\_INCOMPATIBLE\_VIDEO\_PARAM** The function detected some video parameters were incompatible with others; incompatibility resolved.

#### Parameters

- [in] session: SDK session handle.
- [in] par: Pointer to the *mfxVideoParam* structure

*mfxStatus* **MFXVideoVPP\_Close** (*mfxSession* *session*)

This function terminates the current video processing operation and de-allocates any internal tables or structures.

**Return** MFX\_ERR\_NONE The function completed successfully.

**Parameters**

- [in] session: SDK session handle.

---

**Important:** The *MFXVideoVPP\_Close()* function is mandatory when implementing a VPP filter.

---

*mfxStatus* **MFXVideoVPP\_GetVideoParam** (*mfxSession* *session*, *mfxVideoParam* \**par*)

This function retrieves current working parameters to the specified output structure. If extended buffers are to be returned, the application must allocate those extended buffers and attach them as part of the output structure.

**Return** MFX\_ERR\_NONE The function completed successfully.

**Parameters**

- [in] session: SDK session handle.
- [in] par: Pointer to the corresponding parameter structure

*mfxStatus* **MFXVideoVPP\_GetVPPStat** (*mfxSession* *session*, *mfxVPPStat* \**stat*)

This function obtains statistics collected during video processing.

**Return** MFX\_ERR\_NONE The function completed successfully.

**Parameters**

- [in] session: SDK session handle.
- [in] stat: Pointer to the *mfxVPPStat* structure

*mfxStatus* **MFXVideoVPP\_RunFrameVPPAsync** (*mfxSession* *session*, *mfxFrameSurface1* \**in*,  
*mfxFrameSurface1* \**out*, *mfxExtVppAuxData* \**aux*,  
*mfxSyncPoint* \**syncp*)

This function processes a single input frame to a single output frame. Retrieval of the auxiliary data is optional; the encoding process may use it. The video processing process may not generate an instant output given an input. See section Video Processing Procedures for details on how to correctly send input and retrieve output. At the end of the stream, call this function with the input argument in=NULL to retrieve any remaining frames, until the function returns MFX\_ERR\_MORE\_DATA. This function is asynchronous.

**Return** MFX\_ERR\_NONE The output frame is ready after synchronization. MFX\_ERR\_MORE\_DATA Need more input frames before VPP can produce an output

MFX\_ERR\_MORE\_SURFACE The output frame is ready after synchronization. Need more surfaces at output for additional output frames available.

MFX\_ERR\_DEVICE\_LOST Hardware device was lost; See the Working with Microsoft® DirectX® Applications section for further information.

MFX\_WRN\_DEVICE\_BUSY Hardware device is currently busy. Call this function again in a few milliseconds.

## Parameters

- [in] session: SDK session handle.
- [in] in: Pointer to the input video surface structure
- [out] out: Pointer to the output video surface structure
- [in] aux: Optional pointer to the auxiliary data structure
- [out] syncp: Pointer to the output sync point

---

**Important:** The `MFXVideoVPP_RunFrameVPPAsync()` function is mandatory when implementing a VPP filter.

---

### 12.7.6.8 Adapters

`mfxStatus MFXQueryAdapters (mfxComponentInfo *input_info, mfxAdaptersInfo *adapters)`

This function returns a list of adapters that are suitable to handle workload `input_info`. The list is sorted in priority order, with iGPU given the highest precedence. This rule may change in the future. If `input_info` pointer is NULL, the list of all available adapters will be returned.

**Return** `MFX_ERR_NONE` The function completed successfully. `MFX_ERR_NULL_PTR` `input_info` or `adapters` pointer is NULL.

`MFX_ERR_NOT_FOUND` No suitable adapters found.

`MFX_WRN_OUT_OF_RANGE` Not enough memory to report back entire list of adapters. In this case as many adapters as possible will be returned.

## Parameters

- [in] `input_info`: Pointer to workload description. See `mfxComponentInfo` description for details.
- [out] `adapters`: Pointer to output description of all suitable adapters for input workload. See `mfxAdaptersInfo` description for details.

`mfxStatus MFXQueryAdaptersDecode (mfxBitstream *bitstream, mfxU32 codec_id, mfxAdaptersInfo *adapters)`

This function returns list of adapters that are suitable to decode input bitstream. The list is sorted in priority order, with iGPU given the highest precedence. This rule may change in the . This function is a simplification of `MFXQueryAdapters`, because bitstream is a description of the workload itself.

**Return** `MFX_ERR_NONE` The function completed successfully. `MFX_ERR_NULL_PTR` `bitstream` or `adapters` pointer is NULL.

`MFX_ERR_NOT_FOUND` No suitable adapters found.

`MFX_WRN_OUT_OF_RANGE` Not enough memory to report back entire list of adapters. In this case as many adapters as possible will be returned.

## Parameters

- [in] `bitstream`: Pointer to bitstream with input data.
- [in] `codec_id`: Codec ID to determine the type of codec for the input bitstream.

- [out] `adapters`: Pointer to the output list of adapters. Memory should be allocated by user. See [`mfxAdaptersInfo`](#) description for details.

*mfxStatus* **MFXQueryAdaptersNumber** (*mfxU32* \**num\_adapters*)

This function returns the number of detected graphics adapters. It can be used before calling MFX-QueryAdapters to determine the size of input data that the user will need to allocate.

**Return** MFX\_ERR\_NONE The function completed successfully. MFX\_ERR\_NULL\_PTR *num\_adapters* pointer is NULL.

#### Parameters

- [out] `num_adapters`: Pointer for the output number of detected graphics adapters.

## ONEMKL

The oneAPI Math Kernel Library (oneMKL) defines a set of fundamental mathematical routines for use in high-performance computing and other applications. As part of oneAPI, oneMKL is designed to allow execution on a wide variety of computational devices: CPUs, GPUs, FPGAs, and other accelerators. The functionality is subdivided into several domains: dense linear algebra, sparse linear algebra, discrete Fourier transforms, random number generators and vector math.

The general assumptions, design features and requirements for the oneMKL library and host-to-device computational routines will be described in [oneMKL Architecture](#). The individual domains and their APIs are described in [oneMKL Domains](#).

### 13.1 oneMKL Architecture

The oneMKL element of oneAPI has several general assumptions, requirements and recommendations for all domains contained therein. These will be addressed in this architecture section. In particular, DPC++ allows for a great control over the execution of kernels on the various devices. We discuss the supported execution models of oneMKL APIs in [Execution Model](#). A discussion of how data is stored and passed in and out of the APIs is addressed in [Memory Model](#). The general structure and design of oneMKL APIs including namespaces and common data types are expressed in [API Design](#). The exceptions and error handling are described in [Exceptions and Error Handling](#). Finally all the other necessary aspects related to oneMKL architecture can be found in [Other Features](#) including versioning and discussion of pre and post conditions.

#### 13.1.1 Execution Model

This section describes the execution environment common to all oneMKL functionality. The execution environment includes how data is provided to computational routines in [Use of Queues](#), support for several devices in [Device Usage](#), synchronous and asynchronous execution models in [Asynchronous Execution](#) and [Host Thread Safety](#).

##### 13.1.1.1 Use of Queues

The `sycl::queue` defined in the oneAPI DPC++ specification is used to specify the device and features enabled on that device on which a task will be enqueued. There are two forms of computational routines in oneMKL: class based [Member Functions](#) and standalone [Non-Member Functions](#). As these may interact with the `sycl::queue` in different ways, we provide a section for each one to describe assumptions.

### 13.1.1.1.1 Non-Member Functions

Each oneMKL non-member computational routine takes a `sycl::queue` reference as its first parameter:

```
mkl::domain::routine(sycl::queue &q, ...);
```

All computation performed by the routine shall be done on the hardware device(s) associated with this queue, with possible aid from the host, unless otherwise specified. In the case of an ordered queue, all computation shall also be ordered with respect to other kernels as if enqueued on that queue.

A particular oneMKL implementation may not support the execution of a given oneMKL routine on the specified device(s). In this case, the implementation may either perform the computation on the host or throw an exception. See *Exceptions and Error Handling* for the possible exceptions.

### 13.1.1.1.2 Member Functions

oneMKL class-based APIs, such as those in the RNG and DFT domains, require a `sycl::queue` as an argument to the constructor or another setup routine. The execution requirements for computational routines from the previous section also apply to computational class methods.

### 13.1.1.2 Device Usage

oneMKL itself does not currently provide any interfaces for controlling device usage: for instance, controlling the number of cores used on the CPU, or the number of execution units on a GPU. However, such functionality may be available by partitioning a `sycl::device` instance into subdevices, when supported by the device.

When given a queue associated with such a subdevice, a oneMKL implementation shall only perform computation on that subdevice.

### 13.1.1.3 Asynchronous Execution

The oneMKL API is designed to allow asynchronous execution of computational routines, to facilitate concurrent usage of multiple devices in the system. Each computational routine enqueues work to be performed on the selected device, and may (but is not required to) return before execution completes.

Hence, it is the calling application's responsibility to ensure that any inputs are valid until computation is complete, and likewise to wait for computation completion before reading any outputs. This can be done automatically when using DPC++ buffers, or manually when using Unified Shared Memory (USM) pointers, as described in the sections below.

Unless otherwise specified, asynchronous execution is *allowed*, but not *guaranteed*, by any oneMKL computational routine, and may vary between implementations and/or versions. oneMKL implementations must clearly document whether execution is guaranteed to be asynchronous for each supported routine.

### 13.1.1.3.1 Synchronization When Using Buffers

`sycl::buffer` objects automatically manage synchronization between kernel launches linked by a data dependency (either read-after-write, write-after-write, or write-after-read).

oneMKL routines are not required to perform any additional synchronization of `sycl::buffer` arguments.

### 13.1.1.3.2 Synchronization When Using USM APIs

When USM pointers are used as input to, or output from, a oneMKL routine, it becomes the calling application's responsibility to manage possible asynchronicity.

To help the calling application, all oneMKL routines with at least one USM pointer argument also take an optional reference to a list of *input events*, of type `sycl::vector_class<sycl::event>`, and have a return value of type `sycl::event` representing computation completion:

```
sycl::event mkl::domain::routine(..., sycl::vector_class<sycl::event> &in_events = {}  
    ↵);
```

The routine shall ensure that all input events (if the list is present and non-empty) have occurred before any USM pointers are accessed. Likewise, the routine's output event shall not be complete until the routine has finished accessing all USM pointer arguments.

For class methods, "argument" includes any USM pointers previously provided to the object via the class constructor or other class methods.

### 13.1.1.4 Host Thread Safety

All oneMKL member and non-member functions shall be *host thread safe*. That is, they may be safely called simultaneously from concurrent host threads. However, oneMKL objects in class-based APIs may not be shared between concurrent host threads unless otherwise specified.

## 13.1.2 Memory Model

The oneMKL memory model shall follow directly from the oneAPI memory model. Mainly, oneMKL shall support two modes of encapsulating data for consumption on the device: the buffer memory abstraction model and the pointer-based memory model using Unified Shared Memory (USM). These two paradigms shall also support both synchronous and asynchronous execution models as described in [Asynchronous Execution](#).

### 13.1.2.1 The Buffer Memory Model

The SYCL 1.2.1 specification defines the buffer container templated on the provided data type which encapsulates the data in a SYCL application across both host and devices. It provides the concept of accessors as the mechanism to access the buffer data with different modes to read and or write into that data. These accessors allow SYCL to create and manage the data dependencies in the SYCL graph that order the kernel executions. With the buffer model, all data movement is handled by the SYCL runtime supporting both synchronous and asynchronous execution.

oneMKL provides APIs where buffers (in particular 1D buffers, `sycl::buffer<T, 1>`) contain the memory for all non scalar input and output data arguments. See [Synchronization When Using Buffers](#) for details on how oneMKL routines manage any data dependencies with buffer arguments. Any higher dimensional buffer must be converted to a 1D buffer prior to use in oneMKL APIs, e.g., via `buffer::reinterpret`.

### 13.1.2.2 Unified Shared Memory Model

While the buffer model is powerful and elegantly expresses data dependencies, it can be a burden for programmers to replace all pointers and arrays by buffers in their C++ applications. DPC++ also provides pointer-based addressing for device-accessible data, using the Unified Shared Memory (USM) model. Correspondingly, oneMKL provides USM APIs in which non-scalar input and output data arguments are passed by USM pointer.

USM devices and system configurations vary in their ability to share data between devices and between a device and the host. oneMKL implementations may only assume that user-provided USM pointers are accessible by the device associated with the user-provided queue. In particular, an implementation must not assume that USM pointers can be accessed by any other device, or by the host, without querying the DPC++ runtime. An implementation must accept any device-accessible USM pointer regardless of how it was created (`sycl::malloc_device`, `sycl::malloc_shared`, etc.).

Unlike buffers, USM pointers cannot automatically manage data dependencies between kernels. Users may use in-order queues to ensure ordered execution, or explicitly manage dependencies with `sycl::event` objects. To support the second use case, oneMKL USM APIs accept input events (prerequisites before computation can begin) and return an output event (indicating computation is complete). See *Synchronization When Using USM APIs* for details.

### 13.1.3 API Design

This section discusses the general features of oneMKL API design. In particular, it covers the use of namespaces and data types from C++, from DPC++ and new ones introduced for oneMKL APIs.

#### 13.1.3.1 oneMKL namespaces

The oneMKL library uses C++ namespaces to organize routines by mathematical domain. All oneMKL objects and routines shall be contained within the `onemkl` base namespace. The individual oneMKL domains use a secondary namespace layer as follows:

namespace	oneMKL domain or content
<code>onemkl</code>	oneMKL base namespace, contains general oneMKL data types, objects, exceptions and routines
<code>onemkl::blas</code>	Dense linear algebra routines from BLAS and BLAS like extensions. See <a href="#">BLAS Routines</a>
<code>onemkl::lapack</code>	Dense linear algebra routines from LAPACK and LAPACK like extensions. See <a href="#">LAPACK Routines</a>
<code>onemkl::sparse</code>	Sparse linear algebra routines from Sparse BLAS and Sparse Solvers. See <a href="#">Sparse Linear Algebra</a>
<code>onemkl::dft</code>	Discrete and fast Fourier transformations. See <a href="#">Fourier Transform Functions</a>
<code>onemkl::rng</code>	Random number generator routines. See <a href="#">Random Number Generators</a>
<code>onemkl::vm</code>	Vector mathematics routines, e.g. trigonometric, exponential functions acting on elements of a vector. See <a href="#">Vector Math</a>

#### 13.1.3.2 Standard C++ datatype usage

oneMKL uses C++ STL data types for scalars where applicable:

- Integer scalars are C++ fixed-size integer types (`std::intN_t`, `std::uintN_t`).
- Complex numbers are represented by C++ `std::complex` types.

In general, scalar integer arguments to oneMKL routines are 64-bit integers (`std::int64_t` or `std::uint64_t`). Integer vectors and matrices may have varying bit widths, defined on a per-routine basis.

### 13.1.3.3 DPC++ datatype usage

oneMKL uses the following DPC++ data types:

- SYCL queue `sycl::queue` for scheduling kernels on a SYCL device. See [Use of Queues](#) for more details.
- SYCL buffer `sycl::buffer` for buffer-based memory access. See [The Buffer Memory Model](#) for more details.
- Unified Shared Memory (USM) for pointer-based memory access. See [Unified Shared Memory Model](#) for more details.
- SYCL event `sycl::event` for output event synchronization in oneMKL routines with USM pointers. See [Synchronization When Using USM APIs](#) for more details.
- Vector of SYCL events `sycl::vector<sycl::event>` for input events synchronization in oneMKL routines with USM pointers. See [Synchronization When Using USM APIs](#) for more details.

### 13.1.3.4 oneMKL defined datatypes

oneMKL dense and sparse linear algebra routines use scoped enum types as type-safe replacements for the traditional character arguments used in C/Fortran implementations of BLAS and LAPACK. These types all belong to the `onemkl` namespace.

Each enumeration value comes with two names: A single-character name (the traditional BLAS/LAPACK character) and a longer, more descriptive name. The two names are exactly equivalent and may be used interchangeably.

#### **transpose**

The `transpose` type specifies whether an input matrix should be transposed and/or conjugated. It can take the following values:

Short Name	Long Name	Description
<code>transpose::tn</code>	<code>transpose::nontrans</code>	Do not transpose or conjugate the matrix.
<code>transpose::tT</code>	<code>transpose::trans</code>	Transpose the matrix.
<code>transpose::tC</code>	<code>transpose::conj</code>	Perform Hermitian transpose (transpose and conjugate). Only applicable to complex matrices.

#### **uplo**

The `uplo` type specifies whether the lower or upper triangle of a triangular, symmetric, or Hermitian matrix should be accessed. It can take the following values:

Short Name	Long Name	Description
<code>uplo::U</code>	<code>uplo::upper</code>	Access the upper triangle of the matrix.
<code>uplo::L</code>	<code>uplo::lower</code>	Access the lower triangle of the matrix.

In both cases, elements that are not in the selected triangle are not accessed or updated.

**diag**

The `diag` type specifies the values on the diagonal of a triangular matrix. It can take the following values:

Short Name	Long Name	Description
<code>diag::N</code>	<code>diag::nonunit</code>	The matrix is not unit triangular. The diagonal entries are stored with the matrix data.
<code>diag::U</code>	<code>diag::unit</code>	The matrix is unit triangular (the diagonal entries are all 1's). The diagonal entries in the matrix data are not accessed.

**side**

The `side` type specifies the order of matrix multiplication when one matrix has a special form (triangular, symmetric, or Hermitian):

Short Name	Long Name	Description
<code>side::L</code>	<code>side::left</code>	The special form matrix is on the left in the multiplication.
<code>side::R</code>	<code>side::right</code>	The special form matrix is on the right in the multiplication.

**offset**

The `offset` type specifies whether the offset to apply to an output matrix is a fix offset, column offset or row offset. It can take the following values

Short Name	Long Name	Description
<code>offset::off</code>	<code>offset::fix</code>	The offset to apply to the output matrix is fix, all the inputs in the <code>C_offset</code> matrix has the same value given by the first element in the <code>co</code> array.
<code>offset::off</code>	<code>offset::col</code>	The offset to apply to the output matrix is a column offset, that is to say all the columns in the <code>C_offset</code> matrix are the same and given by the elements in the <code>co</code> array.
<code>offset::off</code>	<code>offset::row</code>	The offset to apply to the output matrix is a row offset, that is to say all the rows in the <code>C_offset</code> matrix are the same and given by the elements in the <code>co</code> array.

**index\_base**

The `index_base` type specifies how values in index arrays are interpreted. For instance, a sparse matrix stores nonzero values and the indices that they correspond to. The indices are traditionally provided in one of two forms: C/C++-style using zero-based indices, or Fortran-style using one-based indices. The `index_base` type can take the following values:

Name	Description
<code>index_base::z</code>	Index arrays for an input matrix are provided using zero-based (C/C++ style) index values. That is, indices start at 0.
<code>index_base::o</code>	Index arrays for an input matrix are provided using one-based (Fortran style) index values. That is, indices start at 1.

### 13.1.4 Exceptions and Error Handling

Will be added in a future version.

### 13.1.5 Other Features

This section covers all other features in the design of oneMKL architecture.

#### 13.1.5.1 oneMKL Specification Versioning Strategy

This oneMKL specification uses a MAJOR.MINOR.PATCH approach for its versioning strategy. The MAJOR count is updated when a significant new feature or domain is added or backward compatibility is broken. The MINOR count is updated when new APIs are changed or added or deleted, or language updated with relatively significant change to the meaning but without breaking backwards compatibility. The PATCH count is updated when wording, language or structure is updated without significant change to the meaning or interpretation.

#### 13.1.5.2 Current Version of this oneMKL Specification

This is the oneMKL specification version 0.8.0.

#### 13.1.5.3 Pre/Post Condition Checking

The individual oneMKL computational routines will define any preconditions and postconditions and will define in this specification any specific checks or verifications that should be enabled for all implementations.

## 13.2 oneMKL Domains

This section describes the Data Parallel C++ (DPC++) interface.

### 13.2.1 Dense Linear Algebra

This section contains information about dense linear algebra routines:

*Matrix Storage* provides information about dense matrix and vector storage formats that are used by oneMKL *BLAS Routines* and *LAPACK Routines*.

*BLAS Routines* provides vector, matrix-vector, and matrix-matrix routines for dense matrices and vector operations.

*LAPACK Routines* provides more complex dense linear algebra routines, e.g., matrix factorization, solving dense systems of linear equations, least square problems, eigenvalue and singular value problems, and performing a number of related computational tasks.

### 13.2.1.1 Matrix Storage

The oneMKL BLAS and LAPACK routines for DPC++ use several matrix and vector storage formats. These are the same formats used in traditional Fortran BLAS/LAPACK.

#### General Matrix

A general matrix  $A$  of  $m$  rows and  $n$  columns with leading dimension  $lda$  is represented as a one dimensional array  $a$  of size of at least  $lda * n$ . Before entry in any BLAS function using a general matrix, the leading  $m$  by  $n$  part of the array  $a$  must contain the matrix  $A$ . The elements of each column are contiguous in memory while the elements of each row are at distance  $lda$  from the element in the same row and the previous column.

Visually, the matrix

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & \cdots & A_{1n} \\ A_{21} & A_{22} & A_{23} & \cdots & A_{2n} \\ A_{31} & A_{32} & A_{33} & \cdots & A_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{m1} & A_{m2} & A_{m3} & \cdots & A_{mn} \end{bmatrix}$$

is stored in memory as an array

$$a = \underbrace{[A_{11}, A_{21}, A_{31}, \dots, A_{m1}, *, \dots, *, A_{12}, A_{22}, A_{32}, \dots, A_{m2}, *, \dots, *, \dots, A_{1n}, A_{2n}, A_{3n}, \dots, A_{mn}, *, \dots, *]}_{\underbrace{\text{lda}}_{\text{lda} \times n}}, \underbrace{\text{lda}}_{\text{lda}}, \underbrace{\text{lda}}_{\text{lda}}$$

#### Triangular Matrix

A triangular matrix  $A$  of  $n$  rows and  $n$  columns with leading dimension  $lda$  is represented as a one dimensional array  $a$ , of a size of at least  $lda * n$ . The elements of each column are contiguous in memory while the elements of each row are at distance  $lda$  from the element in the same row and the previous column.

Before entry in any BLAS function using a triangular matrix,

- If `upper_lower = uplo::upper`, the leading  $n$  by  $n$  upper triangular part of the array  $a$  must contain the upper triangular part of the matrix  $A$ . The strictly lower triangular part of the array  $a$  is not referenced. In other words, the matrix

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & \cdots & A_{1n} \\ * & A_{22} & A_{23} & \cdots & A_{2n} \\ * & * & A_{33} & \cdots & A_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ * & * & * & \cdots & A_{mn} \end{bmatrix}$$

is stored in memory as the array

$$a = [\underbrace{A_{11},*,\dots,*}_{\text{lda}}, \underbrace{A_{12},A_{22},*,\dots,*}_{\text{lda}}, \dots, \underbrace{A_{1n},A_{2n},A_{3n},\dots,A_{mn},*,\dots,*}_{\text{lda}}].$$

$\underbrace{\hspace{10em}}_{\text{lda} \times n}$

- If `upper_lower = uplo::lower`, the leading  $n$  by  $n$  lower triangular part of the array  $a$  must contain the lower triangular part of the matrix  $A$ . The strictly upper triangular part of the array  $a$  is not referenced. That is, the matrix

$$A = \begin{bmatrix} A_{11} & * & * & \cdots & * \\ A_{21} & A_{22} & * & \cdots & * \\ A_{31} & A_{32} & A_{33} & \cdots & * \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{m1} & A_{m2} & A_{m3} & \cdots & A_{mn} \end{bmatrix}$$

is stored in memory as the array

$$a = [\underbrace{A_{11},A_{21},A_{31},\dots,A_{m1},*,\dots,*}_{\text{lda}}, \underbrace{A_{22},A_{32},\dots,A_{m2},*,\dots,*}_{\text{lda}}, \dots, \underbrace{*,\dots*,A_{mn},*,\dots,*}_{\text{lda}}].$$

$\underbrace{\hspace{10em}}_{\text{lda} \times n}$

## Band Matrix

A general band matrix  $A$  of  $m$  rows and  $n$  columns with  $k_l$  sub-diagonals,  $k_u$  super-diagonals, and leading dimension  $\text{lda}$  is represented as a one dimensional array  $a$  of a size of at least  $\text{lda} * n$ .

Before entry in any BLAS function using a general band matrix, the leading  $(k_l + k_u + 1)$  by  $n$  part of the array  $a$  must contain the matrix  $A$ . This matrix must be supplied column-by-column, with the main diagonal of the matrix in row  $k_u$  of the array (0-based indexing), the first super-diagonal starting at position 1 in row  $(k_u - 1)$ , the first sub-diagonal starting at position 0 in row  $(k_u + 1)$ , and so on. Elements in the array  $a$  that do not correspond to elements in the band matrix (such as the top left  $k_u$  by  $k_u$  triangle) are not referenced.

Visually, the matrix  $A =$

$$A = \begin{array}{ccccccccc|c} A_{11} & A_{12} & A_{13} & \cdots & A_{1,ku+1} & * & \cdots & \cdots & \cdots & \cdots & * \\ A_{21} & A_{22} & A_{23} & A_{24} & \cdots & A_{2,ku+2} & * & \cdots & \cdots & \cdots & * \\ A_{31} & A_{32} & A_{33} & A_{34} & A_{35} & \cdots & A_{3,ku+3} & * & \cdots & \cdots & * \\ \vdots & & A_{42} & A_{43} & \ddots & \ddots & \ddots & \ddots & * & \cdots & \cdots \\ A_{kl+1,1} & \vdots & & A_{53} & \ddots \\ * & A_{kl+2,2} & \vdots & & \ddots \\ \vdots & * & A_{kl+3,3} & & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & * \\ \vdots & \vdots & * & & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & A_{n-ku,n} \\ \vdots & \vdots & \vdots & * & & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \vdots & \vdots & * & & \ddots & \ddots & \ddots & \ddots & A_{m-2,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & * & & \ddots & \ddots & \ddots & A_{m-1,n} \\ * & * & * & \cdots & \cdots & \cdots & * & A_{m,m-kl} & \cdots & A_{m,n-2} & A_{m,n-1} & A_{mn} \end{array}$$

is stored in memory as an array

The

following program segment transfers a band matrix from conventional full matrix storage (variable `matrix`, with leading dimension `ldm`) to band storage (variable `a`, with leading dimension `lda`):

```

for (j = 0; j < n; j++) {
    k = ku - j;
    for (i = max(0, j - ku); i < min(m, j + kl + 1); i++) {
        a[(k + i) + j * lda] = matrix[i + j * ldm];
    }
}

```

## Triangular Band Matrix

A triangular band matrix  $A$  of  $n$  rows and  $n$  columns with  $k$  sub/super-diagonals and leading dimension  $\text{lda}$  is represented as a one dimensional array  $a$  of size at least  $\text{lda} * n$ .

Before entry in any BLAS function using a triangular band matrix,

- If `upper_lower = uplo::upper`, the leading  $(k + 1)$  by  $n$  part of the array `a` must contain the upper triangular band part of the matrix `A`. This matrix must be supplied column-by-column with the main diagonal of the matrix in row  $(k)$  of the array, the first super-diagonal starting at position 1 in row  $(k - 1)$ , and so on. Elements in the array `a` that do not correspond to elements in the triangular band matrix (such as the top left  $k$  by  $k$  triangle) are not referenced.

Visually, the matrix

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & \cdots & A_{1,k+1} & * & \cdots & \cdots & \cdots & \cdots & \cdots & * \\ * & A_{22} & A_{23} & A_{24} & \cdots & A_{2,k+2} & * & \cdots & \cdots & \cdots & \cdots & * \\ \vdots & * & A_{33} & A_{34} & A_{35} & \cdots & A_{3,k+3} & * & \cdots & \cdots & \cdots & * \\ \vdots & \vdots & * & \ddots & \ddots & \vdots & \vdots & \ddots & * & \cdots & \cdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots & \vdots & \ddots & \vdots & \cdots & \cdots & \vdots \\ \vdots & \ddots & \vdots & * \\ \vdots & \ddots & \vdots & A_{n-k,n} \\ \vdots & \ddots & \vdots & A_{n-2,n} \\ \vdots & \ddots & \vdots & A_{n-1,n} \\ * & * & * & \cdots & * & A_{nn} \end{bmatrix}$$

is stored as an array

$$a = \underbrace{[* \dots *]}_{ku} \underbrace{, A_{11}, * \dots *}_{lda} \underbrace{, * \dots *}_{lda} \underbrace{, A_{\max(1,2-k),2}, \dots, A_{2,2}, * \dots *}_{lda} \dots \underbrace{, * \dots *}_{lda} \underbrace{, A_{\max(1,n-k),n}, \dots, A_{n,n}, * \dots *}_{lda}$$

$\underbrace{\qquad\qquad\qquad}_{lda \times n}$

The following program segment transfers a band matrix from conventional full matrix storage (variable `matrix`, with leading dimension `ldm`) to band storage (variable `a`, with leading dimension `lda`):

```
for (j = 0; j < n; j++) {
    m = k - j;
    for (i = max(0, j - k); i <= j; i++) {
        a[(m + i) + j * lda] = matrix[i + j * ldm];
    }
}
```

- If `upper_lower = uplo::lower`, the leading  $(k + 1)$  by  $n$  part of the array `a` must contain the upper triangular band part of the matrix `A`. This matrix must be supplied column-by-column with the main diagonal of the matrix in row 0 of the array, the first sub-diagonal starting at position 0 in row 1, and so on. Elements in the array `a` that do not correspond to elements in the triangular band matrix (such as the bottom right  $k$  by  $k$  triangle) are not referenced.

That is, the matrix

$$A = \begin{bmatrix} A_{11} & * & \dots & * \\ A_{21} & A_{22} & * & \dots & * \\ A_{31} & A_{32} & A_{33} & * & \dots & \dots & \dots & \dots & \dots & \dots & * \\ \vdots & A_{42} & A_{43} & \ddots & \ddots & \dots & \dots & \dots & \dots & \dots & \vdots \\ A_{k+1,1} & \vdots & A_{53} & \ddots & \ddots & \ddots & \dots & \dots & \dots & \dots & \vdots \\ * & A_{k+2,2} & \vdots & \ddots & \ddots & \ddots & \dots & \dots & \dots & \dots & \vdots \\ \vdots & * & A_{k+3,3} & \ddots & \ddots & \ddots & \ddots & \dots & \dots & \dots & \vdots \\ \vdots & \vdots & * & \ddots & \ddots & \ddots & \ddots & \ddots & \dots & \dots & \vdots \\ \vdots & \vdots & \vdots & * & \ddots & \ddots & \ddots & \ddots & \ddots & \dots & \vdots \\ \vdots & \vdots & \vdots & \vdots & * & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & * & \ddots & \ddots & \ddots & \ddots & * \\ * & * & * & \dots & \dots & \dots & * & A_{n,n-k} & \dots & A_{n,n-2} & A_{n,n-1} & A_{nn} \end{bmatrix}$$

is stored as the array

$$a = [\underbrace{A_{11}, A_{21}, \dots, A_{\min(k+1,n),1}, *}, \dots, *] \underbrace{, A_{2,2}, \dots, A_{\min(k+2,n),2}, *}, \dots, *], \dots, \underbrace{A_{n,n}, *}, \dots, *]$$

$\underbrace{\hspace{10cm}}$   
 $lda \times n$

The following program segment transfers a band matrix from conventional full matrix storage (variable `matrix`, with leading dimension `ldm`) to band storage (variable `a`, with leading dimension `lda`):

```
for (j = 0; j < n; j++) {
    m = -j;
    for (i = j; i < min(n, j + k + 1); i++) {
        a[(m + i) + j * lda] = matrix[i + j * ldm];
    }
}
```

## Packed Triangular Matrix

A triangular matrix  $A$  of  $n$  rows and  $n$  columns is represented in packed format as a one dimensional array  $a$  of size at least  $(n*(n + 1))/2$ . All elements in the upper or lower part of the matrix  $A$  are stored contiguously in the array  $a$ .

Before entry in any BLAS function using a triangular packed matrix,

- If `upper_lower = uplo::upper`, the first  $(n*(n + 1))/2$  elements in the array  $a$  must contain the upper triangular part of the matrix  $A$  packed sequentially, column by column so that  $a[0]$  contains  $A_{11}$ ,  $a[1]$  and  $a[2]$  contain  $A_{12}$  and  $A_{22}$  respectively, and so on. Hence, the matrix

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & \cdots & A_{1n} \\ * & A_{22} & A_{23} & \cdots & A_{2n} \\ * & * & A_{33} & \cdots & A_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ * & * & * & \cdots & A_{mn} \end{bmatrix}$$

is stored as the array

$$a = [A_{11}, A_{12}, A_{22}, A_{13}, A_{23}, A_{33}, \dots, A_{(m-1),n}, A_{mn}]$$

- If `upper_lower = uplo::lower`, the first  $(n*(n + 1))/2$  elements in the array `a` must contain the lower triangular part of the matrix `A` packed sequentially, column by column so that `a[0]` contains  $A_{11}$ , `a[1]` and `a[2]` contain  $A_{21}$  and  $A_{31}$  respectively, and so on. The matrix

$$A = \begin{bmatrix} A_{11} & * & * & \cdots & * \\ A_{21} & A_{22} & * & \cdots & * \\ A_{31} & A_{32} & A_{33} & \cdots & * \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{m1} & A_{m2} & A_{m3} & \cdots & A_{mn} \end{bmatrix}$$

is stored as the array

$$a = [A_{11}, A_{21}, A_{31}, \dots, A_{m1}, A_{22}, A_{32}, \dots, A_{m2}, \dots, A_{(m-1),(n-1)}, A_{m,(n-1)}, A_{mn}]$$

## Vector

A vector `X` of `n` elements with increment `incx` is represented as a one dimensional array `x` of size at least  $(1 + (n - 1) * \text{abs}(\text{incx}))$ .

Visually, the vector

$$X = (X_1 \ X_2 \ X_3 \ \cdots \ X_n)$$

is stored in memory as an array

$$X = [ \underbrace{X_1, * , \dots, *}_{incx}, \underbrace{X_2, * , \dots, *}_{incx}, \dots, \underbrace{X_{n-1}, * , \dots, *}_{incx}, X_n ] \text{ if } incx > 0$$

$1 + (n-1) \times incx$

$$X = [ \underbrace{X_n, * , \dots, *}_{incx}, \underbrace{X_{n-1}, * , \dots, *}_{incx}, \dots, \underbrace{X_2, * , \dots, *}_{incx}, X_1 ] \text{ if } incx < 0$$

$1 + (1-n) \times incx$

**Parent topic:** *Dense Linear Algebra*

### 13.2.1.2 BLAS Routines

oneMKL provides a DPC++ interface to the Basic Linear Algebra Subprograms (BLAS) routines, as well as several BLAS-like extension routines.

#### 13.2.1.2.1 BLAS Level 1 Routines

BLAS Level 1 includes routines which perform vector-vector operations as described in the following table.

Routines	Description
asum	Sum of vector magnitudes
axpy	Scalar-vector product
copy	Copy vector
dot	Dot product
sdsdot	Dot product with double precision
dotc	Dot product conjugated
dotu	Dot product unconjugated
nrm2	Vector 2-norm (Euclidean norm)
rot	Plane rotation of points
rotg	Generate Givens rotation of points
rotm	Modified Givens plane rotation of points
rotmg	Generate modified Givens plane rotation of points
scal	Vector-scalar product
swap	Vector-vector swap
iamax	Index of the maximum absolute value element of a vector
imin	Index of the minimum absolute value element of a vector

### 13.2.1.2.1.1 **asum**

Computes the sum of magnitudes of the vector elements.

**asum** supports the following precisions.

T	T_res
float	float
double	double
<code>std::complex&lt;float&gt;</code>	float
<code>std::complex&lt;double&gt;</code>	double

#### Description

The **asum** routine computes the sum of the magnitudes of elements of a real vector, or the sum of magnitudes of the real and imaginary parts of elements of a complex vector:

$$result = \sum_{i=1}^n \left( \left| \operatorname{Re}(X_i) \right| + \left| \operatorname{Im}(X_i) \right| \right)^{\square}$$

where  $x$  is a vector with  $n$  elements.

### 13.2.1.2.1.2 **asum** (Buffer Version)

#### Syntax

```
void onemkl::blas::asum(sycl::queue &queue, std::int64_t n, sycl::buffer<T, 1> &x, std::int64_t incx,
                        sycl::buffer<T_res, 1> &result)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**n** Number of elements in vector  $x$ .

**x** Buffer holding input vector  $x$ . The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(incx))$ . See Matrix and Vector Storage for more details.

**incx** Stride of vector  $x$ .

#### Output Parameters

**result** Buffer where the scalar result is stored (the sum of magnitudes of the real and imaginary parts of all elements of the vector).

### 13.2.1.2.1.3 `asum` (USM Version)

#### Syntax

```
sycl::event onemkl::blas::asum(sycl::queue &queue, std::int64_t n, const T *x, std::int64_t incx,
                                T_res *result, const sycl::vector_class<sycl::event> &dependencies
                                = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**n** Number of elements in vector **x**.

**x** Pointer to input vector **x**. The array holding the vector **x** must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector **x**.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

#### Output Parameters

**result** Pointer to the output matrix where the scalar result is stored (the sum of magnitudes of the real and imaginary parts of all elements of the vector).

#### Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** [BLAS Level 1 Routines](#)

### 13.2.1.2.1.4 `axpy`

Computes a vector-scalar product and adds the result to a vector.

`axpy` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The axpy routines compute a scalar-vector product and add the result to a vector:

$y \leftarrow \alpha * x + y$

where:

$x$  and  $y$  are vectors of  $n$  elements,

$\alpha$  is a scalar.

### 13.2.1.2.1.5 axpy (Buffer Version)

#### Syntax

```
void onemkl::blas::axpy(sycl::queue &queue, std::int64_t n, T alpha, sycl::buffer<T, 1> &x,
std::int64_t incx, sycl::buffer<T, 1> &y, std::int64_t incy)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**n** Number of elements in vector  $x$ .

**alpha** Specifies the scalar  $\alpha$ .

**x** Buffer holding input vector  $x$ . The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(incx))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector  $x$ .

**y** Buffer holding input vector  $y$ . The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(incy))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector  $y$ .

#### Output Parameters

**y** Buffer holding the updated vector  $y$ .

### 13.2.1.2.1.6 axpy (USM Version)

#### Syntax

```
sycl::event onemkl::blas::axpy(sycl::queue &queue, std::int64_t n, T alpha, const T *x, std::int64_t
incx, T *y, std::int64_t incy, const sycl::vector_class<sycl::event>
&dependencies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**n** Number of elements in vector  $x$ .

**alpha** Specifies the scalar alpha.

**x** Pointer to the input vector  $x$ . The array holding the vector  $x$  must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector  $x$ .

**y** Pointer to the input vector  $y$ . The array holding the vector  $y$  must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incy}))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector  $y$ .

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**y** Pointer to the updated vector  $y$ .

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** [BLAS Level 1 Routines](#)

### 13.2.1.2.1.7 copy

Copies a vector to another vector.

copy supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The copy routines copy one vector to another:

$$y \leftarrow x$$

where  $x$  and  $y$  are vectors of  $n$  elements.

### 13.2.1.2.1.8 copy (Buffer Version)

#### Syntax

```
void onemkl::blas::copy(sycl::queue &queue, std::int64_t n, sycl::buffer<T, 1> &x, std::int64_t incx,
                       sycl::buffer<T, 1> &y, std::int64_t incy)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**n** Number of elements in vector **x**.

**x** Buffer holding input vector **x**. The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(incx))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector **x**.

**incy** Stride of vector **y**.

#### Output Parameters

**y** Buffer holding the updated vector **y**.

### 13.2.1.2.1.9 copy (USM Version)

#### Syntax

```
sycl::event onemkl::blas::copy(sycl::queue &queue, std::int64_t n, const T *x, std::int64_t incx, T
                                *y, std::int64_t incy, const sycl::vector_class<sycl::event> &dependencies = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**n** Number of elements in vector **x**.

**x** Pointer to the input vector **x**. The array holding the vector **x** must be of size at least  $(1 + (n - 1) * \text{abs}(incx))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector **x**.

**incy** Stride of vector **y**.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**y** Pointer to the updated vector  $y$ .

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 1 Routines*

### 13.2.1.2.1.10 dot

Computes the dot product of two real vectors.

dot supports the following precisions.

T	T_res
float	float
double	double
float	double

## Description

The dot routines perform a dot product between two vectors:

$$result = \sum_{i=1}^n X_i Y_i$$

## Note

For the mixed precision version (inputs are float while result is double), the dot product is computed with double precision.

### 13.2.1.2.1.11 dot (Buffer Version)

## Syntax

```
void onemkl::blas::dot(sycl::queue &queue, std::int64_t n, sycl::buffer<T, 1> &x, std::int64_t incx,
                      sycl::buffer<T, 1> &y, std::int64_t incy, sycl::buffer<T_res, 1> &result)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**n** Number of elements in vectors **x** and **y**.

**x** Buffer holding input vector **x**. The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector **x**.

**y** Buffer holding input vector **y**. The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incy}))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector **y**.

## Output Parameters

**result** Buffer where the result (a scalar) will be stored.

### 13.2.1.2.1.12 dot (USM Version)

#### Syntax

```
sycl::event onemkl::blas::dot(sycl::queue &queue, std::int64_t n, const T *x, std::int64_t
                               incx, const T *y, std::int64_t incy, T_res *result, const
                               sycl::vector_class<sycl::event> &dependencies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**n** Number of elements in vectors **x** and **y**.

**x** Pointer to the input vector **x**. The array holding the vector **x** must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector **x**.

**y** Pointer to the input vector **y**. The array holding the vector **y** must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incy}))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector **y**.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**result** Pointer to where the result (a scalar) will be stored.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 1 Routines*

### 13.2.1.2.1.13 `sdsdot`

Computes a vector-vector dot product with double precision.

#### Description

The `sdsdot` routines perform a dot product between two vectors with double precision:

$$\text{result} = sb + \sum_{i=1}^n X_i Y_i$$

### 13.2.1.2.1.14 `sdsdot` (Buffer Version)

#### Syntax

```
void onemkl::blas::sdsdot (sycl::queue &queue, std::int64_t n, float sb, sycl::buffer<float, 1>
    &x, std::int64_t incx, sycl::buffer<float, 1> &y, std::int64_t incy,
    sycl::buffer<float, 1> &result)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**n** Number of elements in vectors **x** and **y**.

**sb** Single precision scalar to be added to the dot product.

**x** Buffer holding input vector **x**. The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(incx))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector **x**.

**y** Buffer holding input vector **y**. The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(incxy))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector **y**.

## Output Parameters

**result** Buffer where the result (a scalar) will be stored. If  $n < 0$  the result is  $sb$ .

### 13.2.1.2.1.15 `sdsdot` (USM Version)

#### Syntax

```
sycl::event onemkl::blas::sdsdot(sycl::queue &queue, std::int64_t n, float sb, const float *x,  
std::int64_t incx, const float *y, std::int64_t incy, float *result,  
const sycl::vector_class<sycl::event> &dependencies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**n** Number of elements in vectors **x** and **y**.

**sb** Single precision scalar to be added to the dot product.

**x** Pointer to the input vector **x**. The array must be of size at least  $(1 + (n - 1)*\text{abs}(\text{incx}))$ . See Matrix and Vector Storage for more details.

**incx** Stride of vector **x**.

**y** Pointer to the input vector **y**. The array must be of size at least  $(1 + (n - 1)*\text{abs}(\text{incy}))$ . See Matrix and Vector Storage for more details.

**incy** Stride of vector **y**.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**result** Pointer to where the result (a scalar) will be stored. If  $n < 0$  the result is  $sb$ .

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 1 Routines*

### 13.2.1.2.1.16 `dotc`

Computes the dot product of two complex vectors, conjugating the first vector.

`dotc` supports the following precisions.

<b>T</b>
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The `dotc` routines perform a dot product between two complex vectors, conjugating the first of them:

$$result = \sum_{i=1}^n \overline{X}_l Y_i$$

### 13.2.1.2.1.17 `dotc` (Buffer Version)

#### Syntax

```
void onemkl::blas::dotc(sycl::queue &queue, std::int64_t n, sycl::buffer<T, 1> &x, std::int64_t incx,
                        sycl::buffer<T, 1> &y, std::int64_t incy, sycl::buffer<T, 1> &result)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**n** The number of elements in vectors **x** and **y**.

**x** Buffer holding input vector **x**. The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(incx))$ . See [Matrix and Vector Storage](#) for more details.

**incx** The stride of vector **x**.

**y** Buffer holding input vector **y**. The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(incy))$ . See [Matrix and Vector Storage](#) for more details..

**incy** The stride of vector **y**.

#### Output Parameters

**result** The buffer where the result (a scalar) is stored.

### 13.2.1.2.1.18 `dotc` (USM Version)

#### Syntax

```
void onemkl::blas::dotc(sycl::queue &queue, std::int64_t n, const T *x, std::int64_t incx, const T
                        *y, std::int64_t incy, T *result, const sycl::vector_class<sycl::event> &de-
                        pendencies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**n** The number of elements in vectors  $x$  and  $y$ .

**x** Pointer to input vector  $x$ . The array holding the input vector  $x$  must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** The stride of vector  $x$ .

**y** Pointer to input vector  $y$ . The array holding the input vector  $y$  must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incy}))$ . See [Matrix and Vector Storage](#) for more details..

**incy** The stride of vector  $y$ .

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**result** The pointer to where the result (a scalar) is stored.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** [BLAS Level 1 Routines](#)

### 13.2.1.2.1.19 dotu

Computes the dot product of two complex vectors.

dotu supports the following precisions.

T
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The dotu routines perform a dot product between two complex vectors:

$$\text{result} = \sum_{i=1}^n X_i Y_i$$

### 13.2.1.2.1.20 dotu (Buffer Version)

#### Syntax

```
void onemkl::blas::dotu(sycl::queue &queue, std::int64_t n, sycl::buffer<T, 1> &x, std::int64_t incx,
                        sycl::buffer<T, 1> &y, std::int64_t incy, sycl::buffer<T, 1> &result)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**n** Number of elements in vectors **x** and **y**.

**x** Buffer holding input vector **x**. The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector **x**.

**y** Buffer holding input vector **y**. The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incy}))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector **y**.

#### Output Parameters

**result** Buffer where the result (a scalar) is stored.

### 13.2.1.2.1.21 dotu (USM Version)

#### Syntax

```
sycl::event onemkl::blas::dotu(sycl::queue &queue, std::int64_t n, const T *x, std::int64_t
                                incx, const T *y, std::int64_t incy, T *result, const
                                sycl::vector_class<sycl::event> &dependencies = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**n** Number of elements in vectors **x** and **y**.

**x** Pointer to the input vector **x**. The array holding input vector **x** must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector **x**.

**y** Pointer to input vector **y**. The array holding input vector **y** must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incy}))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector **y**.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**result** Pointer to where the result (a scalar) is stored.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 1 Routines*

### 13.2.1.2.1.22 nrm2

Computes the Euclidean norm of a vector.

nrm2 supports the following precisions.

T	T_res
float	float
double	double
std::complex<float>	float
std::complex<double>	double

## Description

The nrm2 routines computes Euclidean norm of a vector

$$\text{result} = \|\mathbf{x}\|,$$

where:

$\mathbf{x}$  is a vector of  $n$  elements.

### 13.2.1.2.1.23 nrm2 (Buffer Version)

## Syntax

```
void onemkl::blas::nrm2(sycl::queue &queue, std::int64_t n, sycl::buffer<T, 1> &x, std::int64_t incx,
                        sycl::buffer<T_res, 1> &result)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**n** Number of elements in vector  $\mathbf{x}$ .

**x** Buffer holding input vector  $\mathbf{x}$ . The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incx}))$ . See Matrix and Vector Storage for more details.

**incx** Stride of vector  $\mathbf{x}$ .

## Output Parameters

**result** Buffer where the Euclidean norm of the vector  $x$  will be stored.

### 13.2.1.2.1.24 nrm2 (USM Version)

#### Syntax

```
sycl::event onemkl::blas::nrm2(sycl::queue &queue, std::int64_t n, const T *x, std::int64_t incx,
                                T_res *result, const sycl::vector_class<sycl::event> &dependencies
                                = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**n** Number of elements in vector  $x$ .

**x** Pointer to input vector  $x$ . The array holding input vector  $x$  must be of size at least  $(1 + (n - 1) * \text{abs}(incx))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector  $x$ .

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**result** Pointer to where the Euclidean norm of the vector  $x$  will be stored.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** [BLAS Level 1 Routines](#)

### 13.2.1.2.1.25 rot

Performs rotation of points in the plane.

**rot** supports the following precisions.

T	T_scalar
float	float
double	double
std::complex<float>	float
std::complex<double>	double

## Description

Given two vectors  $x$  and  $y$  of  $n$  elements, the `rot` routines compute four scalar-vector products and update the input vectors with the sum of two of these scalar-vector products as follow:

$$x \leftarrow c*x + s*y$$

$$y \leftarrow c*y - s*x$$

### 13.2.1.2.1.26 rot (Buffer Version)

#### Syntax

```
void onemkl::blas::rot(sycl::queue &queue, std::int64_t n, sycl::buffer<T, 1> &x, std::int64_t incx,
                      sycl::buffer<T, 1> &y, std::int64_t incy, T_scalar c, T_scalar s)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**n** Number of elements in vector  $x$ .

**x** Buffer holding input vector  $x$ . The buffer must be of size at least  $(1 + (n - 1)*\text{abs}(incx))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector  $x$ .

**y** Buffer holding input vector  $y$ . The buffer must be of size at least  $(1 + (n - 1)*\text{abs}(incy))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector  $y$ .

**c** Scaling factor.

**s** Scaling factor.

#### Output Parameters

**x** Buffer holding updated buffer  $x$ .

**y** Buffer holding updated buffer  $y$ .

### 13.2.1.2.1.27 rot (USM Version)

#### Syntax

```
sycl::event onemkl::blas::rot(sycl::queue &queue, std::int64_t n, T *x, std::int64_t incx,
                               T *y, std::int64_t incy, T_scalar c, T_scalar s, const
                               sycl::vector_class<sycl::event> &dependencies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**n** Number of elements in vector  $x$ .

**x** Pointer to input vector  $x$ . The array holding input vector  $x$  must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector  $x$ .

**y** Pointer to input vector  $y$ . The array holding input vector  $y$  must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incy}))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector  $y$ .

**c** Scaling factor.

**s** Scaling factor.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**x** Pointer to the updated matrix  $x$ .

**y** Pointer to the updated matrix  $y$ .

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** [BLAS Level 1 Routines](#)

### 13.2.1.2.1.28 rotg

Computes the parameters for a Givens rotation.

`rotg` supports the following precisions.

T	T_res
float	float
double	double
<code>std::complex&lt;float&gt;</code>	float
<code>std::complex&lt;double&gt;</code>	double

## Description

Given the Cartesian coordinates  $(a, b)$  of a point, the `rotg` routines return the parameters  $c$ ,  $s$ ,  $r$ , and  $z$  associated with the Givens rotation. The parameters  $c$  and  $s$  define a unitary matrix such that:

The parameter  $z$  is defined such that if  $|a| > |b|$ ,  $z$  is  $s$ ; otherwise if  $c$  is not 0  $z$  is  $1/c$ ; otherwise  $z$  is 1.

### 13.2.1.2.1.29 rotg (Buffer Version)

#### Syntax

```
void onemkl::blas::rotg(sycl::queue &queue, sycl::buffer<T, 1> &a, sycl::buffer<T, 1> &b,
                        sycl::buffer<T_real, 1> &c, sycl::buffer<T, 1> &s)
```

#### Input Parameters

**queue** The queue where the routine should be executed

**a** Buffer holding the x-coordinate of the point.

**b** Buffer holding the y-coordinate of the point.

#### Output Parameters

**a** Buffer holding the parameter *r* associated with the Givens rotation.

**b** Buffer holding the parameter *z* associated with the Givens rotation.

**c** Buffer holding the parameter *c* associated with the Givens rotation.

**s** Buffer holding the parameter *s* associated with the Givens rotation.

### 13.2.1.2.1.30 rotg (USM Version)

#### Syntax

```
sycl::event onemkl::blas::rotg(sycl::queue &queue, T *a, T *b, T_real *c, T *s, const
                                sycl::vector_class<sycl::event> &dependencies = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed

**a** Pointer to the x-coordinate of the point.

**b** Pointer to the y-coordinate of the point.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

#### Output Parameters

**a** Pointer to the parameter *r* associated with the Givens rotation.

**b** Pointer to the parameter *z* associated with the Givens rotation.

**c** Pointer to the parameter *c* associated with the Givens rotation.

**s** Pointer to the parameter *s* associated with the Givens rotation.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 1 Routines*

### 13.2.1.2.1.31 rotm

Performs modified Givens rotation of points in the plane.

`rotm` supports the following precisions.

T
float
double

## Description

Given two vectors  $x$  and  $y$ , each vector element of these vectors is replaced as follows:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} = H \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

for  $i$  from 1 to  $n$ , where  $H$  is a modified Givens transformation matrix.

### 13.2.1.2.1.32 rotm (Buffer Version)

#### Syntax

```
void onemkl::blas::rotm(sycl::queue &queue, std::int64_t n, sycl::buffer<T, 1> &x, std::int64_t incx,
                        sycl::buffer<T, 1> &y, std::int64_t incy, sycl::buffer<T, 1> &param)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**n** Number of elements in vector  $x$ .

**x** Buffer holding input vector  $x$ . The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(incx))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector  $x$ .

**y** Buffer holding input vector  $y$ . The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(incy))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector  $y$ .

**param** Buffer holding an array of size 5. The elements of the `param` array are:

`param[0]` contains a switch, `flag`,

`param[1-4]` contain  $h_{11}$ ,  $h_{21}$ ,  $h_{12}$ , and  $h_{22}$  respectively, the components of the modified Givens transformation matrix  $H$ .

Depending on the values of `flag`, the components of  $H$  are set as follows:

`flag == -1.0:`

$$H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$$

`flag == 0.0:`

$$H = \begin{bmatrix} 1.0 & h_{12} \\ h_{21} & 1.0 \end{bmatrix}$$

`flag == 1.0:`

$$H = \begin{bmatrix} h_{11} & 1.0 \\ -1.0 & h_{22} \end{bmatrix}$$

`flag == -2.0:`

$$H = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}$$

In the last three cases, the matrix entries of 1.0, -1.0, 0.0 are assumed based on the value of `flag` and are not required to be set in the `param` vector.

## Output Parameters

**x** Buffer holding updated buffer `x`.

**y** Buffer holding updated buffer `y`.

### 13.2.1.2.1.33 rotm (USM Version)

#### Syntax

```
sycl::event onemkl::blas::rotm(sycl::queue &queue, std::int64_t n, T *x, std::int64_t incx, T *y,
                                std::int64_t incy, T *param, const sycl::vector_class<sycl::event>
                                &dependencies = { })
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**n** Number of elements in vector **x**.

**x** Pointer to the input vector **x**. The array holding the vector **x** must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector **x**.

**yparam** Pointer to the input vector **y**. The array holding the vector **y** must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incy}))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector **y**.

**param** Pointer to an array of size 5. The elements of the **param** array are:

**param[0]** contains a switch, **flag**,

**param[1-4]** contain  $h_{11}$ ,  $h_{21}$ ,  $h_{12}$ , and  $h_{22}$  respectively, the components of the modified Givens transformation matrix **H**.

Depending on the values of **flag**, the components of **H** are set as follows:

**flag** = -1.0:

$$H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$$

**flag** = 0.0:

$$H = \begin{bmatrix} 1.0 & h_{12} \\ h_{21} & 1.0 \end{bmatrix}$$

**flag** = 1.0:

$$H = \begin{bmatrix} h_{11} & 1.0 \\ -1.0 & h_{22} \end{bmatrix}$$

`flag = -2.0:`

$$H = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}$$

In the last three cases, the matrix entries of 1.0, -1.0, 0.0 are assumed based on the value of `flag` and are not required to be set in the `param` vector.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**x** Pointer to the updated array `x`.

**y** Pointer to the updated array `y`.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 1 Routines*

### 13.2.1.2.1.34 rotmg

Computes the parameters for a modified Givens rotation.

`rotmg` supports the following precisions.

T
float
double

## Description

Given Cartesian coordinates ( $x_1, y_1$ ) of an input vector, the `rotmg` routines compute the components of a modified Givens transformation matrix  $H$  that zeros the y-component of the resulting vector:

$$\begin{bmatrix} x_1 \\ 0 \end{bmatrix} = H \begin{bmatrix} x_1 \sqrt{d_1} \\ y_1 \sqrt{d_2} \end{bmatrix}$$

### 13.2.1.2.1.35 rotmg (Buffer Version)

#### Syntax

```
void onemkl::blas::rotmg(sycl::queue &queue, sycl::buffer<T, 1> &d1, sycl::buffer<T, 1> &d2,
                         sycl::buffer<T, 1> &x1, sycl::buffer<T, 1> &y1, sycl::buffer<T, 1> &param)
```

#### Input Parameters

- queue** The queue where the routine should be executed.
- d1** Buffer holding the scaling factor for the x-coordinate of the input vector.
- d2** Buffer holding the scaling factor for the y-coordinate of the input vector.
- x1** Buffer holding the x-coordinate of the input vector.
- y1** Scalar specifying the y-coordinate of the input vector.

#### Output Parameters

- d1** Buffer holding the first diagonal element of the updated matrix.
- d2** Buffer holding the second diagonal element of the updated matrix.
- x1** Buffer holding the x-coordinate of the rotated vector before scaling
- param** Buffer holding an array of size 5.

The elements of the **param** array are:

**param[0]** contains a switch, **flag**. the other array elements **param[1-4]** contain the components of the array **H**:  $h_{11}$ ,  $h_{21}$ ,  $h_{12}$ , and  $h_{22}$ , respectively.

Depending on the values of **flag**, the components of **H** are set as follows:

**flag = -1.0:**

$$H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$$

**flag = 0.0:**

$$H = \begin{bmatrix} 1.0 & h_{12} \\ h_{21} & 1.0 \end{bmatrix}$$

**flag = 1.0:**

$$H = \begin{bmatrix} h_{11} & 1.0 \\ -1.0 & h_{22} \end{bmatrix}$$

`flag == -2.0:`

$$H = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}$$

In the last three cases, the matrix entries of 1.0, -1.0, and 0.0 are assumed based on the value of `flag` and are not required to be set in the `param` vector.

### 13.2.1.2.1.36 `rotmg` (USM Version)

#### Syntax

```
sycl::event onemkl::blas::rotmg (sycl::queue &queue, T *d1, T *d2, T *x1, T *y1, T *param, const
                           sycl::vector_class<sycl::event> &dependencies = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**d1** Pointer to the scaling factor for the x-coordinate of the input vector.

**d2** Pointer to the scaling factor for the y-coordinate of the input vector.

**x1** Pointer to the x-coordinate of the input vector.

**y1** Scalar specifying the y-coordinate of the input vector.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

#### Output Parameters

**d1** Pointer to the first diagonal element of the updated matrix.

**d2** Pointer to the second diagonal element of the updated matrix.

**x1** Pointer to the x-coordinate of the rotated vector before scaling

**param** Pointer to an array of size 5.

The elements of the `param` array are:

`param[0]` contains a switch, `flag`. The other array elements `param[1–4]` contain the components of the array `H`:  $h_{11}$ ,  $h_{21}$ ,  $h_{12}$ , and  $h_{22}$ , respectively.

Depending on the values of `flag`, the components of `H` are set as follows:

`flag == -1.0:`

$$H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$$

`flag = 0.0:`

$$H = \begin{bmatrix} 1.0 & h_{12} \\ h_{21} & 1.0 \end{bmatrix}$$

`flag = 1.0:`

$$H = \begin{bmatrix} h_{11} & 1.0 \\ -1.0 & h_{22} \end{bmatrix}$$

`flag = -2.0:`

$$H = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}$$

In the last three cases, the matrix entries of 1.0, -1.0, and 0.0 are assumed based on the value of `flag` and are not required to be set in the `param` vector.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 1 Routines*

### 13.2.1.2.1.37 scal

Computes the product of a vector by a scalar.

`scal` supports the following precisions.

T	T_scalar
<code>float</code>	<code>float</code>
<code>double</code>	<code>double</code>
<code>std::complex&lt;float&gt;</code>	<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>	<code>std::complex&lt;double&gt;</code>
<code>std::complex&lt;float&gt;</code>	<code>float</code>
<code>std::complex&lt;double&gt;</code>	<code>double</code>

## Description

The `scal` routines computes a scalar-vector product:

$$x \leftarrow \text{alpha} \cdot x$$

where:

`x` is a vector of `n` elements,

`alpha` is a scalar.

### 13.2.1.2.1.38 scal (Buffer Version)

#### Syntax

```
void onemkl::blas::scal(sycl::queue &queue, std::int64_t n, T_scalar alpha, sycl::buffer<T, 1> &x,
std::int64_t incx)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**n** Number of elements in vector `x`.

**alpha** Specifies the scalar `alpha`.

**x** Buffer holding input vector `x`. The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(incx))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector `x`.

#### Output Parameters

**x** Buffer holding updated buffer `x`.

### 13.2.1.2.1.39 scal (USM Version)

#### Syntax

```
sycl::event onemkl::blas::scal(sycl::queue &queue, std::int64_t n, T_scalar alpha, T *x, std::int64_t
incx, const sycl::vector_class<sycl::event> &dependencies = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**n** Number of elements in vector `x`.

**alpha** Specifies the scalar `alpha`.

**x** Pointer to the input vector `x`. The array must be of size at least  $(1 + (n - 1) * \text{abs}(incx))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector `x`.

## Output Parameters

**x** Pointer to the updated array **x**.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 1 Routines*

### 13.2.1.2.1.40 swap

Swaps a vector with another vector.

swap supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

Given two vectors of **n** elements, **x** and **y**, the swap routines return vectors **y** and **x** swapped, each replacing the other.

$y \leftarrow x, x \leftarrow y$

### 13.2.1.2.1.41 swap (Buffer Version)

## Syntax

```
void onemkl::blas::swap(sycl::queue &queue, std::int64_t n, sycl::buffer<T, 1> &x, std::int64_t incx,
                        sycl::buffer<T, 1> &y, std::int64_t incy)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**n** Number of elements in vector **x**.

**x** Buffer holding input vector **x**. The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(incx))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector **x**.

**y** Buffer holding input vector **y**. The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(incy))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector **y**.

## Output Parameters

- x** Buffer holding updated buffer **x**, that is, the input vector **y**.
- y** Buffer holding updated buffer **y**, that is, the input vector **x**.

### 13.2.1.2.1.42 swap (USM Version)

#### Syntax

```
sycl::event onemkl::blas::swap(sycl::queue &queue, std::int64_t n, T *x, std::int64_t incx, T *y,
                               std::int64_t incy, const sycl::vector_class<sycl::event> &dependencies = {})
```

## Input Parameters

- queue** The queue where the routine should be executed.
- n** Number of elements in vector **x**.
- x** Pointer to the input vector **x**. The array must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incx}))$ . See Matrix and Vector Storage for more details.
- incx** Stride of vector **x**.
- y** Pointer to the input vector **y**. The array must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incy}))$ . See Matrix and Vector Storage for more details.
- incy** Stride of vector **y**.
- dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

- x** Pointer to the updated array **x**, that is, the input vector **y**.
- y** Pointer to the updated array **y**, that is, the input vector **x**.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 1 Routines*

### 13.2.1.2.1.43 iamax

Finds the index of the element with the largest absolute value in a vector.

**iamax** supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

## Description

The `iamax` routines return an index  $i$  such that  $x[i]$  has the maximum absolute value of all elements in vector  $x$  (real variants), or such that  $|\operatorname{Re}(x[i])| + |\operatorname{Im}(x[i])|$  is maximal (complex variants).

## Note

The index is zero-based.

If either  $n$  or  $\text{inc}_x$  are not positive, the routine returns 0.

If more than one vector element is found with the same largest absolute value, the index of the first one encountered is returned.

If the vector contains NaN values, then the routine returns the index of the first NaN.

### 13.2.1.2.1.44 `iamax` (Buffer Version)

#### Syntax

```
void onemkl::blas::iamax(sycl::queue &queue, std::int64_t n, sycl::buffer<T, 1> &x, std::int64_t incx,
                           sycl::buffer<std::int64_t, 1> &result)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**n** The number of elements in vector  $x$ .

**x** The buffer that holds the input vector  $x$ . The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(\text{inc}_x))$ . See [Matrix and Vector Storage](#) for more details.

**incx** The stride of vector  $x$ .

#### Output Parameters

**result** The buffer where the zero-based index  $i$  of the maximal element is stored.

### 13.2.1.2.1.45 `iamax` (USM Version)

#### Syntax

```
sycl::event onemkl::blas::iamax(sycl::queue &queue, std::int64_t n, const T *x, std::int64_t incx,
                               T_res *result, const sycl::vector_class<sycl::event> &dependencies
                               = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**n** The number of elements in vector  $x$ .

**x** The pointer to the input vector  $x$ . The array holding the input vector  $x$  must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** The stride of vector  $x$ .

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**result** The pointer to where the zero-based index  $i$  of the maximal element is stored.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** [BLAS Level 1 Routines](#)

### 13.2.1.2.1.46 `iamin`

Finds the index of the element with the smallest absolute value.

`iamin` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The `iamin` routines return an index  $i$  such that  $x[i]$  has the minimum absolute value of all elements in vector  $x$  (real variants), or such that  $|\text{Re}(x[i])| + |\text{Im}(x[i])|$  is maximal (complex variants).

## Note

The index is zero-based.

If either  $n$  or  $\text{incx}$  are not positive, the routine returns 0.

If more than one vector element is found with the same smallest absolute value, the index of the first one encountered is returned.

If the vector contains NaN values, then the routine returns the index of the first NaN.

### 13.2.1.2.1.47 `iamin` (Buffer Version)

#### Syntax

```
void onemkl::blas::iamin(sycl::queue &queue, std::int64_t n, sycl::buffer<T, 1> &x, std::int64_t incx,
                           sycl::buffer<std::int64_t, 1> &result)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**n** Number of elements in vector **x**.

**x** Buffer holding input vector **x**. The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(incx))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector **x**.

#### Output Parameters

**result** Buffer where the zero-based index **i** of the minimum element will be stored.

### 13.2.1.2.1.48 `iamin` (USM Version)

#### Syntax

```
sycl::event onemkl::blas::iamin(sycl::queue &queue, std::int64_t n, const T *x, std::int64_t incx,
                                 T_res *result, const sycl::vector_class<sycl::event> &dependencies
                                 = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**n** Number of elements in vector **x**.

**x** The pointer to input vector **x**. The array holding input vector **x** must be of size at least  $(1 + (n - 1) * \text{abs}(incx))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector **x**.

#### Output Parameters

**result** Pointer to where the zero-based index **i** of the minimum element will be stored.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 1 Routines*

**Parent topic:** *BLAS Routines*

### 13.2.1.2.2 BLAS Level 2 Routines

BLAS Level 2 includes routines which perform matrix-vector operations as described in the following table.

Routines	Description
<code>gbmv</code>	Matrix-vector product using a general band matrix
<code>gemv</code>	Matrix-vector product using a general matrix
<code>ger</code>	Rank-1 update of a general matrix
<code>gerc</code>	Rank-1 update of a conjugated general matrix
<code>geru</code>	Rank-1 update of a general matrix, unconjugated
<code>hbmv</code>	Matrix-vector product using a Hermitian band matrix
<code>hemv</code>	Matrix-vector product using a Hermitian matrix
<code>her</code>	Rank-1 update of a Hermitian matrix
<code>her2</code>	Rank-2 update of a Hermitian matrix
<code>hpmv</code>	Matrix-vector product using a Hermitian packed matrix
<code>hpr</code>	Rank-1 update of a Hermitian packed matrix
<code>hpr2</code>	Rank-2 update of a Hermitian packed matrix
<code>sbmv</code>	Matrix-vector product using symmetric band matrix
<code>spmv</code>	Matrix-vector product using a symmetric packed matrix
<code>spr</code>	Rank-1 update of a symmetric packed matrix
<code>spr2</code>	Rank-2 update of a symmetric packed matrix
<code>symv</code>	Matrix-vector product using a symmetric matrix
<code>syr</code>	Rank-1 update of a symmetric matrix
<code>syr2</code>	Rank-2 update of a symmetric matrix
<code>tbmv</code>	Matrix-vector product using a triangular band matrix
<code>tbsv</code>	Solution of a linear system of equations with a triangular band matrix
<code>tpmv</code>	Matrix-vector product using a triangular packed matrix
<code>tpsv</code>	Solution of a linear system of equations with a triangular packed matrix
<code>trmv</code>	Matrix-vector product using a triangular matrix
<code>trsv</code>	Solution of a linear system of equations with a triangular matrix

#### 13.2.1.2.2.1 `gbmv`

Computes a matrix-vector product with a general band matrix.

`gbmv` supports the following precisions.

T
<code>float</code>
<code>double</code>
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The `gbmv` routines compute a scalar-matrix-vector product and add the result to a scalar-vector product, with a general band matrix. The operation is defined as

$$y \leftarrow \text{alpha} * \text{op}(A) * x + \text{beta} * y$$

where:

- $\text{op}(A)$  is one of  $\text{op}(A) = A$ , or  $\text{op}(A) = A^T$ , or  $\text{op}(A) = A^H$ ,
- $\text{alpha}$  and  $\text{beta}$  are scalars,
- $A$  is an  $m$ -by- $n$  matrix with  $k_l$  sub-diagonals and  $k_u$  super-diagonals,
- $x$  and  $y$  are vectors.

### 13.2.1.2.2.2 `gbmv` (Buffer Version)

#### Syntax

```
void onemkl::blas::gbmv(sycl::queue &queue, onemkl::transpose trans, std::int64_t m, std::int64_t n,
                        std::int64_t kl, std::int64_t ku, T alpha, sycl::buffer<T, 1> &a, std::int64_t lda,
                        sycl::buffer<T, 1> &x, std::int64_t incx, T beta, sycl::buffer<T, 1> &y,
                        std::int64_t incy)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**trans** Specifies  $\text{op}(A)$ , the transposition operation applied to  $A$ . See *oneMKL defined datatypes* for more details.

**m** Number of rows of  $A$ . Must be at least zero.

**n** Number of columns of  $A$ . Must be at least zero.

**kl** Number of sub-diagonals of the matrix  $A$ . Must be at least zero.

**ku** Number of super-diagonals of the matrix  $A$ . Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**a** Buffer holding input matrix  $A$ . Must have size at least  $l_{da} * n$ . See *Matrix and Vector Storage* for more details.

**lda** Leading dimension of matrix  $A$ . Must be at least  $(k_l + k_u + 1)$ , and positive.

**x** Buffer holding input vector  $x$ . The length  $l_{en}$  of vector  $x$  is  $n$  if  $A$  is not transposed, and  $m$  if  $A$  is transposed. The buffer must be of size at least  $(1 + (l_{en} - 1) * \text{abs}(incx))$ . See *Matrix and Vector Storage* for more details.

**incx** Stride of vector  $x$ .

**beta** Scaling factor for vector  $y$ .

**y** Buffer holding input/output vector  $y$ . The length  $l_{en}$  of vector  $y$  is  $m$ , if  $A$  is not transposed, and  $n$  if  $A$  is transposed. The buffer must be of size at least  $(1 + (l_{en} - 1) * \text{abs}(incy))$  where  $l_{en}$  is this length. See *Matrix and Vector Storage* for more details.

**incy** Stride of vector  $y$ .

## Output Parameters

**y** Buffer holding the updated vector  $y$ .

### 13.2.1.2.2.3 `gbmv` (USM Version)

#### Syntax

```
sycl::event onemkl::blas::gbmv(sycl::queue &queue, onemkl::transpose trans, std::int64_t m,
                                std::int64_t n, std::int64_t kl, std::int64_t ku, T alpha, const T
                                *a, std::int64_t lda, const T *x, std::int64_t incx, T beta, T *y,
                                std::int64_t incy, const sycl::vector_class<sycl::event> &dependen-
                                cies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**trans** Specifies  $\text{op}(A)$ , the transposition operation applied to  $A$ . See *oneMKL defined datatypes* for more details.

**m** Number of rows of  $A$ . Must be at least zero.

**n** Number of columns of  $A$ . Must be at least zero.

**kl** Number of sub-diagonals of the matrix  $A$ . Must be at least zero.

**ku** Number of super-diagonals of the matrix  $A$ . Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**a** Pointer to input matrix  $A$ . The array holding input matrix  $A$  must have size at least  $\text{lda} \times n$ . See *Matrix and Vector Storage* for more details.

**lda** Leading dimension of matrix  $A$ . Must be at least  $(\text{kl} + \text{ku} + 1)$ , and positive.

**x** Pointer to input vector  $x$ . The length  $\text{len}$  of vector  $x$  is  $n$  if  $A$  is not transposed, and  $m$  if  $A$  is transposed. The array holding input vector  $x$  must be of size at least  $(1 + (\text{len} - 1) \times \text{abs}(\text{incx}))$ . See *Matrix and Vector Storage* for more details.

**incx** Stride of vector  $x$ .

**beta** Scaling factor for vector  $y$ .

**y** Pointer to input/output vector  $y$ . The length  $\text{len}$  of vector  $y$  is  $m$ , if  $A$  is not transposed, and  $n$  if  $A$  is transposed. The array holding input/output vector  $y$  must be of size at least  $(1 + (\text{len} - 1) \times \text{abs}(\text{incy}))$  where  $\text{len}$  is this length. See *Matrix and Vector Storage* for more details.

**incy** Stride of vector  $y$ .

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

`y` Pointer to the updated vector  $y$ .

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 2 Routines*

### 13.2.1.2.2.4 gemv

Computes a matrix-vector product using a general matrix.

`gemv` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The `gemv` routines compute a scalar-matrix-vector product and add the result to a scalar-vector product, with a general matrix. The operation is defined as

$$y \leftarrow \alpha \cdot \text{op}(A) \cdot x + \beta \cdot y$$

where:

`op(A)` is one of  $\text{op}(A) = A$ , or  $\text{op}(A) = A^T$ , or  $\text{op}(A) = A^H$ ,

`alpha` and `beta` are scalars,

`A` is an  $m$ -by- $n$  matrix, and `x`, `y` are vectors.

### 13.2.1.2.2.5 gemv (Buffer Version)

## Syntax

```
void onemkl::blas::gemv(sycl::queue &queue, onemkl::transpose trans, std::int64_t m, std::int64_t n,
                        T alpha, sycl::buffer<T, 1> &a, std::int64_t lda, sycl::buffer<T, 1> &x,
                        std::int64_t incx, T beta, sycl::buffer<T, 1> &y, std::int64_t incy)
```

## Input Parameters

- queue** The queue where the routine should be executed.
- trans** Specifies  $\text{op}(A)$ , the transposition operation applied to A.
- m** Specifies the number of rows of the matrix A. The value of m must be at least zero.
- n** Specifies the number of columns of the matrix A. The value of n must be at least zero.
- alpha** Scaling factor for the matrix-vector product.
- a** The buffer holding the input matrix A. Must have a size of at least  $\text{lda} \times n$ . See [Matrix and Vector Storage](#) for more details.
- lda** The leading dimension of matrix A. It must be at least m, and positive.
- x** Buffer holding input vector x. The length len of vector x is n if A is not transposed, and m if A is transposed. The buffer must be of size at least  $(1 + (\text{len} - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.
- incx** The stride of vector x.
- beta** The scaling factor for vector y.
- y** Buffer holding input/output vector y. The length len of vector y is m, if A is not transposed, and n if A is transposed. The buffer must be of size at least  $(1 + (\text{len} - 1) * \text{abs}(\text{incy}))$  where len is this length. See [Matrix and Vector Storage](#) for more details.
- incy** The stride of vector y.

## Output Parameters

- y** The buffer holding updated vector y.

### 13.2.1.2.2.6 gemv (USM Version)

#### Syntax

```
sycl::event onemkl::blas::gemv(sycl::queue &queue, onemkl::transpose trans, std::int64_t m,
                                std::int64_t n, T alpha, const T *a, std::int64_t lda, const
                                T *x, std::int64_t incx, T beta, T *y, std::int64_t incy, const
                                sycl::vector_class<sycl::event> &dependencies = {})
```

## Input Parameters

- queue** The queue where the routine should be executed.
- trans** Specifies  $\text{op}(A)$ , the transposition operation applied to A. See [oneMKL defined datatypes](#) for more details.
- m** Specifies the number of rows of the matrix A. The value of m must be at least zero.
- n** Specifies the number of columns of the matrix A. The value of n must be at least zero.
- alpha** Scaling factor for the matrix-vector product.
- a** The pointer to the input matrix A. Must have a size of at least  $\text{lda} \times n$ . See [Matrix and Vector Storage](#) for more details.
- lda** The leading dimension of matrix A. It must be at least m, and positive.

**x** Pointer to the input vector  $x$ . The length  $\text{len}$  of vector  $x$  is  $n$  if  $A$  is not transposed, and  $m$  if  $A$  is transposed. The array holding vector  $x$  must be of size at least  $(1 + (\text{len} - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** The stride of vector  $x$ .

**beta** The scaling factor for vector  $y$ .

**y** Pointer to input/output vector  $y$ . The length  $\text{len}$  of vector  $y$  is  $m$ , if  $A$  is not transposed, and  $n$  if  $A$  is transposed. The array holding input/output vector  $y$  must be of size at least  $(1 + (\text{len} - 1) * \text{abs}(\text{incy}))$  where  $\text{len}$  is this length. See [Matrix and Vector Storage](#) for more details.

**incy** The stride of vector  $y$ .

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**y** The pointer to updated vector  $y$ .

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** [BLAS Level 2 Routines](#)

### 13.2.1.2.2.7 ger

Computes a rank-1 update of a general matrix.

`ger` supports the following precisions.

T
float
double

## Description

The `ger` routines compute a scalar-vector-vector product and add the result to a general matrix. The operation is defined as

$$A \leftarrow \alpha * x * y^T + A$$

where:

$\alpha$  is scalar,

$A$  is an  $m$ -by- $n$  matrix,

$x$  is a vector length  $m$ ,

$y$  is a vector length  $n$ .

### 13.2.1.2.2.8 ger (Buffer Version)

#### Syntax

```
void onemkl::blas::ger(sycl::queue &queue, std::int64_t m, std::int64_t n, T alpha, sycl::buffer<T, 1>
&x, std::int64_t incx, sycl::buffer<T, 1> &y, std::int64_t incy, sycl::buffer<T,
1> &a, std::int64_t lda)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**m** Number of rows of A. Must be at least zero.

**n** Number of columns of A. Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**x** Buffer holding input vector x. The buffer must be of size at least  $(1 + (m - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**y** Buffer holding input/output vector y. The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incy}))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector y.

**a** Buffer holding input matrix A. Must have size at least  $\text{lda} * n$ . See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of matrix A. Must be at least m, and positive.

#### Output Parameters

**a** Buffer holding the updated matrix A.

### 13.2.1.2.2.9 ger (USM Version)

#### Syntax

```
sycl::event onemkl::blas::ger(sycl::queue &queue, std::int64_t m, std::int64_t n, T alpha, const T *x,
std::int64_t incx, const T *y, std::int64_t incy, T *a, std::int64_t lda,
const sycl::vector_class<sycl::event> &dependencies = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**m** Number of rows of A. Must be at least zero.

**n** Number of columns of A. Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**x** Pointer to input vector x. The array holding input vector x must be of size at least  $(1 + (m - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**y** Pointer to input/output vector  $y$ . The array holding input/output vector  $y$  must be of size at least  $(1 + (n - 1)*\text{abs}(\text{incy}))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector  $y$ .

**a** Pointer to input matrix  $A$ . Must have size at least  $\text{lda} \times n$ . See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of matrix  $A$ . Must be at least  $m$ , and positive.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**a** Pointer to the updated matrix  $A$ .

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** [BLAS Level 2 Routines](#)

### 13.2.1.2.2.10 gerc

Computes a rank-1 update (conjugated) of a general complex matrix.

`gerc` supports the following precisions.

T
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The `gerc` routines compute a scalar-vector-vector product and add the result to a general matrix. The operation is defined as

$$A \leftarrow \alpha * x * y^H + A$$

where:

$\alpha$  is a scalar,

$A$  is an  $m$ -by- $n$  matrix,

$x$  is a vector of length  $m$ ,

$y$  is vector of length  $n$ .

### 13.2.1.2.2.11 gerc (Buffer Version)

#### Syntax

```
void onemkl::blas::gerc(sycl::queue &queue, std::int64_t m, std::int64_t n, T alpha, sycl::buffer<T, 1>
&x, std::int64_t incx, sycl::buffer<T, 1> &y, std::int64_t incy, sycl::buffer<T,
1> &a, std::int64_t lda)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**m** Number of rows of A. Must be at least zero.

**n** Number of columns of A. Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**x** Buffer holding input vector x. The buffer must be of size at least  $(1 + (m - 1) * \text{abs}(incx))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**y** Buffer holding input/output vector y. The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(incy))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector y.

**a** Buffer holding input matrix A. Must have size at least  $lda * n$ . See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of matrix A. Must be at least m, and positive.

#### Output Parameters

**a** Buffer holding the updated matrix A.

### 13.2.1.2.2.12 gerc (USM Version)

#### Syntax

```
sycl::event onemkl::blas::gerc(sycl::queue &queue, std::int64_t m, std::int64_t n, T alpha, const T
*x, std::int64_t incx, const T *y, std::int64_t incy, T *a, std::int64_t
lda, const sycl::vector_class<sycl::event> &dependencies = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**m** Number of rows of A. Must be at least zero.

**n** Number of columns of A. Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**x** Pointer to the input vector x. The array holding input vector x must be of size at least  $(1 + (m - 1) * \text{abs}(incx))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**y** Pointer to the input/output vector  $y$ . The array holding the input/output vector  $y$  must be of size at least  $(1 + (n - 1)*\text{abs}(\text{incy}))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector  $y$ .

**a** Pointer to input matrix  $A$ . The array holding input matrix  $A$  must have size at least  $\text{lda}^*n$ . See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of matrix  $A$ . Must be at least  $m$ , and positive.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**a** Pointer to the updated matrix  $A$ .

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** [BLAS Level 2 Routines](#)

### 13.2.1.2.2.13 geru

Computes a rank-1 update (unconjugated) of a general complex matrix.

geru supports the following precisions.

T
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The `geru` routines compute a scalar-vector-vector product and add the result to a general matrix. The operation is defined as

$$A \leftarrow \alpha * x * y^T + A$$

where:

$\alpha$  is a scalar,

$A$  is an  $m$ -by- $n$  matrix,

$x$  is a vector of length  $m$ ,

$y$  is a vector of length  $n$ .

### 13.2.1.2.2.14 geru (Buffer Version)

#### Syntax

```
void onemkl::blas::geru(sycl::queue &queue, std::int64_t m, std::int64_t n, T alpha, sycl::buffer<T, 1>
&x, std::int64_t incx, sycl::buffer<T, 1> &y, std::int64_t incy, sycl::buffer<T,
1> &a, std::int64_t lda)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**m** Number of rows of A. Must be at least zero.

**n** Number of columns of A. Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**x** Buffer holding input vector x. The buffer must be of size at least  $(1 + (m - 1) * \text{abs}(incx))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**y** Buffer holding input/output vector y. The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(incy))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector y.

**a** Buffer holding input matrix A. Must have size at least  $lda * n$ . See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of matrix A. Must be at least m, and positive.

#### Output Parameters

**a** Buffer holding the updated matrix A.

### 13.2.1.2.2.15 geru (USM Version)

#### Syntax

```
sycl::event onemkl::blas::geru(sycl::queue &queue, std::int64_t m, std::int64_t n, T alpha, const T
*x, std::int64_t incx, const T *y, std::int64_t incy, T *a, std::int64_t
lda, const sycl::vector_class<sycl::event> &dependencies = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**m** Number of rows of A. Must be at least zero.

**n** Number of columns of A. Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**x** Pointer to the input vector x. The array holding input vector x must be of size at least  $(1 + (m - 1) * \text{abs}(incx))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**y** Pointer to input/output vector  $y$ . The array holding input/output vector  $y$  must be of size at least  $(1 + (n - 1)*\text{abs}(\text{incy}))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector  $y$ .

**a** Pointer to input matrix  $A$ . The array holding input matrix  $A$  must have size at least  $\text{l da} * n$ . See [Matrix and Vector Storage](#) for more details.

**l da** Leading dimension of matrix  $A$ . Must be at least  $m$ , and positive.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**a** Pointer to the updated matrix  $A$ .

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** [BLAS Level 2 Routines](#)

### 13.2.1.2.2.16 `hbmv`

Computes a matrix-vector product using a Hermitian band matrix.

`hbmv` supports the following precisions.

T
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The `hbmv` routines compute a scalar-matrix-vector product and add the result to a scalar-vector product, with a Hermitian band matrix. The operation is defined as

$$y \leftarrow \alpha * A * x + \beta * y$$

where:

`alpha` and `beta` are scalars,

`A` is an  $n$ -by- $n$  Hermitian band matrix, with  $k$  super-diagonals,

`x` and `y` are vectors of length  $n$ .

### 13.2.1.2.2.17 `hbmv` (Buffer Version)

#### Syntax

```
void onemkl::blas::hbmv(sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, std::int64_t k, T alpha, sycl::buffer<T, 1> &a, std::int64_t lda, sycl::buffer<T, 1> &x, std::int64_t incx, T beta, sycl::buffer<T, 1> &y, std::int64_t incy)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**n** Number of rows and columns of A. Must be at least zero.

**k** Number of super-diagonals of the matrix A. Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**a** Buffer holding input matrix A. Must have size at least *lda*\**n*. See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of matrix A. Must be at least (*k* + 1), and positive.

**x** Buffer holding input vector x. The buffer must be of size at least (1 + (*m* - 1)\*abs(*incx*)). See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**beta** Scaling factor for vector y.

**y** Buffer holding input/output vector y. The buffer must be of size at least (1 + (*n* - 1)\*abs(*incy*)). See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector y.

#### Output Parameters

**y** Buffer holding the updated vector y.

### 13.2.1.2.2.18 `hbmv` (USM Version)

#### Syntax

```
sycl::event onemkl::blas::hbmv(sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, std::int64_t k, T alpha, const T *a, std::int64_t lda, const T *x, std::int64_t incx, T beta, T *y, std::int64_t incy, const sycl::vector_class<sycl::event> &dependencies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**n** Number of rows and columns of A. Must be at least zero.

**k** Number of super-diagonals of the matrix A. Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**a** Pointer to the input matrix A. The array holding input matrix A must have size at least  $\text{lda} \times n$ . See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of matrix A. Must be at least ( $k + 1$ ), and positive.

**x** Pointer to input vector x. The array holding input vector x must be of size at least  $(1 + (m - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**beta** Scaling factor for vector y.

**y** Pointer to input/output vector y. The array holding input/output vector y must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incy}))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector y.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**y** Pointer to the updated vector y.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** [BLAS Level 2 Routines](#)

### 13.2.1.2.2.19 hemv

Computes a matrix-vector product using a Hermitian matrix.

hemv supports the following precisions.

T
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The `hemv` routines compute a scalar-matrix-vector product and add the result to a scalar-vector product, with a Hermitian matrix. The operation is defined as

$$y \leftarrow \alpha * A * x + \beta * y$$

where:

`alpha` and `beta` are scalars,  
`A` is an  $n$ -by- $n$  Hermitian matrix,  
`x` and `y` are vectors of length  $n$ .

### 13.2.1.2.2.20 `hemv` (Buffer Version)

#### Syntax

```
void onemkl::blas::hemv(sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, T alpha,
                       sycl::buffer<T, 1> &a, std::int64_t lda, sycl::buffer<T, 1> &x, std::int64_t
                       incx, T beta, sycl::buffer<T, 1> &y, std::int64_t incy)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether `A` is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**n** Number of rows and columns of `A`. Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**a** Buffer holding input matrix `A`. Must have size at least `lda`\*`n`. See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of matrix `A`. Must be at least `m`, and positive.

**x** Buffer holding input vector `x`. The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(incx))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector `x`.

**beta** Scaling factor for vector `y`.

**y** Buffer holding input/output vector `y`. The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(incy))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector `y`.

#### Output Parameters

**y** Buffer holding the updated vector `y`.

### 13.2.1.2.2.21 `hemv` (USM Version)

#### Syntax

```
sycl::event onemkl::blas::hemv(sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, T alpha,
                                const T *a, std::int64_t lda, const T *x, std::int64_t incx, T beta, T
                                *y, std::int64_t incy, const sycl::vector_class<sycl::event> &dependencies = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether  $A$  is upper or lower triangular. See *oneMKL defined datatypes* for more details.

**n** Number of rows and columns of  $A$ . Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**a** Pointer to input matrix  $A$ . The array holding input matrix  $A$  must have size at least  $\text{lda} \times n$ . See *Matrix and Vector Storage* for more details.

**lda** Leading dimension of matrix  $A$ . Must be at least  $m$ , and positive.

**x** Pointer to input vector  $x$ . The array holding input vector  $x$  must be of size at least  $(1 + (n - 1) \times \text{abs}(\text{incx}))$ . See *Matrix and Vector Storage* for more details.

**incx** Stride of vector  $x$ .

**beta** Scaling factor for vector  $y$ .

**y** Pointer to input/output vector  $y$ . The array holding input/output vector  $y$  must be of size at least  $(1 + (n - 1) \times \text{abs}(\text{incy}))$ . See *Matrix and Vector Storage* for more details.

**incy** Stride of vector  $y$ .

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

#### Output Parameters

**y** Pointer to the updated vector  $y$ .

#### Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 2 Routines*

### 13.2.1.2.2.22 her

Computes a rank-1 update of a Hermitian matrix.

her supports the following precisions.

T
std::complex<float>
std::complex<double>

#### Description

The her routines compute a scalar-vector-vector product and add the result to a Hermitian matrix. The operation is defined as

$$A \leftarrow \alpha * x * x^H + A$$

where:

$\alpha$  is scalar,

$A$  is an n-by-n Hermitian matrix,

$x$  is a vector of length n.

### 13.2.1.2.2.23 her (Buffer Version)

#### Syntax

```
void onemkl::blas::her(sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, T alpha,
                      sycl::buffer<T, 1> &x, std::int64_t incx, sycl::buffer<T, 1> &a, std::int64_t lda)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether  $A$  is upper or lower triangular. See *oneMKL defined datatypes* for more details.

**n** Number of rows and columns of  $A$ . Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**x** Buffer holding input vector  $x$ . The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(incx))$ . See *Matrix and Vector Storage* for more details.

**incx** Stride of vector  $x$ .

**a** Buffer holding input matrix  $A$ . Must have size at least  $lda * n$ . See *Matrix and Vector Storage* for more details.

**lda** Leading dimension of matrix  $A$ . Must be at least  $n$ , and positive.

## Output Parameters

- a** Buffer holding the updated upper triangular part of the Hermitian matrix A if `upper_lower = upper` or the updated lower triangular part of the Hermitian matrix A if `upper_lower = lower`.

The imaginary parts of the diagonal elements are set to zero.

### 13.2.1.2.2.24 her (USM Version)

#### Syntax

```
sycl::event onemkl::blas::her(sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, T alpha, const T *x, std::int64_t incx, T *a, std::int64_t lda, const sycl::vector_class<sycl::event> &dependencies = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See *oneMKL defined datatypes* for more details.

**n** Number of rows and columns of A. Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**x** Pointer to input vector x. The array holding input vector x must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**a** Pointer to input matrix A. The array holding input matrix A must have size at least `lda*n`. See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of matrix A. Must be at least n, and positive.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

#### Output Parameters

- a** Pointer to the updated upper triangular part of the Hermitian matrix A if `upper_lower = upper` or the updated lower triangular part of the Hermitian matrix A if `upper_lower = lower`.

The imaginary parts of the diagonal elements are set to zero.

#### Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** [BLAS Level 2 Routines](#)

### 13.2.1.2.2.25 her2

Computes a rank-2 update of a Hermitian matrix.

her2 supports the following precisions.

T
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

#### Description

The her2 routines compute two scalar-vector-vector products and add them to a Hermitian matrix. The operation is defined as

$$A \leftarrow \alpha * x * y^H + \text{conj}(\alpha) * y * x^H + A$$

where:

`alpha` is a scalar,

`A` is an n-by-n Hermitian matrix.

`x` and `y` are vectors or length n.

### 13.2.1.2.2.26 her2 (Buffer Version)

#### Syntax

```
void onemkl::blas::her2(sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, T alpha,
                      sycl::buffer<T, 1> &x, std::int64_t incx, sycl::buffer<T, 1> &y, std::int64_t
                      incy, sycl::buffer<T, 1> &a, std::int64_t lda)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**n** Number of columns of A. Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**x** Buffer holding input vector x. The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**y** Buffer holding input/output vector y. The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incy}))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector y.

**a** Buffer holding input matrix A. Must have size at least `lda*n`. See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of matrix A. Must be at least n, and positive.

## Output Parameters

- a** Buffer holding the updated upper triangular part of the Hermitian matrix A if `upper_lower = upper`, or the updated lower triangular part of the Hermitian matrix A if `upper_lower = lower`.

The imaginary parts of the diagonal elements are set to zero.

### 13.2.1.2.2.27 her2 (USM Version)

#### Syntax

```
sycl::event onemkl::blas::her2(sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, T alpha, const T *x, std::int64_t incx, const T *y, std::int64_t incy, T *a, std::int64_t lda, const sycl::vector_class<sycl::event> &dependencies = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See *oneMKL defined datatypes* for more details.

**n** Number of columns of A. Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**x** Pointer to input vector x. The array holding input vector x must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**y** Pointer to input/output vector y. The array holding input/output vector y must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incy}))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector y.

**a** Pointer to input matrix A. The array holding input matrix A must have size at least `lda*n`. See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of matrix A. Must be at least n, and positive.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

#### Output Parameters

- a** Pointer to the updated upper triangular part of the Hermitian matrix A if `upper_lower = upper`, or the updated lower triangular part of the Hermitian matrix A if `upper_lower = lower`.

The imaginary parts of the diagonal elements are set to zero.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 2 Routines*

### 13.2.1.2.2.28 hpmv

Computes a matrix-vector product using a Hermitian packed matrix.

hpmv supports the following precisions.

T
std::complex<float>
std::complex<double>

## Description

The hpmv routines compute a scalar-matrix-vector product and add the result to a scalar-vector product, with a Hermitian packed matrix. The operation is defined as

$$y \leftarrow \alpha * A * x + \beta * y$$

where:

$\alpha$  and  $\beta$  are scalars,

$A$  is an  $n$ -by- $n$  Hermitian matrix supplied in packed form,

$x$  and  $y$  are vectors of length  $n$ .

### 13.2.1.2.2.29 hpmv (Buffer Version)

## Syntax

```
void onemkl::blas::hpmv(sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, T alpha,
                        sycl::buffer<T, 1> &a, sycl::buffer<T, 1> &x, std::int64_t incx, T beta,
                        sycl::buffer<T, 1> &y, std::int64_t incy)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether  $A$  is upper or lower triangular. See *oneMKL defined datatypes* for more details.

**n** Number of rows and columns of  $A$ . Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**a** Buffer holding input matrix  $A$ . Must have size at least  $(n*(n+1))/2$ . See *Matrix and Vector Storage* for more details.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

**x** Buffer holding input vector  $x$ . The buffer must be of size at least  $(1 + (n - 1)*\text{abs}(\text{incx}))$ . See *Matrix and Vector Storage* for more details.

**incx** Stride of vector  $x$ .

**beta** Scaling factor for vector  $y$ .

**y** Buffer holding input/output vector  $y$ . The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incy}))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector  $y$ .

## Output Parameters

**y** Buffer holding the updated vector  $y$ .

### 13.2.1.2.2.30 `hpmv` (USM Version)

#### Syntax

```
sycl::event onemkl::blas::hpmv(sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, T alpha,  
          const T *a, const T *x, std::int64_t incx, T beta, T *y, std::int64_t  
          incy, const sycl::vector_class<sycl::event> &dependencies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether  $A$  is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**n** Number of rows and columns of  $A$ . Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**a** Pointer to input matrix  $A$ . The array holding input matrix  $A$  must have size at least  $(n * (n + 1)) / 2$ . See [Matrix and Vector Storage](#) for more details.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

**x** Pointer to input vector  $x$ . The array holding input vector  $x$  must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector  $x$ .

**beta** Scaling factor for vector  $y$ .

**y** Pointer to input/output vector  $y$ . The array holding input/output vector  $y$  must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incy}))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector  $y$ .

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**y** Pointer to the updated vector  $y$ .

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 2 Routines*

### 13.2.1.2.2.31 **hpr**

Computes a rank-1 update of a Hermitian packed matrix.

**hpr** supports the following precisions.

T
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The **hpr** routines compute a scalar-vector-vector product and add the result to a Hermitian packed matrix. The operation is defined as

$$A \leftarrow \alpha * x * x^H + A$$

where:

$\alpha$  is scalar,

$A$  is an  $n$ -by- $n$  Hermitian matrix, supplied in packed form,

$x$  is a vector of length  $n$ .

### 13.2.1.2.2.32 **hpr (Buffer Version)**

## Syntax

```
void onemkl::blas::hpr(sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, T alpha,
                      sycl::buffer<T, 1> &x, std::int64_t incx, sycl::buffer<T, 1> &a)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether  $A$  is upper or lower triangular. See *oneMKL defined datatypes* for more details.

**n** Number of rows and columns of  $A$ . Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**x** Buffer holding input vector  $x$ . The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(incx))$ . See *Matrix and Vector Storage* for more details.

**incx** Stride of vector  $x$ .

**a** Buffer holding input matrix  $A$ . Must have size at least  $(n*(n-1))/2$ . See [Matrix and Vector Storage](#) for more details.

The imaginary part of the diagonal elements need not be set and are assumed to be zero.

## Output Parameters

**a** Buffer holding the updated upper triangular part of the Hermitian matrix  $A$  if `upper_lower =upper`, or the updated lower triangular part of the Hermitian matrix  $A$  if `upper_lower =lower`.

The imaginary parts of the diagonal elements are set to zero.

### 13.2.1.2.2.33 hpr (USM Version)

#### Syntax

```
sycl::event onemkl::blas::hpr(sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t
                               n, T alpha, const T *x, std::int64_t incx, T *a, const
                               sycl::vector_class<sycl::event> &dependencies = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether  $A$  is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**n** Number of rows and columns of  $A$ . Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**x** Pointer to input vector  $x$ . The array holding input vector  $x$  must be of size at least  $(1 + (n - 1)*\text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector  $x$ .

**a** Pointer to input matrix  $A$ . The array holding input matrix  $A$  must have size at least  $(n*(n-1))/2$ . See [Matrix and Vector Storage](#) for more details.

The imaginary part of the diagonal elements need not be set and are assumed to be zero.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

#### Output Parameters

**a** Pointer to the updated upper triangular part of the Hermitian matrix  $A$  if `upper_lower =upper`, or the updated lower triangular part of the Hermitian matrix  $A$  if `upper_lower =lower`.

The imaginary parts of the diagonal elements are set to zero.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 2 Routines*

### 13.2.1.2.2.34 hpr2

Performs a rank-2 update of a Hermitian packed matrix.

hpr2 supports the following precisions.

T
std::complex<float>
std::complex<double>

## Description

The hpr2 routines compute two scalar-vector-vector products and add them to a Hermitian packed matrix. The operation is defined as

$$A \leftarrow \alpha * x * y^H + \text{conj}(\alpha) * y * x^H + A$$

where:

$\alpha$  is a scalar,

$A$  is an  $n$ -by- $n$  Hermitian matrix, supplied in packed form,

$x$  and  $y$  are vectors of length  $n$ .

### 13.2.1.2.2.35 hpr2 (Buffer Version)

## Syntax

```
void onemkl::blas::hpr2(sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, T alpha,
                        sycl::buffer<T, 1> &x, std::int64_t incx, sycl::buffer<T, 1> &y, std::int64_t
                        incy, sycl::buffer<T, 1> &a)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether  $A$  is upper or lower triangular. See *oneMKL defined datatypes* for more details.

**n** Number of rows and columns of  $A$ . Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**x** Buffer holding input vector  $x$ . The buffer must be of size at least  $(1 + (n - 1)*\text{abs}(\text{incx}))$ . See *Matrix and Vector Storage* for more details.

**incx** Stride of vector  $x$ .

**y** Buffer holding input/output vector  $y$ . The buffer must be of size at least  $(1 + (n - 1)*\text{abs}(\text{incy}))$ . See *Matrix and Vector Storage* for more details.

**incy** Stride of vector  $y$ .

**a** Buffer holding input matrix  $A$ . Must have size at least  $(n*(n-1))/2$ . See [Matrix and Vector Storage](#) for more details.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

## Output Parameters

**a** Buffer holding the updated upper triangular part of the Hermitian matrix  $A$  if `upper_lower =upper`, or the updated lower triangular part of the Hermitian matrix  $A$  if `upper_lower =lower`.

The imaginary parts of the diagonal elements are set to zero.

### 13.2.1.2.2.36 `hpr2` (USM Version)

#### Syntax

```
sycl::event onemkl::blas::hpr2 (sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, T alpha,
                           const T *x, std::int64_t incx, const T *y, std::int64_t incy, T *a,
                           const sycl::vector_class<sycl::event> &dependencies = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether  $A$  is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**n** Number of rows and columns of  $A$ . Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**x** Pointer to input vector  $x$ . The array holding input vector  $x$  must be of size at least  $(1 + (n - 1)*\text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector  $x$ .

**y** Pointer to input/output vector  $y$ . The array holding input/output vector  $y$  must be of size at least  $(1 + (n - 1)*\text{abs}(\text{incy}))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector  $y$ .

**a** Pointer to input matrix  $A$ . The array holding input matrix  $A$  must have size at least  $(n*(n-1))/2$ . See [Matrix and Vector Storage](#) for more details.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

- a** Pointer to the updated upper triangular part of the Hermitian matrix A if `upper_lower =upper`, or the updated lower triangular part of the Hermitian matrix A if `upper_lower =lower`.

The imaginary parts of the diagonal elements are set to zero.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 2 Routines*

### 13.2.1.2.2.37 sbmv

Computes a matrix-vector product with a symmetric band matrix.

`sbmv` supports the following precisions.

T
float
double

## Description

The `sbmv` routines compute a scalar-matrix-vector product and add the result to a scalar-vector product, with a symmetric band matrix. The operation is defined as

$$y \leftarrow \alpha * A * x + \beta * y$$

where:

`alpha` and `beta` are scalars,

`A` is an n-by-n symmetric matrix with `k` super-diagonals,

`x` and `y` are vectors of length n.

### 13.2.1.2.2.38 sbmv (Buffer Version)

#### Syntax

```
void onemkl::blas::sbmv(sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, std::int64_t k, T alpha, sycl::buffer<T, 1> &a, std::int64_t lda, sycl::buffer<T, 1> &x, std::int64_t incx, T beta, sycl::buffer<T, 1> &y, std::int64_t incy)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**n** Number of rows and columns of A. Must be at least zero.

**k** Number of super-diagonals of the matrix A. Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**a** Buffer holding input matrix A. Must have size at least  $\text{lda} \times n$ . See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of matrix A. Must be at least ( $k + 1$ ), and positive.

**x** Buffer holding input vector x. The buffer must be of size at least  $(1 + (n - 1) \times \text{abs(incx)})$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**beta** Scaling factor for vector y.

**y** Buffer holding input/output vector y. The buffer must be of size at least  $(1 + (n - 1) \times \text{abs(incy)})$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector y.

## Output Parameters

**y** Buffer holding the updated vector y.

### 13.2.1.2.2.39 sbmv (USM Version)

#### Syntax

```
sycl::event onemkl::blas::sbmv(sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n,
                                std::int64_t k, T alpha, const T *a, std::int64_t lda, const
                                T *x, std::int64_t incx, T beta, T *y, std::int64_t incy, const
                                sycl::vector_class<sycl::event> &dependencies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**n** Number of rows and columns of A. Must be at least zero.

**k** Number of super-diagonals of the matrix A. Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**a** Pointer to input matrix A. The array holding input matrix A must have size at least  $\text{lda} \times n$ . See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of matrix A. Must be at least ( $k + 1$ ), and positive.

**x** Pointer to input vector x. The array holding input vector x must be of size at least  $(1 + (n - 1) \times \text{abs(incx)})$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector  $x$ .

**beta** Scaling factor for vector  $y$ .

**y** Pointer to input/output vector  $y$ . The array holding input/output vector  $y$  must be of size at least  $(1 + (n - 1) * \text{abs}(incy))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector  $y$ .

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**y** Pointer to the updated vector  $y$ .

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** [BLAS Level 2 Routines](#)

### 13.2.1.2.2.40 spmv

Computes a matrix-vector product with a symmetric packed matrix.

spmv supports the following precisions.

T
float
double

## Description

The spmv routines compute a scalar-matrix-vector product and add the result to a scalar-vector product, with a symmetric packed matrix. The operation is defined as

$$y \leftarrow \alpha * A * x + \beta * y$$

where:

$\alpha$  and  $\beta$  are scalars,

$A$  is an  $n$ -by- $n$  symmetric matrix, supplied in packed form.

$x$  and  $y$  are vectors of length  $n$ .

### 13.2.1.2.2.41 spmv (Buffer Version)

#### Syntax

```
void onemkl::blas::spmv(sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, T alpha,
                        sycl::buffer<T, 1> &a, sycl::buffer<T, 1> &x, std::int64_t incx, T beta,
                        sycl::buffer<T, 1> &y, std::int64_t incy)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**n** Number of rows and columns of A. Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**a** Buffer holding input matrix A. Must have size at least  $(n*(n+1))/2$ . See [Matrix and Vector Storage](#) for more details.

**x** Buffer holding input vector x. The buffer must be of size at least  $(1 + (n - 1)*\text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**beta** Scaling factor for vector y.

**y** Buffer holding input/output vector y. The buffer must be of size at least  $(1 + (n - 1)*\text{abs}(\text{incy}))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector y.

#### Output Parameters

**y** Buffer holding the updated vector y.

### 13.2.1.2.2.42 spmv (USM Version)

#### Syntax

```
sycl::event onemkl::blas::spmv(sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, T alpha,
                                const T *a, const T *x, std::int64_t incx, T beta, T *y, std::int64_t
                                incy, const sycl::vector_class<sycl::event> &dependencies = { })
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**n** Number of rows and columns of A. Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**a** Pointer to input matrix A. The array holding input matrix A must have size at least  $(n*(n+1))/2$ . See [Matrix and Vector Storage](#) for more details.

**x** Pointer to input vector  $x$ . The array holding input vector  $x$  must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector  $x$ .

**beta** Scaling factor for vector  $y$ .

**y** Pointer to input/output vector  $y$ . The array holding input/output vector  $y$  must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incy}))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector  $y$ .

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**y** Pointer to the updated vector  $y$ .

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** [BLAS Level 2 Routines](#)

### 13.2.1.2.2.43 spr

Performs a rank-1 update of a symmetric packed matrix.

spr supports the following precisions.

T
float
double

## Description

The spr routines compute a scalar-vector-vector product and add the result to a symmetric packed matrix. The operation is defined as

$$A \leftarrow \alpha * x * x^T + A$$

where:

$\alpha$  is scalar,

$A$  is an  $n$ -by- $n$  symmetric matrix, supplied in packed form,

$x$  is a vector of length  $n$ .

### 13.2.1.2.2.44 spr (Buffer Version)

#### Syntax

```
void onemkl::blas::spr(sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, T alpha,
                      sycl::buffer<T, 1> &x, std::int64_t incx, sycl::buffer<T, 1> &a)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**n** Number of rows and columns of A. Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**x** Buffer holding input vector x. The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**a** Buffer holding input matrix A. Must have size at least  $(n * (n - n)) / 2$ . See [Matrix and Vector Storage](#) for more details.

#### Output Parameters

**a** Buffer holding the updated upper triangular part of the symmetric matrix A if `upper_lower =upper`, or the updated lower triangular part of the symmetric matrix A if `upper_lower =lower`.

### 13.2.1.2.2.45 spr (USM Version)

#### Syntax

```
sycl::event onemkl::blas::spr(sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, T alpha, const T *x, std::int64_t incx, T *a, const sycl::vector_class<sycl::event> &dependencies = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**n** Number of rows and columns of A. Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**x** Pointer to input vector x. The array holding input vector x must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**a** Pointer to input matrix A. The array holding input matrix A must have size at least  $(n * (n - n)) / 2$ . See [Matrix and Vector Storage](#) for more details.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

- a Pointer to the updated upper triangular part of the symmetric matrix A if `upper_lower =upper`, or the updated lower triangular part of the symmetric matrix A if `upper_lower =lower`.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 2 Routines*

### 13.2.1.2.2.46 spr2

Computes a rank-2 update of a symmetric packed matrix.

`spr` supports the following precisions.

T
float
double

## Description

The `spr2` routines compute two scalar-vector-vector products and add them to a symmetric packed matrix. The operation is defined as

$$A \leftarrow \alpha x^* y^T + \alpha y^* x^T + A$$

where:

`alpha` is scalar,

`A` is an n-by-n symmetric matrix, supplied in packed form,

`x` and `y` are vectors of length n.

### 13.2.1.2.2.47 spr2 (Buffer Version)

## Syntax

```
void onemkl::blas::spr2(sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, T alpha,
    sycl::buffer<T, 1> &x, std::int64_t incx, sycl::buffer<T, 1> &y, std::int64_t incy, sycl::buffer<T, 1> &a)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**n** Number of rows and columns of A. Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**x** Buffer holding input vector x. The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**y** Buffer holding input/output vector y. The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incy}))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector y.

**a** Buffer holding input matrix A. Must have size at least  $(n * (n - 1)) / 2$ . See [Matrix and Vector Storage](#) for more details.

## Output Parameters

**a** Buffer holding the updated upper triangular part of the symmetric matrix A if `upper_lower =upper` or the updated lower triangular part of the symmetric matrix A if `upper_lower =lower`.

### 13.2.1.2.2.48 spr2 (USM Version)

#### Syntax

```
sycl::event onemkl::blas::spr2(sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, T alpha,
                                const T *x, std::int64_t incx, const T *y, std::int64_t incy, T *a)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**n** Number of rows and columns of A. Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**x** Pointer to input vector x. The array holding input vector x must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**y** Pointer to input/output vector y. The array holding input/output vector y must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incy}))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector y.

**a** Pointer to input matrix A. The array holding input matrix A must have size at least  $(n * (n - 1)) / 2$ . See [Matrix and Vector Storage](#) for more details.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

- a Pointer to the updated upper triangular part of the symmetric matrix A if `upper_lower =upper` or the updated lower triangular part of the symmetric matrix A if `upper_lower =lower`.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 2 Routines*

### 13.2.1.2.2.49 `symv`

Computes a matrix-vector product for a symmetric matrix.

`symv` supports the following precisions.

T
float
double

## Description

The `symv` routines compute a scalar-matrix-vector product and add the result to a scalar-vector product, with a symmetric matrix. The operation is defined as

$$y \leftarrow \alpha * A * x + \beta * y$$

where:

`alpha` and `beta` are scalars,

`A` is an n-by-n symmetric matrix,

`x` and `y` are vectors of length n.

### 13.2.1.2.2.50 `symv` (Buffer Version)

## Syntax

```
void onemkl::blas::symv(sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, T alpha,
                       sycl::buffer<T, 1> &a, std::int64_t lda, sycl::buffer<T, 1> &x, std::int64_t
incx, T beta, sycl::buffer<T, 1> &y, std::int64_t incy)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**n** Number of rows and columns of A. Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**a** Buffer holding input matrix A. Must have size at least  $l\text{da} \times n$ . See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of matrix A. Must be at least m, and positive.

**x** Buffer holding input vector x. The buffer must be of size at least  $(1 + (n - 1) \times \text{abs}(incx))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**y** Buffer holding input/output vector y. The buffer must be of size at least  $(1 + (n - 1) \times \text{abs}(incy))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector y.

## Output Parameters

**y** Buffer holding the updated vector y.

### 13.2.1.2.2.51 symv (USM Version)

#### Syntax

```
sycl::event onemkl::blas::symv(sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, T alpha,
                                const T *a, std::int64_t lda, const T *x, std::int64_t incx, T beta, T
                                *y, std::int64_t incy, const sycl::vector_class<sycl::event> &dependencies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**n** Number of rows and columns of A. Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**a** Pointer to input matrix A. The array holding input matrix A must have size at least  $l\text{da} \times n$ . See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of matrix A. Must be at least m, and positive.

**x** Pointer to input vector x. The array holding input vector x must be of size at least  $(1 + (n - 1) \times \text{abs}(incx))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**y** Pointer to input/output vector y. The array holding input/output vector y must be of size at least  $(1 + (n - 1) \times \text{abs}(incy))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector y.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**y** Pointer to the updated vector  $y$ .

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 2 Routines*

### 13.2.1.2.2.52 `syr`

Computes a rank-1 update of a symmetric matrix.

`syr` supports the following precisions.

T
float
double

## Description

The `syr` routines compute a scalar-vector-vector product add them and add the result to a matrix, with a symmetric matrix. The operation is defined as

$$A \leftarrow \alpha * x * x^T + A$$

where:

$\alpha$  is scalar,

$A$  is an  $n$ -by- $n$  symmetric matrix,

$x$  is a vector of length  $n$ .

### 13.2.1.2.2.53 `syr` (Buffer Version)

## Syntax

```
void onemkl::blas::syr(sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, T alpha,
                      sycl::buffer<T, 1> &x, std::int64_t incx, sycl::buffer<T, 1> &a, std::int64_t lda)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**n** Number of columns of A. Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**x** Buffer holding input vector x. The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**a** Buffer holding input matrix A. Must have size at least  $\text{lda} * n$ . See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of matrix A. Must be at least n, and positive.

## Output Parameters

**a** Buffer holding the updated upper triangular part of the symmetric matrix A if `upper_lower =upper` or the updated lower triangular part of the symmetric matrix A if `upper_lower =lower`.

### 13.2.1.2.2.54 `syr` (USM Version)

#### Syntax

```
sycl::event onemkl::blas::syr(sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, T alpha, const T *x, std::int64_t incx, T *a, std::int64_t lda, const sycl::vector_class<sycl::event> &dependencies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**n** Number of columns of A. Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**x** Pointer to input vector x. The array holding input vector x must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**a** Pointer to input matrix A. The array holding input matrix A must have size at least  $\text{lda} * n$ . See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of matrix A. Must be at least n, and positive.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

- a Pointer to the updated upper triangular part of the symmetric matrix A if `upper_lower =upper` or the updated lower triangular part of the symmetric matrix A if `upper_lower =lower`.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 2 Routines*

### 13.2.1.2.2.55 `syr2`

Computes a rank-2 update of a symmetric matrix.

`syr2` supports the following precisions.

T
float
double

## Description

The `syr2` routines compute two scalar-vector-vector product add them and add the result to a matrix, with a symmetric matrix. The operation is defined as

$$A \leftarrow \alpha * x * y^T + \alpha * y * x^T + A$$

where:

`alpha` is a scalar,

`A` is an n-by-n symmetric matrix,

`x` and `y` are vectors of length n.

### 13.2.1.2.2.56 `syr2 (Buffer Version)`

## Syntax

```
void onemkl::blas::syr2(sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, T alpha,
    sycl::buffer<T, 1> &x, std::int64_t incx, sycl::buffer<T, 1> &y, std::int64_t incy, sycl::buffer<T, 1> &a, std::int64_t lda)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**n** Number of columns of A. Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**x** Buffer holding input vector x. The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**y** Buffer holding input/output vector y. The buffer must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incy}))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector y.

**a** Buffer holding input matrix A. Must have size at least  $\text{lda} * n$ . See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of matrix A. Must be at least n, and positive.

## Output Parameters

**a** Buffer holding the updated upper triangular part of the symmetric matrix A if **upper\_lower** =upper, or the updated lower triangular part of the symmetric matrix A if **upper\_lower** =lower.

### 13.2.1.2.2.57 **syr2** (USM Version)

#### Syntax

```
sycl::event onemkl::blas::syr2 (sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, T alpha, const T *x, std::int64_t incx, const T *y, std::int64_t incy, T *a, std::int64_t lda, const sycl::vector_class<sycl::event> &dependencies = {} )
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**n** Number of columns of A. Must be at least zero.

**alpha** Scaling factor for the matrix-vector product.

**x** Pointer to input vector x. The array holding input vector x must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incx}))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**y** Pointer to input/output vector y. The array holding input/output vector y must be of size at least  $(1 + (n - 1) * \text{abs}(\text{incy}))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Stride of vector y.

**a** Pointer to input matrix A. The array holding input matrix A must have size at least  $\text{lda} * n$ . See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of matrix A. Must be at least n, and positive.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

- a Pointer to the updated upper triangular part of the symmetric matrix A if `upper_lower =upper`, or the updated lower triangular part of the symmetric matrix A if `upper_lower =lower`.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 2 Routines*

### 13.2.1.2.2.58 tbmv

Computes a matrix-vector product using a triangular band matrix.

`tbmv` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The `tbmv` routines compute a matrix-vector product with a triangular band matrix. The operation is defined as

$$x \leftarrow op(A)*x$$

where:

`op(A)` is one of  $op(A) = A$ , or  $op(A) = A^T$ , or  $op(A) = A^H$ ,

A is an n-by-n unit or non-unit, upper or lower triangular band matrix, with  $(k + 1)$  diagonals,

x is a vector of length n.

### 13.2.1.2.2.59 tbmv (Buffer Version)

## Syntax

```
void onemkl::blas::tbmv(sycl::queue &queue, onemkl::uplo upper_lower, onemkl::transpose trans,
                        onemkl::diag unit_nonunit, std::int64_t n, std::int64_t k, sycl::buffer<T, 1>
                        &a, std::int64_t lda, sycl::buffer<T, 1> &x, std::int64_t incx)
```

## Input Parameters

- queue** The queue where the routine should be executed.
- upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.
- trans** Specifies op(A), the transposition operation applied to A. See [oneMKL defined datatypes](#) for more details.
- unit\_nonunit** Specifies whether the matrix A is unit triangular or not. See [oneMKL defined datatypes](#) for more details.
- n** Numbers of rows and columns of A. Must be at least zero.
- k** Number of sub/super-diagonals of the matrix A. Must be at least zero.
- a** Buffer holding input matrix A. Must have size at least lda\*n. See [Matrix and Vector Storage](#) for more details.
- lda** Leading dimension of matrix A. Must be at least (k + 1), and positive.
- x** Buffer holding input vector x. The buffer must be of size at least (1 + (n - 1)\*abs(incx)). See [Matrix and Vector Storage](#) for more details.
- incx** Stride of vector x.

## Output Parameters

- x** Buffer holding the updated vector x.

### 13.2.1.2.2.60 tbmv (USM Version)

#### Syntax

```
sycl::event onemkl::blas::tbmv(sycl::queue &queue, onemkl::uplo upper_lower, onemkl::transpose
                                trans, onemkl::diag unit_nonunit, std::int64_t n, std::int64_t k,
                                const T *a, std::int64_t lda, T *x, std::int64_t incx, const
                                sycl::vector_class<sycl::event> &dependencies = {})
```

## Input Parameters

- queue** The queue where the routine should be executed.
- upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.
- trans** Specifies op(A), the transposition operation applied to A. See [oneMKL defined datatypes](#) for more details.
- unit\_nonunit** Specifies whether the matrix A is unit triangular or not. See [oneMKL defined datatypes](#) for more details.
- n** Numbers of rows and columns of A. Must be at least zero.
- k** Number of sub/super-diagonals of the matrix A. Must be at least zero.
- a** Pointer to input matrix A. The array holding input matrix A must have size at least lda\*n. See [Matrix and Vector Storage](#) for more details.
- lda** Leading dimension of matrix A. Must be at least (k + 1), and positive.
- x** Pointer to input vector x. The array holding input vector x must be of size at least (1 + (n - 1)\*abs(incx)). See [Matrix and Vector Storage](#) for more details.
- incx** Stride of vector x.
- dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**x** Pointer to the updated vector  $x$ .

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 2 Routines*

### 13.2.1.2.2.61 tbsv

Solves a system of linear equations whose coefficients are in a triangular band matrix.

tbsv supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The tbsv routines solve a system of linear equations whose coefficients are in a triangular band matrix. The operation is defined as

$$\text{op}(A)^*x = b$$

where:

$\text{op}(A)$  is one of  $\text{op}(A) = A$ , or  $\text{op}(A) = A^T$ , or  $\text{op}(A) = A^H$ ,

$A$  is an  $n$ -by- $n$  unit or non-unit, upper or lower triangular band matrix, with  $(k + 1)$  diagonals,

$b$  and  $x$  are vectors of length  $n$ .

### 13.2.1.2.2.62 tbsv (Buffer Version)

## Syntax

```
void onemkl::blas::tbsv(sycl::queue &queue, onemkl::uplo upper_lower, onemkl::transpose trans,
onemkl::diag unit_nonunit, std::int64_t n, std::int64_t k, sycl::buffer<T, 1>
&a, std::int64_t lda, sycl::buffer<T, 1> &x, std::int64_t incx)
```

## Input Parameters

- queue** The queue where the routine should be executed.
- upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.
- trans** Specifies op(A), the transposition operation applied to A. See [oneMKL defined datatypes](#) for more details.
- unit\_nonunit** Specifies whether the matrix A is unit triangular or not. See [oneMKL defined datatypes](#) for more details.
- n** Number of rows and columns of A. Must be at least zero.
- k** Number of sub/super-diagonals of the matrix A. Must be at least zero.
- a** Buffer holding input matrix A. Must have size at least lda\*n. See [Matrix and Vector Storage](#) for more details.
- lda** Leading dimension of matrix A. Must be at least (k + 1), and positive.
- x** Buffer holding input vector x. The buffer must be of size at least (1 + (n - 1)\*abs(incx)). See [Matrix and Vector Storage](#) for more details.
- incx** Stride of vector x.

## Output Parameters

- x** Buffer holding the solution vector x.

### 13.2.1.2.2.63 tbsv (USM Version)

#### Syntax

```
sycl::event onemkl::blas::tbsv(sycl::queue &queue, onemkl::uplo upper_lower, onemkl::transpose
    trans, onemkl::diag unit_nonunit, std::int64_t n, std::int64_t k,
    const T *a, std::int64_t lda, T *x, std::int64_t incx, const
    sycl::vector_class<sycl::event> &dependencies = {})
```

## Input Parameters

- queue** The queue where the routine should be executed.
- upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.
- trans** Specifies op(A), the transposition operation applied to A. See [oneMKL defined datatypes](#) for more details.
- unit\_nonunit** Specifies whether the matrix A is unit triangular or not. See [oneMKL defined datatypes](#) for more details.
- n** Number of rows and columns of A. Must be at least zero.
- k** Number of sub/super-diagonals of the matrix A. Must be at least zero.
- a** Pointer to input matrix A. The array holding input matrix A must have size at least lda\*n. See [Matrix and Vector Storage](#) for more details.
- lda** Leading dimension of matrix A. Must be at least (k + 1), and positive.
- x** Pointer to input vector x. The array holding input vector x must be of size at least (1 + (n - 1)\*abs(incx)). See [Matrix and Vector Storage](#) for more details.
- incx** Stride of vector x.
- dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**x** Pointer to the solution vector  $x$ .

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 2 Routines*

### 13.2.1.2.2.64 tpmv

Computes a matrix-vector product using a triangular packed matrix.

$\text{tpmv}$  supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The  $\text{tpmv}$  routines compute a matrix-vector product with a triangular packed matrix. The operation is defined as

$$x \leftarrow \text{op}(A)^*x$$

where:

$\text{op}(A)$  is one of  $\text{op}(A) = A$ , or  $\text{op}(A) = A^T$ , or  $\text{op}(A) = A^H$ ,

$A$  is an  $n$ -by- $n$  unit or non-unit, upper or lower triangular band matrix, supplied in packed form,

$x$  is a vector of length  $n$ .

### 13.2.1.2.2.65 tpmv (Buffer Version)

## Syntax

```
void onemkl::blas::tpmv(sycl::queue &queue, onemkl::uplo upper_lower, onemkl::transpose trans, onemkl::diag unit_nonunit, std::int64_t n, sycl::buffer<T, 1> &a, sycl::buffer<T, 1> &x, std::int64_t incx)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**trans** Specifies op(A), the transposition operation applied to A. See [oneMKL defined datatypes](#) for more details.

**unit\_nonunit** Specifies whether the matrix A is unit triangular or not. See [oneMKL defined datatypes](#) for more details.

**n** Numbers of rows and columns of A. Must be at least zero.

**a** Buffer holding input matrix A. Must have size at least  $(n*(n+1))/2$ . See [Matrix and Vector Storage](#) for more details.

**x** Buffer holding input vector x. The buffer must be of size at least  $(1 + (n - 1)*abs(incx))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

## Output Parameters

**x** Buffer holding the updated vector x.

### 13.2.1.2.2.66 tpmv (USM Version)

#### Syntax

```
sycl::event onemkl::blas::tpmv(sycl::queue &queue, onemkl::uplo upper_lower, onemkl::transpose
                                trans, onemkl::diag unit_nonunit, std::int64_t n, const T *a, T *x,
                                std::int64_t incx, const sycl::vector_class<sycl::event> &dependencies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**trans** Specifies op(A), the transposition operation applied to A. See [oneMKL defined datatypes](#) for more details.

**unit\_nonunit** Specifies whether the matrix A is unit triangular or not. See [oneMKL defined datatypes](#) for more details.

**n** Numbers of rows and columns of A. Must be at least zero.

**a** Pointer to input matrix A. The array holding input matrix A must have size at least  $(n*(n+1))/2$ . See [Matrix and Vector Storage](#) for more details.

**x** Pointer to input vector x. The array holding input vector x must be of size at least  $(1 + (n - 1)*abs(incx))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**x** Pointer to the updated vector  $x$ .

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 2 Routines*

### 13.2.1.2.2.67 tpsv

Solves a system of linear equations whose coefficients are in a triangular packed matrix.

`tpsv` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The `tpsv` routines solve a system of linear equations whose coefficients are in a triangular packed matrix. The operation is defined as

$$\text{op}(A)^*x = b$$

where:

`op(A)` is one of  $\text{op}(A) = A$ , or  $\text{op}(A) = A^T$ , or  $\text{op}(A) = A^H$ ,

$A$  is an  $n$ -by- $n$  unit or non-unit, upper or lower triangular band matrix, supplied in packed form,

$b$  and  $x$  are vectors of length  $n$ .

### 13.2.1.2.2.68 tpsv (Buffer Version)

## Syntax

```
void onemkl::blas::tpsv(sycl::queue &queue, onemkl::uplo upper_lower, onemkl::transpose trans,
onemkl::diag unit_nonunit, std::int64_t n, std::int64_t k, sycl::buffer<T, 1>
&a, sycl::buffer<T, 1> &x, std::int64_t incx)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**trans** Specifies op(A), the transposition operation applied to A. See [oneMKL defined datatypes](#) for more details.

**unit\_nonunit** Specifies whether the matrix A is unit triangular or not. See [oneMKL defined datatypes](#) for more details.

**n** Numbers of rows and columns of A. Must be at least zero.

**a** Buffer holding input matrix A. Must have size at least  $(n*(n+1))/2$ . See [Matrix and Vector Storage](#) for more details.

**x** Buffer holding the n-element right-hand side vector b. The buffer must be of size at least  $(1 + (n - 1)*abs(incx))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

## Output Parameters

**x** Buffer holding the solution vector x.

### 13.2.1.2.2.69 tpsv (USM Version)

#### Syntax

```
sycl::event onemkl::blas::tpsv(sycl::queue &queue, onemkl::uplo upper_lower, onemkl::transpose
trans, onemkl::diag unit_nonunit, std::int64_t n, std::int64_t k, const
T *a, T *x, std::int64_t incx, const sycl::vector_class<sycl::event>
&dependencies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**trans** Specifies op(A), the transposition operation applied to A. See [oneMKL defined datatypes](#) for more details.

**unit\_nonunit** Specifies whether the matrix A is unit triangular or not. See [oneMKL defined datatypes](#) for more details.

**n** Numbers of rows and columns of A. Must be at least zero.

**a** Pointer to input matrix A. The array holding input matrix A must have size at least  $(n*(n+1))/2$ . See [Matrix and Vector Storage](#) for more details.

**x** Pointer to the n-element right-hand side vector b. The array holding the n-element right-hand side vector b must be of size at least  $(1 + (n - 1)*abs(incx))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**x** Pointer to the solution vector  $x$ .

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 2 Routines*

### 13.2.1.2.2.70 `trmv`

Computes a matrix-vector product using a triangular matrix.

`trmv` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The `trmv` routines compute a matrix-vector product with a triangular matrix. The operation is defined

$$x \leftarrow \text{op}(A)*x$$

where:

`op(A)` is one of  $\text{op}(A) = A$ , or  $\text{op}(A) = A^T$ , or  $\text{op}(A) = A^H$ ,

$A$  is an  $n$ -by- $n$  unit or non-unit, upper or lower triangular band matrix,

$x$  is a vector of length  $n$ .

### 13.2.1.2.2.71 `trmv` (Buffer Version)

## Syntax

```
void onemkl::blas::trmv(sycl::queue &queue, onemkl::uplo upper_lower, onemkl::transpose trans,
onemkl::diag unit_nonunit, std::int64_t n, sycl::buffer<T, 1> &a, std::int64_t
lda, sycl::buffer<T, 1> &x, std::int64_t incx)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**trans** Specifies op(A), the transposition operation applied to A. See [oneMKL defined datatypes](#) for more details.

**unit\_nonunit** Specifies whether the matrix A is unit triangular or not. See [oneMKL defined datatypes](#) for more details.

**n** Numbers of rows and columns of A. Must be at least zero.

**a** Buffer holding input matrix A. Must have size at least lda\*n. See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of matrix A. Must be at least n, and positive.

**x** Buffer holding input vector x. The buffer must be of size at least (1 + (n - 1)\*abs(incx)). See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

## Output Parameters

**x** Buffer holding the updated vector x.

### 13.2.1.2.2.72 trmv (USM Version)

#### Syntax

```
sycl::event onemkl::blas::trmv(sycl::queue &queue, onemkl::uplo upper_lower, onemkl::transpose
                                trans, onemkl::diag unit_nonunit, std::int64_t n, const
                                T *a, std::int64_t lda, T *x, std::int64_t incx, const
                                sycl::vector_class<sycl::event> &dependencies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**trans** Specifies op(A), the transposition operation applied to A. See [oneMKL defined datatypes](#) for more details.

**unit\_nonunit** Specifies whether the matrix A is unit triangular or not. See [oneMKL defined datatypes](#) for more details.

**n** Numbers of rows and columns of A. Must be at least zero.

**a** Pointer to input matrix A. The array holding input matrix A must have size at least lda\*n. See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of matrix A. Must be at least n, and positive.

**x** Pointer to input vector x. The array holding input vector x must be of size at least (1 + (n - 1)\*abs(incx)). See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**x** Pointer to the updated vector  $x$ .

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 2 Routines*

### 13.2.1.2.2.73 `trsv`

Solves a system of linear equations whose coefficients are in a triangular matrix.

`trsv` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The `trsv` routines compute a matrix-vector product with a triangular band matrix. The operation is defined as

$$\text{op}(A)^*x = b$$

where:

`op(A)` is one of  $\text{op}(A) = A$ , or  $\text{op}(A) = A^T$ , or  $\text{op}(A) = A^H$ ,

$A$  is an  $n$ -by- $n$  unit or non-unit, upper or lower triangular matrix,

$b$  and  $x$  are vectors of length  $n$ .

### 13.2.1.2.2.74 `trsv` (Buffer Version)

## Syntax

```
void onemkl::blas::trsv(sycl::queue &queue, onemkl::uplo upper_lower, onemkl::transpose trans,
onemkl::diag unit_nonunit, std::int64_t n, std::int64_t k, sycl::buffer<T, 1>
&a, std::int64_t lda, sycl::buffer<T, 1> &x, std::int64_t incx)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**trans** Specifies op(A), the transposition operation applied to A. See [oneMKL defined datatypes](#) for more details.

**unit\_nonunit** Specifies whether the matrix A is unit triangular or not. See [oneMKL defined datatypes](#) for more details.

**n** Numbers of rows and columns of A. Must be at least zero.

**a** Buffer holding input matrix A. Must have size at least lda\*n. See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of matrix A. Must be at least n, and positive.

**x** Buffer holding the n-element right-hand side vector b. The buffer must be of size at least (1 + (n - 1)\*abs(incx)). See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

## Output Parameters

**x** Buffer holding the solution vector x.

### 13.2.1.2.2.75 trsv (USM Version)

#### Syntax

```
sycl::event onemkl::blas::trsv(sycl::queue &queue, onemkl::uplo upper_lower, onemkl::transpose
                                trans, onemkl::diag unit_nonunit, std::int64_t n, std::int64_t k,
                                const T *a, std::int64_t lda, T *x, std::int64_t incx, const
                                sycl::vector_class<sycl::event> &dependencies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**trans** Specifies op(A), the transposition operation applied to A. See [oneMKL defined datatypes](#) for more details.

**unit\_nonunit** Specifies whether the matrix A is unit triangular or not. See [oneMKL defined datatypes](#) for more details.

**n** Numbers of rows and columns of A. Must be at least zero.

**a** Pointer to input matrix A. The array holding input matrix A must have size at least lda\*n. See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of matrix A. Must be at least n, and positive.

**x** Pointer to the n-element right-hand side vector b. The array holding the n-element right-hand side vector b must be of size at least (1 + (n - 1)\*abs(incx)). See [Matrix and Vector Storage](#) for more details.

**incx** Stride of vector x.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**x** Pointer to the solution vector  $x$ .

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 2 Routines*

**Parent topic:** *BLAS Routines*

### 13.2.1.2.3 BLAS Level 3 Routines

BLAS Level 3 includes routines which perform matrix-matrix operations as described in the following table.

Routines	Description
gemm	Computes a matrix-matrix product with general matrices.
hemm	Computes a matrix-matrix product where one input matrix is Hermitian and one is general.
herk	Performs a Hermitian rank-k update.
her2k	Performs a Hermitian rank-2k update.
symm	Computes a matrix-matrix product where one input matrix is symmetric and one matrix is general.
syrk	Performs a symmetric rank-k update.
syr2k	Performs a symmetric rank-2k update.
trmm	Computes a matrix-matrix product where one input matrix is triangular and one input matrix is general.
trsm	Solves a triangular matrix equation (forward or backward solve).

#### 13.2.1.2.3.1 gemm

Computes a matrix-matrix product with general matrices.

`gemm` supports the following precisions.

Ts	Ta	Tb	Tc
float	half	half	float
half	half	half	half
float	float	float	float
double	double	double	double
<code>std::complex&lt;float&gt;</code>	<code>std::complex&lt;float&gt;</code>	<code>std::complex&lt;float&gt;</code>	<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>	<code>std::complex&lt;double&gt;</code>	<code>std::complex&lt;double&gt;</code>	<code>std::complex&lt;double&gt;</code>

## Description

The `gemm` routines compute a scalar-matrix-matrix product and add the result to a scalar-matrix product, with general matrices. The operation is defined as

$$C \leftarrow \alpha \cdot op(A) \cdot op(B) + \beta \cdot C$$

where:

`op(X)` is one of `op(X) = X`, or `op(X) = XT`, or `op(X) = XH`,

`alpha` and `beta` are scalars,

`A`, `B` and `C` are matrices:

`op(A)` is an  $m$ -by- $k$  matrix,

`op(B)` is a  $k$ -by- $n$  matrix,

`C` is an  $m$ -by- $n$  matrix.

### 13.2.1.2.3.2 gemm (Buffer Version)

#### Syntax

```
void onemkl::blas::gemm(sycl::queue &queue, onemkl::transpose transa, onemkl::transpose transb,
                        std::int64_t m, std::int64_t n, std::int64_t k, Ts alpha, sycl::buffer<Ta, 1>
                        &a, std::int64_t lda, sycl::buffer<Tb, 1> &b, std::int64_t ldb, Ts beta,
                        sycl::buffer<Tc, 1> &c, std::int64_t ldc)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**transa** Specifies the form of `op(A)`, the transposition operation applied to `A`.

**transb** Specifies the form of `op(B)`, the transposition operation applied to `B`.

**m** Specifies the number of rows of the matrix `op(A)` and of the matrix `C`. The value of `m` must be at least zero.

**n** Specifies the number of columns of the matrix `op(B)` and the number of columns of the matrix `B`. The value of `n` must be at least zero.

**k** Specifies the number of columns of the matrix `op(A)` and the number of rows of the matrix `op(B)`. The value of `k` must be at least zero.

**alpha** Scaling factor for the matrix-matrix product.

**a** The buffer holding the input matrix `A`. If `A` is not transposed, `A` is an  $m$ -by- $k$  matrix so the array `a` must have size at least `lda*k`. If `A` is transposed, `A` is an  $k$ -by- $m$  matrix so the array `a` must have size at least `lda*m`. See [Matrix and Vector Storage](#) for more details.

**lda** The leading dimension of `A`. Must be at least `m` if `A` is not transposed, and at least `k` if `A` is transposed. It must be positive.

**b** The buffer holding the input matrix `B`. If `B` is not transposed, `B` is an  $k$ -by- $n$  matrix so the array `b` must have size at least `ldb*n`. If `B` is transposed, `B` is an  $n$ -by- $k$  matrix so the array `b` must have size at least `ldb*k`. See [Matrix and Vector Storage](#) for more details.

**ldb** The leading dimension of `B`. Must be at least `k` if `B` is not transposed, and at least `n` if `B` is transposed. It must be positive.

**beta** Scaling factor for matrix C.

**c** The buffer holding the input/output matrix C. It must have a size of at least ldc\*n. See [Matrix and Vector Storage](#) for more details.

**ldc** The leading dimension of C. It must be positive and at least the size of m.

## Output Parameters

**c** The buffer, which is overwritten by  $\text{alpha} \cdot \text{op}(A) \cdot \text{op}(B) + \text{beta} \cdot C$ .

## Notes

If  $\text{beta} = 0$ , matrix C does not need to be initialized before calling gemm.

### 13.2.1.2.3.3 gemm (USM Version)

#### Syntax

```
sycl::event onemkl::blas::gemm(sycl::queue &queue, onemkl::transpose transa, onemkl::transpose
transb, std::int64_t m, std::int64_t n, std::int64_t k, Ts alpha, const
Ta *a, std::int64_t lda, const Tb *b, std::int64_t ldb, Ts beta, Tc *c,
std::int64_t ldc, const sycl::vector_class<sycl::event> &dependencies
= {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**transa** Specifies the form of  $\text{op}(A)$ , the transposition operation applied to A.

**transb** Specifies the form of  $\text{op}(B)$ , the transposition operation applied to B.

**m** Specifies the number of rows of the matrix  $\text{op}(A)$  and of the matrix C. The value of m must be at least zero.

**n** Specifies the number of columns of the matrix  $\text{op}(B)$  and the number of columns of the matrix C. The value of n must be at least zero.

**k** Specifies the number of columns of the matrix  $\text{op}(A)$  and the number of rows of the matrix  $\text{op}(B)$ . The value of k must be at least zero.

**alpha** Scaling factor for the matrix-matrix product.

**a** Pointer to input matrix A. If A is not transposed, A is an m-by-k matrix so the array a must have size at least lda\*k. If A is transposed, A is an k-by-m matrix so the array a must have size at least lda\*m. See [Matrix and Vector Storage](#) for more details.

**lda** The leading dimension of A. Must be at least m if A is not transposed, and at least k if A is transposed. It must be positive.

**b** Pointer to input matrix B. If B is not transposed, B is an k-by-n matrix so the array b must have size at least ldb\*n. If B is transposed, B is an n-by-k matrix so the array b must have size at least ldb\*k. See [Matrix and Vector Storage](#) for more details.

**ldb** The leading dimension of B. Must be at least k if B is not transposed, and at least n if B is transposed. It must be positive.

**beta** Scaling factor for matrix C.

**c** The pointer to input/output matrix C. It must have a size of at least ldc\*n. See [Matrix and Vector Storage](#) for more details.

**ldc** The leading dimension of C. It must be positive and at least the size of m.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**c** Pointer to the output matrix, overwritten by  $\alpha \cdot \text{op}(A) \cdot \text{op}(B) + \beta \cdot C$ .

## Notes

If  $\beta = 0$ , matrix C does not need to be initialized before calling gemm.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** [BLAS Level 3 Routines](#)

### 13.2.1.2.3.4 hemm

Computes a matrix-matrix product where one input matrix is Hermitian and one is general.

hemm supports the following precisions:

T
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The hemm routines compute a scalar-matrix-matrix product and add the result to a scalar-matrix product, where one of the matrices in the multiplication is Hermitian. The argument `left_right` determines if the Hermitian matrix, A, is on the left of the multiplication (`left_right = side::left`) or on the right (`left_right = side::right`). Depending on `left_right`, the operation is defined as

$$C \leftarrow \alpha \cdot A \cdot B + \beta \cdot C$$

or

$$C \leftarrow \alpha \cdot B \cdot A + \beta \cdot C$$

where:

`alpha` and `beta` are scalars,

A is a Hermitian matrix, either m-by-m or n-by-n matrices,

B and C are m-by-n matrices.

### 13.2.1.2.3.5 hemm (Buffer Version)

#### Syntax

```
void onemkl::blas::hemm(sycl::queue &queue, onemkl::side left_right, onemkl::uplo upper_lower,
std::int64_t m, std::int64_t n, T alpha, sycl::buffer<T, 1> &a, std::int64_t
lda, sycl::buffer<T, 1> &b, std::int64_t ldb, T beta, sycl::buffer<T, 1> &c,
std::int64_t ldc)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**left\_right** Specifies whether A is on the left side of the multiplication (`side::left`) or on the right side (`side::right`). See [oneMKL defined datatypes](#) for more details.

**uplo** Specifies whether A's data is stored in its upper or lower triangle. See [oneMKL defined datatypes](#) for more details.

**m** Specifies the number of rows of the matrix B and C.

The value of **m** must be at least zero.

**n** Specifies the number of columns of the matrix B and C.

The value of **n** must be at least zero.

**alpha** Scaling factor for the matrix-matrix product.

**a** Buffer holding input matrix A. Must have size at least `lda*m` if A is on the left of the multiplication, or `lda*n` if A is on the right. See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of A. Must be at least **m** if A is on the left of the multiplication, or at least **n** if A is on the right. Must be positive.

**b** Buffer holding input matrix B. Must have size at least `ldb*n`. See [Matrix and Vector Storage](#) for more details.

**ldb** Leading dimension of B. Must be positive and at least **m**.

**beta** Scaling factor for matrix C.

**c** Buffer holding input/output matrix C. Must have size at least `ldc*n`. See [Matrix and Vector Storage](#) for more details.

**ldc** Leading dimension of C. Must be positive and at least **m**.

#### Output Parameters

**c** Output buffer, overwritten by  $\alpha * A * B + \beta * C$  (`left_right = side::left`) or  $\alpha * B * A + \beta * C$  (`left_right = side::right`).

## Notes

If **beta** = 0, matrix C does not need to be initialized before calling **hemm**.

### 13.2.1.2.3.6 **hemm** (USM Version)

#### Syntax

```
sycl::event onemkl::blas::hemm(sycl::queue &queue, onemkl::side left_right, onemkl::uplo up-
    per_lower, std::int64_t m, std::int64_t n, T alpha, const T *a,
    std::int64_t lda, const T *b, std::int64_t ldb, T beta, T *c, std::int64_t
    ldc, const sycl::vector_class<sycl::event> &dependencies = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**left\_right** Specifies whether A is on the left side of the multiplication (`side::left`) or on the right side (`side::right`). See [oneMKL defined datatypes](#) for more details.

**uplo** Specifies whether A's data is stored in its upper or lower triangle. See [oneMKL defined datatypes](#) for more details.

**m** Specifies the number of rows of the matrix B and C.

The value of **m** must be at least zero.

**n** Specifies the number of columns of the matrix B and C.

The value of **n** must be at least zero.

**alpha** Scaling factor for the matrix-matrix product.

**a** Pointer to input matrix A. Must have size at least `lda*m` if A is on the left of the multiplication, or `lda*n` if A is on the right. See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of A. Must be at least **m** if A is on the left of the multiplication, or at least **n** if A is on the right. Must be positive.

**b** Pointer to input matrix B. Must have size at least `ldb*n`. See [Matrix and Vector Storage](#) for more details.

**ldb** Leading dimension of B. Must be positive and at least **m**.

**beta** Scaling factor for matrix C.

**c** Pointer to input/output matrix C. Must have size at least `ldc*n`. See [Matrix and Vector Storage](#) for more details.

**ldc** Leading dimension of C. Must be positive and at least **m**.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

- c** Pointer to the output matrix, overwritten by  $\alpha A^*B + \beta C$  (`left_right = side::left`) or  $\alpha B^*A + \beta C$  (`left_right = side::right`).

## Notes

If `beta = 0`, matrix `C` does not need to be initialized before calling `hemm`.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 3 Routines*

### 13.2.1.2.3.7 herk

Performs a Hermitian rank-k update.

`herk` supports the following precisions:

T	T_real
<code>std::complex&lt;float&gt;</code>	<code>float</code>
<code>std::complex&lt;double&gt;</code>	<code>double</code>

## Description

The `herk` routines compute a rank-k update of a Hermitian matrix `C` by a general matrix `A`. The operation is defined as:

$$C \leftarrow \alpha \operatorname{op}(A) \operatorname{op}(A)^H + \beta C$$

where:

`op(X)` is one of `op(X) = X` or `op(X) = XH`,

`alpha` and `beta` are real scalars,

`C` is a Hermitian matrix and `A` is a general matrix.

Here `op(A)` is  $n \times k$ , and `C` is  $n \times n$ .

### 13.2.1.2.3.8 herk (Buffer Version)

## Syntax

```
void onemkl::blas::herk(sycl::queue &queue, onemkl::uplo upper_lower, onemkl::transpose trans,
                       std::int64_t n, std::int64_t k, T_real alpha, sycl::buffer<T, 1> &a, std::int64_t
                       lda, T_real beta, sycl::buffer<T, 1> &c, std::int64_t ldc)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A's data is stored in its upper or lower triangle. See [oneMKL defined datatypes](#) for more details.

**trans** Specifies op(A), the transposition operation applied to A. See [oneMKL defined datatypes](#) for more details. Supported operations are transpose::nontrans and transpose::conjtrans.

**n** The number of rows and columns in C. The value of n must be at least zero.

**k** Number of columns in op(A).

The value of k must be at least zero.

**alpha** Real scaling factor for the rank-k update.

**a** Buffer holding input matrix A. If trans = transpose::nontrans, A is an n-by-k matrix so the array a must have size at least lda\*k. Otherwise, A is an k-by-n matrix so the array a must have size at least lda\*n. See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of A. Must be at least n if A is not transposed, and at least k if A is transposed. Must be positive.

**beta** Real scaling factor for matrix C.

**c** Buffer holding input/output matrix C. Must have size at least ldc\*n. See [Matrix and Vector Storage](#) for more details.

**ldc** Leading dimension of C. Must be positive and at least n.

## Output Parameters

**c** The output buffer, overwritten by alpha\*op(A)\*op(A)<sup>T</sup> + beta\*C. The imaginary parts of the diagonal elements are set to zero.

### 13.2.1.2.3.9 herk (USM Version)

#### Syntax

```
sycl::event onemkl::blas::herk(sycl::queue &queue, onemkl::uplo upper_lower, onemkl::transpose
                               trans, std::int64_t n, std::int64_t k, T_real alpha, const T
                               *a, std::int64_t lda, T_real beta, T *c, std::int64_t ldc, const
                               sycl::vector_class<sycl::event> &dependencies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A's data is stored in its upper or lower triangle. See [oneMKL defined datatypes](#) for more details.

**trans** Specifies op(A), the transposition operation applied to A. See [oneMKL defined datatypes](#) for more details. Supported operations are transpose::nontrans and transpose::conjtrans.

**n** The number of rows and columns in C. The value of n must be at least zero.

**k** Number of columns in  $\text{op}(A)$ .

The value of **k** must be at least zero.

**alpha** Real scaling factor for the rank-k update.

**a** Pointer to input matrix  $A$ . If  $\text{trans} = \text{transpose::nontrans}$ ,  $A$  is an  $n$ -by- $k$  matrix so the array  $a$  must have size at least  $\text{l da} * k$ . Otherwise,  $A$  is an  $k$ -by- $n$  matrix so the array  $a$  must have size at least  $\text{l da} * n$ . See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of  $A$ . Must be at least  $n$  if  $A$  is not transposed, and at least  $k$  if  $A$  is transposed. Must be positive.

**beta** Real scaling factor for matrix  $C$ .

**c** Pointer to input/output matrix  $C$ . Must have size at least  $\text{l dc} * n$ . See [Matrix and Vector Storage](#) for more details.

**ldc** Leading dimension of  $C$ . Must be positive and at least  $n$ .

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**c** Pointer to the output matrix, overwritten by  $\text{alpha} * \text{op}(A) * \text{op}(A)^T + \text{beta} * C$ . The imaginary parts of the diagonal elements are set to zero.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** [BLAS Level 3 Routines](#)

### 13.2.1.2.3.10 her2k

Performs a Hermitian rank-2k update.

her2k supports the following precisions:

T	T_real
<code>std::complex&lt;float&gt;</code>	<code>float</code>
<code>std::complex&lt;double&gt;</code>	<code>double</code>

## Description

The her2k routines perform a rank-2k update of an  $n$  x  $n$  Hermitian matrix  $C$  by general matrices  $A$  and  $B$ . If  $\text{trans} = \text{transpose::nontrans}$ . The operation is defined as

$$C \leftarrow \text{alpha} * A * B^H + \text{conjg}(\text{alpha}) * B * A^H + \text{beta} * C$$

where  $A$  is  $n$  x  $k$  and  $B$  is  $k$  x  $n$ .

If  $\text{trans} = \text{transpose::conjtrans}$ , the operation is defined as:

$$C \leftarrow \text{alpha} * B * A^H + \text{conjg}(\text{alpha}) * A * B^H + \text{beta} * C$$

where A is  $k \times n$  and B is  $n \times k$ .

In both cases:

alpha is a complex scalar and beta is a real scalar.

C is a Hermitian matrix and A, B are general matrices.

The inner dimension of both matrix multiplications is k.

### 13.2.1.2.3.11 her2k (Buffer Version)

#### Syntax

```
void onemkl::blas::her2k(sycl::queue &queue, onemkl::uplo upper_lower, onemkl::transpose trans,
                        std::int64_t n, std::int64_t k, T alpha, sycl::buffer<T, 1> &a, std::int64_t
                        lda, sycl::buffer<T, 1> &b, std::int64_t ldb, T_real beta, sycl::buffer<T, 1>
                        &c, std::int64_t ldc)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A's data is stored in its upper or lower triangle. See [oneMKL defined datatypes](#) for more details.

**trans** Specifies the operation to apply, as described above. Supported operations are transpose::nontrans and transpose::conjtrans.

**n** The number of rows and columns in C. The value of n must be at least zero.

**k** The inner dimension of matrix multiplications. The value of k must be at least equal to zero.

**alpha** Complex scaling factor for the rank-2k update.

**a** Buffer holding input matrix A. If trans = transpose::nontrans, A is an n-by-k matrix so the array a must have size at least lda\*k. Otherwise, A is an k-by-n matrix so the array a must have size at least lda\*n. See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of A. Must be at least n if trans = transpose::nontrans, and at least k otherwise. Must be positive.

**beta** Real scaling factor for matrix C.

**b** Buffer holding input matrix B. If trans = transpose::nontrans, B is an k-by-n matrix so the array b must have size at least ldb\*n. Otherwise, B is an n-by-k matrix so the array b must have size at least ldb\*k. See [Matrix and Vector Storage](#) for more details.

**ldb** Leading dimension of B. Must be at least k if trans = transpose::nontrans, and at least n otherwise. Must be positive.

**c** Buffer holding input/output matrix C. Must have size at least ldc\*n. See [Matrix and Vector Storage](#) for more details.

**ldc** Leading dimension of C. Must be positive and at least n.

## Output Parameters

- c** Output buffer, overwritten by the updated C matrix.

### 13.2.1.2.3.12 her2k (USM Version)

#### Syntax

```
sycl::event onemkl::blas::her2k(sycl::queue &queue, onemkl::uplo upper_lower, onemkl::transpose trans, std::int64_t n, std::int64_t k, T alpha, const T *a, std::int64_t lda, const T *b, std::int64_t ldb, T_beta, T *c, std::int64_t ldc, const sycl::vector_class<sycl::event> &dependencies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A's data is stored in its upper or lower triangle. See [oneMKL defined datatypes](#) for more details.

**trans** Specifies the operation to apply, as described above. Supported operations are transpose::nontrans and transpose::conjtrans.

**n** The number of rows and columns in C. The value of n must be at least zero.

**k** The inner dimension of matrix multiplications. The value of k must be at least equal to zero.

**alpha** Complex scaling factor for the rank-2k update.

**a** Pointer to input matrix A. If trans = transpose::nontrans, A is an n-by-k matrix so the array a must have size at least lda\*k. Otherwise, A is an k-by-n matrix so the array a must have size at least lda\*n. See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of A. Must be at least n if trans = transpose::nontrans, and at least k otherwise. Must be positive.

**beta** Real scaling factor for matrix C.

**b** Pointer to input matrix B. If trans = transpose::nontrans, B is an k-by-n matrix so the array b must have size at least ldb\*n. Otherwise, B is an n-by-k matrix so the array b must have size at least ldb\*k. See [Matrix and Vector Storage](#) for more details.

**ldb** Leading dimension of B. Must be at least k if trans = transpose::nontrans, and at least n otherwise. Must be positive.

**c** Pointer to input/output matrix C. Must have size at least ldc\*n. See [Matrix and Vector Storage](#) for more details.

**ldc** Leading dimension of C. Must be positive and at least n.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**c** Pointer to the output matrix, overwritten by the updated C matrix.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 3 Routines*

### 13.2.1.2.3.13 `symm`

Computes a matrix-matrix product where one input matrix is symmetric and one matrix is general.

`symm` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The `symm` routines compute a scalar-matrix-matrix product and add the result to a scalar-matrix product, where one of the matrices in the multiplication is symmetric. The argument `left_right` determines if the symmetric matrix, A, is on the left of the multiplication (`left_right = side::left`) or on the right (`left_right = side::right`). Depending on `left_right`, the operation is defined as

$$C \leftarrow \alpha * A * B + \beta * C,$$

or

$$C \leftarrow \alpha * B * A + \beta * C,$$

where:

`alpha` and `beta` are scalars,

A is a symmetric matrix, either m-by-m or n-by-n,

B and C are m-by-n matrices.

### 13.2.1.2.3.14 `symm` (Buffer Version)

## Syntax

```
void onemkl::blas::symm(sycl::queue &queue, onemkl::side left_right, onemkl::uplo upper_lower,
                      std::int64_t m, std::int64_t n, T alpha, sycl::buffer<T, 1> &a, std::int64_t
                      lda, sycl::buffer<T, 1> &b, std::int64_t ldb, T beta, sycl::buffer<T, 1> &c,
                      std::int64_t ldc)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**left\_right** Specifies whether A is on the left side of the multiplication (`side::left`) or on the right side (`side::right`). See [oneMKL defined datatypes](#) for more details.

**upper\_lower** Specifies whether A's data is stored in its upper or lower triangle. See [oneMKL defined datatypes](#) for more details.

**m** Number of rows of B and C. The value of m must be at least zero.

**n** Number of columns of B and C. The value of n must be at least zero.

**alpha** Scaling factor for the matrix-matrix product.

**a** Buffer holding input matrix A. Must have size at least `lda*m` if A is on the left of the multiplication, or `lda*n` if A is on the right. See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of A. Must be at least m if A is on the left of the multiplication, or at least n if A is on the right. Must be positive.

**b** Buffer holding input matrix B. Must have size at least `ldb*n`. See [Matrix and Vector Storage](#) for more details.

**ldb** Leading dimension of B. Must be positive and at least m.

**beta** Scaling factor for matrix C.

**c** Buffer holding input/output matrix C. Must have size at least `ldc*n`. See [Matrix and Vector Storage](#) for more details.

**ldc** Leading dimension of C. Must be positive and at least m.

## Output Parameters

**c** Output buffer, overwritten by  $\text{alpha} * \text{A} * \text{B} + \text{beta} * \text{C}$  (`left_right = side::left`) or  $\text{alpha} * \text{B} * \text{A} + \text{beta} * \text{C}$  (`left_right = side::right`).

## Notes

If `beta = 0`, matrix C does not need to be initialized before calling `symm`.

### 13.2.1.2.3.15 `symm` (USM Version)

#### Syntax

```
sycl::event onemkl::blas::symm(sycl::queue &queue, onemkl::side left_right, onemkl::uplo upper_lower, std::int64_t m, std::int64_t n, T alpha, const T *a, std::int64_t lda, const T *b, std::int64_t ldb, T beta, T *c, std::int64_t ldc, const sycl::vector_class<sycl::event> &dependencies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**left\_right** Specifies whether A is on the left side of the multiplication (`side::left`) or on the right side (`side::right`). See [oneMKL defined datatypes](#) for more details.

**upper\_lower** Specifies whether A's data is stored in its upper or lower triangle. See [oneMKL defined datatypes](#) for more details.

**m** Number of rows of B and C. The value of m must be at least zero.

**n** Number of columns of B and C. The value of n must be at least zero.

**alpha** Scaling factor for the matrix-matrix product.

**a** Pointer to input matrix A. Must have size at least `lda*m` if A is on the left of the multiplication, or `lda*n` if A is on the right. See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of A. Must be at least m if A is on the left of the multiplication, or at least n if A is on the right. Must be positive.

**b** Pointer to input matrix B. Must have size at least `ldb*n`. See [Matrix and Vector Storage](#) for more details.

**ldb** Leading dimension of B. Must be positive and at least m.

**beta** Scaling factor for matrix C.

**c** Pointer to input/output matrix C. Must have size at least `ldc*n`. See [Matrix and Vector Storage](#) for more details.

**ldc** Leading dimension of C. Must be positive and at least m.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**c** Pointer to the output matrix, overwritten by  $\text{alpha} \cdot A \cdot B + \text{beta} \cdot C$  (`left_right = side::left`) or  $\text{alpha} \cdot B \cdot A + \text{beta} \cdot C$  (`left_right = side::right`).

## Notes

If `beta = 0`, matrix C does not need to be initialized before calling `symm`.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** [BLAS Level 3 Routines](#)

### 13.2.1.2.3.16 syrk

Performs a symmetric rank-k update.

`syrk` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

#### Description

The `syrk` routines perform a rank-k update of a symmetric matrix  $C$  by a general matrix  $A$ . The operation is defined as:

$$C \leftarrow \alpha \cdot \text{op}(A) \cdot \text{op}(A)^T + \beta \cdot C$$

where:

`op(X)` is one of `op(X) = X` or `op(X) = XT`,

`alpha` and `beta` are scalars,

$C$  is a symmetric matrix and  $A$  is a general matrix.

Here `op(A)` is n-by-k, and  $C$  is n-by-n.

### 13.2.1.2.3.17 syrk (Buffer Version)

#### Syntax

```
void onemkl::blas::syrk(sycl::queue &queue, onemkl::uplo upper_lower, onemkl::transpose trans,
                       std::int64_t n, std::int64_t k, T alpha, sycl::buffer<T, 1> &a, std::int64_t lda,
                       T beta, sycl::buffer<T, 1> &c, std::int64_t ldc)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether  $A$ 's data is stored in its upper or lower triangle. See [oneMKL defined datatypes](#) for more details.

**trans** Specifies `op(A)`, the transposition operation applied to  $A$  (See [oneMKL defined datatypes](#) for more details). Conjugation is never performed, even if `trans = transpose::conjtrans`.

**n** Number of rows and columns in  $C$ . The value of  $n$  must be at least zero.

**k** Number of columns in `op(A)`. The value of  $k$  must be at least zero.

**alpha** Scaling factor for the rank-k update.

**a** Buffer holding input matrix  $A$ . If `trans = transpose::nontrans`,  $A$  is an n-by-k matrix so the array `a` must have size at least `lda*k`. Otherwise,  $A$  is an k-by-n matrix so the array `a` must have size at least `lda*n`. See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of A. Must be at least n if A is not transposed, and at least k if A is transposed. Must be positive.

**beta** Scaling factor for matrix C.

**c** Buffer holding input/output matrix C. Must have size at least ldc\*n. See [Matrix and Vector Storage](#) for more details.

**ldc** Leading dimension of C. Must be positive and at least n.

## Output Parameters

**c** Output buffer, overwritten by  $\alpha \operatorname{op}(A) \operatorname{op}(A)^T + \beta C$ .

### 13.2.1.2.3.18 syrk (USM Version)

#### Syntax

```
sycl::event onemkl::blas::syrk(sycl::queue &queue, onemkl::uplo upper_lower, onemkl::transpose
trans, std::int64_t n, std::int64_t k, T alpha, const T
*ia, std::int64_t lda, T beta, T *c, std::int64_t ldc, const
sycl::vector_class<sycl::event> &dependencies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A's data is stored in its upper or lower triangle. See [oneMKL defined datatypes](#) for more details.

**trans** Specifies op(A), the transposition operation applied to A (See [oneMKL defined datatypes](#) for more details). Conjugation is never performed, even if *trans* = transpose::conjtrans.

**n** Number of rows and columns in C. The value of n must be at least zero.

**k** Number of columns in op(A). The value of k must be at least zero.

**alpha** Scaling factor for the rank-k update.

**a** Pointer to input matrix A. If *trans* = transpose::nontrans, A is an n-by-k matrix so the array a must have size at least lda\*k. Otherwise, A is an k-by-n matrix so the array a must have size at least lda\*n. See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of A. Must be at least n if A is not transposed, and at least k if A is transposed. Must be positive.

**beta** Scaling factor for matrix C.

**c** Pointer to input/output matrix C. Must have size at least ldc\*n. See [Matrix and Vector Storage](#) for more details.

**ldc** Leading dimension of C. Must be positive and at least n.

## Output Parameters

**c** Pointer to the output matrix, overwritten by  $\alpha \cdot \text{op}(A) \cdot \text{op}(A)^T + \beta \cdot C$ .

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 3 Routines*

### 13.2.1.2.3.19 `syr2k`

Performs a symmetric rank-2k update.

`syr2k` supports the following precisions:

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The `syr2k` routines perform a rank-2k update of an  $n \times n$  symmetric matrix  $C$  by general matrices  $A$  and  $B$ . If `trans = transpose::nontrans`, the operation is defined as:

$$C \leftarrow \alpha \cdot (A \cdot B^T + B \cdot A^T) + \beta \cdot C$$

where  $A$  is  $n \times k$  and  $B$  is  $k \times n$ .

If `trans = transpose::trans`, the operation is defined as:

$$C \leftarrow \alpha \cdot (A^T \cdot B + B^T \cdot A) + \beta \cdot C$$

where  $A$  is  $k \times n$  and  $B$  is  $n \times k$ .

In both cases:

`alpha` and `beta` are scalars,

$C$  is a symmetric matrix and  $A, B$  are general matrices,

The inner dimension of both matrix multiplications is  $k$ .

### 13.2.1.2.3.20 `syr2k` (Buffer Version)

## Syntax

```
void onemkl::blas::syr2k(sycl::queue &queue, onemkl::uplo upper_lower, onemkl::transpose trans,
                         std::int64_t n, std::int64_t k, T alpha, sycl::buffer<T, 1> &a, std::int64_t lda,
                         sycl::buffer<T, 1> &b, std::int64_t ldb, T beta, sycl::buffer<T, 1> &c,
                         std::int64_t ldc)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A's data is stored in its upper or lower triangle. See [oneMKL defined datatypes](#) for more details.

**trans** Specifies the operation to apply, as described above. Conjugation is never performed, even if `trans = transpose::conjtrans`.

**n** Number of rows and columns in C. The value of n must be at least zero.

**k** Inner dimension of matrix multiplications. The value of k must be at least zero.

**alpha** Scaling factor for the rank-2k update.

**a** Buffer holding input matrix A. If A is not transposed, A is an m-by-k matrix so the array a must have size at least `lda*k`. If A is transposed, A is an k-by-m matrix so the array a must have size at least `lda*m`. See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of A. Must be at least n if `trans = transpose::nontrans`, and at least k otherwise. Must be positive.

**b** Buffer holding input matrix B. If `trans = transpose::nontrans`, B is an k-by-n matrix so the array b must have size at least `ldb*n`. Otherwise, B is an n-by-k matrix so the array b must have size at least `ldb*k`. See [Matrix and Vector Storage](#) for more details.

**ldb** Leading dimension of B. Must be at least k if `trans = transpose::nontrans`, and at least n otherwise. Must be positive.

**beta** Scaling factor for matrix C.

**c** Buffer holding input/output matrix C. Must have size at least `ldc*n`. See [Matrix and Vector Storage](#) for more details

**ldc** Leading dimension of C. Must be positive and at least n.

## Output Parameters

**c** Output buffer, overwritten by the updated C matrix.

### 13.2.1.2.3.21 `syr2k` (USM Version)

#### Syntax

```
sycl::event onemkl::blas::syr2k(sycl::queue &queue, onemkl::uplo upper_lower, onemkl::transpose trans, std::int64_t n, std::int64_t k, T alpha, const T *a, std::int64_t lda, const T *b, std::int64_t ldb, T beta, T *c, std::int64_t ldc, const sycl::vector_class<sycl::event> &dependencies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether A's data is stored in its upper or lower triangle. See [oneMKL defined datatypes](#) for more details.

**trans** Specifies the operation to apply, as described above. Conjugation is never performed, even if `trans = transpose::conjtrans`.

**n** Number of rows and columns in C. The value of n must be at least zero.

**k** Inner dimension of matrix multiplications. The value of k must be at least zero.

**alpha** Scaling factor for the rank-2k update.

**a** Pointer to input matrix A. If A is not transposed, A is an m-by-k matrix so the array a must have size at least `lda*k`. If A is transposed, A is an k-by-m matrix so the array a must have size at least `lda*m`. See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of A. Must be at least n if `trans = transpose::nontrans`, and at least k otherwise. Must be positive.

**b** Pointer to input matrix B. If `trans = transpose::nontrans`, B is an k-by-n matrix so the array b must have size at least `ldb*n`. Otherwise, B is an n-by-k matrix so the array b must have size at least `ldb*k`. See [Matrix and Vector Storage](#) for more details.

**ldb** Leading dimension of B. Must be at least k if `trans = transpose::nontrans`, and at least n otherwise. Must be positive.

**beta** Scaling factor for matrix C.

**c** Pointer to input/output matrix C. Must have size at least `ldc*n`. See [Matrix and Vector Storage](#) for more details

**ldc** Leading dimension of C. Must be positive and at least n.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**c** Pointer to the output matrix, overwritten by the updated C matrix.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** [BLAS Level 3 Routines](#)

### 13.2.1.2.3.22 trmm

Computes a matrix-matrix product where one input matrix is triangular and one input matrix is general.

`trmm` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The `trmm` routines compute a scalar-matrix-matrix product where one of the matrices in the multiplication is triangular. The argument `left_right` determines if the triangular matrix,  $A$ , is on the left of the multiplication (`left_right = side::left`) or on the right (`left_right = side::right`). Depending on `left_right`. The operation is defined as

$$B \leftarrow \alpha * \text{op}(A) * B$$

or

$$B \leftarrow \alpha * B * \text{op}(A)$$

where:

`op(A)` is one of  $\text{op}(A) = A$ , or  $\text{op}(A) = A^T$ , or  $\text{op}(A) = A^H$ ,

`alpha` is a scalar,

$A$  is a triangular matrix, and  $B$  is a general matrix.

Here  $B$  is  $m \times n$  and  $A$  is either  $m \times m$  or  $n \times n$ , depending on `left_right`.

### 13.2.1.2.3.23 `trmm` (Buffer Version)

#### Syntax

```
void onemkl::blas::trmm(sycl::queue &queue, onemkl::uplo upper_lower, onemkl::transpose transa,
                        onemkl::diag unit_diag, std::int64_t m, std::int64_t n, T alpha, sycl::buffer<T,
                        1> &a, std::int64_t lda, sycl::buffer<T, 1> &b, std::int64_t ldb)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**left\_right** Specifies whether  $A$  is on the left side of the multiplication (`side::left`) or on the right side (`side::right`). See [oneMKL defined datatypes](#) for more details.

**uplo** Specifies whether the matrix  $A$  is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**trans** Specifies  $\text{op}(A)$ , the transposition operation applied to  $A$ . See [oneMKL defined datatypes](#) for more details.

**unit\_diag** Specifies whether  $A$  is assumed to be unit triangular (all diagonal elements are 1). See [oneMKL defined datatypes](#) for more details.

**m** Specifies the number of rows of  $B$ . The value of  $m$  must be at least zero.

**n** Specifies the number of columns of  $B$ . The value of  $n$  must be at least zero.

**alpha** Scaling factor for the matrix-matrix product.

**a** Buffer holding input matrix  $A$ . Must have size at least  $\text{lda} * m$  if `left_right = side::left`, or  $\text{lda} * n$  if `left_right = side::right`. See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of  $A$ . Must be at least  $m$  if `left_right = side::left`, and at least  $n$  if `left_right = side::right`. Must be positive.

**b** Buffer holding input/output matrix  $B$ . Must have size at least  $\text{ldb} * n$ . See [Matrix and Vector Storage](#) for more details.

**ldb** Leading dimension of  $B$ . Must be at least  $m$  and positive.

## Output Parameters

**b** Output buffer, overwritten by  $\alpha \cdot \text{op}(A) \cdot B$  or  $\alpha \cdot B \cdot \text{op}(A)$ .

## Notes

If  $\alpha = 0$ , matrix  $B$  is set to zero, and  $A$  and  $B$  do not need to be initialized at entry.

### 13.2.1.2.3.24 trmm (USM Version)

#### Syntax

```
sycl::event onemkl::blas::trmm(sycl::queue &queue, onemkl::uplo upper_lower, onemkl::transpose
    transa, onemkl::diag unit_diag, std::int64_t m, std::int64_t n, T alpha, const T *a, std::int64_t lda, T *b, std::int64_t ldb, const
    sycl::vector_class<sycl::event> &dependencies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**left\_right** Specifies whether  $A$  is on the left side of the multiplication (`side::left`) or on the right side (`side::right`). See [oneMKL defined datatypes](#) for more details.

**uplo** Specifies whether the matrix  $A$  is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**trans** Specifies  $\text{op}(A)$ , the transposition operation applied to  $A$ . See [oneMKL defined datatypes](#) for more details.

**unit\_diag** Specifies whether  $A$  is assumed to be unit triangular (all diagonal elements are 1). See [oneMKL defined datatypes](#) for more details.

**m** Specifies the number of rows of  $B$ . The value of  $m$  must be at least zero.

**n** Specifies the number of columns of  $B$ . The value of  $n$  must be at least zero.

**alpha** Scaling factor for the matrix-matrix product.

**a** Pointer to input matrix  $A$ . Must have size at least  $\text{lda} \cdot m$  if `left_right = side::left`, or  $\text{lda} \cdot n$  if `left_right = side::right`. See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of  $A$ . Must be at least  $m$  if `left_right = side::left`, and at least  $n$  if `left_right = side::right`. Must be positive.

**b** Pointer to input/output matrix  $B$ . Must have size at least  $\text{ldb} \cdot n$ . See [Matrix and Vector Storage](#) for more details.

**ldb** Leading dimension of  $B$ . Must be at least  $m$  and positive.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**b** Pointer to the output matrix, overwritten by  $\alpha \cdot \text{op}(A) \cdot B$  or  $\alpha \cdot B \cdot \text{op}(A)$ .

## Notes

If  $\alpha = 0$ , matrix  $B$  is set to zero, and  $A$  and  $B$  do not need to be initialized at entry.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 3 Routines*

### 13.2.1.2.3.25 trsm

Solves a triangular matrix equation (forward or backward solve).

`trsm` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The `trsm` routines solve one of the following matrix equations:

$$\text{op}(A) \cdot X = \alpha \cdot B,$$

or

$$X \cdot \text{op}(A) = \alpha \cdot B,$$

where:

$\text{op}(A)$  is one of  $\text{op}(A) = A$ , or  $\text{op}(A) = A^T$ , or  $\text{op}(A) = A^H$ ,

$\alpha$  is a scalar,

$A$  is a triangular matrix, and

$B$  and  $X$  are  $m \times n$  general matrices.

$A$  is either  $m \times m$  or  $n \times n$ , depending on whether it multiplies  $X$  on the left or right. On return, the matrix  $B$  is overwritten by the solution matrix  $X$ .

### 13.2.1.2.3.26 trsm (Buffer Version)

#### Syntax

```
void onemkl::blas::trsm(sycl::queue &queue, onemkl::side left_right, onemkl::uplo upper_lower,
onemkl::transpose transa, onemkl::diag unit_diag, std::int64_t m, std::int64_t n, T alpha, sycl::buffer<T, 1> &a, std::int64_t lda, sycl::buffer<T, 1> &b, std::int64_t ldb)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**left\_right** Specifies whether A multiplies X on the left (`side::left`) or on the right (`side::right`). See [oneMKL defined datatypes](#) for more details.

**uplo** Specifies whether the matrix A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**trans** Specifies  $\text{op}(A)$ , the transposition operation applied to A. See [oneMKL defined datatypes](#) for more details.

**unit\_diag** Specifies whether A is assumed to be unit triangular (all diagonal elements are 1). See [oneMKL defined datatypes](#) for more details.

**m** Specifies the number of rows of B. The value of m must be at least zero.

**n** Specifies the number of columns of B. The value of n must be at least zero.

**alpha** Scaling factor for the solution.

**a** Buffer holding input matrix A. Must have size at least  $\text{lda} \times m$  if `left_right = side::left`, or  $\text{lda} \times n$  if `left_right = side::right`. See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of A. Must be at least m if `left_right = side::left`, and at least n if `left_right = side::right`. Must be positive.

**b** Buffer holding input/output matrix B. Must have size at least  $\text{ldb} \times n$ . See [Matrix and Vector Storage](#) for more details.

**ldb** Leading dimension of B. Must be at least m and positive.

#### Output Parameters

**b** Output buffer. Overwritten by the solution matrix X.

#### Notes

If `alpha = 0`, matrix B is set to zero, and A and B do not need to be initialized at entry.

### 13.2.1.2.3.27 trsm (USM Version)

#### Syntax

```
sycl::event onemkl::blas::trsm(sycl::queue &queue, onemkl::side left_right, onemkl::uplo upper_lower, onemkl::transpose transa, onemkl::diag unit_diag, std::int64_t m, std::int64_t n, T alpha, const T *a, std::int64_t lda, T *b, std::int64_t ldb, const sycl::vector_class<sycl::event> &dependencies = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**left\_right** Specifies whether A multiplies X on the left (`side::left`) or on the right (`side::right`). See [oneMKL defined datatypes](#) for more details.

**uplo** Specifies whether the matrix A is upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**transa** Specifies op(A), the transposition operation applied to A. See [oneMKL defined datatypes](#) for more details.

**unit\_diag** Specifies whether A is assumed to be unit triangular (all diagonal elements are 1). See [oneMKL defined datatypes](#) for more details.

**m** Specifies the number of rows of B. The value of m must be at least zero.

**n** Specifies the number of columns of B. The value of n must be at least zero.

**alpha** Scaling factor for the solution.

**a** Pointer to input matrix A. Must have size at least lda\*m if `left_right = side::left`, or lda\*n if `left_right = side::right`. See [Matrix and Vector Storage](#) for more details.

**lda** Leading dimension of A. Must be at least m if `left_right = side::left`, and at least n if `left_right = side::right`. Must be positive.

**b** Pointer to input/output matrix B. Must have size at least ldb\*n. See [Matrix and Vector Storage](#) for more details.

**ldb** Leading dimension of B. Must be at least m and positive.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

#### Output Parameters

**b** Pointer to the output matrix. Overwritten by the solution matrix X.

#### Notes

If `alpha = 0`, matrix B is set to zero, and A and B do not need to be initialized at entry.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS Level 3 Routines*

**Parent topic:** *BLAS Routines*

### 13.2.1.2.4 BLAS-like Extensions

oneAPI Math Kernel Library DPC++ provides additional routines to extend the functionality of the BLAS routines. These include routines to compute many independent vector-vector and matrix-matrix operations.

The following table lists the BLAS-like extensions with their descriptions.

Routines	Description
<code>axpy_batch</code>	Computes groups of vector-scalar products added to a vector.
<code>gemm_batch</code>	Computes groups of matrix-matrix products with general matrices.
<code>trsm_batch</code>	Solves a triangular matrix equation for a group of matrices.
<code>gemmt</code>	Computes a matrix-matrix product with general matrices, but updates only the upper or lower triangular part of the result matrix.
<code>gemm_bias</code>	Computes a matrix-matrix product using general integer matrices with bias

#### 13.2.1.2.4.1 `axpy_batch`

The `axpy_batch` routines are batched versions of `axpy`, performing multiple `axpy` operations in a single call. Each `axpy` operation adds a scalar-vector product to a vector.

`axpy_batch` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

#### 13.2.1.2.4.2 `axpy_batch` (Buffer Version)

##### Description

The buffer version of `axpy_batch` supports only the strided API.

The strided API operation is defined as

```
for i = 0 ... batch_size - 1
    X and Y are vectors at offset i * stridex, i * stridey in x and y
    Y := alpha * X + Y
end for
```

where:

`alpha` is scalar

`X` and `Y` are vectors.

## Strided API

### Syntax

```
void onemkl::blas::axpy_batch(sycl::queue &queue, std::int64_t n, T alpha, sycl::buffer<T, 1> &x, std::int64_t incx, std::int64_t stridex, sycl::buffer<T, 1> &y, std::int64_t incy, std::int64_t stridey, std::int64_t batch_size)
```

### Input Parameters

**queue** The queue where the routine should be executed.

**n** Number of elements in X and Y.

**alpha** Specifies the scalar alpha.

**x** Buffer holding input vectors X with size stridex\*batch\_size.

**incx** Stride of vector X.

**stridex** Stride between different X vectors.

**y** Buffer holding input/output vectors Y with size stridey\*batch\_size.

**incy** Stride of vector Y.

**stridey** Stride between different Y vectors.

**batch\_size** Specifies the number of axpy operations to perform.

### Output Parameters

**y** Output buffer, overwritten by batch\_size axpy operations of the form  $\text{alpha} \cdot \text{X} + \text{Y}$ .

#### 13.2.1.2.4.3 **axpy\_batch** (USM Version)

### Description

The USM version of **axpy\_batch** supports the group API and strided API.

The group API operation is defined as

```
idx = 0
for i = 0 ... group_count - 1
    for j = 0 ... group_size - 1
        X and Y are vectors in x[idx] and y[idx]
        Y := alpha[i] * X + Y
        idx := idx + 1
    end for
end for
```

The strided API operation is defined as

```
for i = 0 ... batch_size - 1
    X and Y are vectors at offset i * stridex, i * stridey in x and y
    Y := alpha * X + Y
end for
```

where:

`alpha` is scalar

`X` and `Y` are vectors.

For group API, `x` and `y` arrays contain the pointers for all the input vectors. The total number of vectors in `x` and `y` are given by:

`total_batch_count` = sum of all of the `group_size` entries

For strided API, `x` and `y` arrays contain all the input vectors. The total number of vectors in `x` and `y` are given by the `batch_size` parameter.

## Group API

### Syntax

```
sycl::event onemkl::blas::axpy_batch(sycl::queue &queue, std::int64_t *n, T *alpha, const
                                         T **x, std::int64_t *incx, T **y, std::int64_t *incy,
                                         std::int64_t group_count, std::int64_t *group_size, const
                                         sycl::vector_class<sycl::event> &dependencies = {})
```

### Input Parameters

**queue** The queue where the routine should be executed.

**n** Array of `group_count` integers. `n[i]` specifies the number of elements in vectors `X` and `Y` for every vector in group `i`.

**alpha** Array of `group_count` scalar elements. `alpha[i]` specifies the scaling factor for vector `X` in group `i`.

**x** Array of pointers to input vectors `X` with size `total_batch_count`. The size of array allocated for the `X` vector of the group `i` must be at least  $(1 + (n[i] - 1) * \text{abs}(incx[i]))$ . See [Matrix and Vector Storage](#) for more details.

**incx** Array of `group_count` integers. `incx[i]` specifies the stride of vector `X` in group `i`.

**y** Array of pointers to input/output vectors `Y` with size `total_batch_count`. The size of array allocated for the `Y` vector of the group `i` must be at least  $(1 + (n[i] - 1) * \text{abs}(incy[i]))$ . See [Matrix and Vector Storage](#) for more details.

**incy** Array of `group_count` integers. `incy[i]` specifies the stride of vector `Y` in group `i`.

**group\_count** Number of groups. Must be at least 0.

**group\_size** Array of `group_count` integers. `group_size[i]` specifies the number of `axpy` operations in group `i`. Each element in `group_size` must be at least 0.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**y** Array of pointers holding the  $\mathbf{Y}$  vectors, overwritten by `total_batch_count` `axpy` operations of the form  $\alpha * \mathbf{X} + \mathbf{Y}$ .

## Return Values

Output event to wait on to ensure computation is complete.

## Strided API

### Syntax

```
sycl::event onemkl::blas::axpy_batch(sycl::queue &queue, std::int64_t n, T alpha, const T
                                         *x, std::int64_t incx, std::int64_t stridex, T *y, std::int64_t
                                         incy, std::int64_t stridey, std::int64_t batch_size, const
                                         sycl::vector_class<sycl::event> &dependencies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**n** Number of elements in  $\mathbf{X}$  and  $\mathbf{Y}$ .

**alpha** Specifies the scalar  $\alpha$ .

**x** Pointer to input vectors  $\mathbf{X}$  with size `stridex*batch_size`.

**incx** Stride of vector  $\mathbf{X}$ .

**stridex** Stride between different  $\mathbf{X}$  vectors.

**y** Pointer to input/output vectors  $\mathbf{Y}$  with size `stridey*batch_size`.

**incy** Stride of vector  $\mathbf{Y}$ .

**stridey** Stride between different  $\mathbf{Y}$  vectors.

**batch\_size** Specifies the number of `axpy` operations to perform.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**y** Output vectors, overwritten by `batch_size` `axpy` operations of the form  $\alpha * \mathbf{X} + \mathbf{Y}$ .

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS-like Extensions*

### 13.2.1.2.4.4 gemm\_batch

The `gemm_batch` routines are batched versions of `gemm`, performing multiple `gemm` operations in a single call. Each `gemm` operation perform a matrix-matrix product with general matrices.

`gemm_batch` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

### 13.2.1.2.4.5 gemm\_batch (Buffer Version)

#### Description

The buffer version of `gemm_batch` supports only the strided API.

The strided API operation is defined as

```
for i = 0 ... batch_size - 1
    A, B and C are matrices at offset i * stridea, i * strideb, i * stridec in a, b,
    ↵and c.
    C := alpha * op(A) * op(B) + beta * C
end for
```

where:

`op(X)` is one of  $op(X) = X$ , or  $op(X) = X^T$ , or  $op(X) = X^H$

`alpha` and `beta` are scalars

`A`, `B`, and `C` are matrices

`op(A)` is  $m \times k$ , `op(B)` is  $k \times n$ , and `C` is  $m \times n$ .

The `a`, `b` and `c` buffers contain all the input matrices. The stride between matrices is given by the `stride` parameter. The total number of matrices in `a`, `b` and `c` buffers is given by the `batch_size` parameter.

#### Strided API

#### Syntax

```
void onemkl::blas::gemm_batch(sycl::queue &queue, onemkl::transpose transa, onemkl::transpose
                               transb, std::int64_t m, std::int64_t n, std::int64_t k, T alpha,
                               sycl::buffer<T, 1> &a, std::int64_t lda, std::int64_t stridea,
                               sycl::buffer<T, 1> &b, std::int64_t ldb, std::int64_t strideb, T
                               beta, sycl::buffer<T, 1> &c, std::int64_t ldc, std::int64_t stridec,
                               std::int64_t batch_size)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**transa** Specifies  $\text{op}(A)$  the transposition operation applied to the matrices A. See [oneMKL defined datatypes](#) for more details.

**transb** Specifies  $\text{op}(B)$  the transposition operation applied to the matrices B. See [oneMKL defined datatypes](#) for more details.

**m** Number of rows of  $\text{op}(A)$  and C. Must be at least zero.

**n** Number of columns of  $\text{op}(B)$  and C. Must be at least zero.

**k** Number of columns of  $\text{op}(A)$  and rows of  $\text{op}(B)$ . Must be at least zero.

**alpha** Scaling factor for the matrix-matrix products.

**a** Buffer holding the input matrices A with size `stridea*batch_size`.

**lda** Leading dimension of the matrices A. Must be at least m if the matrices A are not transposed, and at least k if the matrices A are transposed. Must be positive.

**stridea** Stride between different A matrices.

**b** Buffer holding the input matrices B with size `strideb*batch_size`.

**ldb** Leading dimension of the matrices B. Must be at least k if the matrices B are not transposed, and at least n if the matrices B are transposed. Must be positive.

**strideb** Stride between different B matrices.

**beta** Scaling factor for the matrices C.

**c** Buffer holding input/output matrices C with size `stridec*batch_size`.

**ldc** Leading dimension of C. Must be positive and at least m.

**stridec** Stride between different C matrices. Must be at least ldc\*n.

**batch\_size** Specifies the number of matrix multiply operations to perform.

## Output Parameters

**c** Output buffer, overwritten by `batch_size` matrix multiply operations of the form  $\text{alpha} \cdot \text{op}(A) \cdot \text{op}(B) + \text{beta} \cdot C$ .

## Notes

If  $\text{beta} = 0$ , matrix C does not need to be initialized before calling `gemm_batch`.

### 13.2.1.2.4.6 gemm\_batch (USM Version)

#### Description

The USM version of `gemm_batch` supports the group API and strided API.

The group API operation is defined as

```
idx = 0
for i = 0 ... group_count - 1
    for j = 0 ... group_size - 1
        A, B, and C are matrices in a[idx], b[idx] and c[idx]
        C := alpha[i] * op(A) * op(B) + beta[i] * C
        idx = idx + 1
    end for
end for
```

The strided API operation is defined as

```
for i = 0 ... batch_size - 1
    A, B and C are matrices at offset i * stridea, i * strideb, i * stridec in a, b
    ↪and c.
    C := alpha * op(A) * op(B) + beta * C
end for
```

where:

`op(X)` is one of  $\text{op}(X) = X$ , or  $\text{op}(X) = X^T$ , or  $\text{op}(X) = X^H$

`alpha` and `beta` are scalars

`A`, `B`, and `C` are matrices

`op(A)` is  $m \times k$ , `op(B)` is  $k \times n$ , and `C` is  $m \times n$ .

For group API, `a`, `b` and `c` arrays contain the pointers for all the input matrices. The total number of matrices in `a`, `b` and `c` are given by:

`total_batch_count` = sum of all of the `group_size` entries

For strided API, `a`, `b`, `c` arrays contain all the input matrices. The total number of matrices in `a`, `b` and `c` are given by the `batch_size` parameter.

#### Group API

#### Syntax

```
sycl::event onemkl::blas::gemm_batch(sycl::queue      &queue,      onemkl::transpose      *transa,
                                         onemkl::transpose      *transb,      std::int64_t      *m,      std::int64_t
                                         *n,      std::int64_t      *k,      T      *alpha,      const T      **a,      std::int64_t      *lda,
                                         const T      **b,      std::int64_t      *ldb,      T      *beta,      T      **c,      std::int64_t
                                         *ldc,      std::int64_t      group_count,      std::int64_t      *group_size,
                                         const sycl::vector_class<sycl::event>      &dependencies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**transa** Array of group\_count onemkl::transpose values. `transa[i]` specifies the form of  $\text{op}(A)$  used in the matrix multiplication in group  $i$ . See [oneMKL defined datatypes](#) for more details.

**transb** Array of group\_count onemkl::transpose values. `transb[i]` specifies the form of  $\text{op}(B)$  used in the matrix multiplication in group  $i$ . See [oneMKL defined datatypes](#) for more details.

**m** Array of group\_count integers. `m[i]` specifies the number of rows of  $\text{op}(A)$  and  $C$  for every matrix in group  $i$ . All entries must be at least zero.

**n** Array of group\_count integers. `n[i]` specifies the number of columns of  $\text{op}(B)$  and  $C$  for every matrix in group  $i$ . All entries must be at least zero.

**k** Array of group\_count integers. `k[i]` specifies the number of columns of  $\text{op}(A)$  and rows of  $\text{op}(B)$  for every matrix in group  $i$ . All entries must be at least zero.

**alpha** Array of group\_count scalar elements. `alpha[i]` specifies the scaling factor for every matrix-matrix product in group  $i$ .

**a** Array of pointers to input matrices  $A$  with size `total_batch_count`.

See [Matrix Storage](#) for more details.

**lda** Array of group\_count integers. `lda[i]` specifies the leading dimension of  $A$  for every matrix in group  $i$ . All entries must be at least  $m$  if  $A$  is not transposed, and at least  $k$  if  $A$  is transposed. All entries must be positive.

**b** Array of pointers to input matrices  $B$  with size `total_batch_count`.

See [Matrix Storage](#) for more details.

**ldb** Array of group\_count integers. `ldb[i]` specifies the leading dimension of  $B$  for every matrix in group  $i$ . All entries must be at least  $k$  if  $B$  is not transposed, and at least  $n$  if  $B$  is transposed. All entries must be positive.

**beta** Array of group\_count scalar elements. `beta[i]` specifies the scaling factor for matrix  $C$  for every matrix in group  $i$ .

**c** Array of pointers to input/output matrices  $C$  with size `total_batch_count`.

See [Matrix Storage](#) for more details.

**ldc** Array of group\_count integers. `ldc[i]` specifies the leading dimension of  $C$  for every matrix in group  $i$ . All entries must be positive and at least  $m$ .

**group\_count** Specifies the number of groups. Must be at least 0.

**group\_size** Array of group\_count integers. `group_size[i]` specifies the number of matrix multiply products in group  $i$ . All entries must be at least 0.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**c** Overwritten by the  $m[i]$ -by- $n[i]$  matrix calculated by  $(\text{alpha}[i] * \text{op}(A) * \text{op}(B) + \text{beta}[i] * C)$  for group  $i$ .

## Notes

If  $\text{beta} = 0$ , matrix  $C$  does not need to be initialized before calling `gemm_batch`.

## Return Values

Output event to wait on to ensure computation is complete.

## Strided API

### Syntax

```
sycl::event onemkl::blas::gemm_batch(sycl::queue &queue, onemkl::transpose transa,
                                      onemkl::transpose transb, std::int64_t m, std::int64_t
                                      n, std::int64_t k, T alpha, const T *a, std::int64_t
                                      lda, std::int64_t stridea, const T *b, std::int64_t
                                      ldb, std::int64_t strideb, T beta, T *c, std::int64_t
                                      ldc, std::int64_t stridec, std::int64_t batch_size, const
                                      sycl::vector_class<sycl::event> &dependencies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**transa** Specifies  $\text{op}(A)$  the transposition operation applied to the matrices  $A$ . See [oneMKL defined datatypes](#) for more details.

**transb** Specifies  $\text{op}(B)$  the transposition operation applied to the matrices  $B$ . See [oneMKL defined datatypes](#) for more details.

**m** Number of rows of  $\text{op}(A)$  and  $C$ . Must be at least zero.

**n** Number of columns of  $\text{op}(B)$  and  $C$ . Must be at least zero.

**k** Number of columns of  $\text{op}(A)$  and rows of  $\text{op}(B)$ . Must be at least zero.

**alpha** Scaling factor for the matrix-matrix products.

**a** Pointer to input matrices  $A$  with size `stridea*batch_size`.

**lda** Leading dimension of the matrices  $A$ . Must be at least  $m$  if the matrices  $A$  are not transposed, and at least  $k$  if the matrices  $A$  are transposed. Must be positive.

**stridea** Stride between different  $A$  matrices.

**b** Pointer to input matrices  $B$  with size `strideb*batch_size`.

**ldb** Leading dimension of the matrices  $B$ . Must be at least  $k$  if the matrices  $B$  are not transposed, and at least  $n$  if the matrices  $B$  are transposed. Must be positive.

**strideb** Stride between different  $B$  matrices.

**beta** Scaling factor for the matrices  $C$ .

**c** Pointer to input/output matrices C with size `stridec*batch_size`.

**ldc** Leading dimension of C. Must be positive and at least m.

**stridec** Stride between different C matrices.

**batch\_size** Specifies the number of matrix multiply operations to perform.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**c** Output matrices, overwritten by `batch_size` matrix multiply operations of the form `alpha*op(A) *op(B) + beta*C`.

## Notes

If `beta = 0`, matrix C does not need to be initialized before calling `gemm_batch`.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS-like Extensions*

### 13.2.1.2.4.7 `trsm_batch`

The `trsm_batch` routines are batched versions of `trsm`, performing multiple `trsm` operations in a single call. Each `trsm` solves an equation of the form `op(A) * X = alpha * B` or `X * op(A) = alpha * B`.

`trsm_batch` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

### 13.2.1.2.4.8 `trsm_batch` (Buffer Version)

## Description

The buffer version of `trsm_batch` supports only the strided API.

The strided API operation is defined as

```
for i = 0 ... batch_size - 1
    A and B are matrices at offset i * stridea and i * strideb in a and b.
    if (left_right == onemkl::side::left) then
        compute X such that op(A) * X = alpha * B
    else
        compute X such that X * op(A) = alpha * B
    end if
```

(continues on next page)

(continued from previous page)

```
B := X
end for
```

where:

$\text{op}(A)$  is one of  $\text{op}(A) = A$ , or  $\text{op}(A) = A^T$ , or  $\text{op}(A) = A^H$

$\alpha$  is a scalar

$A$  is a triangular matrix

$B$  and  $X$  are  $m \times n$  general matrices

$A$  is either  $m \times m$  or  $n \times n$ , depending on whether it multiplies  $X$  on the left or right. On return, the matrix  $B$  is overwritten by the solution matrix  $X$ .

The  $a$  and  $b$  buffers contain all the input matrices. The stride between matrices is given by the stride parameter. The total number of matrices in  $a$  and  $b$  buffers are given by the `batch_size` parameter.

## Strided API

### Syntax

```
void onemkl::blas::trsm_batch(sycl::queue &queue, onemkl::side left_right, onemkl::uplo upper_lower, onemkl::transpose trans, onemkl::diag unit_diag, std::int64_t m, std::int64_t n, T alpha, sycl::buffer<T, 1> &a, std::int64_t lda, std::int64_t stridea, sycl::buffer<T, 1> &b, std::int64_t ldb, std::int64_t strideb, std::int64_t batch_size)
```

### Input Parameters

**queue** The queue where the routine should be executed.

**left\_right** Specifies whether the matrices  $A$  multiply  $X$  on the left (`side::left`) or on the right (`side::right`). See [oneMKL defined datatypes](#) for more details.

**upper\_lower** Specifies whether the matrices  $A$  are upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**trans** Specifies  $\text{op}(A)$ , the transposition operation applied to the matrices  $A$ . See [oneMKL defined datatypes](#) for more details.

**unit\_diag** Specifies whether the matrices  $A$  are assumed to be unit triangular (all diagonal elements are 1). See [oneMKL defined datatypes](#) for more details.

**m** Number of rows of the  $B$  matrices. Must be at least zero.

**n** Number of columns of the  $B$  matrices. Must be at least zero.

**alpha** Scaling factor for the solutions.

**a** Buffer holding the input matrices  $A$  with size `stridea*batch_size`.

**lda** Leading dimension of the matrices  $A$ . Must be at least  $m$  if `left_right = side::left`, and at least  $n$  if `left_right = side::right`. Must be positive.

**stridea** Stride between different  $A$  matrices.

**b** Buffer holding the input matrices  $B$  with size `strideb*batch_size`.

**ldb** Leading dimension of the matrices  $B$ . Must be at least  $m$ . Must be positive.

**strideb** Stride between different B matrices.

**batch\_size** Specifies the number of triangular linear systems to solve.

## Output Parameters

**b** Output buffer, overwritten by batch\_size solution matrices X.

## Notes

If alpha = 0, matrix B is set to zero and the matrices A and B do not need to be initialized before calling trsm\_batch.

### 13.2.1.2.4.9 trsm\_batch (USM Version)

#### Description

The USM version of trsm\_batch supports the group API and strided API.

The group API operation is defined as

```
idx = 0
for i = 0 ... group_count - 1
    for j = 0 ... group_size - 1
        A and B are matrices in a[idx] and b[idx]
        if (left_right == onemkl::side::left) then
            compute X such that op(A) * X = alpha[i] * B
        else
            compute X such that X * op(A) = alpha[i] * B
        end if
        B := X
        idx = idx + 1
    end for
end for
```

The strided API operation is defined as

```
for i = 0 ... batch_size - 1
    A and B are matrices at offset i * stridea and i * strideb in a and b.
    if (left_right == onemkl::side::left) then
        compute X such that op(A) * X = alpha * B
    else
        compute X such that X * op(A) = alpha * B
    end if
    B := X
end for
```

where:

op(A) is one of  $\text{op}(A) = A$ , or  $\text{op}(A) = A^T$ , or  $\text{op}(A) = A^H$

alpha is a scalar

A is a triangular matrix

B and X are  $m \times n$  general matrices

$A$  is either  $m \times m$  or  $n \times n$ , depending on whether it multiplies  $X$  on the left or right. On return, the matrix  $B$  is overwritten by the solution matrix  $X$ .

For group API,  $a$  and  $b$  arrays contain the pointers for all the input matrices. The total number of matrices in  $a$  and  $b$  are given by:

`total_batch_count = sum of all of the group_size entries`

For strided API,  $a$  and  $b$  arrays contain all the input matrices. The total number of matrices in  $a$  and  $b$  are given by the `batch_size` parameter.

## Group API

### Syntax

```
sycl::event onemkl::blas::trsm_batch(sycl::queue &queue, onemkl::side *left_right, onemkl::uplo
                                     *upper_lower, onemkl::transpose *trans, onemkl::diag
                                     *unit_diag, std::int64_t *m, std::int64_t *n, T *alpha,
                                     const T **a, std::int64_t *lda, T **b, std::int64_t *ldb,
                                     std::int64_t group_count, std::int64_t *group_size, const
                                     sycl::vector_class<sycl::event> &dependencies = {})
```

### Input Parameters

**queue** The queue where the routine should be executed.

**left\_right** Array of `group_count` `onemkl::side` values. `left_right[i]` specifies whether  $A$  multiplies  $X$  on the left (`side::left`) or on the right (`side::right`) for every `trsm` operation in group  $i$ . See [oneMKL defined datatypes](#) for more details.

**upper\_lower** Array of `group_count` `onemkl::uplo` values. `upper_lower[i]` specifies whether  $A$  is upper or lower triangular for every matrix in group  $i$ . See [oneMKL defined datatypes](#) for more details.

**trans** Array of `group_count` `onemkl::transpose` values. `trans[i]` specifies the form of  $\text{op}(A)$  used for every `trsm` operation in group  $i$ . See [oneMKL defined datatypes](#) for more details.

**unit\_diag** Array of `group_count` `onemkl::diag` values. `unit_diag[i]` specifies whether  $A$  is assumed to be unit triangular (all diagonal elements are 1) for every matrix in group  $i$ . See [oneMKL defined datatypes](#) for more details.

**m** Array of `group_count` integers. `m[i]` specifies the number of rows of  $B$  for every matrix in group  $i$ . All entries must be at least zero.

**n** Array of `group_count` integers. `n[i]` specifies the number of columns of  $B$  for every matrix in group  $i$ . All entries must be at least zero.

**alpha** Array of `group_count` scalar elements. `alpha[i]` specifies the scaling factor in group  $i$ .

**a** Array of pointers to input matrices  $A$  with size `total_batch_count`. See [Matrix Storage](#) for more details.

**lda** Array of `group_count` integers. `lda[i]` specifies the leading dimension of  $A$  for every matrix in group  $i$ . All entries must be at least  $m$  if `left_right` is `side::left`, and at least  $n$  if `left_right` is `side::right`. All entries must be positive.

**b** Array of pointers to input matrices  $B$  with size `total_batch_count`. See [Matrix Storage](#) for more details.

**ldb** Array of `group_count` integers. `ldb[i]` specifies the leading dimension of  $B$  for every matrix in group  $i$ . All entries must be at least  $m$  and positive.

**group\_count** Specifies the number of groups. Must be at least 0.

**group\_size** Array of group\_count integers. `group_size[i]` specifies the number of `trsm` operations in group `i`. All entries must be at least 0.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**b** Output buffer, overwritten by the `total_batch_count` solution matrices `X`.

## Notes

If `alpha = 0`, matrix `B` is set to zero and the matrices `A` and `B` do not need to be initialized before calling `trsm_batch`.

## Return Values

Output event to wait on to ensure computation is complete.

## Strided API

### Syntax

```
sycl::event onemkl::blas::trsm_batch(sycl::queue &queue, onemkl::side left_right, onemkl::uplo upper_lower, onemkl::transpose trans, onemkl::diag unit_diag, std::int64_t m, std::int64_t n, T alpha, const T *a, std::int64_t lda, std::int64_t stridea, T *b, std::int64_t ldb, std::int64_t strideb, std::int64_t batch_size, const sycl::vector_class<sycl::event> &dependencies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**left\_right** Specifies whether the matrices `A` multiply `X` on the left (`side::left`) or on the right (`side::right`). See [oneMKL defined datatypes](#) for more details.

**upper\_lower** Specifies whether the matrices `A` are upper or lower triangular. See [oneMKL defined datatypes](#) for more details.

**trans** Specifies op (`A`), the transposition operation applied to the matrices `A`. See [oneMKL defined datatypes](#) for more details.

**unit\_diag** Specifies whether the matrices `A` are assumed to be unit triangular (all diagonal elements are 1). See [oneMKL defined datatypes](#) for more details.

**m** Number of rows of the `B` matrices. Must be at least zero.

**n** Number of columns of the `B` matrices. Must be at least zero.

**alpha** Scaling factor for the solutions.

**a** Pointer to input matrices `A` with size `stridea*batch_size`.

**lda** Leading dimension of the matrices `A`. Must be at least `m` if `left_right` = ``side::left`, and at least `n` if `left_right = side::right`. Must be positive.

**stridea** Stride between different A matrices.

**b** Pointer to input matrices B with size `strideb*batch_size`.

**ldb** Leading dimension of the matrices B. Must be at least m. Must be positive.

**strideb** Stride between different B matrices.

**batch\_size** Specifies the number of triangular linear systems to solve.

## Output Parameters

**b** Output matrices, overwritten by `batch_size` solution matrices X.

## Notes

If `alpha = 0`, matrix B is set to zero and the matrices A and B do not need to be initialized before calling `trsm_batch`.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS-like Extensions*

### 13.2.1.2.4.10 gemmt

Computes a matrix-matrix product with general matrices, but updates only the upper or lower triangular part of the result matrix.

`gemmt` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The `gemmt` routines compute a scalar-matrix-matrix product and add the result to the upper or lower part of a scalar-matrix product, with general matrices. The operation is defined as:

```
C <- alpha*op(A)*op(B) + beta*C
```

where:

`op(X)` is one of  $op(X) = X$ , or  $op(X) = X^T$ , or  $op(X) = X^H$

`alpha` and `beta` are scalars

A, B, and C are matrices

`op(A)` is  $n \times k$ , `op(B)` is  $k \times n$ , and C is  $n \times n$ .

### 13.2.1.2.4.11 gemmt (Buffer Version)

#### Syntax

```
void onemkl::blas::gemmt(sycl::queue &queue, onemkl::uplo upper_lower, onemkl::transpose transa,
                         onemkl::transpose transb, std::int64_t n, std::int64_t k, T alpha,
                         sycl::buffer<T, 1> &a, std::int64_t lda, sycl::buffer<T, 1> &b, std::int64_t
                         ldb, T beta, sycl::buffer<T, 1> &c, std::int64_t ldc)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether C's data is stored in its upper or lower triangle. See [oneMKL defined datatypes](#) for more details.

**transa** Specifies op(A), the transposition operation applied to A. See [oneMKL defined datatypes](#) for more details.

**transb** Specifies op(B), the transposition operation applied to B. See [oneMKL defined datatypes](#) for more details.

**n** Number of columns of op(A), columns of op(B), and columns of C. Must be at least zero.

**k** Number of columns of op(A) and rows of op(B). Must be at least zero.

**alpha** Scaling factor for the matrix-matrix product.

**a** Buffer holding the input matrix A.

If A is not transposed, A is an n-by-k matrix so the array a must have size at least lda\*k.

If A is transposed, A is a k-by-n matrix so the array a must have size at least lda\*n.

See [Matrix Storage](#) for more details.

**lda** Leading dimension of A. Must be at least n if A is not transposed, and at least k if A is transposed. Must be positive.

**b** Buffer holding the input matrix B.

If B is not transposed, B is a k-by-n matrix so the array b must have size at least ldb\*n.

If B is transposed, B is an n-by-k matrix so the array b must have size at least ldb\*k.

See [Matrix Storage](#) for more details.

**ldb** Leading dimension of B. Must be at least k if B is not transposed, and at least n if B is transposed. Must be positive.

**beta** Scaling factor for matrix C.

**c** Buffer holding the input/output matrix C. Must have size at least ldc \* n. See [Matrix Storage](#) for more details.

**ldc** Leading dimension of C. Must be positive and at least m.

## Output Parameters

- c** Output buffer, overwritten by the upper or lower triangular part of  $\alpha \cdot \text{op}(A) \cdot \text{op}(B) + \beta \cdot C$ .

## Notes

If  $\beta = 0$ , matrix  $C$  does not need to be initialized before calling gemmt.

### 13.2.1.2.4.12 gemmt (USM Version)

## Syntax

```
sycl::event onemkl::blas::gemmt(sycl::queue &queue, onemkl::uplo upper_lower, onemkl::transpose transa, onemkl::transpose transb, std::int64_t n, std::int64_t k, T alpha, const T *a, std::int64_t lda, const T *b, std::int64_t ldb, T beta, T *c, std::int64_t ldc, const sycl::vector_class<sycl::event> &dependencies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Specifies whether  $C$ 's data is stored in its upper or lower triangle. See *oneMKL defined datatypes* for more details.

**transa** Specifies  $\text{op}(A)$ , the transposition operation applied to  $A$ . See *oneMKL defined datatypes* for more details.

**transb** Specifies  $\text{op}(B)$ , the transposition operation applied to  $B$ . See *oneMKL defined datatypes* for more details.

**n** Number of columns of  $\text{op}(A)$ , columns of  $\text{op}(B)$ , and columns of  $C$ . Must be at least zero.

**k** Number of columns of  $\text{op}(A)$  and rows of  $\text{op}(B)$ . Must be at least zero.

**alpha** Scaling factor for the matrix-matrix product.

**a** Pointer to input matrix  $A$ .

If  $A$  is not transposed,  $A$  is an  $n$ -by- $k$  matrix so the array  $a$  must have size at least  $\text{lda} \cdot k$ .

If  $A$  is transposed,  $A$  is a  $k$ -by- $n$  matrix so the array  $a$  must have size at least  $\text{lda} \cdot n$ .

See [Matrix Storage](#) for more details.

**lda** Leading dimension of  $A$ . Must be at least  $n$  if  $A$  is not transposed, and at least  $k$  if  $A$  is transposed. Must be positive.

**b** Pointer to input matrix  $B$ .

If  $B$  is not transposed,  $B$  is a  $k$ -by- $n$  matrix so the array  $b$  must have size at least  $\text{ldb} \cdot n$ .

If  $B$  is transposed,  $B$  is an  $n$ -by- $k$  matrix so the array  $b$  must have size at least  $\text{ldb} \cdot k$ .

See [Matrix Storage](#) for more details.

**ldb** Leading dimension of  $B$ . Must be at least  $k$  if  $B$  is not transposed, and at least  $n$  if  $B$  is transposed. Must be positive.

**beta** Scaling factor for matrix  $C$ .

**c** Pointer to input/output matrix  $C$ . Must have size at least  $\text{ldc} \cdot n$ . See [Matrix Storage](#) for more details.

**l<sub>dc</sub>** Leading dimension of C. Must be positive and at least m.

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**c** Pointer to the output matrix, overwritten by the upper or lower triangular part of alpha\*op(A)\*op(B) + beta\*C.

## Notes

If beta = 0, matrix C does not need to be initialized before calling gemmt.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *BLAS-like Extensions*

### 13.2.1.2.4.13 gemm\_bias

Computes a matrix-matrix product using general integer matrices with bias.

gemm\_bias supports the following precisions.

Ts	Ta	Tb	Tc
float	std::uint8_t	std::uint8_t	std::int32_t
float	std::int8_t	std::uint8_t	std::int32_t
float	std::uint8_t	std::int8_t	std::int32_t
float	std::int8_t	std::int8_t	std::int32_t

## Description

The gemm\_bias routines compute a scalar-matrix-matrix product and add the result to a scalar-matrix product, using general integer matrices with biases/offsets. The operation is defined as:

$$C \leftarrow \alpha * (\text{op}(A) - A\_offset) * (\text{op}(B) - B\_offset) + \beta * C + C\_offset$$

where:

$\text{op}(X)$  is one of  $\text{op}(X) = X$ , or  $\text{op}(X) = X^T$ , or  $\text{op}(X) = X^H$

$\alpha$  and  $\beta$  are scalars

$A\_offset$  is an  $m \times k$  matrix with every element equal to the value  $a_0$

$B\_offset$  is a  $k \times n$  matrix with every element equal to the value  $b_0$

$C\_offset$  is an  $m \times n$  matrix defined by the co buffer as described below.

$A$ ,  $B$ , and  $C$  are matrices

$\text{op}(A)$  is  $m \times k$ ,  $\text{op}(B)$  is  $k \times n$ , and  $C$  is  $m \times n$ .

### 13.2.1.2.4.14 gemm\_bias (Buffer Version)

#### Syntax

```
void onemkl::blas::gemm_bias(sycl::queue &queue, onemkl::transpose transa, onemkl::transpose
transb, onemkl::offset offset_type, std::int64_t m, std::int64_t n,
std::int64_t k, Ts alpha, sycl::buffer<Ta, 1> &a, std::int64_t lda,
Ta ao, sycl::buffer<Tb, 1> &b, std::int64_t ldb, Tb bo, Ts beta,
sycl::buffer<Tc, 1> &c, std::int64_t ldc, sycl::buffer<Tc, 1> &co)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**transa** Specifies op(A), the transposition operation applied to A. See [oneMKL defined datatypes](#) for more details.

**transb** Specifies op(B), the transposition operation applied to B. See [oneMKL defined datatypes](#) for more details.

**offset\_type** Specifies the form of C\_offset used in the matrix multiplication. See [oneMKL defined datatypes](#) for more details.

**m** Number of rows of op(A) and C. Must be at least zero.

**n** Number of columns of op(B) and C. Must be at least zero.

**k** Number of columns of op(A) and rows of op(B). Must be at least zero.

**alpha** Scaling factor for the matrix-matrix product.

**a** Buffer holding the input matrix A.

If A is not transposed, A is an m-by-k matrix so the array a must have size at least lda\*k.

If A is transposed, A is a k-by-m matrix so the array a must have size at least lda\*m.

See [Matrix Storage](#) for more details.

**lda** Leading dimension of A. Must be at least m if A is not transposed, and at least k if A is transposed. Must be positive.

**ao** Specifies the scalar offset value for matrix A.

**b** Buffer holding the input matrix B.

If B is not transposed, B is a k-by-n matrix so the array b must have size at least ldb\*n.

If B is transposed, B is an n-by-k matrix so the array b must have size at least ldb\*k.

See [Matrix Storage](#) for more details.

**ldb** Leading dimension of B. Must be at least k if B is not transposed, and at least n if B is transposed. Must be positive.

**bo** Specifies the scalar offset value for matrix B.

**beta** Scaling factor for matrix C.

**c** Buffer holding the input/output matrix C. Must have size at least ldc \* n. See [Matrix Storage](#) for more details.

**ldc** Leading dimension of C. Must be positive and at least m.

**co** Buffer holding the offset values for matrix C.

If offset\_type = offset::fix, the co array must have size at least 1.

If `offset_type` = `offset::col`, the `co` array must have size at least `max(1, m)`.

If `offset_type` = `offset::row`, the `co` array must have size at least `max(1, n)`.

## Output Parameters

`c` Output buffer, overwritten by  $\alpha * \text{op}(A) - A\_offset) * (\text{op}(B) - B\_offset) + \beta * C + C\_offset$ .

## Notes

If `beta` = 0, matrix `C` does not need to be initialized before calling `gemm_bias`.

### 13.2.1.2.4.15 `gemm_bias` (USM Version)

#### Syntax

```
sycl::event onemkl::blas::gemm_bias(sycl::queue      &queue,      onemkl::transpose      transa,
                                         onemkl::transpose      transb,      onemkl::offset      offset_type,
                                         std::int64_t m, std::int64_t n, std::int64_t k, Ts alpha, const
                                         Ta *a, std::int64_t lda, Ta ao, const Tb *b, std::int64_t ldb,
                                         Tb bo, Ts beta, Tc *c, std::int64_t ldc, const Tc *co, const
                                         sycl::vector_class<sycl::event> &dependencies = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**transa** Specifies `op(A)`, the transposition operation applied to `A`. See [oneMKL defined datatypes](#) for more details.

**transb** Specifies `op(B)`, the transposition operation applied to `B`. See [oneMKL defined datatypes](#) for more details.

**offset\_type** Specifies the form of `C_offset` used in the matrix multiplication. See [oneMKL defined datatypes](#) for more details.

**m** Number of rows of `op(A)` and `C`. Must be at least zero.

**n** Number of columns of `op(B)` and `C`. Must be at least zero.

**k** Number of columns of `op(A)` and rows of `op(B)`. Must be at least zero.

**alpha** Scaling factor for the matrix-matrix product.

**a** Pointer to input matrix `A`.

If `A` is not transposed, `A` is an `m`-by-`k` matrix so the array `a` must have size at least `lda*k`.

If `A` is transposed, `A` is a `k`-by-`m` matrix so the array `a` must have size at least `lda*m`.

See [Matrix Storage](#) for more details.

**lda** Leading dimension of `A`. Must be at least `m` if `A` is not transposed, and at least `k` if `A` is transposed. Must be positive.

**ao** Specifies the scalar offset value for matrix `A`.

**b** Pointer to input matrix B.

If B is not transposed, B is a k-by-n matrix so the array b must have size at least ldb\*n.

If B is transposed, B is an n-by-k matrix so the array b must have size at least ldb\*k.

See [Matrix Storage](#) for more details.

**ldb** Leading dimension of B. Must be at least k if B is not transposed, and at least n if B is transposed. Must be positive.

**bo** Specifies the scalar offset value for matrix B.

**beta** Scaling factor for matrix C.

**c** Pointer to input/output matrix C. Must have size at least ldc \* n. See [Matrix Storage](#) for more details.

**ldc** Leading dimension of C. Must be positive and at least m.

**co** Pointer to offset values for matrix C.

If offset\_type = offset::fix, the co array must have size at least 1.

If offset\_type = offset::col, the co array must have size at least max(1, m).

If offset\_type = offset::row, the co array must have size at least max(1, n).

**dependencies** List of events to wait for before starting computation, if any. If omitted, defaults to no dependencies.

## Output Parameters

**c** Pointer to the output matrix, overwritten by alpha \* (op(A) - A\_offset) \* (op(B) - B\_offset) + beta \* C + C\_offset.

## Notes

If beta = 0, matrix C does not need to be initialized before calling gemm\_bias.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** [BLAS-like Extensions](#)

**Parent topic:** [BLAS Routines](#)

**Parent topic:** [Dense Linear Algebra](#)

### 13.2.1.3 LAPACK Routines

oneMKL provides a DPC++ interface to select routines from the Linear Algebra PACKage (LAPACK), as well as several LAPACK-like extension routines.

### 13.2.1.3.1 LAPACK Linear Equation Routines

LAPACK Linear Equation routines are used for factoring a matrix, solving a system of linear equations, solving linear least squares problems, and inverting a matrix. The following table lists the LAPACK Linear Equation routine groups.

Routines	Scratchpad Size Routines	Description
<code>geqrf</code>	<code>geqrf_scratchpad</code>	Computes the QR factorization of a general m-by-n matrix.
<code>getrf</code>	<code>getrf_scratchpad</code>	Computes the LU factorization of a general m-by-n matrix.
<code>getri</code>	<code>getri_scratchpad</code>	Computes the inverse of an LU-factored general matrix.
<code>getrs</code>	<code>getrs_scratchpad</code>	Solves a system of linear equations with an LU-factored square coefficient matrix, with multiple right-hand sides.
<code>orgqr</code>	<code>orgqr_scratchpad</code>	Generates the real orthogonal matrix $Q$ of the QR factorization formed by <code>geqrf</code> .
<code>or-mqr</code>	<code>or-mqr_scratchpad</code>	Multiplies a real matrix by the orthogonal matrix $Q$ of the QR factorization formed by <code>geqrf</code> .
<code>potrf</code>	<code>potrf_scratchpad</code>	Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix.
<code>potri</code>	<code>potri_scratchpad</code>	Computes the inverse of a Cholesky-factored symmetric (Hermitian) positive-definite matrix.
<code>potrs</code>	<code>potrs_scratchpad</code>	Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite coefficient matrix, with multiple right-hand sides.
<code>sytrf</code>	<code>sytrf_scratchpad</code>	Computes the Bunch-Kaufman factorization of a symmetric matrix.
<code>trtrs</code>	<code>trtrs_scratchpad</code>	Solves a system of linear equations with a triangular coefficient matrix, with multiple right-hand sides.
<code>ungqr</code>	<code>ungqr_scratchpad</code>	Generates the complex unitary matrix $Q$ of the QR factorization formed by <code>geqrf</code> .
<code>un-mqr</code>	<code>un-mqr_scratchpad</code>	Multiplies a complex matrix by the unitary matrix $Q$ of the QR factorization formed by <code>geqrf</code> .

#### 13.2.1.3.1.1 `geqrf`

Computes the QR factorization of a general m-by-n matrix.

`geqrf` supports the following precisions:

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

#### Description

The routine forms the QR factorization of a general m-by-n matrix A. No pivoting is performed.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of min (m, n) elementary reflectors. Routines are provided to work with Q in this representation.

### 13.2.1.3.1.2 `geqrf` (BUFFER Version)

#### Syntax

```
void onemkl::lapack::geqrf(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, cl::sycl::buffer<T, 1> &a, std::int64_t lda, cl::sycl::buffer<T, 1> &tau, cl::sycl::buffer<T, 1> &scratchpad, std::int64_t scratchpad_size)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**m** The number of rows in the matrix A ( $0 \leq m$ ).

**n** The number of columns in A ( $0 \leq n$ ).

**a** Buffer holding input matrix A. Must have size at least  $lda * n$ .

**lda** The leading dimension of A; at least  $\max(1, m)$ .

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by `geqrf_scratchpad_size` function.

#### Output Parameters

**a** Output buffer, overwritten by the factorization data as follows:

The elements on and above the diagonal of the array contain the  $\min(m, n)$ -by-n upper trapezoidal matrix R (R is upper triangular if  $m \geq n$ ); the elements below the diagonal, with the array tau, represent the orthogonal matrix Q as a product of  $\min(m, n)$  elementary reflectors.

**tau** Output buffer, size at least  $\max(1, \min(m, n))$ . Contains scalars that define elementary reflectors for the matrix Q in its decomposition in a product of elementary reflectors.

**scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

#### Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info=-i`, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

### 13.2.1.3.1.3 `geqrf` (USM Version)

#### Syntax

```
cl::sycl::event onemkl::lapack::geqrf(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, T *a, std::int64_t lda, T *tau, T *scratchpad, std::int64_t scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**m** The number of rows in the matrix A ( $0 \leq m$ ).

**n** The number of columns in A ( $0 \leq n$ ).

**a** Pointer to memory holding input matrix A. Must have size at least  $lda * n$ .

**lda** The leading dimension of A; at least  $\max(1, m)$ .

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by `geqrf_scratchpad_size` function.

**events** List of events to wait for before starting computation. Defaults to empty list.

#### Output Parameters

**a** Overwritten by the factorization data as follows:

The elements on and above the diagonal of the array contain the  $\min(m, n)$ -by-n upper trapezoidal matrix R (R is upper triangular if  $m \geq n$ ); the elements below the diagonal, with the array tau, represent the orthogonal matrix Q as a product of  $\min(m, n)$  elementary reflectors.

**tau** Array, size at least  $\max(1, \min(m, n))$ . Contains scalars that define elementary reflectors for the matrix Q in its decomposition in a product of elementary reflectors.

**scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

#### Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info == -i`, the `i`-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *LAPACK Linear Equation Routines*

### 13.2.1.3.1.4 `geqrf_scratchpad_size`

Computes size of scratchpad memory required for `geqrf` function.

`geqrf_scratchpad_size` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

Computes the number of elements of type T the scratchpad memory to be passed to `geqrf` function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.1.5 `geqrf_scratchpad_size`

#### Syntax

```
template<typename T>
std::int64_t onemkl::lapack::geqrf_scratchpad_size(cl::sycl::queue &queue, std::int64_t m,
                                                    std::int64_t n, std::int64_t lda)
```

#### Input Parameters

**queue** Device queue where calculations by `geqrf` function will be performed.

**m** The number of rows in the matrix A ( $0 \leq m$ ).

**n** The number of columns in the matrix A ( $0 \leq n$ ).

**lda** The leading dimension of a.

#### Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by `get_info()` method of exception object.

## Return Value

The number of elements of type T the scratchpad memory to be passed to [geqrf](#) function should be able to hold.

**Parent topic:** [LAPACK Linear Equation Routines](#)

### 13.2.1.3.1.6 getrf

Computes the LU factorization of a general m-by-n matrix.

getrf supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

## Description

The routine computes the LU factorization of a general m-by-n matrix A as

```
A = P * L * U,
```

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n) and U is upper triangular (upper trapezoidal if m < n). The routine uses partial pivoting, with row interchanges.

### 13.2.1.3.1.7 getrf (BUFFER Version)

#### Syntax

```
void onemkl::lapack::getrf(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, cl::sycl::buffer<T,
                           1> &a, std::int64_t lda, cl::sycl::buffer<std::int64_t, 1> &ipiv,
                           cl::sycl::buffer<T, 1> &scratchpad, std::int64_t scratchpad_size)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**m** The number of rows in the matrix A ( $0 \leq m$ ).

**n** The number of columns in A ( $0 \leq n$ ).

**a** Buffer holding input matrix A. The buffer a contains the matrix A. The second dimension of a must be at least  $\max(1, n)$ .

**lda** The leading dimension of a.

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by [getrf\\_scratchpad\\_size](#) function.

## Output Parameters

- a** Overwritten by L and U. The unit diagonal elements of L are not stored.
- ipiv** Array, size at least  $\max(1, \min(m, n))$ . Contains the pivot indices; for  $1 \leq i \leq \min(m, n)$ , row  $i$  was interchanged with row  $\text{ipiv}(i)$ .
- scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info == -i`, the  $i$ -th parameter had an illegal value.

If `info == i`,  $\text{ui}_i$  is 0. The factorization has been completed, but U is exactly singular. Division by 0 will occur if you use the factor U for solving a system of linear equations.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

### 13.2.1.3.1.8 `getrf` (USM Version)

#### Syntax

```
cl::sycl::event onemkl::lapack::getrf(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, T *a,
                                      std::int64_t lda, std::int64_t *ipiv, T *scratchpad, std::int64_t
                                      scratchpad_size, const cl::sycl::vector_class<cl::sycl::event>
                                      &events = {})
```

## Input Parameters

- queue** The queue where the routine should be executed.
- m** The number of rows in the matrix A ( $0 \leq m$ ).
- n** The number of columns in A ( $0 \leq n$ ).
- a** Pointer to array holding input matrix A. The second dimension of a must be at least  $\max(1, n)$ .
- lda** The leading dimension of a.
- scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by `getrf_scratchpad_size` function.
- events** List of events to wait for before starting computation. Defaults to empty list.

## Output Parameters

- a** Overwritten by L and U. The unit diagonal elements of L are not stored.
- ipiv** Array, size at least  $\max(1, \min(m, n))$ . Contains the pivot indices; for  $1 \leq i \leq \min(m, n)$ , row  $i$  was interchanged with row  $\text{ipiv}(i)$ .
- scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info == -i`, the  $i$ -th parameter had an illegal value.

If `info == i`,  $u_{ii} == 0$ . The factorization has been completed, but U is exactly singular. Division by 0 will occur if you use the factor U for solving a system of linear equations.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *LAPACK Linear Equation Routines*

### 13.2.1.3.1.9 `getrf_scratchpad_size`

Computes size of scratchpad memory required for `getrf` function.

`getrf_scratchpad_size` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

Computes the number of elements of type T the scratchpad memory to be passed to `getrf` function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.1.10 `getrf_scratchpad_size`

#### Syntax

```
template<typename T>
std::int64_t onemkl::lapack::getrf_scratchpad_size(cl::sycl::queue &queue, std::int64_t m,
                                                    std::int64_t n, std::int64_t lda)
```

#### Input Parameters

**queue** Device queue where calculations by `getrf` function will be performed.

**m** The number of rows in the matrix A ( $0 \leq m$ ).

**n** The number of columns in A ( $0 \leq n$ ).

**lda** The leading dimension of a ( $n \leq \text{lda}$ ).

#### Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by `get_info()` method of exception object.

#### Return Value

The number of elements of type T the scratchpad memory to be passed to `getrf` function should be able to hold.

**Parent topic:** *LAPACK Linear Equation Routines*

### 13.2.1.3.1.11 `getri`

Computes the inverse of an LU-factored general matrix determined by `getrf`.

`getri` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

#### Description

The routine computes the inverse `inv(A)` of a general matrix A. Before calling this routine, call `getrf` to factorize A.

### 13.2.1.3.1.12 `getri` (BUFFER Version)

#### Syntax

```
void onemkl::lapack::getri(cl::sycl::queue &queue, std::int64_t n, cl::sycl::buffer<T, 1> &a,
                           std::int64_t lda, cl::sycl::buffer<std::int64_t, 1> &ipiv, cl::sycl::buffer<T,
                           1> &scratchpad, std::int64_t scratchpad_size)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**n** The order of the matrix A ( $0 \leq n$ ).

**a** The buffer *a* as returned by `getrf`. Must be of size at least  $\text{lda} \times \max(1, n)$ .

**lda** The leading dimension of *a* ( $n \leq \text{lda}$ ).

**ipiv** The buffer as returned by `getrf`. The dimension of *ipiv* must be at least  $\max(1, n)$ .

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type *T*. Size should not be less than the value returned by `getri_scratchpad_size` function.

#### Output Parameters

**a** Overwritten by the *n*-by-*n* matrix A.

**scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

#### Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The *info* code of the problem can be obtained by `get_info()` method of exception object:

If *info*=-*i*, the *i*-th parameter had an illegal value.

If *info* equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

### 13.2.1.3.1.13 `getri` (USM Version)

#### Syntax

```
cl::sycl::event onemkl::lapack::getri(cl::sycl::queue &queue, std::int64_t n, T *a, std::int64_t lda,
                                       std::int64_t *ipiv, T *scratchpad, std::int64_t scratchpad_size,
                                       const cl::sycl::vector_class<cl::sycl::event> &events = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**n** The order of the matrix A ( $0 \leq n$ ).

**a** The array as returned by [getrf](#). Must be of size at least  $\text{lda} * \max(1, n)$ .

**lda** The leading dimension of a ( $n \leq \text{lda}$ ).

**ipiv** The array as returned by [getrf](#). The dimension of ipiv must be at least  $\max(1, n)$ .

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by [getri\\_scratchpad\\_size](#) function.

**events** List of events to wait for before starting computation. Defaults to empty list.

## Output Parameters

**a** Overwritten by the n-by-n matrix A.

**scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info == -i`, the `i`-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** [LAPACK Linear Equation Routines](#)

### 13.2.1.3.1.14 `getri_scratchpad_size`

Computes size of scratchpad memory required for [getri](#) function.

`getri_scratchpad_size` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

Computes the number of elements of type T the scratchpad memory to be passed to [getri](#) function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.1.15 getri\_scratchpad\_size

#### Syntax

```
template<typename T>
std::int64_t onemkl::lapack::getri_scratchpad_size(cl::sycl::queue &queue, std::int64_t n,
                                                    std::int64_t lda)
```

#### Input Parameters

**queue** Device queue where calculations by [getri](#) function will be performed.

**n** The order of the matrix A ( $0 \leq n$ ).

**lda** The leading dimension of a ( $n \leq 1 \text{da}$ ).

#### Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by *get\_info()* method of exception object.

#### Return Value

The number of elements of type T the scratchpad memory to be passed to [getri](#) function should be able to hold.

**Parent topic:** [LAPACK Linear Equation Routines](#)

### 13.2.1.3.1.16 getrs

Solves a system of linear equations with an LU-factored square coefficient matrix, with multiple right-hand sides.

`getrs` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The routine solves for X the following systems of linear equations:

$A \times X = B$	if <code>trans=onemkl::transpose::nontrans</code>
$AT \times X = B$	if <code>trans=onemkl::transpose::trans</code>
$AH \times X = B$	if <code>trans=onemkl::transpose::conjtrans</code>

Before calling this routine, you must call `getrf` to compute the LU factorization of A.

### 13.2.1.3.1.17 `getrs` (BUFFER Version)

#### Syntax

```
void onemkl::lapack::getrs(cl::sycl::queue &queue, onemkl::transpose trans, std::int64_t n, std::int64_t nrhs, cl::sycl::buffer<T, 1> &a, std::int64_t lda, cl::sycl::buffer<std::int64_t, 1> &ipiv, cl::sycl::buffer<T, 1> &b, std::int64_t ldb, cl::sycl::buffer<T, 1> &scratchpad, std::int64_t scratchpad_size)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**trans** Indicates the form of the equations:

If `trans=onemkl::transpose::nontrans`, then  $A \times X = B$  is solved for X.

If `trans=onemkl::transpose::trans`, then  $AT \times X = B$  is solved for X.

If `trans=onemkl::transpose::conjtrans`, then  $AH \times X = B$  is solved for X.

**n** The order of the matrix A and the number of rows in matrix B ( $0 \leq n$ ).

**nrhs** The number of right-hand sides ( $0 \leq nrhs$ ).

**a** Buffer containing the factorization of the matrix A, as returned by `getrf`. The second dimension of a must be at least  $\max(1, n)$ .

**lda** The leading dimension of a.

**ipiv** Array, size at least  $\max(1, n)$ . The ipiv array, as returned by `getrf`.

**b** The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least  $\max(1, nrhs)$ .

**ldb** The leading dimension of b.

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by `getrs_scratchpad_size` function.

## Output Parameters

**b** The buffer b is overwritten by the solution matrix X.

**scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info==i`, the `i`-th parameter had an illegal value.

If `info=i`, the `i`-th diagonal element of U is zero, and the solve could not be completed.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

### 13.2.1.3.1.18 `getrs` (USM Version)

## Syntax

```
cl::sycl::event onemkl::lapack::getrs(cl::sycl::queue &queue, onemkl::transpose trans, std::int64_t n,
                                      std::int64_t nrhs, T *a, std::int64_t lda, std::int64_t *ipiv, T *b,
                                      std::int64_t ldb, T *scratchpad, std::int64_t scratchpad_size,
                                      const cl::sycl::vector_class<cl::sycl::event> &events = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**trans** Indicates the form of the equations:

If `trans=onemkl::transpose::nontrans`, then  $A \times X = B$  is solved for X.

If `trans=onemkl::transpose::trans`, then  $AT \times X = B$  is solved for X.

If `trans=onemkl::transpose::conjtrans`, then  $AH \times X = B$  is solved for X.

**n** The order of the matrix A and the number of rows in matrix B ( $0 \leq n$ ).

**nrhs** The number of right-hand sides ( $0 \leq nrhs$ ).

**a** Pointer to array containing the factorization of the matrix A, as returned by `getrf`. The second dimension of a must be at least `max(1, n)`.

**lda** The leading dimension of a.

**ipiv** Array, size at least `max(1, n)`. The ipiv array, as returned by `getrf`.

**b** The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least `max(1, nrhs)`.

**ldb** The leading dimension of b.

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by `getrs_scratchpad_size` function.

**events** List of events to wait for before starting computation. Defaults to empty list.

## Output Parameters

**b** The array b is overwritten by the solution matrix X.

**scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info==i`, the `i`-th parameter had an illegal value.

If `info=i`, the `i`-th diagonal element of U is zero, and the solve could not be completed.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *LAPACK Linear Equation Routines*

### 13.2.1.3.1.19 `getrs_scratchpad_size`

Computes size of scratchpad memory required for `getrs` function.

`getrs_scratchpad_size` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

Computes the number of elements of type T the scratchpad memory to be passed to `getrs` function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.1.20 `getrs_scratchpad_size`

#### Syntax

```
template<typename T>
std::int64_t onemkl::lapack::getrs_scratchpad_size(cl::sycl::queue &queue,
                                                    onemkl::transpose trans, std::int64_t n,
                                                    std::int64_t nrhs, std::int64_t lda,
                                                    std::int64_t ldb)
```

## Input Parameters

**queue** Device queue where calculations by [getrs](#) function will be performed.

**trans** Indicates the form of the equations:

If `trans=onemkl::transpose::nontrans`, then  $A \times X = B$  is solved for  $X$ .

If `trans=onemkl::transpose::trans`, then  $AT \times X = B$  is solved for  $X$ .

If `trans=onemkl::transpose::conjtrans`, then  $AH \times X = B$  is solved for  $X$ .

**n** The order of the matrix  $A$  ( $0 \leq n$ ) and the number of rows in matrix  $B$  ( $0 \leq n$ ).

**nrhs** The number of right-hand sides ( $0 \leq nrhs$ ).

**lda** The leading dimension of  $a$ .

**ldb** The leading dimension of  $b$ .

## Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by `get_info()` method of exception object.

## Return Value

The number of elements of type  $T$  the scratchpad memory to be passed to [getrs](#) function should be able to hold.

**Parent topic:** [LAPACK Linear Equation Routines](#)

### 13.2.1.3.1.21 orgqr

Generates the real orthogonal matrix  $Q$  of the QR factorization formed by `geqrf`.

`orgqr` supports the following precisions.

<b>T</b>
float
double

## Description

The routine generates the whole or part of  $m$ -by- $m$  orthogonal matrix  $Q$  of the QR factorization formed by the routine `geqrf`.

Usually  $Q$  is determined from the QR factorization of an  $m$  by  $p$  matrix  $A$  with  $m \geq p$ . To compute the whole matrix  $Q$ , use:

```
onemkl::orgqr(queue, m, m, p, a, lda, tau, scratchpad, scratchpad_size)
```

To compute the leading  $p$  columns of  $Q$  (which form an orthonormal basis in the space spanned by the columns of  $A$ ):

```
onemkl::orgqr(queue, m, p, p, a, lda, tau, scratchpad, scratchpad_size)
```

To compute the matrix  $Q^k$  of the QR factorization of leading  $k$  columns of the matrix  $A$ :

```
onemkl::orgqr(queue, m, m, k, a, lda, tau, scratchpad, scratchpad_size)
```

To compute the leading  $k$  columns of  $Q^k$  (which form an orthonormal basis in the space spanned by leading  $k$  columns of the matrix  $A$ ):

```
onemkl::orgqr(queue, m, k, k, a, lda, tau, scratchpad, scratchpad_size)
```

### 13.2.1.3.1.22 orgqr (BUFFER Version)

#### Syntax

```
void onemkl::lapack::orgqr(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, std::int64_t k,
                           cl::sycl::buffer<T, 1> &a, std::int64_t lda, cl::sycl::buffer<T, 1> &tau,
                           cl::sycl::buffer<T, 1> &scratchpad, std::int64_t scratchpad_size)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**m** The number of rows in the matrix  $A$  ( $0 \leq m$ ).

**n** The number of columns in the matrix  $A$  ( $0 \leq n$ ).

**k** The number of elementary reflectors whose product defines the matrix  $Q$  ( $0 \leq k \leq n$ ).

**a** The buffer  $a$  as returned by [geqrf](#).

**lda** The leading dimension of  $a$  ( $1 \leq lda \leq m$ ).

**tau** The buffer  $\tau$  as returned by [geqrf](#).

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type  $T$ . Size should not be less than the value returned by [orgqr\\_scratchpad\\_size](#) function.

#### Output Parameters

**a** Overwritten by  $n$  leading columns of the  $m$ -by- $m$  orthogonal matrix  $Q$ .

**scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

#### Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info=-i`, the  $i$ -th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

### 13.2.1.3.1.23 `orgqr` (USM Version)

#### Syntax

```
cl::sycl::event onemkl::lapack::orgqr(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, std::int64_t k, T *a, std::int64_t lda, T *tau, T *scratchpad, std::int64_t scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**m** The number of rows in the matrix A ( $0 \leq m$ ).

**n** The number of columns in the matrix A ( $0 \leq n$ ).

**k** The number of elementary reflectors whose product defines the matrix Q ( $0 \leq k \leq n$ ).

**a** The pointer to a as returned by [geqrf](#).

**lda** The leading dimension of a ( $1 \leq lda \leq m$ ).

**tau** The pointer to tau as returned by [geqrf](#).

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by [orgqr\\_scratchpad\\_size](#) function.

**events** List of events to wait for before starting computation. Defaults to empty list.

#### Output Parameters

**a** Overwritten by n leading columns of the m-by-m orthogonal matrix Q.

**scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

#### Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info=-i`, the `i`-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** [LAPACK Linear Equation Routines](#)

### 13.2.1.3.1.24 `orgqr_scratchpad_size`

Computes size of scratchpad memory required for `orgqr` function.

`orgqr_scratchpad_size` supports the following precisions.

T
float
double

## Description

Computes the number of elements of type T the scratchpad memory to be passed to `orgqr` function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.1.25 `orgqr_scratchpad_size`

## Syntax

```
template<typename T>
std::int64_t onemkl::lapack::orgqr_scratchpad_size(cl::sycl::queue &queue, std::int64_t m,
                                                    std::int64_t n, std::int64_t k, std::int64_t
                                                    lda)
```

## Input Parameters

**queue** Device queue where calculations by `orgqr` function will be performed.

**m** The number of rows in the matrix A ( $0 \leq m$ ).

**n** The number of columns in the matrix A ( $0 \leq n \leq m$ ).

**k** The number of elementary reflectors whose product defines the matrix Q ( $0 \leq k \leq n$ ).

**lda** The leading dimension of a.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by `get_info()` method of exception object.

## Return Value

The number of elements of type T the scratchpad memory to be passed to [orgqr](#) function should be able to hold.

**Parent topic:** [LAPACK Linear Equation Routines](#)

### 13.2.1.3.1.26 ormqr

Multiplies a real matrix by the orthogonal matrix Q of the QR factorization formed by [geqrf](#).

`ormqr` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The routine multiplies a real matrix C by Q or  $Q^T$ , where Q is the orthogonal matrix Q of the QR factorization formed by the routine [geqrf](#).

Depending on the parameters `left_right` and `trans`, the routine can form one of the matrix products  $Q \times C$ ,  $Q^T \times C$ ,  $C \times Q$ , or  $C \times Q^T$  (overwriting the result on C).

### 13.2.1.3.1.27 ormqr (BUFFER Version)

## Syntax

```
void onemkl::lapack::ormqr(cl::sycl::queue &queue, onemkl::side left_right, onemkl::transpose trans,
                           std::int64_t m, std::int64_t n, std::int64_t k, cl::sycl::buffer<T, 1> &a,
                           std::int64_t lda, cl::sycl::buffer<T, 1> &tau, cl::sycl::buffer<T, 1> &c,
                           std::int64_t ldc, cl::sycl::buffer<T, 1> &scratchpad, std::int64_t scratch-
                           pad_size)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**left\_right** If `left_right=onemkl::side::left`, Q or  $Q^T$  is applied to C from the left.

If `left_right=onemkl::side::right`, Q or  $Q^T$  is applied to C from the right.

**trans** If `trans=onemkl::transpose::trans`, the routine multiplies C by Q.

If `trans=onemkl::transpose::nontrans`, the routine multiplies C by  $Q^T$ .

**m** The number of rows in the matrix A ( $0 \leq m$ ).

**n** The number of columns in the matrix A ( $0 \leq n \leq m$ ).

**k** The number of elementary reflectors whose product defines the matrix Q ( $0 \leq k \leq n$ ).

**a** The buffer a as returned by [geqrf](#). The second dimension of a must be at least  $\max(1, k)$ .

**lda** The leading dimension of a.

**tau** The buffer tau as returned by [geqrf](#). The second dimension of a must be at least  $\max(1, k)$ .

**c** The buffer c contains the matrix C. The second dimension of c must be at least  $\max(1, n)$ .

**ldc** The leading dimension of c.

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by [ormqr\\_scratchpad\\_size](#) function.

## Output Parameters

**c** Overwritten by the product  $Q^*C$ ,  $Q^T*C$ ,  $C*Q$ , or  $C*Q^T$  (as specified by left\_right and trans).

**scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The info code of the problem can be obtained by `get_info()` method of exception object:

If `info==i`, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

### 13.2.1.3.1.28 ormqr (USM Version)

## Syntax

```
cl::sycl::event onemkl::lapack::ormqr(cl::sycl::queue &queue, onemkl::side left_right,
onemkl::transpose trans, std::int64_t m, std::int64_t n,
std::int64_t k, T *a, std::int64_t lda, T *tau, T *c, std::int64_t ldc,
T *scratchpad, std::int64_t scratchpad_size, const
cl::sycl::vector_class<cl::sycl::event> &events = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**left\_right** If `left_right=onemkl::side::left`, Q or  $Q^T$  is applied to C from the left.

If `left_right=onemkl::side::right`, Q or  $Q^T$  is applied to C from the right.

**trans** If `trans=onemkl::transpose::trans`, the routine multiplies C by Q.

If `trans=onemkl::transpose::nontrans`, the routine multiplies C by  $Q^T$ .

**m** The number of rows in the matrix A ( $0 \leq m$ ).

**n** The number of columns in the matrix A ( $0 \leq n \leq m$ ).

**k** The number of elementary reflectors whose product defines the matrix Q ( $0 \leq k \leq n$ ).

**a** The pointer to a as returned by [geqrf](#). The second dimension of a must be at least  $\max(1, k)$ .

**lda** The leading dimension of a.

**tau** The pointer to `tau` as returned by `geqrf`. The second dimension of `c` must be at least  $\max(1, k)$ .

**c** The pointer to the matrix `C`. The second dimension of `c` must be at least  $\max(1, n)$ .

**ldc** The leading dimension of `c`.

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by `ormqr_scratchpad_size` function.

**events** List of events to wait for before starting computation. Defaults to empty list.

## Output Parameters

**c** Overwritten by the product  $Q^*C$ ,  $Q^T*C$ ,  $C^*Q$ , or  $C^*Q^T$  (as specified by `left_right` and `trans`).

**scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info==i`, the `i`-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *LAPACK Linear Equation Routines*

### 13.2.1.3.1.29 `ormqr_scratchpad_size`

Computes size of scratchpad memory required for `ormqr` function.

`ormqr_scratchpad_size` supports the following precisions.

<code>T</code>
<code>float</code>
<code>double</code>

## Description

Computes the number of elements of type `T` the scratchpad memory to be passed to `ormqr` function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.1.30 ormqr\_scratchpad\_size

#### Syntax

```
template<typename T>
std::int64_t onemkl::lapack::ormqr_scratchpad_size(cl::sycl::queue &queue, onemkl::side
left_right, onemkl::transpose trans,
std::int64_t m, std::int64_t n, std::int64_t
k, std::int64_t lda, std::int64_t ldc,
std::int64_t &scratchpad_size)
```

#### Input Parameters

**queue** Device queue where calculations by *ormqr* function will be performed.

**left\_right** If `left_right=onemkl::side::left`,  $Q$  or  $Q^T$  is applied to  $C$  from the left.

If `left_right=onemkl::side::right`,  $Q$  or  $Q^T$  is applied to  $C$  from the right.

**trans** If `trans=onemkl::transpose::trans`, the routine multiplies  $C$  by  $Q$ .

If `trans=onemkl::transpose::nontrans`, the routine multiplies  $C$  by  $Q^T$ .

**m** The number of rows in the matrix  $A$  ( $0 \leq m$ ).

**n** The number of columns in the matrix  $A$  ( $0 \leq n \leq m$ ).

**k** The number of elementary reflectors whose product defines the matrix  $Q$  ( $0 \leq k \leq n$ ).

**lda** The leading dimension of  $a$ .

**ldc** The leading dimension of  $c$ .

#### Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by `get_info()` method of exception object.

#### Return Value

The number of elements of type  $T$  the scratchpad memory to be passed to *ormqr* function should be able to hold.

**Parent topic:** *LAPACK Linear Equation Routines*

### 13.2.1.3.1.31 potrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix.

`potrf` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A:

$A = U^T * U$ for real data, $A = U^H * U$ for complex data	if <code>upper_lower=onemkl::uplo::upper</code>
$A = L * L^T$ for real data, $A = L * L^H$ for complex data	if <code>upper_lower=onemkl::uplo::lower</code>

where L is a lower triangular matrix and U is upper triangular.

### 13.2.1.3.1.32 `potrf` (BUFFER Version)

#### Syntax

```
void onemkl::lapack::potrf(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n,
                           cl::sycl::buffer<T, 1> &a, std::int64_t lda, cl::sycl::buffer<T, 1>
                           &scratchpad, std::int64_t scratchpad_size)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If `upper_lower=onemkl::uplo::upper`, the array a stores the upper triangular part of the matrix A, and the strictly lower triangular part of the matrix is not referenced.

If `upper_lower=onemkl::uplo::lower`, the array a stores the lower triangular part of the matrix A, and the strictly upper triangular part of the matrix is not referenced.

**n** Specifies the order of the matrix A ( $0 \leq n$ ).

**a** Buffer holding input matrix A. The buffer a contains either the upper or the lower triangular part of the matrix A (see `upper_lower`). The second dimension of a must be at least  $\max(1, n)$ .

**lda** The leading dimension of a.

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by `potrf_scratchpad_size` function.

#### Output Parameters

**a** The buffer a is overwritten by the Cholesky factor U or L, as specified by `upper_lower`.

**scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info==i`, the `i`-th parameter had an illegal value.

If `info=i`, and `get_detail()` returns 0, then the leading minor of order `i` (and therefore the matrix `A` itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix `A`.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

### 13.2.1.3.1.33 `potrf` (USM Version)

#### Syntax

```
cl::sycl::event onemkl::lapack::potrf(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, T *a, std::int64_t lda, T *scratchpad, std::int64_t scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Indicates whether the upper or lower triangular part of `A` is stored and how `A` is factored:

If `upper_lower=onemkl::uplo::upper`, the array `a` stores the upper triangular part of the matrix `A`, and the strictly lower triangular part of the matrix is not referenced.

If `upper_lower=onemkl::uplo::lower`, the array `a` stores the lower triangular part of the matrix `A`, and the strictly upper triangular part of the matrix is not referenced.

**n** Specifies the order of the matrix `A` ( $0 \leq n$ ).

**a** Pointer to input matrix `A`. The array `a` contains either the upper or the lower triangular part of the matrix `A` (see `upper_lower`). The second dimension of `a` must be at least `max(1, n)`.

**lda** The leading dimension of `a`.

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by `potrf_scratchpad_size` function.

**events** List of events to wait for before starting computation. Defaults to empty list.

## Output Parameters

**a** The memory pointer to by pointer **a** is overwritten by the Cholesky factor **U** or **L**, as specified by **upper\_lower**.

**scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The **info** code of the problem can be obtained by **get\_info()** method of exception object:

If **info==-i**, the **i**-th parameter had an illegal value.

If **info=i**, and **get\_detail()** returns 0, then the leading minor of order **i** (and therefore the matrix **A** itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix **A**.

If **info** equals to value passed as scratchpad size, and **get\_detail()** returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by **get\_detail()** method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *LAPACK Linear Equation Routines*

### 13.2.1.3.1.34 `potrf_scratchpad_size`

Computes size of scratchpad memory required for ***potrf*** function.

**potrf\_scratchpad\_size** supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

Computes the number of elements of type **T** the scratchpad memory to be passed to ***potrf*** function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.1.35 `potrf_scratchpad_size`

#### Syntax

```
template<typename T>
std::int64_t onemkl::lapack::potrf_scratchpad_size(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, std::int64_t lda)
```

#### Input Parameters

**queue** Device queue where calculations by `potrf` function will be performed.

**upper\_lower** Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If `upper_lower=onemkl::uplo::upper`, the array a stores the upper triangular part of the matrix A, and the strictly lower triangular part of the matrix is not referenced.

If `upper_lower=onemkl::uplo::lower`, the array a stores the lower triangular part of the matrix A, and the strictly upper triangular part of the matrix is not referenced.

**n** Specifies the order of the matrix A ( $0 \leq n$ ).

**lda** The leading dimension of a.

#### Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by `get_info()` method of exception object.

#### Return Value

The number of elements of type T the scratchpad memory to be passed to `potrf` function should be able to hold.

**Parent topic:** *LAPACK Linear Equation Routines*

### 13.2.1.3.1.36 `potri`

Computes the inverse of a symmetric (Hermitian) positive-definite matrix using the Cholesky factorization.

`potri` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The routine computes the inverse `inv(A)` of a symmetric positive definite or, for complex flavors, Hermitian positive-definite matrix `A`. Before calling this routine, call `potrf` to factorize `A`.

### 13.2.1.3.1.37 `potri` (BUFFER Version)

#### Syntax

```
void onemkl::lapack::potri(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n,
                           cl::sycl::buffer<T, 1> &a, std::int64_t lda, cl::sycl::buffer<T, 1>
                           &scratchpad, std::int64_t scratchpad_size)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Indicates how the input matrix `A` has been factored:

If `upper_lower = onemkl::uplo::upper`, the upper triangle of `A` is stored.

If `upper_lower = onemkl::uplo::lower`, the lower triangle of `A` is stored.

**n** Specifies the order of the matrix `A` ( $0 \leq n$ ).

**a** Contains the factorization of the matrix `A`, as returned by `potrf`. The second dimension of `a` must be at least  $\max(1, n)$ .

**lda** The leading dimension of `a`.

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by `potri\_scratchpad\_size` function.

#### Output Parameters

**a** Overwritten by the upper or lower triangle of the inverse of `A`. Specified by `upper_lower`.

**scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

#### Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info=-i`, the `i`-th parameter had an illegal value.

If `info=i`, the `i`-th diagonal element of the Cholesky factor (and therefore the factor itself) is zero, and the inversion could not be completed.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

### 13.2.1.3.1.38 `potri` (USM Version)

#### Syntax

```
cl::sycl::event onemkl::lapack::potri(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, T *a, std::int64_t lda, T *scratchpad, std::int64_t scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Indicates how the input matrix A has been factored:

If *upper\_lower* = `onemkl::uplo::upper`, the upper triangle of A is stored.

If *upper\_lower* = `onemkl::uplo::lower`, the lower triangle of A is stored.

**n** Specifies the order of the matrix A( $0 \leq n$ ).

**a** Contains the factorization of the matrix A, as returned by `potrf`. The second dimension of a must be at least  $\max(1, n)$ .

**lda** The leading dimension of a.

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by `potri_scratchpad_size` function.

**events** List of events to wait for before starting computation. Defaults to empty list.

#### Output Parameters

**a** Overwritten by the upper or lower triangle of the inverse of A. Specified by *upper\_lower*.

**scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

#### Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The *info* code of the problem can be obtained by `get_info()` method of exception object:

If *info*=-*i*, the *i*-th parameter had an illegal value.

If *info*=*i*, the *i*-th diagonal element of the Cholesky factor (and therefore the factor itself) is zero, and the inversion could not be completed.

If *info* equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** [LAPACK Linear Equation Routines](#)

### 13.2.1.3.1.39 `potri_scratchpad_size`

Computes size of scratchpad memory required for `potri` function.

`potri_scratchpad_size` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

Computes the number of elements of type T the scratchpad memory to be passed to `potri` function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.1.40 `potri_scratchpad_size`

## Syntax

```
template<typename T>
std::int64_t onemkl::lapack::potri_scratchpad_size(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, std::int64_t lda)
```

## Input Parameters

**queue** Device queue where calculations by `potri` function will be performed.

**upper\_lower** Indicates how the input matrix A has been factored:

If `upper_lower` = `onemkl::uplo::upper`, the upper triangle of A is stored.

If `upper_lower` = `onemkl::uplo::lower`, the lower triangle of A is stored.

**n** Specifies the order of the matrix A( $0 \leq n$ ).

**lda** The leading dimension of a.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by *get\_info()* method of exception object.

## Return Value

The number of elements of type T the scratchpad memory to be passed to *potri* function should be able to hold.

**Parent topic:** *LAPACK Linear Equation Routines*

### 13.2.1.3.1.41 potrs

Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite coefficient matrix.

potrs supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The routine solves for X the system of linear equations  $A \cdot X = B$  with a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A, given the Cholesky factorization of A:

$A = UT \cdot U$ for real data, $A = UH \cdot U$ for complex data	if <code>upper_lower=onemkl::uplo::upper</code>
$A = L \cdot LT$ for real data, $A = L \cdot LH$ for complex data	if <code>upper_lower=onemkl::uplo::lower</code>

where L is a lower triangular matrix and U is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix B.

Before calling this routine, you must call *potrf* to compute the Cholesky factorization of A.

### 13.2.1.3.1.42 potrs (BUFFER Version)

## Syntax

```
void onemkl::lapack::potrs(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, std::int64_t nrhs, cl::sycl::buffer<T, 1> &a, std::int64_t lda, cl::sycl::buffer<T, 1> &b, std::int64_t ldb, cl::sycl::buffer<T, 1> &scratchpad, std::int64_t scratchpad_size)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Indicates how the input matrix has been factored:

If `upper_lower = onemkl::uplo::upper`, the upper triangle  $U$  of  $A$  is stored, where  $A = U^T * U$  for real data,  $A = U^H * U$  for complex data.

If `upper_lower = onemkl::uplo::lower`, the lower triangle  $L$  of  $A$  is stored, where  $A = L * L^T$  for real data,  $A = L * L^H$  for complex data.

**n** The order of matrix  $A$  ( $0 \leq n$ ).

**nrhs** The number of right-hand sides ( $0 \leq nrhs$ ).

**a** Buffer containing the factorization of the matrix  $A$ , as returned by `potrf`. The second dimension of  $a$  must be at least `max(1, n)`.

**lda** The leading dimension of  $a$ .

**b** The array  $b$  contains the matrix  $B$  whose columns are the right-hand sides for the systems of equations. The second dimension of  $b$  must be at least `max(1, nrhs)`.

**ldb** The leading dimension of  $b$ .

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type  $T$ . Size should not be less than the value returned by `potrs_scratchpad_size` function.

## Output Parameters

**b** Overwritten by the solution matrix  $X$ .

**scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info=-i`, the  $i$ -th parameter had an illegal value.

If `info=i`, the  $i$ -th diagonal element of the Cholesky factor is zero, and the solve could not be completed.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

### 13.2.1.3.1.43 `potrs` (USM Version)

#### Syntax

```
cl::sycl::event onemkl::lapack::potrs(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, std::int64_t nrhs, T *a, std::int64_t lda, T *b, std::int64_t ldb, T *scratchpad, std::int64_t scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Indicates how the input matrix has been factored:

If `upper_lower = onemkl::uplo::upper`, the upper triangle  $U$  of  $A$  is stored, where  $A = U^T * U$  for real data,  $A = U^H * U$  for complex data.

If `upper_lower = onemkl::uplo::lower`, the lower triangle  $L$  of  $A$  is stored, where  $A = L * L^T$  for real data,  $A = L * L^H$  for complex data.

**n** The order of matrix  $A$  ( $0 \leq n$ ).

**nrhs** The number of right-hand sides ( $0 \leq nrhs$ ).

**a** Pointer to array containing the factorization of the matrix  $A$ , as returned by `potrf`. The second dimension of  $a$  must be at least `max(1, n)`.

**lda** The leading dimension of  $a$ .

**b** The array  $b$  contains the matrix  $B$  whose columns are the right-hand sides for the systems of equations. The second dimension of  $b$  must be at least `max(1, nrhs)`.

**ldb** The leading dimension of  $b$ .

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type  $T$ . Size should not be less than the value returned by `potrs_scratchpad_size` function.

**events** List of events to wait for before starting computation. Defaults to empty list.

## Output Parameters

**b** Overwritten by the solution matrix  $X$ .

**scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info == -i`, the  $i$ -th parameter had an illegal value.

If `info == i`, the  $i$ -th diagonal element of the Cholesky factor is zero, and the solve could not be completed.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** [LAPACK Linear Equation Routines](#)

### 13.2.1.3.1.44 `potrs_scratchpad_size`

Computes size of scratchpad memory required for `potrs` function.

`potrs_scratchpad_size` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

Computes the number of elements of type T the scratchpad memory to be passed to `potrs` function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.1.45 `potrs_scratchpad_size`

## Syntax

```
template<typename T>
std::int64_t onemkl::lapack::potrs_scratchpad_size(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, std::int64_t nrhs, std::int64_t lda, std::int64_t ldb)
```

## Input Parameters

**queue** Device queue where calculations by `potrs` function will be performed.

**upper\_lower** Indicates how the input matrix has been factored:

If `upper_lower` = `onemkl::uplo::upper`, the upper triangle U of A is stored, where  $A = U^T * U$  for real data,  $A = U^H * U$  for complex data.

If `upper_lower` = `onemkl::uplo::lower`, the lower triangle L of A is stored, where  $A = L * L^T$  for real data,  $A = L * L^H$  for complex data.

**n** The order of matrix A ( $0 \leq n$ ).

**nrhs** The number of right-hand sides ( $0 \leq nrhs$ ).

**lda** The leading dimension of a.

**ldb** The leading dimension of b.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by *get\_info()* method of exception object.

## Return Value

The number of elements of type T the scratchpad memory to be passed to *potrs* function should be able to hold.

**Parent topic:** *LAPACK Linear Equation Routines*

### 13.2.1.3.1.46 sytrf

Computes the Bunch-Kaufman factorization of a symmetric matrix.

sytrf supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

## Description

The routine computes the factorization of a real/complex symmetric matrix A using the Bunch-Kaufman diagonal pivoting method. The form of the factorization is:

- if *upper\_lower*=*uplo*::upper,  $A = U \star D \star U^T$
- if *upper\_lower*=*uplo*::lower,  $A = L \star D \star L^T$

where A is the input matrix, U and L are products of permutation and triangular matrices with unit diagonal (upper triangular for U and lower triangular for L), and D is a symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D.

### 13.2.1.3.1.47 sytrf (BUFFER Version)

## Syntax

```
void onemkl::lapack::sytrf(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n,
                           cl::sycl::buffer<T, 1> &a, std::int64_t lda, cl::sycl::buffer<int_64, 1>
                           &ipiv, cl::sycl::buffer<T, 1> &scratchpad, std::int64_t scratchpad_size)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If `upper_lower=uplo::upper`, the buffer a stores the upper triangular part of the matrix A, and A is factored as  $U \star D \star UT$ .

If `upper_lower=uplo::lower`, the buffer a stores the lower triangular part of the matrix A, and A is factored as  $L \star D \star LT$ .

**n** The order of matrix A ( $0 \leq n$ ).

**a** The buffer a, size  $\max(1, \text{lda} * n)$ . The buffer a contains either the upper or the lower triangular part of the matrix A (see `upper_lower`). The second dimension of a must be at least  $\max(1, n)$ .

**lda** The leading dimension of a.

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by `sytrf_scratchpad_size` function.

## Output Parameters

**a** The upper or lower triangular part of a is overwritten by details of the block-diagonal matrix D and the multipliers used to obtain the factor U (or L).

**ipiv** Buffer, size at least  $\max(1, n)$ . Contains details of the interchanges and the block structure of D. If `ipiv(i)=k>0`, then  $d_{ii}$  is a 1-by-1 block, and the i-th row and column of A was interchanged with the k-th row and column.

If `upper_lower=onemkl::uplo::upper` and `ipiv(i)=ipiv(i-1)=-m<0`, then D has a 2-by-2 block in rows/columns i and i-1, and (i-1)-th row and column of A was interchanged with the m-th row and column.

If `upper_lower=onemkl::uplo::lower` and `ipiv(i)=ipiv(i+1)=-m<0`, then D has a 2-by-2 block in rows/columns i and i+1, and (i+1)-th row and column of A was interchanged with the m-th row and column.

**scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info=-i`, the i-th parameter had an illegal value.

If `info=i`,  $d_{ii}$  is 0. The factorization has been completed, but D is exactly singular. Division by 0 will occur if you use D for solving a system of linear equations.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

### 13.2.1.3.1.48 sytrf (USM Version)

#### Syntax

```
cl::sycl::event onemkl::lapack::sytrf(cl::sycl::queue &queue, onemkl::uplo upper_lower,
std::int64_t n, T *a, std::int64_t lda, int_64 *ipiv,
T *scratchpad, std::int64_t scratchpad_size, const
cl::sycl::vector_class<cl::sycl::event> &events = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If `upper_lower=uplo::upper`, the array a stores the upper triangular part of the matrix A, and A is factored as  $U \star D \star UT$ .

If `upper_lower=uplo::lower`, the array a stores the lower triangular part of the matrix A, and A is factored as  $L \star D \star LT$ .

**n** The order of matrix A ( $0 \leq n$ ).

**a** The pointer to A, size  $\max(1, lda * n)$ , containing either the upper or the lower triangular part of the matrix A (see `upper_lower`). The second dimension of a must be at least  $\max(1, n)$ .

**lda** The leading dimension of a.

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by `sytrf_scratchpad_size` function.

**events** List of events to wait for before starting computation. Defaults to empty list.

#### Output Parameters

**a** The upper or lower triangular part of a is overwritten by details of the block-diagonal matrix D and the multipliers used to obtain the factor U (or L).

**ipiv** Pointer to array of size at least  $\max(1, n)$ . Contains details of the interchanges and the block structure of D. If `ipiv(i)=k>0`, then  $d_{ii}$  is a 1-by-1 block, and the i-th row and column of A was interchanged with the k-th row and column.

If `upper_lower=onemkl::uplo::upper` and `ipiv(i)=ipiv(i-1)=-m<0`, then D has a 2-by-2 block in rows/columns i and i-1, and (i-1)-th row and column of A was interchanged with the m-th row and column.

If `upper_lower=onemkl::uplo::lower` and `ipiv(i)=ipiv(i+1)=-m<0`, then D has a 2-by-2 block in rows/columns i and i+1, and (i+1)-th row and column of A was interchanged with the m-th row and column.

**scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info==i`, the `i`-th parameter had an illegal value.

If `info==i`,  $d_{ii}$  is 0. The factorization has been completed, but  $D$  is exactly singular. Division by 0 will occur if you use  $D$  for solving a system of linear equations.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *LAPACK Linear Equation Routines*

### 13.2.1.3.1.49 `sytrf_scratchpad_size`

Computes size of scratchpad memory required for `sytrf` function.

`sytrf_scratchpad_size` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

Computes the number of elements of type `T` the scratchpad memory to be passed to `sytrf` function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.1.50 `sytrf_scratchpad_size`

## Syntax

```
template<typename T>
std::int64_t onemkl::lapack::sytrf_scratchpad_size(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, std::int64_t lda)
```

## Input Parameters

**queue** Device queue where calculations by [sytrf](#) function will be performed.

**upper\_lower** Indicates whether the upper or lower triangular part of  $A$  is stored and how  $A$  is factored:

If `upper_lower=uplo::upper`, the buffer `a` stores the upper triangular part of the matrix  $A$ , and  $A$  is factored as  $U \star D \star UT$ .

If `upper_lower=uplo::lower`, the buffer `a` stores the lower triangular part of the matrix  $A$ , and  $A$  is factored as  $L \star D \star LT$

**n** The order of the matrix  $A$  ( $0 \leq n$ ).

**lda** The leading dimension of `a`.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by `get_info()` method of exception object.

## Return Value

The number of elements of type  $T$  the scratchpad memory to be passed to [sytrf](#) function should be able to hold.

**Parent topic:** [LAPACK Linear Equation Routines](#)

### 13.2.1.3.1.51 trtrs

Solves a system of linear equations with a triangular coefficient matrix, with multiple right-hand sides.

`trtrs` supports the following precisions.

$T$
<code>float</code>
<code>double</code>
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The routine solves for  $X$  the following systems of linear equations with a triangular matrix  $A$ , with multiple right-hand sides stored in  $B$ :

$A^*X = B$	if <code>transa=transpose::nontrans</code> ,
$AT^*X = B$	if <code>transa=transpose::trans</code> ,
$A^H*X = B$	if <code>transa=transpose::conjtrans</code> (for complex matrices only).

### 13.2.1.3.1.52 `trtrs` (BUFFER Version)

#### Syntax

```
void onemkl::lapack::trtrs(cl::sycl::queue &queue, onemkl::uplo upper_lower, onemkl::transpose transa, onemkl::diag unit_diag, std::int64_t n, std::int64_t nrhs, cl::sycl::buffer<T, 1> &a, std::int64_t lda, cl::sycl::buffer<T, 1> &b, std::int64_t ldb, cl::sycl::buffer<T, 1> &scratchpad, std::int64_t scratchpad_size)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Indicates whether A is upper or lower triangular:

If `upper_lower = uplo::upper`, then A is upper triangular.

If `upper_lower = uplo::lower`, then A is lower triangular.

**transa** If `transa = transpose::nontrans`, then  $A^*X = B$  is solved for X.

If `transa = transpose::trans`, then  $A^T*X = B$  is solved for X.

If `transa = transpose::conjtrans`, then  $A^H*X = B$  is solved for X.

**unit\_diag** If `unit_diag = diag::nonunit`, then A is not a unit triangular matrix.

If `unit_diag = diag::unit`, then A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array a.

**n** The order of A; the number of rows in B;  $n \geq 0$ .

**nrhs** The number of right-hand sides;  $nrhs \geq 0$ .

**a** Buffer containing the matrix A. The second dimension of a must be at least  $\max(1, n)$ .

**lda** The leading dimension of a;  $lda \geq \max(1, n)$ .

**b** Buffer containing the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b at least  $\max(1, nrhs)$ .

**ldb** The leading dimension of b;  $ldb \geq \max(1, n)$ .

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by `trtrs_scratchpad_size` function.

#### Output Parameters

**b** Overwritten by the solution matrix X.

**scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info==i`, the `i`-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

### 13.2.1.3.1.53 `trtrs` (USM Version)

#### Syntax

```
cl::sycl::event onemkl::lapack::trtrs(cl::sycl::queue &queue, onemkl::uplo upper_lower,
                                      onemkl::transpose transa, onemkl::diag unit_diag, std::int64_t n,
                                      std::int64_t nrhs, T *a, std::int64_t lda, T *b, std::int64_t ldb,
                                      T *scratchpad, std::int64_t scratchpad_size, const
                                      cl::sycl::vector_class<cl::sycl::event> &events = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Indicates whether `A` is upper or lower triangular:

If `upper_lower = uplo::upper`, then `A` is upper triangular.

If `upper_lower = uplo::lower`, then `A` is lower triangular.

**transa** If `transa = transpose::nontrans`, then  $A^T X = B$  is solved for `X`.

If `transa = transpose::trans`, then  $A^T X = B$  is solved for `X`.

If `transa = transpose::conjtrans`, then  $A^H X = B$  is solved for `X`.

**unit\_diag** If `unit_diag = diag::nonunit`, then `A` is not a unit triangular matrix.

If `unit_diag = diag::unit`, then `A` is unit triangular: diagonal elements of `A` are assumed to be 1 and not referenced in the array `a`.

**n** The order of `A`; the number of rows in `B`;  $n \geq 0$ .

**nrhs** The number of right-hand sides;  $nrhs \geq 0$ .

**a** Array containing the matrix `A`. The second dimension of `a` must be at least  $\max(1, n)$ .

**lda** The leading dimension of `a`;  $lda \geq \max(1, n)$ .

**b** Array containing the matrix `B` whose columns are the right-hand sides for the systems of equations. The second dimension of `b` at least  $\max(1, nrhs)$ .

**ldb** The leading dimension of `b`;  $ldb \geq \max(1, n)$ .

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by `trtrs_scratchpad_size` function.

**events** List of events to wait for before starting computation. Defaults to empty list.

## Output Parameters

**b** Overwritten by the solution matrix X.

**scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info==i`, the `i`-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *LAPACK Linear Equation Routines*

### 13.2.1.3.1.54 `trtrs_scratchpad_size`

Computes size of scratchpad memory required for `trtrs` function.

`trtrs_scratchpad_size` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

Computes the number of elements of type T the scratchpad memory to be passed to `trtrs` function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.1.55 `trtrs_scratchpad_size`

## Syntax

```
template<typename T>
std::int64_t onemkl::lapack::trtrs_scratchpad_size(cl::sycl::queue &queue, onemkl::uplo upper_lower,
                                                    onemkl::transpose trans,
                                                    onemkl::diag diag, std::int64_t n,
                                                    std::int64_t nrhs, std::int64_t lda,
                                                    std::int64_t ldb)
```

## Input Parameters

**queue** Device queue where calculations by `trtrs` function will be performed.

**upper\_lower** Indicates whether A is upper or lower triangular:

If `upper_lower = uplo::upper`, then A is upper triangular.

If `upper_lower = uplo::lower`, then A is lower triangular.

**trans** Indicates the form of the equations:

If `trans=onemkl::transpose::nontrans`, then  $A \times X = B$  is solved for X.

If `trans=onemkl::transpose::trans`, then  $A^T \times X = B$  is solved for X.

If `trans=onemkl::transpose::conjtrans`, then  $A^H \times X = B$  is solved for X.

**diag** If `diag = onemkl::diag::nonunit`, then A is not a unit triangular matrix.

If `unit_diag = diag::unit`, then A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array a.

**n** The order of A; the number of rows in B;  $n \geq 0$ .

**nrhs** The number of right-hand sides ( $0 \leq \text{nrhs}$ ).

**lda** The leading dimension of a;  $\text{lda} \geq \max(1, n)$ .

**ldb** The leading dimension of b;  $\text{ldb} \geq \max(1, n)$ .

## Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by `get_info()` method of exception object.

## Return Value

The number of elements of type T the scratchpad memory to be passed to `trtrs` function should be able to hold.

**Parent topic:** *LAPACK Linear Equation Routines*

### 13.2.1.3.1.56 ungqr

Generates the complex unitary matrix Q of the QR factorization formed by `geqrf`.

`ungqr` supports the following precisions.

T
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The routine generates the whole or part of  $m$ -by- $m$  unitary matrix  $Q$  of the QR factorization formed by the routines [geqrf](#).

Usually  $Q$  is determined from the QR factorization of an  $m$  by  $p$  matrix  $A$  with  $m \geq p$ . To compute the whole matrix  $Q$ , use:

```
onemkl::ungqr(queue, m, m, p, a, lda, tau, scratchpad, scratchpad_size)
```

To compute the leading  $p$  columns of  $Q$  (which form an orthonormal basis in the space spanned by the columns of  $A$ ):

```
onemkl::ungqr(queue, m, p, p, a, lda, tau, scratchpad, scratchpad_size)
```

To compute the matrix  $Q^k$  of the QR factorization of the leading  $k$  columns of the matrix  $A$ :

```
onemkl::ungqr(queue, m, m, k, a, lda, tau, scratchpad, scratchpad_size)
```

To compute the leading  $k$  columns of  $Q^k$  (which form an orthonormal basis in the space spanned by the leading  $k$  columns of the matrix  $A$ ):

```
onemkl::ungqr(queue, m, k, k, a, lda, tau, scratchpad, scratchpad_size)
```

### 13.2.1.3.1.57 ungqr (BUFFER Version)

#### Syntax

```
void onemkl::lapack::ungqr(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, std::int64_t k,
                           cl::sycl::buffer<T, 1> &a, std::int64_t lda, cl::sycl::buffer<T, 1> &tau,
                           cl::sycl::buffer<T, 1> &scratchpad, std::int64_t scratchpad_size)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**m** The number of rows in the matrix  $A$  ( $m \leq 0$ ).

**n** The number of columns in the matrix  $A$  ( $0 \leq n$ ).

**k** The number of elementary reflectors whose product defines the matrix  $Q$  ( $0 \leq k \leq n$ ).

**a** The buffer  $a$  as returned by [geqrf](#).

**lda** The leading dimension of  $a$  ( $lda \leq m$ ).

**tau** The buffer  $\tau$  as returned by [geqrf](#).

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type  $T$ . Size should not be less than the value returned by [ungqr\\_scratchpad\\_size](#) function.

## Output Parameters

**a** Overwritten by  $n$  leading columns of the  $m$ -by- $m$  orthogonal matrix  $Q$ .

**scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info==i`, the  $i$ -th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

### 13.2.1.3.1.58 `ungqr` (USM Version)

#### Syntax

```
cl::sycl::event onemkl::lapack::ungqr(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, std::int64_t k, T *a, std::int64_t lda, T *tau, T *scratchpad, std::int64_t scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**m** The number of rows in the matrix  $A$  ( $m \leq 0$ ).

**n** The number of columns in the matrix  $A$  ( $0 \leq n$ ).

**k** The number of elementary reflectors whose product defines the matrix  $Q$  ( $0 \leq k \leq n$ ).

**a** The pointer to  $a$  as returned by `geqrf`.

**lda** The leading dimension of  $a$  ( $lda \leq m$ ).

**tau** The pointer to  $\tau$  as returned by `geqrf`.

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type  $T$ . Size should not be less than the value returned by `ungqr_scratchpad_size` function.

**events** List of events to wait for before starting computation. Defaults to empty list.

## Output Parameters

**a** Overwritten by  $n$  leading columns of the  $m$ -by- $m$  orthogonal matrix  $Q$ .

**scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info==i`, the  $i$ -th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *LAPACK Linear Equation Routines*

### 13.2.1.3.1.59 `ungqr_scratchpad_size`

Computes size of scratchpad memory required for `ungqr` function.

`ungqr_scratchpad_size` supports the following precisions.

T
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

Computes the number of elements of type T the scratchpad memory to be passed to `ungqr` function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.1.60 `ungqr_scratchpad_size`

#### Syntax

```
template<typename T>
std::int64_t onemkl::lapack::ungqr_scratchpad_size(cl::sycl::queue &queue, std::int64_t m,
                                                    std::int64_t n, std::int64_t k, std::int64_t lda)
```

## Input Parameters

- queue** Device queue where calculations by [ungqr](#) function will be performed.
- m** The number of rows in the matrix A ( $0 \leq m$ ).
- n** The number of columns the matrix A ( $0 \leq n \leq m$ ).
- k** The number of elementary reflectors whose product defines the matrix Q ( $0 \leq k \leq n$ ).
- lda** The leading dimension of a.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by `get_info()` method of exception object.

## Return Value

The number of elements of type T the scratchpad memory to be passed to [ungqr](#) function should be able to hold.

**Parent topic:** [LAPACK Linear Equation Routines](#)

### 13.2.1.3.1.61 unmqr

Multiplies a complex matrix by the unitary matrix Q of the QR factorization formed by unmqr.

unmqr supports the following precisions.

T
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The routine multiplies a rectangular complex matrix C by Q or  $Q^H$ , where Q is the unitary matrix Q of the QR factorization formed by the routines [geqrf](#).

Depending on the parameters `left_right` and `trans`, the routine can form one of the matrix products  $Q * C$ ,  $Q^H * C$ ,  $C * Q$ , or  $C * Q^H$  (overwriting the result on C).

### 13.2.1.3.1.62 unmqr (BUFFER Version)

## Syntax

```
void onemkl::lapack::unmqr(cl::sycl::queue &queue, onemkl::side left_right, onemkl::transpose trans,
                           std::int64_t m, std::int64_t n, std::int64_t k, cl::sycl::buffer<T, 1> &a,
                           std::int64_t lda, cl::sycl::buffer<T, 1> &tau, cl::sycl::buffer<T, 1> &c,
                           std::int64_t ldc, cl::sycl::buffer<T, 1> &scratchpad, std::int64_t scratch-
                           pad_size)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**left\_right** If `left_right=onemkl::side::left`,  $Q$  or  $Q^H$  is applied to  $C$  from the left.

If `left_right=onemkl::side::right`,  $Q$  or  $Q^H$  is applied to  $C$  from the right.

**trans** If `trans=onemkl::transpose::trans`, the routine multiplies  $C$  by  $Q$ .

If `trans=onemkl::transpose::nontrans`, the routine multiplies  $C$  by  $Q^H$ .

**m** The number of rows in the matrix  $A$  ( $m \leq 0$ ).

**n** The number of columns in the matrix  $A$  ( $0 \leq n \leq m$ ).

**k** The number of elementary reflectors whose product defines the matrix  $Q$  ( $0 \leq k \leq n$ ).

**a** The buffer  $a$  as returned by [`geqrf`](#). The second dimension of  $a$  must be at least  $\max(1, k)$ .

**lda** The leading dimension of  $a$ .

**tau** The buffer  $\tau$  as returned by [`geqrf`](#). The second dimension of  $a$  must be at least  $\max(1, k)$ .

**c** The buffer  $c$  contains the matrix  $C$ . The second dimension of  $c$  must be at least  $\max(1, n)$ .

**ldc** The leading dimension of  $c$ .

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type  $T$ . Size should not be less than the value returned by [`unmqr\_scratchpad\_size`](#) function.

## Output Parameters

**c** Overwritten by the product  $Q^H * C$ ,  $Q^H * C$ ,  $C * Q$ , or  $C * Q^H$  (as specified by `left_right` and `trans`).

**scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info=-i`, the  $i$ -th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## 13.2.1.3.1.63 `unmqr` (USM Version)

### Syntax

```
cl::sycl::event onemkl::lapack::unmqr(cl::sycl::queue      &queue,      onemkl::side      left_right,
onemkl::transpose trans, std::int64_t m, std::int64_t n,
std::int64_t k, T *a, std::int64_t lda, T *tau, T *c, std::int64_t
ldc, T *scratchpad, std::int64_t scratchpad_size, const
cl::sycl::vector_class<cl::sycl::event> &events = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**left\_right** If `left_right=onemkl::side::left`,  $Q$  or  $Q^H$  is applied to  $C$  from the left.

If `left_right=onemkl::side::right`,  $Q$  or  $Q^H$  is applied to  $C$  from the right.

**trans** If `trans=onemkl::transpose::trans`, the routine multiplies  $C$  by  $Q$ .

If `trans=onemkl::transpose::nontrans`, the routine multiplies  $C$  by  $Q^H$ .

**m** The number of rows in the matrix  $A$  ( $m \leq 0$ ).

**n** The number of columns in the matrix  $A$  ( $0 \leq n \leq m$ ).

**k** The number of elementary reflectors whose product defines the matrix  $Q$  ( $0 \leq k \leq n$ ).

**a** The pointer to  $a$  as returned by `geqrf`. The second dimension of  $a$  must be at least  $\max(1, k)$ .

**lda** The leading dimension of  $a$ .

**tau** The pointer to  $\tau$  as returned by `geqrf`. The second dimension of  $a$  must be at least  $\max(1, k)$ .

**c** The array  $c$  contains the matrix  $C$ . The second dimension of  $c$  must be at least  $\max(1, n)$ .

**ldc** The leading dimension of  $c$ .

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type  $T$ . Size should not be less than the value returned by `unmqr_scratchpad_size` function.

**events** List of events to wait for before starting computation. Defaults to empty list.

## Output Parameters

**c** Overwritten by the product  $Q^H * C$ ,  $Q^H * C$ ,  $C * Q$ , or  $C * Q^H$  (as specified by `left_right` and `trans`).

**scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info=-i`, the  $i$ -th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *LAPACK Linear Equation Routines*

### 13.2.1.3.1.64 unmqr\_scratchpad\_size

Computes size of scratchpad memory required for [unmqr](#) function.

`unmqr_scratchpad_size` supports the following precisions.

T
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

#### Description

Computes the number of elements of type T the scratchpad memory to be passed to [unmqr](#) function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.1.65 unmqr\_scratchpad\_size

#### Syntax

```
template<typename T>
std::int64_t onemkl::lapack::unmqr_scratchpad_size(cl::sycl::queue &queue, onemkl::side
left_right, onemkl::transpose trans,
std::int64_t m, std::int64_t n, std::int64_t
k, std::int64_t lda, std::int64_t ldc,
std::int64_t &scratchpad_size)
```

#### Input Parameters

**queue** Device queue where calculations by [unmqr](#) function will be performed.

**left\_right** If `left_right=onemkl::side::left`, Q or  $Q^H$  is applied to C from the left.

If `left_right=onemkl::side::right`, Q or  $Q^H$  is applied to C from the right.

**trans** If `trans=onemkl::transpose::trans`, the routine multiplies C by Q.

If `trans=onemkl::transpose::conjtrans`, the routine multiplies C by  $Q^H$ .

**m** The number of rows in the matrix A ( $0 \leq m$ ).

**n** The number of columns the matrix A ( $0 \leq n \leq m$ ).

**k** The number of elementary reflectors whose product defines the matrix Q ( $0 \leq k \leq n$ ).

**lda** The leading dimension of a.

**ldc** The leading dimension of c.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by *get\_info()* method of exception object.

## Return Value

The number of elements of type T the scratchpad memory to be passed to *unmqr* function should be able to hold.

**Parent topic:** *LAPACK Linear Equation Routines*

### 13.2.1.3.2 LAPACK Singular Value and Eigenvalue Problem Routines

LAPACK Singular Value and Eigenvalue Problem routines are used for singular value and eigenvalue problems, and for performing a number of related computational tasks. The following table lists the LAPACK Singular Value and Eigenvalue Problem routine groups.

Routines	Scratchpad Size Routines	Description
<i>ge-brd</i>	<i>ge-brd_scratchpad_size</i>	Reduces a general matrix to bidiagonal form.
<i>gesvd</i>	<i>gesvd_scratchpad</i>	Computes the singular value decomposition of a general rectangular matrix.
<i>heevd</i>	<i>heevd_scratchpad</i>	Computes all eigenvalues and, optionally, all eigenvectors of a complex Hermitian matrix using divide and conquer algorithm.
<i>hegvd</i>	<i>hegvd_scratchpad</i>	Computes all eigenvalues and, optionally, all eigenvectors of a complex generalized Hermitian definite eigenproblem using divide and conquer algorithm.
<i>hetrd</i>	<i>hetrd_scratchpad</i>	Reduces a complex Hermitian matrix to tridiagonal form.
<i>orgbr</i>	<i>orgbr_scratchpad</i>	Generates the real orthogonal matrix Q or P <sup>T</sup> determined by <i>gebrd</i> .
<i>orgtr</i>	<i>orgtr_scratchpad</i>	Generates the real orthogonal matrix Q determined by <i>sytrd</i> .
<i>ormtr</i>	<i>ormtr_scratchpad</i>	Multiples a real matrix by the orthogonal matrix Q determined by <i>sytrd</i> .
<i>syevd</i>	<i>syevd_scratchpad</i>	Computes all eigenvalues and, optionally, all eigenvectors of a real symmetric matrix using divide and conquer algorithm.
<i>sygvd</i>	<i>sygvd_scratchpad</i>	Computes all eigenvalues and, optionally, all eigenvectors of a real generalized symmetric definite eigenproblem using divide and conquer algorithm.
<i>sytrd</i>	<i>sytrd_scratchpad</i>	Reduces a real symmetric matrix to tridiagonal form.
<i>ungbr</i>	<i>ungbr_scratchpad</i>	Generates the complex unitary matrix Q or P <sup>T</sup> determined by <i>gebrd</i> .
<i>ungtr</i>	<i>ungtr_scratchpad</i>	Generates the complex unitary matrix Q determined by <i>hetrd</i> .
<i>un-mtr</i>	<i>un-mtr_scratchpad_size</i>	Multiples a complex matrix by the unitary matrix Q determined by <i>hetrd</i> .

#### 13.2.1.3.2.1 *gebrd*

Reduces a general matrix to bidiagonal form.

*gebrd* supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The routine reduces a general  $m$ -by- $n$  matrix  $A$  to a bidiagonal matrix  $B$  by an orthogonal (unitary) transformation.

$$A = QBP^H = \begin{pmatrix} B_1 \\ 0 \end{pmatrix} P^H = Q_1 B_1 P_H,$$

If  $m \geq n$ , the reduction is given by

where  $B_1$  is an  $n$ -by- $n$  upper diagonal matrix,  $Q$  and  $P$  are orthogonal or, for a complex  $A$ , unitary matrices;  $Q_1$  consists of the first  $n$  columns of  $Q$ .

If  $m < n$ , the reduction is given by

$$A = Q * B * P^H = Q * (B_{10}) * P^H = Q_1 * B_1 * P_1^H,$$

where  $B_1$  is an  $m$ -by- $m$  lower diagonal matrix,  $Q$  and  $P$  are orthogonal or, for a complex  $A$ , unitary matrices;  $P_1$  consists of the first  $m$  columns of  $P$ .

The routine does not form the matrices  $Q$  and  $P$  explicitly, but represents them as products of elementary reflectors. Routines are provided to work with the matrices  $Q$  and  $P$  in this representation:

If the matrix  $A$  is real,

- to compute  $Q$  and  $P$  explicitly, call [orgbr](#).

If the matrix  $A$  is complex,

- to compute  $Q$  and  $P$  explicitly, call [ungbr](#)

### 13.2.1.3.2.2 gebrd (BUFFER Version)

#### Syntax

```
void onemkl::lapack::gebrd(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, cl::sycl::buffer<T,
                           1> &a, std::int64_t lda, cl::sycl::buffer<realT, 1> &d,
                           cl::sycl::buffer<realT, 1> &e, cl::sycl::buffer<T, 1> &tauq,
                           cl::sycl::buffer<T, 1> &taup, cl::sycl::buffer<T, 1> &scratchpad,
                           std::int64_t scratchpad_size)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**m** The number of rows in the matrix  $A$  ( $0 \leq m$ ).

**n** The number of columns in the matrix  $A$  ( $0 \leq n$ ).

**a** The buffer  $a$ , size ( $lda, *$ ). The buffer  $a$  contains the matrix  $A$ . The second dimension of  $a$  must be at least  $\max(1, m)$ .

**lda** The leading dimension of  $a$ .

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type  $T$ . Size should not be less than the value returned by [gebrd\\_scratchpad\\_size](#) function.

## Output Parameters

- a** If  $m \geq n$ , the diagonal and first super-diagonal of  $A$  are overwritten by the upper bidiagonal matrix  $B$ . The elements below the diagonal, with the buffer  $\text{tau}_Q$ , represent the orthogonal matrix  $Q$  as a product of elementary reflectors, and the elements above the first superdiagonal, with the buffer  $\text{tau}_P$ , represent the orthogonal matrix  $P$  as a product of elementary reflectors.
- If  $m < n$ , the diagonal and first sub-diagonal of  $A$  are overwritten by the lower bidiagonal matrix  $B$ . The elements below the first subdiagonal, with the buffer  $\text{tau}_Q$ , represent the orthogonal matrix  $Q$  as a product of elementary reflectors, and the elements above the diagonal, with the buffer  $\text{tau}_P$ , represent the orthogonal matrix  $P$  as a product of elementary reflectors.
- d** Buffer, size at least  $\max(1, \min(m, n))$ . Contains the diagonal elements of  $B$ .
- e** Buffer, size at least  $\max(1, \min(m, n) - 1)$ . Contains the off-diagonal elements of  $B$ .
- tau<sub>Q</sub>** Buffer, size at least  $\max(1, \min(m, n))$ . The scalar factors of the elementary reflectors which represent the orthogonal or unitary matrix  $Q$ .
- tau<sub>P</sub>** Buffer, size at least  $\max(1, \min(m, n))$ . The scalar factors of the elementary reflectors which represent the orthogonal or unitary matrix  $P$ .
- scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info == -i`, the `i`-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

### 13.2.1.3.2.3 `gebrd` (USM Version)

#### Syntax

```
cl::sycl::event onemkl::lapack::gebrd(cl::sycl::queue &queue, std::int64_t m, std::int64_t n, T
                                         *a, std::int64_t lda, RealT *d, RealT *e, T *tauq, T
                                         *taup, T *scratchpad, std::int64_t scratchpad_size, const
                                         cl::sycl::vector_class<cl::sycl::event> &events = {})
```

#### Input Parameters

- queue** The queue where the routine should be executed.
- m** The number of rows in the matrix  $A$  ( $0 \leq m$ ).
- n** The number of columns in the matrix  $A$  ( $0 \leq n$ ).
- a** Pointer to matrix  $A$ . The second dimension of  $a$  must be at least  $\max(1, m)$ .
- lda** The leading dimension of  $a$ .
- scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type  $T$ . Size should not be less than the value returned by `gebrd_scratchpad_size` function.

**events** List of events to wait for before starting computation. Defaults to empty list.

## Output Parameters

**a** If  $m \geq n$ , the diagonal and first super-diagonal of  $a$  are overwritten by the upper bidiagonal matrix  $B$ . The elements below the diagonal, with the array **tauq**, represent the orthogonal matrix  $Q$  as a product of elementary reflectors, and the elements above the first superdiagonal, with the array **taup**, represent the orthogonal matrix  $P$  as a product of elementary reflectors.

If  $m < n$ , the diagonal and first sub-diagonal of  $a$  are overwritten by the lower bidiagonal matrix  $B$ . The elements below the first subdiagonal, with the array **tauq**, represent the orthogonal matrix  $Q$  as a product of elementary reflectors, and the elements above the diagonal, with the array **taup**, represent the orthogonal matrix  $P$  as a product of elementary reflectors.

**d** Pointer to memory of size at least  $\max(1, \min(m, n))$ . Contains the diagonal elements of  $B$ .

**e** Pointer to memory of size at least  $\max(1, \min(m, n) - 1)$ . Contains the off-diagonal elements of  $B$ .

**tauq** Pointer to memory of size at least  $\max(1, \min(m, n))$ . The scalar factors of the elementary reflectors which represent the orthogonal or unitary matrix  $Q$ .

**taup** Pointer to memory of size at least  $\max(1, \min(m, n))$ . The scalar factors of the elementary reflectors which represent the orthogonal or unitary matrix  $P$ .

**scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The **info** code of the problem can be obtained by `get_info()` method of exception object:

If `info == -i`, the `i`-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *LAPACK Singular Value and Eigenvalue Problem Routines*

### 13.2.1.3.2.4 `gebrd_scratchpad_size`

Computes size of scratchpad memory required for `gebrd` function.

`gebrd_scratchpad_size` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

Computes the number of elements of type T the scratchpad memory to be passed to `gebrd` function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.2.5 `gebrd_scratchpad_size`

#### Syntax

```
template<typename T>
std::int64_t onemkl::lapack::gebrd_scratchpad_size(cl::sycl::queue &queue, std::int64_t m,
                                                    std::int64_t n, std::int64_t lda)
```

#### Input Parameters

**queue** Device queue where calculations by `gebrd` function will be performed.

**m** The number of rows in the matrix A ( $0 \leq m$ ).

**n** The number of columns in the matrix A ( $0 \leq n$ ).

**lda** The leading dimension of a.

#### Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by `get_info()` method of exception object.

#### Return Value

The number of elements of type T the scratchpad memory to be passed to `gebrd` function should be able to hold.

**Parent topic:** *LAPACK Singular Value and Eigenvalue Problem Routines*

### 13.2.1.3.2.6 `gesvd`

Computes the singular value decomposition of a general rectangular matrix.

`gesvd` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

### 13.2.1.3.2.7 gesvd (BUFFER Version)

#### Syntax

```
void onemkl::lapack::gesvd(cl::sycl::queue &queue, onemkl::job jobu, onemkl::job jobvt,
                           std::int64_t m, std::int64_t n, cl::sycl::buffer<T, 1> &a, std::int64_t lda,
                           cl::sycl::buffer<realT, 1> &s, cl::sycl::buffer<T, 1> &u, std::int64_t ldu,
                           cl::sycl::buffer<T, 1> &vt, std::int64_t ldvt, cl::sycl::buffer<T, 1>
                           &scratchpad, std::int64_t scratchpad_size)
```

#### Description

The routine computes the singular value decomposition (SVD) of a real/complex  $m$ -by- $n$  matrix  $A$ , optionally computing the left and/or right singular vectors. The SVD is written as

$A = U \Sigma V^T$  for real routines

$A = U \Sigma V^H$  for complex routines

where  $\Sigma$  is an  $m$ -by- $n$  diagonal matrix,  $U$  is an  $m$ -by- $m$  orthogonal/unitary matrix, and  $V$  is an  $n$ -by- $n$  orthogonal/unitary matrix. The diagonal elements of  $\Sigma$  are the singular values of  $A$ ; they are real and non-negative, and are returned in descending order. The first  $\min(m, n)$  columns of  $U$  and  $V$  are the left and right singular vectors of  $A$ .

#### Input Parameters

**queue** The queue where the routine should be executed.

**jobu** Must be `job::allvec`, `job::somevec`, `job::overwritevec`, or `job::novec`. Specifies options for computing all or part of the matrix  $U$ .

If `jobu = job::allvec`, all  $m$  columns of  $U$  are returned in the buffer  $u$ ;

if `jobu = job::somevec`, the first  $\min(m, n)$  columns of  $U$  (the left singular vectors) are returned in the buffer  $u$ ;

if `jobu = job::overwritevec`, the first  $\min(m, n)$  columns of  $U$  (the left singular vectors) are overwritten on the buffer  $a$ ;

if `jobu = job::novec`, no columns of  $U$  (no left singular vectors) are computed.

**jobvt** Must be `job::allvec`, `job::somevec`, `job::overwritevec`, or `job::novec`. Specifies options for computing all or part of the matrix  $VT/VH$ .

If `jobvt = job::allvec`, all  $n$  columns of  $VT/VH$  are returned in the buffer  $vt$ ;

if `jobvt = job::somevec`, the first  $\min(m, n)$  columns of  $VT/VH$  (the left singular vectors) are returned in the buffer  $vt$ ;

if `jobvt = job::overwritevec`, the first  $\min(m, n)$  columns of  $VT/VH$  (the left singular vectors) are overwritten on the buffer  $a$ ;

if `jobvt = job::novec`, no columns of  $VT/VH$  (no left singular vectors) are computed.

`jobvt` and `jobu` cannot both be `job::overwritevec`.

**m** The number of rows in the matrix  $A$  ( $0 \leq m$ ).

**a** The buffer  $a$ , size  $(\text{lda}, *)$ . The buffer  $a$  contains the matrix  $A$ . The second dimension of  $a$  must be at least  $\max(1, m)$ .

**lda** The leading dimension of  $a$ .

**lDU** The leading dimension of u.

**lDVT** The leading dimension of vt.

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by `gesvd_scratchpad_size` function.

## Output Parameters

**a** On exit,

If `jobu = job::overwritevec`, a is overwritten with the first  $\min(m, n)$  columns of U (the left singular vectors stored columnwise);

If `jobvt = job::overwritevec`, a is overwritten with the first  $\min(m, n)$  rows of  $V^T/V^H$  (the right singular vectors stored rowwise);

If `jobu ≠ job::overwritevec` and `jobvt ≠ job::overwritevec`, the contents of a are destroyed.

**s** Buffer containing the singular values, size at least  $\max(1, \min(m, n))$ . Contains the singular values of A sorted so that  $s(i) ≥ s(i+1)$ .

**u** Buffer containing U; the second dimension of u must be at least  $\max(1, m)$  if `jobu = job::allvec`, and at least  $\max(1, \min(m, n))$  if `jobu = job::somevec`.

If `jobu = job::allvec`, u contains the m-by-m orthogonal/unitary matrix U.

If `jobu = job::somevec`, u contains the first  $\min(m, n)$  columns of U (the left singular vectors stored column-wise).

If `jobu = job::novec` or `job::overwritevec`, u is not referenced.

**vt** Buffer containing  $V^T$ ; the second dimension of vt must be at least  $\max(1, n)$ .

If `jobvt = job::allvec`, vt contains the n-by-n orthogonal/unitary matrix  $V^T/V^H$ .

If `jobvt = job::somevec`, vt contains the first  $\min(m, n)$  rows of  $V^T/V^H$  (the right singular vectors stored row-wise).

If `jobvt = job::novec` or `job::overwritevec`, vt is not referenced.

**scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info=-i`, the i-th parameter had an illegal value.

If `info=i`, then if `bdsqr` did not converge, i specifies how many superdiagonals of the intermediate bidiagonal form B did not converge to zero, and `scratchpad(2:min(m, n))` contains the unconverged superdiagonal elements of an upper bidiagonal matrix B whose diagonal is in s (not necessarily sorted). B satisfies  $A = U * B * V^T$ , so it has the same singular values as A, and singular vectors related by U and  $V^T$ .

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

### 13.2.1.3.2.8 gesvd (USM Version)

#### Syntax

```
cl::sycl::event onemkl::lapack::gesvd(cl::sycl::queue &queue, onemkl::job jobu, onemkl::job jobvt,
                                      std::int64_t m, std::int64_t n, T *a, std::int64_t lda,
                                      RealT *s, T *u, std::int64_t ldu, T *vt, std::int64_t ldvt,
                                      T *scratchpad, std::int64_t scratchpad_size, const
                                      cl::sycl::vector_class<cl::sycl::event> &events = {})
```

#### Description

The routine computes the singular value decomposition (SVD) of a real/complex  $m$ -by- $n$  matrix  $A$ , optionally computing the left and/or right singular vectors. The SVD is written as

$A = U \Sigma V^T$  for real routines

$A = U \Sigma V^H$  for complex routines

where  $\Sigma$  is an  $m$ -by- $n$  diagonal matrix,  $U$  is an  $m$ -by- $m$  orthogonal/unitary matrix, and  $V$  is an  $n$ -by- $n$  orthogonal/unitary matrix. The diagonal elements of  $\Sigma$  are the singular values of  $A$ ; they are real and non-negative, and are returned in descending order. The first  $\min(m, n)$  columns of  $U$  and  $V$  are the left and right singular vectors of  $A$ .

#### Input Parameters

**queue** The queue where the routine should be executed.

**jobu** Must be `job::allvec`, `job::somevec`, `job::overwritevec`, or `job::novec`. Specifies options for computing all or part of the matrix  $U$ .

If `jobu = job::allvec`, all  $m$  columns of  $U$  are returned in the array  $u$ ;

if `jobu = job::somevec`, the first  $\min(m, n)$  columns of  $U$  (the left singular vectors) are returned in the array  $u$ ;

if `jobu = job::overwritevec`, the first  $\min(m, n)$  columns of  $U$  (the left singular vectors) are overwritten on the array  $a$ ;

if `jobu = job::novec`, no columns of  $U$  (no left singular vectors) are computed.

**jobvt** Must be `job::allvec`, `job::somevec`, `job::overwritevec`, or `job::novec`. Specifies options for computing all or part of the matrix  $VT/VH$ .

If `jobvt = job::allvec`, all  $n$  columns of  $VT/VH$  are returned in the array  $vt$ ;

if `jobvt = job::somevec`, the first  $\min(m, n)$  columns of  $VT/VH$  (the left singular vectors) are returned in the array  $vt$ ;

if `jobvt = job::overwritevec`, the first  $\min(m, n)$  columns of  $VT/VH$  (the left singular vectors) are overwritten on the array  $a$ ;

if `jobvt = job::novec`, no columns of  $VT/VH$  (no left singular vectors) are computed.

`jobvt` and `jobu` cannot both be `job::overwritevec`.

**m** The number of rows in the matrix  $A$  ( $0 \leq m$ ).

**a** Pointer to array  $a$ , size `(lda, *)`, containing the matrix  $A$ . The second dimension of  $a$  must be at least  $\max(1, m)$ .

**lda** The leading dimension of  $a$ .

**lDU** The leading dimension of u.

**lDVT** The leading dimension of vt.

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by `gesvd_scratchpad_size` function.

**events** List of events to wait for before starting computation. Defaults to empty list.

## Output Parameters

**a** On exit,

If `jobu = job::overwritevec`, a is overwritten with the first  $\min(m, n)$  columns of U (the left singular vectors stored columnwise);

If `jobvt = job::overwritevec`, a is overwritten with the first  $\min(m, n)$  rows of  $V^T/V^H$  (the right singular vectors stored rowwise);

If `jobu ≠ job::overwritevec` and `jobvt ≠ job::overwritevec`, the contents of a are destroyed.

**s** Array containing the singular values, size at least  $\max(1, \min(m, n))$ . Contains the singular values of A sorted so that  $s(i) ≥ s(i+1)$ .

**u** Array containing U; the second dimension of u must be at least  $\max(1, m)$  if `jobu = job::allvec`, and at least  $\max(1, \min(m, n))$  if `jobu = job::somevec`.

If `jobu = job::allvec`, u contains the m-by-m orthogonal/unitary matrix U.

If `jobu = job::somevec`, u contains the first  $\min(m, n)$  columns of U (the left singular vectors stored column-wise).

If `jobu = job::novec` or `job::overwritevec`, u is not referenced.

**vt** Array containing  $V^T$ ; the second dimension of vt must be at least  $\max(1, n)$ .

If `jobvt = job::allvec`, vt contains the n-by-n orthogonal/unitary matrix  $V^T/V^H$ .

If `jobvt = job::somevec`, vt contains the first  $\min(m, n)$  rows of  $V^T/V^H$  (the right singular vectors stored row-wise).

If `jobvt = job::novec` or `job::overwritevec`, vt is not referenced.

**scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info=-i`, the i-th parameter had an illegal value.

If `info=i`, then if `bdsqr` did not converge, i specifies how many superdiagonals of the intermediate bidiagonal form B did not converge to zero, and `scratchpad(2:min(m, n))` contains the unconverged superdiagonal elements of an upper bidiagonal matrix B whose diagonal is in `s` (not necessarily sorted). B satisfies  $A = U * B * V^T$ , so it has the same singular values as A, and singular vectors related by U and  $V^T$ .

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *LAPACK Singular Value and Eigenvalue Problem Routines*

### 13.2.1.3.2.9 gesvd\_scratchpad\_size

Computes size of scratchpad memory required for *gesvd* function.

*gesvd\_scratchpad\_size* supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

## Description

Computes the number of elements of type T the scratchpad memory to be passed to *gesvd* function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.2.10 gesvd\_scratchpad\_size

## Syntax

```
template<typename T>
std::int64_t onemkl::lapack::gesvd_scratchpad_size(cl::sycl::queue &queue, onemkl::job
jobu, onemkl::job jobvt, std::int64_t m,
std::int64_t n, std::int64_t lda, std::int64_t
ldu, std::int64_t ldvt)
```

## Input Parameters

**queue** Device queue where calculations by *gesvd* function will be performed.

**jobu** Must be *job::allvec*, *job::somevec*, *job::overwritevec*, or *job::novec*. Specifies options for computing all or part of the matrix U.

If *jobu* = *job::allvec*, all m columns of U are returned in the buffer u;

if *jobu* = *job::somevec*, the first *min(m, n)* columns of U (the left singular vectors) are returned in the buffer v;

if *jobu* = *job::overwritevec*, the first *min(m, n)* columns of U (the left singular vectors) are overwritten on the buffer a;

if *jobu* = *job::novec*, no columns of U (no left singular vectors) are computed.

**jobvt** Must be *job::allvec*, *job::somevec*, *job::overwritevec*, or *job::novec*. Specifies options for computing all or part of the matrix VT/VH.

If *jobvt* = *job::allvec*, all n columns of VT/VH are returned in the buffer vt;

if `jobvt = job::somevec`, the first  $\min(m, n)$  columns of  $VT/VH$  (the left singular vectors) are returned in the buffer `vt`;

if `jobvt = job::overwritevec`, the first  $\min(m, n)$  columns of  $VT/VH$  (the left singular vectors) are overwritten on the buffer `a`;

if `jobvt = job::novec`, no columns of  $VT/VH$  (no left singular vectors) are computed.

**m** The number of rows in the matrix `A` ( $0 \leq m$ ).

**n** The number of columns in the matrix `A` ( $0 \leq n$ ).

**lda** The leading dimension of `a`.

**ldu** The leading dimension of `u`.

**ldvt** The leading dimension of `vt`.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by `get_info()` method of exception object.

## Return Value

The number of elements of type `T` the scratchpad memory to be passed to `gesvd` function should be able to hold.

**Parent topic:** *LAPACK Singular Value and Eigenvalue Problem Routines*

### 13.2.1.3.2.11 heevd

Computes all eigenvalues and, optionally, all eigenvectors of a complex Hermitian matrix using divide and conquer algorithm.

`heevd` supports the following precisions.

<code>T</code>
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian matrix `A`. In other words, it can compute the spectral factorization of `A` as:  $A = Z * \Lambda * Z^H$ .

Here  $\Lambda$  is a real diagonal matrix whose diagonal elements are the eigenvalues  $\lambda_i$ , and  $Z$  is the (complex) unitary matrix whose columns are the eigenvectors  $z_i$ . Thus,

$$A * z_i = \lambda_i * z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

### 13.2.1.3.2.12 heevd (BUFFER Version)

#### Syntax

```
void onemkl::lapack::heevd(cl::sycl::queue &queue, onemkl::job jobz, onemkl::uplo upper_lower,
                           std::int64_t n, butter<T, 1> &a, std::int64_t lda, cl::sycl::buffer<realT, 1>
                           &w, cl::sycl::buffer<T, 1> &scratchpad, std::int64_t scratchpad_size)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**jobz** Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

**upper\_lower** Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = job::upper`, a stores the upper triangular part of A.

If `upper_lower = job::lower`, a stores the lower triangular part of A.

**n** The order of the matrix A ( $0 \leq n$ ).

**a** The buffer a, size (`lda, *`). The buffer a contains the matrix A. The second dimension of a must be at least `max(1, n)`.

**lda** The leading dimension of a. Must be at least `max(1, n)`.

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by `heevd_scratchpad_size` function.

#### Output Parameters

**a** If `jobz = job::vec`, then on exit this buffer is overwritten by the unitary matrix Z which contains the eigenvectors of A.

**w** Buffer, size at least n. Contains the eigenvalues of the matrix A in ascending order.

**scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

#### Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info=-i`, the i-th parameter had an illegal value.

If `info=i`, and `jobz = onemkl::job::novec`, then the algorithm failed to converge; i indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero.

If `info=i`, and `jobz = onemkl::job::vec`, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns `info/(n+1)` through `mod(info, n+1)`.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

### 13.2.1.3.2.13 heevd (USM Version)

#### Syntax

```
cl::sycl::event onemkl::lapack::heevd(cl::sycl::queue &queue, onemkl::job jobz, onemkl::uplo upper_lower, std::int64_t n, butter<T, 1> &a, std::int64_t lda, RealT *w, T *scratchpad, std::int64_t scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**jobz** Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

**upper\_lower** Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = job::upper`, `a` stores the upper triangular part of `A`.

If `upper_lower = job::lower`, `a` stores the lower triangular part of `A`.

**n** The order of the matrix `A` ( $0 \leq n$ ).

**a** Pointer to array containing `A`, size `(lda, *)`. The second dimension of `a` must be at least `max(1, n)`.

**lda** The leading dimension of `a`. Must be at least `max(1, n)`.

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by `heevd_scratchpad_size` function.

**events** List of events to wait for before starting computation. Defaults to empty list.

#### Output Parameters

**a** If `jobz = job::vec`, then on exit this array is overwritten by the unitary matrix `Z` which contains the eigenvectors of `A`.

**w** Pointer to array of size at least `n`. Contains the eigenvalues of the matrix `A` in ascending order.

**scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

#### Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info=-i`, the `i`-th parameter had an illegal value.

If `info=i`, and `jobz = onemkl::job::novec`, then the algorithm failed to converge; `i` indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero.

If `info=i`, and `jobz = onemkl::job::vec`, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns `info/(n+1)` through `mod(info, n+1)`.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *LAPACK Singular Value and Eigenvalue Problem Routines*

### 13.2.1.3.2.14 heevd\_scratchpad\_size

Computes size of scratchpad memory required for `heevd` function.

`heevd_scratchpad_size` supports the following precisions.

<b>T</b>
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

Computes the number of elements of type T the scratchpad memory to be passed to `heevd` function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.2.15 heevd\_scratchpad\_size

## Syntax

```
template<typename T>
std::int64_t onemkl::lapack::heevd_scratchpad_size(cl::sycl::queue &queue, onemkl::job jobz,
                                                    onemkl::uplo upper_lower, std::int64_t n,
                                                    std::int64_t lda)
```

## Input Parameters

**queue** Device queue where calculations by `heevd` function will be performed.

**jobz** Must be `job::novec` or `job::vec`.

If `jobz` = `job::novec`, then only eigenvalues are computed.

If `jobz` = `job::vec`, then eigenvalues and eigenvectors are computed.

**upper\_lower** Must be `uplo::upper` or `uplo::lower`.

If `upper_lower` = `job::upper`, a stores the upper triangular part of A.

If `upper_lower` = `job::lower`, a stores the lower triangular part of A.

**n** The order of the matrix A ( $0 \leq n$ ).

**lda** The leading dimension of a.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by *get\_info()* method of exception object.

## Return Value

The number of elements of type T the scratchpad memory to be passed to *heevd* function should be able to hold.

**Parent topic:** *LAPACK Singular Value and Eigenvalue Problem Routines*

### 13.2.1.3.2.16 hegvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem using a divide and conquer method.

*hegvd* supports the following precisions.

T
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

$$A \times = \lambda \times B \times, \quad A \times B \times = \lambda \times, \text{ or } B \times A \times = \lambda \times.$$

Here A and B are assumed to be Hermitian and B is also positive definite.

It uses a divide and conquer algorithm.

### 13.2.1.3.2.17 hegvd (BUFFER Version)

## Syntax

```
void onemkl::lapack::hegvd(cl::sycl::queue &queue, std::int64_t itype, onemkl::job jobz, onemkl::uplo upper_lower, std::int64_t n, cl::sycl::buffer<T, 1> &a, std::int64_t lda, cl::sycl::buffer<T, 1> &b, std::int64_t ldb, cl::sycl::buffer<realT, 1> &w, cl::sycl::buffer<T, 1> &scratchpad, std::int64_t scratchpad_size)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**itype** Must be 1 or 2 or 3. Specifies the problem type to be solved:

if  $\text{itype} = 1$ , the problem type is  $A \times x = \lambda B \times x$ ;

if  $\text{itype} = 2$ , the problem type is  $A \times B \times x = \lambda x$ ;

if  $\text{itype} = 3$ , the problem type is  $B \times A \times x = \lambda x$ .

**jobz** Must be `job::novec` or `job::vec`.

If  $\text{jobz} = \text{job::novec}$ , then only eigenvalues are computed.

If  $\text{jobz} = \text{job::vec}$ , then eigenvalues and eigenvectors are computed.

**upper\_lower** Must be `uplo::upper` or `uplo::lower`.

If  $\text{upper\_lower} = \text{uplo::upper}$ , a and b store the upper triangular part of A and B.

If  $\text{upper\_lower} = \text{uplo::lower}$ , a and b stores the lower triangular part of A and B.

**n** The order of the matrices A and B ( $0 \leq n$ ).

**a** Buffer, size a (`lda, *`) contains the upper or lower triangle of the Hermitian matrix A, as specified by `upper_lower`.

The second dimension of a must be at least  $\max(1, n)$ .

**lda** The leading dimension of a; at least  $\max(1, n)$ .

**b** Buffer, size b (`ldb, *`) contains the upper or lower triangle of the Hermitian matrix B, as specified by `upper_lower`.

The second dimension of b must be at least  $\max(1, n)$ .

**ldb** The leading dimension of b; at least  $\max(1, n)$ .

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by [`hegvd\_scratchpad\_size`](#) function.

## Output Parameters

**a** On exit, if  $\text{jobz} = \text{job::vec}$ , then if  $\text{info} = 0$ , a contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows:

if  $\text{itype} = 1$  or  $2$ ,  $Z^H \times B \times Z = I$ ;

if  $\text{itype} = 3$ ,  $Z^H \times \text{inv}(B) \times Z = I$ ;

If  $\text{jobz} = \text{job::novec}$ , then on exit the upper triangle (if  $\text{upper\_lower} = \text{uplo::upper}$ ) or the lower triangle (if  $\text{upper\_lower} = \text{uplo::lower}$ ) of A, including the diagonal, is destroyed.

**b** On exit, if  $\text{info} \leq n$ , the part of b containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization  $B = U^H \times U$  or  $B = L \times L^H$ .

**w** Buffer, size at least n. If  $\text{info} = 0$ , contains the eigenvalues of the matrix A in ascending order.

**scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info==i`, the `i`-th parameter had an illegal value.

For `info≤n`:

If `info=i`, and `jobz = onemkl::job::novec`, then the algorithm failed to converge; `i` indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero;

If `info=i`, and `jobz = onemkl::job::vec`, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns `info/(n+1)` through `mod(info, n+1)`.

For `info>n`:

If `info=n+i`, for  $1 \leq i \leq n$ , then the leading minor of order `i` of `B` is not positive-definite. The factorization of `B` could not be completed and no eigenvalues or eigenvectors were computed.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

### 13.2.1.3.2.18 hegvd (USM Version)

#### Syntax

```
cl::sycl::event onemkl::lapack::hegvd(cl::sycl::queue &queue, std::int64_t itype, onemkl::job jobz,
                                       onemkl::uplo upper_lower, std::int64_t n, T *a, std::int64_t lda,
                                       T *b, std::int64_t ldb, RealT *w, T *scratchpad, std::int64_t
                                       scratchpad_size, const cl::sycl::vector_class<cl::sycl::event>
                                       &events = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**itype** Must be 1 or 2 or 3. Specifies the problem type to be solved:

- if `itype= 1`, the problem type is  $A*x = \lambda B*x$ ;
- if `itype= 2`, the problem type is  $A*B*x = \lambda x$ ;
- if `itype= 3`, the problem type is  $B*A*x = \lambda x$ .

**jobz** Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

**upper\_lower** Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = uplo::upper`, `a` and `b` store the upper triangular part of `A` and `B`.

If `upper_lower = uplo::lower`, `a` and `b` stores the lower triangular part of `A` and `B`.

**n** The order of the matrices `A` and `B` ( $0 \leq n$ ).

**a** Pointer to array of size  $a(\text{lda}, *)$  containing the upper or lower triangle of the Hermitian matrix A, as specified by `upper_lower`. The second dimension of a must be at least  $\max(1, n)$ .

**lda** The leading dimension of a; at least  $\max(1, n)$ .

**b** Pointer to array of size  $b(\text{ldb}, *)$  containing the upper or lower triangle of the Hermitian matrix B, as specified by `upper_lower`. The second dimension of b must be at least  $\max(1, n)$ .

**ldb** The leading dimension of b; at least  $\max(1, n)$ .

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by `hegvd_scratchpad_size` function.

**events** List of events to wait for before starting computation. Defaults to empty list.

## Output Parameters

**a** On exit, if `jobz = job::vec`, then if `info = 0`, a contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows:

if `itype= 1 or 2`,  $Z^H * B * Z = I$ ;

if `itype= 3`,  $Z^H * \text{inv}(B) * Z = I$ ;

If `jobz = job::novec`, then on exit the upper triangle (if `upper_lower = uplo::upper`) or the lower triangle (if `upper_lower = uplo::lower`) of A, including the diagonal, is destroyed.

**b** On exit, if `info≤n`, the part of b containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization  $B = U^H * U$  or  $B = L * L^H$ .

**w** Pointer to array of size at least n. If `info = 0`, contains the eigenvalues of the matrix A in ascending order.

**scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info=-i`, the i-th parameter had an illegal value.

For `info≤n`:

If `info=i`, and `jobz = onemkl::job::novec`, then the algorithm failed to converge; i indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero;

If `info=i`, and `jobz = onemkl::job::vec`, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns  $\text{info}/(n+1)$  through  $\text{mod}(\text{info}, n+1)$ .

For `info>n`:

If `info=n+i`, for  $1≤i≤n$ , then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** [LAPACK Singular Value and Eigenvalue Problem Routines](#)

### 13.2.1.3.2.19 `hegvd_scratchpad_size`

Computes size of scratchpad memory required for `hegvd` function.

`hegvd_scratchpad_size` supports the following precisions.

T
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

Computes the number of elements of type T the scratchpad memory to be passed to `hegvd` function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.2.20 `hegvd_scratchpad_size`

## Syntax

```
template<typename T>
std::int64_t onemkl::lapack::hegvd_scratchpad_size(cl::sycl::queue &queue, std::int64_t
itotype, onemkl::job jobz, onemkl::uplo
upper_lower, std::int64_t n, std::int64_t
lda, std::int64_t ldb)
```

## Input Parameters

**queue** Device queue where calculations by `hegvd` function will be performed.

**itype** Must be 1 or 2 or 3. Specifies the problem type to be solved:

- if `itype = 1`, the problem type is  $A \cdot x = \lambda \cdot B \cdot x$ ;
- if `itype = 2`, the problem type is  $A \cdot B \cdot x = \lambda \cdot x$ ;
- if `itype = 3`, the problem type is  $B \cdot A \cdot x = \lambda \cdot x$ .

**jobz** Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

**upper\_lower** Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = uplo::upper`, a and b store the upper triangular part of A and B.

If `upper_lower = uplo::lower`, a and b store the lower triangular part of A and B.

**n** The order of the matrices A and B ( $0 \leq n$ ).

**lda** The leading dimension of a. Currently lda is not referenced in this function.

**ldb** The leading dimension of b. Currently ldb is not referenced in this function.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by `get_info()` method of exception object.

## Return Value

The number of elements of type T the scratchpad memory to be passed to `hegvd` function should be able to hold.

**Parent topic:** *LAPACK Singular Value and Eigenvalue Problem Routines*

### 13.2.1.3.2.21 `hetrd`

Reduces a complex Hermitian matrix to tridiagonal form.

`hetrd` supports the following precisions.

Routine name	T
<code>chetrd</code>	<code>std::complex&lt;float&gt;</code>
<code>zhetrd</code>	<code>std::complex&lt;double&gt;</code>

## Description

The routine reduces a complex Hermitian matrix A to symmetric tridiagonal form T by a unitary similarity transformation:  $A = Q \cdot T \cdot Q^H$ . The unitary matrix Q is not formed explicitly but is represented as a product of n-1 elementary reflectors. Routines are provided to work with Q in this representation.

### 13.2.1.3.2.22 `hetrd` (BUFFER Version)

## Syntax

```
void onemkl::lapack::hetrd(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n,
                           cl::sycl::buffer<T, 1> &a, std::int64_t lda, cl::sycl::buffer<realT,
                           1> &d, cl::sycl::buffer<realT, 1> &e, cl::sycl::buffer<T, 1> &tau,
                           cl::sycl::buffer<T, 1> &scratchpad, std::int64_t scratchpad_size)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = uplo::upper`, a stores the upper triangular part of A.

If `upper_lower = uplo::lower`, a stores the lower triangular part of A.

**n** The order of the matrices A ( $0 \leq n$ ).

**a** Buffer, size  $(\text{lda}, *)$ . The buffer  $a$  contains either the upper or lower triangle of the Hermitian matrix  $A$ , as specified by `upper_lower`.

The second dimension of  $a$  must be at least  $\max(1, n)$ .

**lda** The leading dimension of  $a$ ; at least  $\max(1, n)$

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type  $T$ . Size should not be less than the value returned by `hetrd_scratchpad_size` function.

## Output Parameters

**a** On exit,

if `upper_lower = uplo::upper`, the diagonal and first superdiagonal of  $A$  are overwritten by the corresponding elements of the tridiagonal matrix  $T$ , and the elements above the first superdiagonal, with the buffer  $\tau$ , represent the orthogonal matrix  $Q$  as a product of elementary reflectors;

if `upper_lower = uplo::lower`, the diagonal and first subdiagonal of  $A$  are overwritten by the corresponding elements of the tridiagonal matrix  $T$ , and the elements below the first subdiagonal, with the buffer  $\tau$ , represent the orthogonal matrix  $Q$  as a product of elementary reflectors.

**d** Buffer containing the diagonal elements of the matrix  $T$ . The dimension of  $d$  must be at least  $\max(1, n)$ .

**e** Buffer containing the off diagonal elements of the matrix  $T$ . The dimension of  $e$  must be at least  $\max(1, n-1)$ .

**tau** Buffer, size at least  $\max(1, n-1)$ . Stores  $(n-1)$  scalars that define elementary reflectors in decomposition of the unitary matrix  $Q$  in a product of  $n-1$  elementary reflectors.

**scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info=-i`, the  $i$ -th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

### 13.2.1.3.2.23 `hetrd` (USM Version)

#### Syntax

```
cl::sycl::event onemkl::lapack::hetrd(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, T *a, std::int64_t lda, RealT *d, RealT *e, T *tau, T *scratchpad, std::int64_t scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = uplo::upper`,  $A$  stores the upper triangular part of  $A$ .

If `upper_lower = uplo::lower`,  $A$  stores the lower triangular part of  $A$ .

**n** The order of the matrices  $A$  ( $0 \leq n$ ).

**a** The pointer to matrix  $A$ , size `(lda, *)`. Contains either the upper or lower triangle of the Hermitian matrix  $A$ , as specified by `upper_lower`. The second dimension of  $a$  must be at least `max(1, n)`.

**lda** The leading dimension of  $a$ ; at least `max(1, n)`

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type  $T$ . Size should not be less than the value returned by `hetrd_scratchpad_size` function.

**events** List of events to wait for before starting computation. Defaults to empty list.

## Output Parameters

**a** On exit,

if `upper_lower = uplo::upper`, the diagonal and first superdiagonal of  $A$  are overwritten by the corresponding elements of the tridiagonal matrix  $T$ , and the elements above the first superdiagonal, with the array  $\tau$ , represent the orthogonal matrix  $Q$  as a product of elementary reflectors;

if `upper_lower = uplo::lower`, the diagonal and first subdiagonal of  $A$  are overwritten by the corresponding elements of the tridiagonal matrix  $T$ , and the elements below the first subdiagonal, with the array  $\tau$ , represent the orthogonal matrix  $Q$  as a product of elementary reflectors.

**d** Pointer to diagonal elements of the matrix  $T$ . The dimension of  $d$  must be at least `max(1, n)`.

**e** Pointer to off diagonal elements of the matrix  $T$ . The dimension of  $e$  must be at least `max(1, n-1)`.

**tau** Pointer to array of size at least `max(1, n-1)`. Stores  $(n-1)$  scalars that define elementary reflectors in decomposition of the unitary matrix  $Q$  in a product of  $n-1$  elementary reflectors.

**scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info=-i`, the  $i$ -th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *LAPACK Singular Value and Eigenvalue Problem Routines*

### 13.2.1.3.2.24 `hetrd_scratchpad_size`

Computes size of scratchpad memory required for `hetrd` function.

`hetrd_scratchpad_size` supports the following precisions.

T
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

Computes the number of elements of type T the scratchpad memory to be passed to `hetrd` function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.2.25 `hetrd_scratchpad_size`

## Syntax

```
template<typename T>
std::int64_t onemkl::lapack::hetrd_scratchpad_size(cl::sycl::queue &queue, onemkl::uplo upper_lower,
                                                    std::int64_t n, std::int64_t lda)
```

## Input Parameters

**queue** Device queue where calculations by `hetrd` function will be performed.

**upper\_lower** Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = uplo::upper`, a stores the upper triangular part of A and B.

If `upper_lower = uplo::lower`, a stores the lower triangular part of A.

**n** The order of the matrices A and B ( $0 \leq n$ ).

**lda** The leading dimension of a. Currently, lda is not referenced in this function.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by `get_info()` method of exception object.

## Return Value

The number of elements of type T the scratchpad memory to be passed to `hetrd` function should be able to hold.

**Parent topic:** *LAPACK Singular Value and Eigenvalue Problem Routines*

### 13.2.1.3.2.26 orgbr

Generates the real orthogonal matrix  $Q$  or  $P^T$  determined by `gebrd`.

`orgbr` supports the following precisions.

T
float
double

## Description

The routine generates the whole or part of the orthogonal matrices  $Q$  and  $P^T$  formed by the routines `gebrd`. All valid combinations of arguments are described in *Input parameters*. In most cases you need the following:

To compute the whole m-by-m matrix  $Q$ :

```
orgbr(queue, generate::q, m, m, n, a, ...)
```

(note that the array `a` must have at least `m` columns).

To form the `n` leading columns of  $Q$  if `m > n`:

```
orgbr(queue, generate::q, m, n, n, a, ...)
```

To compute the whole n-by-n matrix  $P^T$ :

```
orgbr(queue, generate::p, n, n, m, a, ...)
```

(note that the array `a` must have at least `n` rows).

To form the `m` leading rows of  $P^T$  if `m < n`:

```
orgbr(queue, generate::p, m, n, m, a, ...)
```

### 13.2.1.3.2.27 orgbr (BUFFER Version)

#### Syntax

```
void onemkl::lapack::orgbr(cl::sycl::queue &queue, onemkl::generate gen, std::int64_t m,
                           std::int64_t n, std::int64_t k, cl::sycl::buffer<T, 1> &a, std::int64_t lda,
                           cl::sycl::buffer<T, 1> &tau, cl::sycl::buffer<T, 1> &scratchpad,
                           std::int64_t scratchpad_size)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**gen** Must be `generate::q` or `generate::p`.

If `gen= generate::q`, the routine generates the matrix  $Q$ .

If `gen= generate::p`, the routine generates the matrix  $P^T$ .

**m** The number of rows in the matrix  $Q$  or  $P^T$  to be returned ( $0 \leq m$ ).

If `gen= generate::q, m \geq n \geq \min(m, k)`.

If `gen= generate::p, n \geq m \geq \min(n, k)`.

**n** The number of rows in the matrix  $Q$  or  $P^T$  to be returned ( $0 \leq n$ ). See **m** for constraints.

**k** If `gen= generate::q`, the number of columns in the original  $m$ -by- $k$  matrix reduced by `gebrd`.

If `gen= generate::p`, the number of rows in the original  $k$ -by- $n$  matrix reduced by `gebrd`.

**a** The buffer **a** as returned by `gebrd`.

**lda** The leading dimension of **a**.

**tau** Buffer, size  $\min(m, k)$  if `gen= generate::q`, size  $\min(n, k)$  if `gen= generate::p`. Scalar factor of the elementary reflectors, as returned by `gebrd` in the array `tauq` or `taup`.

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type **T**. Size should not be less than the value returned by `orgbr_scratchpad_size` function.

#### Output Parameters

**a** Overwritten by **n** leading columns of the  $m$ -by- $m$  orthogonal matrix  $Q$  or  $P^T$  (or the leading rows or columns thereof) as specified by **gen**, **m**, and **n**.

**scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

#### Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info=-i`, the *i*-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

### 13.2.1.3.2.28 `orgbr` (USM Version)

#### Syntax

```
cl::sycl::event onemkl::lapack::orgbr(cl::sycl::queue &queue, onemkl::generate gen, std::int64_t m, std::int64_t n, std::int64_t k, T *a, std::int64_t lda, T *tau, T *scratchpad, std::int64_t scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**gen** Must be `generate::q` or `generate::p`.

If `gen= generate::q`, the routine generates the matrix  $Q$ .

If `gen= generate::p`, the routine generates the matrix  $P^T$ .

**m** The number of rows in the matrix  $Q$  or  $P^T$  to be returned ( $0 \leq m$ ).

If `gen= generate::q, m \geq n \geq \min(m, k)`.

If `gen= generate::p, n \geq m \geq \min(n, k)`.

**n** The number of rows in the matrix  $Q$  or  $P^T$  to be returned ( $0 \leq n$ ). See **m** for constraints.

**k** If `gen= generate::q`, the number of columns in the original  $m$ -by- $k$  matrix reduced by `gebrd`.

If `gen= generate::p`, the number of rows in the original  $k$ -by- $n$  matrix reduced by `gebrd`.

**a** Pointer to array **a** as returned by `gebrd`.

**lda** The leading dimension of **a**.

**tau** Pointer to array of size  $\min(m, k)$  if `gen= generate::q`, size  $\min(n, k)$  if `gen= generate::p`. Scalar factor of the elementary reflectors, as returned by `gebrd` in the array tauq or taup.

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type **T**. Size should not be less than the value returned by `orgbr_scratchpad_size` function.

**events** List of events to wait for before starting computation. Defaults to empty list.

#### Output Parameters

**a** Overwritten by **n** leading columns of the  $m$ -by- $m$  orthogonal matrix  $Q$  or  $P^T$  (or the leading rows or columns thereof) as specified by **gen**, **m**, and **n**.

**scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info==i`, the `i`-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *LAPACK Singular Value and Eigenvalue Problem Routines*

### 13.2.1.3.2.29 `orgbr_scratchpad_size`

Computes size of scratchpad memory required for `orgbr` function.

`orgbr_scratchpad_size` supports the following precisions.

T
float
double

## Description

Computes the number of elements of type T the scratchpad memory to be passed to `orgbr` function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.2.30 `orgbr_scratchpad_size`

## Syntax

```
template<typename T>
std::int64_t onemkl::lapack::orgbr_scratchpad_size(cl::sycl::queue &queue, onemkl::generate
gen, std::int64_t m, std::int64_t n,
std::int64_t k, std::int64_t lda, std::int64_t
&scratchpad_size)
```

## Input Parameters

**queue** Device queue where calculations by [orgbr](#) function will be performed.

**gen** Must be `generate::q` or `generate::p`.

If `gen = generate::q`, the routine generates the matrix  $Q$ .

If `gen = generate::p`, the routine generates the matrix  $P^T$ .

**m** The number of rows in the matrix  $Q$  or  $P^T$  to be returned ( $0 \leq m$ ).

If `gen = generate::q, m \geq n \geq \min(m, k)`.

If `gen = generate::p, n \geq m \geq \min(n, k)`.

**n** The number of rows in the matrix  $Q$  or  $P^T$  to be returned ( $0 \leq n$ ). See **m** for constraints.

**k** If `gen = generate::q`, the number of columns in the original  $m$ -by- $k$  matrix returned by [gebrd](#).

If `gen = generate::p`, the number of rows in the original  $k$ -by- $n$  matrix returned by [gebrd](#).

**lda** The leading dimension of  $a$ .

## Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by `get_info()` method of exception object.

## Return Value

The number of elements of type  $T$  the scratchpad memory to be passed to [orgbr](#) function should be able to hold.

**Parent topic:** [LAPACK Singular Value and Eigenvalue Problem Routines](#)

### 13.2.1.3.2.31 orgtr

Generates the real orthogonal matrix  $Q$  determined by [sytrd](#).

`orgtr` supports the following precisions.

<b>T</b>
<code>float</code>
<code>double</code>

## Description

The routine explicitly generates the  $n$ -by- $n$  orthogonal matrix  $Q$  formed by [sytrd](#) when reducing a real symmetric matrix  $A$  to tridiagonal form:  $A = Q \cdot T \cdot Q^T$ . Use this routine after a call to [sytrd](#).

### 13.2.1.3.2.32 `orgtr` (BUFFER Version)

#### Syntax

```
void onemkl::lapack::orgtr(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n,
                           cl::sycl::buffer<T, 1> &a, std::int64_t lda, cl::sycl::buffer<T, 1> &tau,
                           cl::sycl::buffer<T, 1> &scratchpad, std::int64_t scratchpad_size)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Must be `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to `sytrd`.

**n** The order of the matrix  $Q$  ( $0 \leq n$ ).

**a** The buffer  $a$  as returned by `sytrd`. The second dimension of  $a$  must be at least  $\max(1, n)$ .

**lda** The leading dimension of  $a$  ( $n \leq lda$ ).

**tau** The buffer  $\tau$  as returned by `sytrd`. The dimension of  $\tau$  must be at least  $\max(1, n-1)$ .

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type  $T$ . Size should not be less than the value returned by `orgtr_scratchpad_size` function.

#### Output Parameters

**a** Overwritten by the orthogonal matrix  $Q$ .

**scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

#### Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info == -i`, the  $i$ -th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

### 13.2.1.3.2.33 `orgtr` (USM Version)

#### Syntax

```
cl::sycl::event onemkl::lapack::orgtr(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n,
                                       T *a, std::int64_t lda, T *tau, T *scratchpad, std::int64_t scratchpad_size,
                                       const cl::sycl::vector_class<cl::sycl::event> &events = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Must be `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to `sytrd`.

**n** The order of the matrix  $Q$  ( $0 \leq n$ ).

**a** The pointer to  $a$  as returned by `sytrd`. The second dimension of  $a$  must be at least  $\max(1, n)$ .

**lda** The leading dimension of  $a$  ( $n \leq lda$ ).

**tau** The pointer to  $\tau$  as returned by `sytrd`. The dimension of  $\tau$  must be at least  $\max(1, n-1)$ .

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type  $T$ . Size should not be less than the value returned by `orgtr_scratchpad_size` function.

**events** List of events to wait for before starting computation. Defaults to empty list.

## Output Parameters

**a** Overwritten by the orthogonal matrix  $Q$ .

**scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info=-i`, the  $i$ -th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *LAPACK Singular Value and Eigenvalue Problem Routines*

### 13.2.1.3.2.34 `orgtr_scratchpad_size`

Computes size of scratchpad memory required for `orgtr` function.

`orgtr_scratchpad_size` supports the following precisions.

$T$
<code>float</code>
<code>double</code>

## Description

Computes the number of elements of type T the scratchpad memory to be passed to [orgtr](#) function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.2.35 orgtr\_scratchpad\_size

#### Syntax

```
template<typename T>
std::int64_t onemkl::lapack::orgtr_scratchpad_size(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, std::int64_t lda)
```

#### Input Parameters

**queue** Device queue where calculations by [orgtr](#) function will be performed.

**upper\_lower** Must be `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to [sytrd](#).

**n** The order of the matrix Q ( $0 \leq n$ ).

**lda** The leading dimension of a ( $n \leq lda$ ).

#### Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by `get_info()` method of exception object.

#### Return Value

The number of elements of type T the scratchpad memory to be passed to [orgtr](#) function should be able to hold.

**Parent topic:** [LAPACK Singular Value and Eigenvalue Problem Routines](#)

### 13.2.1.3.2.36 ormtr

Multiplies a real matrix by the real orthogonal matrix Q determined by [sytrd](#).

`ormtr` supports the following precisions.

T
float
double

## Description

The routine multiplies a real matrix  $C$  by  $Q$  or  $Q^T$ , where  $Q$  is the orthogonal matrix  $Q$  formed by [sytrd](#) when reducing a real symmetric matrix  $A$  to tridiagonal form:  $A = Q \cdot T \cdot Q^T$ . Use this routine after a call to [sytrd](#).

Depending on the parameters `left_right` and `trans`, the routine can form one of the matrix products  $Q \cdot C$ ,  $Q^T \cdot C$ ,  $C \cdot Q$ , or  $C \cdot Q^T$  (overwriting the result on  $C$ ).

### 13.2.1.3.2.37 ormtr (BUFFER Version)

#### Syntax

```
void onemkl::lapack::ormtr(cl::sycl::queue &queue, onemkl::side left_right, onemkl::uplo upper_lower, onemkl::transpose trans, std::int64_t m, std::int64_t n, cl::sycl::buffer<T, 1> &a, std::int64_t lda, cl::sycl::buffer<T, 1> &tau, cl::sycl::buffer<T, 1> &c, std::int64_t ldc, cl::sycl::buffer<T, 1> &scratchpad, std::int64_t scratchpad_size)
```

#### Input Parameters

In the descriptions below,  $r$  denotes the order of  $Q$ :

$r = m$	if <code>left_right</code> = <code>side::left</code>
$r = n$	if <code>left_right</code> = <code>side::right</code>

**queue** The queue where the routine should be executed.

**left\_right** Must be either `side::left` or `side::right`.

If `left_right` = `side::left`,  $Q$  or  $Q^T$  is applied to  $C$  from the left.

If `left_right` = `side::right`,  $Q$  or  $Q^T$  is applied to  $C$  from the right.

**upper\_lower** Must be either `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to [sytrd](#).

**trans** Must be either `transpose::nontrans` or `transpose::trans`.

If `trans` = `transpose::nontrans`, the routine multiplies  $C$  by  $Q$ .

If `trans` = `transpose::trans`, the routine multiplies  $C$  by  $Q^T$ .

**m** The number of rows in the matrix  $C$  ( $m \geq 0$ ).

**n** The number of columns in the matrix  $C$  ( $n \geq 0$ ).

**a** The buffer  $a$  as returned by [sytrd](#).

**lda** The leading dimension of  $a$  ( $\max(1, r) \leq \text{lda}$ ).

**tau** The buffer  $\tau$  as returned by [sytrd](#). The dimension of  $\tau$  must be at least  $\max(1, r-1)$ .

**c** The buffer  $c$  contains the matrix  $C$ . The second dimension of  $c$  must be at least  $\max(1, n)$ .

**ldc** The leading dimension of  $c$  ( $\max(1, n) \leq \text{ldc}$ ).

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type  $T$ . Size should not be less than the value returned by [ormtr\\_scratchpad\\_size](#) function.

## Output Parameters

**c** Overwritten by the product  $Q \times C$ ,  $QT \times C$ ,  $C \times Q$ , or  $C \times QT$  (as specified by `left_right` and `trans`).

**scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info == -i`, the `i`-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

### 13.2.1.3.2.38 ormtr (USM Version)

#### Syntax

```
cl::sycl::event onemkl::lapack::ormtr(cl::sycl::queue &queue, onemkl::side left_right, onemkl::uplo upper_lower, onemkl::transpose trans, std::int64_t m, std::int64_t n, T *a, std::int64_t lda, T *tau, T *c, std::int64_t ldc, T *scratchpad, std::int64_t scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
```

## Input Parameters

In the descriptions below, `r` denotes the order of  $Q$ :

$r = m$	if <code>left_right = side::left</code>
$r = n$	if <code>left_right = side::right</code>

**queue** The queue where the routine should be executed.

**left\_right** Must be either `side::left` or `side::right`.

If `left_right = side::left`,  $Q$  or  $Q^T$  is applied to  $C$  from the left.

If `left_right = side::right`,  $Q$  or  $Q^T$  is applied to  $C$  from the right.

**upper\_lower** Must be either `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to `sytrd`.

**trans** Must be either `transpose::nontrans` or `transpose::trans`.

If `trans = transpose::nontrans`, the routine multiplies  $C$  by  $Q$ .

If `trans = transpose::trans`, the routine multiplies  $C$  by  $Q^T$ .

**m** The number of rows in the matrix  $C$  ( $m \geq 0$ ).

**n** The number of columns in the matrix  $C$  ( $n \geq 0$ ).

**a** The pointer to `a` as returned by `sytrd`.

**lda** The leading dimension of `a` ( $\max(1, r) \leq lda$ ).

**tau** The buffer `tau` as returned by `sytrd`. The dimension of `tau` must be at least `max(1, r-1)`.

**c** The pointer to memory containing the matrix `C`. The second dimension of `c` must be at least `max(1, n)`.

**ldc** The leading dimension of `c` (`max(1, n) ≤ ldc`).

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by `ormtr_scratchpad_size` function.

**events** List of events to wait for before starting computation. Defaults to empty list.

## Output Parameters

**c** Overwritten by the product `Q*C`, `QT*C`, `C*Q`, or `C*QT` (as specified by `left_right` and `trans`).

**scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info==i`, the `i`-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *LAPACK Singular Value and Eigenvalue Problem Routines*

### 13.2.1.3.2.39 `ormtr_scratchpad_size`

Computes size of scratchpad memory required for `ormtr` function.

`ormtr_scratchpad_size` supports the following precisions.

<code>T</code>
<code>float</code>
<code>double</code>

## Description

Computes the number of elements of type `T` the scratchpad memory to be passed to `ormtr` function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.2.40 `ormtr_scratchpad_size`

#### Syntax

```
template<typename T>
std::int64_t onemkl::lapack::ormtr_scratchpad_size(cl::sycl::queue &queue, onemkl::side
left_right, onemkl::uplo upper_lower,
onemkl::transpose trans, std::int64_t m,
std::int64_t n, std::int64_t lda, std::int64_t
ldc)
```

#### Input Parameters

In the descriptions below,  $r$  denotes the order of  $Q$ :

$r = m$	if <code>left_right</code> = <code>side::left</code>
$r = n$	if <code>left_right</code> = <code>side::right</code>

**queue** Device queue where calculations by `ormtr` function will be performed.

**left\_right** Must be either `side::left` or `side::right`.

If `left_right` = `side::left`,  $Q$  or  $Q^T$  is applied to  $C$  from the left.

If `left_right` = `side::right`,  $Q$  or  $Q^T$  is applied to  $C$  from the right.

**upper\_lower** Must be either `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to `sytrd`.

**trans** Must be either `transpose::nontrans` or `transpose::trans`.

If `trans` = `transpose::nontrans`, the routine multiplies  $C$  by  $Q$ .

If `trans` = `transpose::trans`, the routine multiplies  $C$  by  $Q^T$ .

**m** The number of rows in the matrix  $C$  ( $m \geq 0$ ).

**n** The number of rows in the matrix  $C$  ( $n \geq 0$ ).

**lda** The leading dimension of  $A$  ( $\max(1, r) \leq lda$ ).

**ldc** The leading dimension of  $C$  ( $\max(1, n) \leq ldc$ ).

#### Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by `get_info()` method of exception object.

## Return Value

The number of elements of type T the scratchpad memory to be passed to *ormtr* function should be able to hold.

**Parent topic:** *LAPACK Singular Value and Eigenvalue Problem Routines*

### 13.2.1.3.2.41 syevd

Computes all eigenvalues and, optionally, all eigenvectors of a real symmetric matrix using divide and conquer algorithm.

*syevd* supports the following precisions.

T
float
double

## Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric matrix A. In other words, it can compute the spectral factorization of A as:  $A = Z \lambda Z^T$ .

Here  $\Lambda$  is a diagonal matrix whose diagonal elements are the eigenvalues  $\lambda_i$ , and Z is the orthogonal matrix whose columns are the eigenvectors  $z_i$ . Thus,

$$A \cdot z_i = \lambda_i \cdot z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

### 13.2.1.3.2.42 syevd (BUFFER Version)

## Syntax

```
void onemkl::lapack::syevd(cl::sycl::queue &queue, jobz jobz, onemkl::uplo upper_lower, std::int64_t n, cl::sycl::buffer<T, 1> &a, std::int64_t lda, cl::sycl::buffer<T, 1> &w, cl::sycl::buffer<T, 1> &scratchpad, std::int64_t scratchpad_size)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**jobz** Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

**upper\_lower** Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = job::upper`, a stores the upper triangular part of A.

If `upper_lower = job::lower`, a stores the lower triangular part of A.

**n** The order of the matrix A ( $0 \leq n$ ).

**a** The buffer `a`, size `(lda, *)`. The buffer `a` contains the matrix `A`. The second dimension of `a` must be at least `max(1, n)`.

**lda** The leading dimension of `a`. Must be at least `max(1, n)`.

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by `syevd_scratchpad_size` function.

## Output Parameters

**a** If `jobz = job::vec`, then on exit this buffer is overwritten by the orthogonal matrix `Z` which contains the eigenvectors of `A`.

**w** Buffer, size at least `n`. Contains the eigenvalues of the matrix `A` in ascending order.

**scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info == -i`, the `i`-th parameter had an illegal value.

If `info == i`, and `jobz = onemkl::job::novec`, then the algorithm failed to converge; `i` indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero.

If `info == i`, and `jobz = onemkl::job::vec`, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns `info / (n+1)` through `mod(info, n+1)`.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## 13.2.1.3.2.43 syevd (USM Version)

### Syntax

```
cl::sycl::event onemkl::lapack::syevd(cl::sycl::queue &queue, jobz jobz, onemkl::uplo upper_lower, std::int64_t n, T *a, std::int64_t lda, T *w, T *scratchpad, std::int64_t scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**jobz** Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

**upper\_lower** Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = job::upper`, `a` stores the upper triangular part of `A`.

If `upper_lower = job::lower`, `a` stores the lower triangular part of `A`.

**n** The order of the matrix A ( $0 \leq n$ ).

**a** Pointer to array containing A, size (lda, \*). The second dimension of a must be at least  $\max(1, n)$ .

**lda** The leading dimension of a. Must be at least  $\max(1, n)$ .

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by [syevd\\_scratchpad\\_size](#) function.

**events** List of events to wait for before starting computation. Defaults to empty list.

## Output Parameters

**a** If `jobz = job::vec`, then on exit this array is overwritten by the orthogonal matrix Z which contains the eigenvectors of A.

**w** Pointer to array of size at least n. Contains the eigenvalues of the matrix A in ascending order.

**scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info == -i`, the i-th parameter had an illegal value.

If `info == i`, and `jobz = onemkl::job::novec`, then the algorithm failed to converge; `i` indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero.

If `info == i`, and `jobz = onemkl::job::vec`, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns `info / (n+1)` through `mod(info, n+1)`.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** [LAPACK Singular Value and Eigenvalue Problem Routines](#)

### 13.2.1.3.2.44 syevd\_scratchpad\_size

Computes size of scratchpad memory required for [syevd](#) function.

`syevd_scratchpad_size` supports the following precisions.

T
float
double

## Description

Computes the number of elements of type T the scratchpad memory to be passed to `syevd` function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.2.45 `syevd_scratchpad_size`

#### Syntax

```
template<typename T>
std::int64_t onemkl::lapack::syevd_scratchpad_size(cl::sycl::queue &queue, onemkl::job jobz,
                                                    onemkl::uplo upper_lower, std::int64_t n,
                                                    std::int64_t lda)
```

#### Input Parameters

**queue** Device queue where calculations by `syevd` function will be performed.

**jobz** Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

**upper\_lower** Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = job::upper`, a stores the upper triangular part of A.

If `upper_lower = job::lower`, a stores the lower triangular part of A.

**n** The order of the matrix A ( $0 \leq n$ ).

**lda** The leading dimension of a. Currently lda is not referenced in this function.

#### Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by `get_info()` method of exception object.

#### Return Value

The number of elements of type T the scratchpad memory to be passed to `syevd` function should be able to hold.

**Parent topic:** *LAPACK Singular Value and Eigenvalue Problem Routines*

### 13.2.1.3.2.46 sygvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem using a divide and conquer method.

sygvd supports the following precisions.

T
float
double

#### Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$A \times x = \lambda \times B \times x, A \times B \times x = \lambda \times x, \text{ or } B \times A \times x = \lambda \times x.$$

Here A and B are assumed to be symmetric and B is also positive definite.

It uses a divide and conquer algorithm.

### 13.2.1.3.2.47 sygvd (BUFFER Version)

#### Syntax

```
void onemkl::lapack::sygvd(cl::sycl::queue &queue, std::int64_t itype, onemkl::job jobz, onemkl::uplo upper_lower, std::int64_t n, cl::sycl::buffer<T, 1> &a, std::int64_t lda, cl::sycl::buffer<T, 1> &b, std::int64_t ldb, cl::sycl::buffer<T, 1> &w, cl::sycl::buffer<T, 1> &scratchpad, std::int64_t scratchpad_size)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**itype** Must be 1 or 2 or 3. Specifies the problem type to be solved:

- if itype= 1, the problem type is  $A \times x = \lambda \times B \times x$ ;
- if itype= 2, the problem type is  $A \times B \times x = \lambda \times x$ ;
- if itype= 3, the problem type is  $B \times A \times x = \lambda \times x$ .

**jobz** Must be `job::novec` or `job::vec`.

If `jobz = job::novec`, then only eigenvalues are computed.

If `jobz = job::vec`, then eigenvalues and eigenvectors are computed.

**upper\_lower** Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = job::upper`, a and b store the upper triangular part of A and B.

If `upper_lower = job::lower`, a and b stores the lower triangular part of A and B.

**n** The order of the matrices A and B ( $0 \leq n$ ).

**a** Buffer, size a(`lda, *`) contains the upper or lower triangle of the symmetric matrix A, as specified by `upper_lower`. The second dimension of a must be at least  $\max(1, n)$ .

**lда** The leading dimension of a; at least  $\max(1, n)$ .

**b** Buffer, size  $b(\text{ldb}, *)$  contains the upper or lower triangle of the symmetric matrix B, as specified by `upper_lower`. The second dimension of b must be at least  $\max(1, n)$ .

**ldb** The leading dimension of b; at least  $\max(1, n)$ .

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by `sygvd_scratchpad_size` function.

## Output Parameters

**a** On exit, if `jobz = job::vec`, then if `info = 0`, a contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows:

if `itype= 1 or 2`,  $Z^T * B * Z = I$ ;

if `itype= 3`,  $Z^T * \text{inv}(B) * Z = I$ ;

If `jobz = job::novec`, then on exit the upper triangle (if `upper_lower = uplo::upper`) or the lower triangle (if `upper_lower = uplo::lower`) of A, including the diagonal, is destroyed.

**b** On exit, if `info \leq n`, the part of b containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization  $B = U^T * U$  or  $B = L * L^T$ .

**w** Buffer, size at least n. If `info = 0`, contains the eigenvalues of the matrix A in ascending order.

**scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info=-i`, the i-th parameter had an illegal value.

For `info \leq n`:

If `info=i`, and `jobz = onemkl::job::novec`, then the algorithm failed to converge; i indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero.

If `info=i`, and `jobz = onemkl::job::vec`, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns  $\text{info}/(n+1)$  through  $\text{mod}(\text{info}, n+1)$ .

For `info > n`:

If `info=n+i`, for  $1 \leq i \leq n$ , then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

### 13.2.1.3.2.48 sygvd (USM Version)

#### Syntax

```
cl::sycl::event onemkl::lapack::sygvd(cl::sycl::queue &queue, std::int64_t itype, onemkl::job jobz,
onemkl::uplo upper_lower, std::int64_t n, T *a, std::int64_t lda, T *b, std::int64_t ldb, T *w, T *scratchpad, std::int64_t scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**itype** Must be 1 or 2 or 3. Specifies the problem type to be solved:

- if  $\text{itype} = 1$ , the problem type is  $A \times x = \lambda B \times x$ ;
- if  $\text{itype} = 2$ , the problem type is  $A \times B \times x = \lambda A \times x$ ;
- if  $\text{itype} = 3$ , the problem type is  $B \times A \times x = \lambda B \times x$ .

**jobz** Must be `job::novec` or `job::vec`.

If  $\text{jobz} = \text{job::novec}$ , then only eigenvalues are computed.

If  $\text{jobz} = \text{job::vec}$ , then eigenvalues and eigenvectors are computed.

**upper\_lower** Must be `uplo::upper` or `uplo::lower`.

If  $\text{upper\_lower} = \text{job::upper}$ ,  $a$  and  $b$  store the upper triangular part of  $A$  and  $B$ .

If  $\text{upper\_lower} = \text{job::lower}$ ,  $a$  and  $b$  stores the lower triangular part of  $A$  and  $B$ .

**n** The order of the matrices  $A$  and  $B$  ( $0 \leq n$ ).

**a** Pointer to array of size  $a(\text{lda}, *)$  containing the upper or lower triangle of the symmetric matrix  $A$ , as specified by `upper_lower`. The second dimension of  $a$  must be at least  $\max(1, n)$ .

**lda** The leading dimension of  $a$ ; at least  $\max(1, n)$ .

**b** Pointer to array of size  $b(\text{ldb}, *)$  contains the upper or lower triangle of the symmetric matrix  $B$ , as specified by `upper_lower`. The second dimension of  $b$  must be at least  $\max(1, n)$ .

**ldb** The leading dimension of  $b$ ; at least  $\max(1, n)$ .

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type  $T$ . Size should not be less than the value returned by `sygvd_scratchpad_size` function.

**events** List of events to wait for before starting computation. Defaults to empty list.

## Output Parameters

- a** On exit, if `jobz = job::vec`, then if `info = 0`, `a` contains the matrix `Z` of eigenvectors. The eigenvectors are normalized as follows:

```
if itype= 1 or 2,  $Z^T \cdot B \cdot Z = I$ ;  
if itype= 3,  $Z^T \cdot \text{inv}(B) \cdot Z = I$ ;
```

If `jobz = job::novec`, then on exit the upper triangle (if `upper_lower = uplo::upper`) or the lower triangle (if `upper_lower = uplo::lower`) of `A`, including the diagonal, is destroyed.

- b** On exit, if `info ≤ n`, the part of `b` containing the matrix is overwritten by the triangular factor `U` or `L` from the Cholesky factorization  $B = U^T \cdot U$  or  $B = L \cdot L^T$ .

- w** Pointer to array of size at least `n`. If `info = 0`, contains the eigenvalues of the matrix `A` in ascending order.

**scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info=-i`, the `i`-th parameter had an illegal value.

For `info≤n`:

If `info=i`, and `jobz = onemkl::job::novec`, then the algorithm failed to converge; `i` indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero.

If `info=i`, and `jobz = onemkl::job::vec`, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns `info/(n+1)` through `mod(info, n+1)`.

For `info>n`:

If `info=n+i`, for  $1 \leq i \leq n$ , then the leading minor of order `i` of `B` is not positive-definite. The factorization of `B` could not be completed and no eigenvalues or eigenvectors were computed.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *LAPACK Singular Value and Eigenvalue Problem Routines*

### 13.2.1.3.2.49 `sygvd_scratchpad_size`

Computes size of scratchpad memory required for `sygvd` function.

`sygvd_scratchpad_size` supports the following precisions.

T
float
double

#### Description

Computes the number of elements of type T the scratchpad memory to be passed to `sygvd` function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.2.50 `sygvd_scratchpad_size`

#### Syntax

```
template<typename T>
std::int64_t onemkl::lapack::sygvd_scratchpad_size(cl::sycl::queue &queue, std::int64_t
    itype, onemkl::job jobz, onemkl::uplo
    upper_lower, std::int64_t n, std::int64_t
    lda, std::int64_t ldb)
```

#### Input Parameters

**queue** Device queue where calculations by `sygvd` function will be performed.

**itype** Must be 1 or 2 or 3. Specifies the problem type to be solved:

if `itype` = 1, the problem type is  $A \times x = \lambda B \times x$ ;  
 if `itype` = 2, the problem type is  $A \times B \times x = \lambda A \times x$ ;  
 if `itype` = 3, the problem type is  $B \times A \times x = \lambda B \times x$ .

**jobz** Must be `job::novec` or `job::vec`.

If `jobz` = `job::novec`, then only eigenvalues are computed.

If `jobz` = `job::vec`, then eigenvalues and eigenvectors are computed.

**upper\_lower** Must be `uplo::upper` or `uplo::lower`.

If `upper_lower` = `job::upper`, a and b store the upper triangular part of A and B.

If `upper_lower` = `job::lower`, a and b stores the lower triangular part of A and B.

**n** The order of the matrices A and B ( $0 \leq n$ ).

**lda** The leading dimension of a.

**ldb** The leading dimension of b.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by `get_info()` method of exception object.

## Return Value

The number of elements of type T the scratchpad memory to be passed to `sygvd` function should be able to hold.

**Parent topic:** *LAPACK Singular Value and Eigenvalue Problem Routines*

### 13.2.1.3.2.51 sytrd

Reduces a real symmetric matrix to tridiagonal form.

`sytrd` supports the following precisions.

T
float
double

## Description

The routine reduces a real symmetric matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation:  $A = Q*T*Q^T$ . The orthogonal matrix Q is not formed explicitly but is represented as a product of n-1 elementary reflectors. Routines are provided for working with Q in this representation .

### 13.2.1.3.2.52 sytrd (BUFFER Version)

## Syntax

```
void onemkl::lapack::sytrd(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n,
                           cl::sycl::buffer<T, 1> &a, std::int64_t lda, cl::sycl::buffer<T, 1> &d,
                           cl::sycl::buffer<T, 1> &e, cl::sycl::buffer<T, 1> &tau, cl::sycl::buffer<T,
                           1> &scratchpad, std::int64_t scratchpad_size)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = uplo::upper`, a stores the upper triangular part of A.

If `upper_lower = uplo::lower`, a stores the lower triangular part of A.

**n** The order of the matrices A ( $0 \leq n$ ).

**a** The buffer a, size `(lda, *)`. Contains the upper or lower triangle of the symmetric matrix A, as specified by `upper_lower`.

The second dimension of a must be at least `max(1, n)`.

**lda** The leading dimension of a; at least `max(1, n)`.

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by [sytrd\\_scratchpad\\_size](#) function.

## Output Parameters

**a** On exit,

if `upper_lower = uplo::upper`, the diagonal and first superdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements above the first superdiagonal, with the buffer tau, represent the orthogonal matrix Q as a product of elementary reflectors;

if `upper_lower = uplo::lower`, the diagonal and first subdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements below the first subdiagonal, with the buffer tau, represent the orthogonal matrix Q as a product of elementary reflectors.

**d** Buffer containing the diagonal elements of the matrix T. The dimension of d must be at least `max(1, n)`.

**e** Buffer containing the off diagonal elements of the matrix T. The dimension of e must be at least `max(1, n-1)`.

**tau** Buffer, size at least `max(1, n)`. Stores  $(n-1)$  scalars that define elementary reflectors in decomposition of the unitary matrix Q in a product of  $n-1$  elementary reflectors. tau(n) is used as workspace.

**scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info=-i`, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

### 13.2.1.3.2.53 sytrd (USM Version)

#### Syntax

```
cl::sycl::event onemkl::lapack::sytrd(cl::sycl::queue &queue, onemkl::uplo upper_lower,
                                         std::int64_t n, T *a, std::int64_t lda, T *d, T *e, T
                                         *tau, T *scratchpad, std::int64_t scratchpad_size, const
                                         cl::sycl::vector_class<cl::sycl::event> &events = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = uplo::upper`, a stores the upper triangular part of A.

If `upper_lower = uplo::lower`, a stores the lower triangular part of A.

**n** The order of the matrices A ( $0 \leq n$ ).

**a** The pointer to matrix A, size  $(\text{lda}, *)$ . Contains the upper or lower triangle of the symmetric matrix A, as specified by `upper_lower`. The second dimension of a must be at least  $\max(1, n)$ .

**lda** The leading dimension of a; at least  $\max(1, n)$ .

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by `sytrd_scratchpad_size` function.

**events** List of events to wait for before starting computation. Defaults to empty list.

## Output Parameters

**a** On exit,

if `upper_lower = uplo::upper`, the diagonal and first superdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements above the first superdiagonal, with the array tau, represent the orthogonal matrix Q as a product of elementary reflectors;

if `upper_lower = uplo::lower`, the diagonal and first subdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T, and the elements below the first subdiagonal, with the array tau, represent the orthogonal matrix Q as a product of elementary reflectors.

**d** Pointer to diagonal elements of the matrix T. The dimension of d must be at least  $\max(1, n)$ .

**e** Pointer to off diagonal elements of the matrix T. The dimension of e must be at least  $\max(1, n-1)$ .

**tau** Pointer to array of size at least  $\max(1, n)$ . Stores  $(n-1)$  scalars that define elementary reflectors in decomposition of the unitary matrix Q in a product of  $n-1$  elementary reflectors. tau(n) is used as workspace.

**scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info=-i`, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *LAPACK Singular Value and Eigenvalue Problem Routines*

### 13.2.1.3.2.54 `sytrd_scratchpad_size`

Computes size of scratchpad memory required for `sytrd` function.

`sytrd_scratchpad_size` supports the following precisions.

T
float
double

#### Description

Computes the number of elements of type T the scratchpad memory to be passed to `sytrd` function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.2.55 `sytrd_scratchpad_size`

#### Syntax

```
template<typename T>
std::int64_t onemkl::lapack::sytrd_scratchpad_size(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, std::int64_t lda)
```

#### Input Parameters

**queue** Device queue where calculations by `sytrd` function will be performed.

**upper\_lower** Must be `uplo::upper` or `uplo::lower`.

If `upper_lower = uplo::upper`, a stores the upper triangular part of A.

If `upper_lower = uplo::lower`, a stores the lower triangular part of A.

**n** The order of the matrices A ( $0 \leq n$ ).

**lda** The leading dimension of a.

#### Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by `get_info()` method of exception object.

## Return Value

The number of elements of type T the scratchpad memory to be passed to `sytrd` function should be able to hold.

**Parent topic:** *LAPACK Singular Value and Eigenvalue Problem Routines*

### 13.2.1.3.2.56 `ungbr`

Generates the complex unitary matrix Q or P<sup>T</sup> determined by `gebrd`.

`ungbr` supports the following precisions.

T
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The routine generates the whole or part of the unitary matrices Q and P<sup>H</sup> formed by the routines `gebrd`. All valid combinations of arguments are described in *Input Parameters*; in most cases you need the following:

To compute the whole m-by-m matrix Q, use:

```
ungbr(queue, generate::q, m, m, n, a, ...)
```

(note that the buffer `a` must have at least m columns).

To form the n leading columns of Q if m > n, use:

```
ungbr(queue, generate::q, m, n, n, a, ...)
```

To compute the whole n-by-n matrix P<sup>T</sup>, use:

```
ungbr(queue, generate::p, n, n, m, a, ...)
```

(note that the array `a` must have at least n rows).

To form the m leading rows of P<sup>T</sup> if m < n, use:

```
ungbr(queue, generate::p, m, n, m, a, ...)
```

### 13.2.1.3.2.57 `ungbr` (BUFFER Version)

## Syntax

```
void onemkl::lapack::ungbr(cl::sycl::queue &queue, onemkl::generate gen, std::int64_t m,
                           std::int64_t n, std::int64_t k, cl::sycl::buffer<T, 1> &a, std::int64_t lda,
                           cl::sycl::buffer<T, 1> &tau, cl::sycl::buffer<T, 1> &scratchpad,
                           std::int64_t scratchpad_size)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**gen** Must be `generate::q` or `generate::p`.

If `gen = generate::q`, the routine generates the matrix  $Q$ .

If `gen = generate::p`, the routine generates the matrix  $P^T$ .

**m** The number of rows in the matrix  $Q$  or  $P^T$  to be returned ( $0 \leq m$ ).

If `gen = generate::q, m \geq n \geq \min(m, k)`.

If `gen = generate::p, n \geq m \geq \min(n, k)`.

**n** The number of columns in the matrix  $Q$  or  $P^T$  to be returned ( $0 \leq n$ ). See **m** for constraints.

**k** If `gen = generate::q`, the number of columns in the original  $m$ -by- $k$  matrix returned by `gebrd`.

If `gen = generate::p`, the number of rows in the original  $k$ -by- $n$  matrix returned by `gebrd`.

**a** The buffer `a` as returned by `gebrd`.

**lda** The leading dimension of `a`.

**tau** For `gen= generate::q`, the array `tauq` as returned by `gebrd`. For `gen= generate::p`, the array `taup` as returned by `gebrd`.

The dimension of `tau` must be at least `max(1, min(m, k))` for `gen=generate::q`, or `max(1, min(m, k))` for `gen= generate::p`.

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by `ungbr_scratchpad_size` function.

## Output Parameters

**a** Overwritten by `n` leading columns of the  $m$ -by- $m$  unitary matrix  $Q$  or  $P^T$ , (or the leading rows or columns thereof) as specified by `gen`, `m`, and `n`.

**scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info=-i`, the `i`-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

### 13.2.1.3.2.58 `ungbr` (USM Version)

#### Syntax

```
cl::sycl::event onemkl::lapack::ungbr(cl::sycl::queue &queue, onemkl::generate gen, std::int64_t m, std::int64_t n, std::int64_t k, T *a, std::int64_t lda, T *tau, T *scratchpad, std::int64_t scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**gen** Must be `generate::q` or `generate::p`.

If `gen = generate::q`, the routine generates the matrix  $Q$ .

If `gen = generate::p`, the routine generates the matrix  $P^T$ .

**m** The number of rows in the matrix  $Q$  or  $P^T$  to be returned ( $0 \leq m$ ).

If `gen = generate::q, m \geq n \geq \min(m, k)`.

If `gen = generate::p, n \geq m \geq \min(n, k)`.

**n** The number of columns in the matrix  $Q$  or  $P^T$  to be returned ( $0 \leq n$ ). See **m** for constraints.

**k** If `gen = generate::q`, the number of columns in the original  $m$ -by- $k$  matrix returned by `gebrd`.

If `gen = generate::p`, the number of rows in the original  $k$ -by- $n$  matrix returned by `gebrd`.

**a** The pointer to `a` as returned by `gebrd`.

**lda** The leading dimension of `a`.

**tau** For `gen= generate::q`, the array `tauq` as returned by `gebrd`. For `gen= generate::p`, the array `taup` as returned by `gebrd`.

The dimension of `tau` must be at least  $\max(1, \min(m, k))$  for `gen=generate::q`, or  $\max(1, \min(m, k))$  for `gen= generate::p`.

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type `T`. Size should not be less than the value returned by `ungbr_scratchpad_size` function.

**events** List of events to wait for before starting computation. Defaults to empty list.

#### Output Parameters

**a** Overwritten by `n` leading columns of the  $m$ -by- $m$  unitary matrix  $Q$  or  $P^T$ , (or the leading rows or columns thereof) as specified by `gen`, `m`, and `n`.

**scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info==i`, the `i`-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *LAPACK Singular Value and Eigenvalue Problem Routines*

### 13.2.1.3.2.59 `ungbr_scratchpad_size`

Computes size of scratchpad memory required for `ungbr` function.

`ungbr_scratchpad_size` supports the following precisions.

T
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

Computes the number of elements of type `T` the scratchpad memory to be passed to `ungbr` function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.2.60 `ungbr_scratchpad_size`

## Syntax

```
template<typename T>
std::int64_t onemkl::lapack::ungbr_scratchpad_size(cl::sycl::queue &queue, onemkl::generate
gen, std::int64_t m, std::int64_t n,
std::int64_t k, std::int64_t lda, std::int64_t
&scratchpad_size)
```

## Input Parameters

**queue** Device queue where calculations by [ungbr](#) function will be performed.

**gen** Must be `generate::q` or `generate::p`.

If `gen = generate::q`, the routine generates the matrix  $Q$ .

If `gen = generate::p`, the routine generates the matrix  $P^T$ .

**m** The number of rows in the matrix  $Q$  or  $P^T$  to be returned ( $0 \leq m$ ).

If `gen = generate::q, m \geq n \geq \min(m, k)`.

If `gen = generate::p, n \geq m \geq \min(n, k)`.

**n** The number of columns in the matrix  $Q$  or  $P^T$  to be returned ( $0 \leq n$ ). See **m** for constraints.

**k** If `gen = generate::q`, the number of columns in the original  $m$ -by- $k$  matrix reduced by [gebrd](#).

If `gen = generate::p`, the number of rows in the original  $k$ -by- $n$  matrix reduced by [gebrd](#).

**lda** The leading dimension of  $a$ .

## Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by `get_info()` method of exception object.

## Return Value

The number of elements of type  $T$  the scratchpad memory to be passed to [ungbr](#) function should be able to hold.

**Parent topic:** *LAPACK Singular Value and Eigenvalue Problem Routines*

### 13.2.1.3.2.61 ungtr

Generates the complex unitary matrix  $Q$  determined by [hetrd](#).

`ungtr` supports the following precisions.

$T$
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The routine explicitly generates the  $n$ -by- $n$  unitary matrix  $Q$  formed by [hetrd](#) when reducing a complex Hermitian matrix  $A$  to tridiagonal form:  $A = Q \cdot T \cdot Q^H$ . Use this routine after a call to [hetrd](#).

### 13.2.1.3.2.62 `ungtr` (BUFFER Version)

#### Syntax

```
void onemkl::lapack::ungtr(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n,
                           cl::sycl::buffer<T, 1> &a, std::int64_t lda, cl::sycl::buffer<T, 1> &tau,
                           cl::sycl::buffer<T, 1> &scratchpad, std::int64_t scratchpad_size)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Must be `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to `hetrd`.

**n** The order of the matrix  $Q$  ( $0 \leq n$ ).

**a** The buffer  $a$  as returned by `hetrd`. The second dimension of  $a$  must be at least  $\max(1, n)$ .

**lda** The leading dimension of  $a$  ( $n \leq lda$ ).

**tau** The buffer  $\tau$  as returned by `hetrd`. The dimension of  $\tau$  must be at least  $\max(1, n-1)$ .

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type  $T$ . Size should not be less than the value returned by `ungtr_scratchpad_size` function.

#### Output Parameters

**a** Overwritten by the unitary matrix  $Q$ .

**scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

#### Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info == -i`, the  $i$ -th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

### 13.2.1.3.2.63 `ungtr` (USM Version)

#### Syntax

```
cl::sycl::event onemkl::lapack::ungtr(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n,
                                         T *a, std::int64_t lda, T *tau, T *scratchpad, std::int64_t scratchpad_size,
                                         const cl::sycl::vector_class<cl::sycl::event> &events = {})
```

## Input Parameters

**queue** The queue where the routine should be executed.

**upper\_lower** Must be `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to `hetrd`.

**n** The order of the matrix  $Q$  ( $0 \leq n$ ).

**a** The pointer to  $a$  as returned by `hetrd`. The second dimension of  $a$  must be at least  $\max(1, n)$ .

**lda** The leading dimension of  $a$  ( $n \leq lda$ ).

**tau** The pointer to  $\tau$  as returned by `hetrd`. The dimension of  $\tau$  must be at least  $\max(1, n-1)$ .

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type  $T$ . Size should not be less than the value returned by `ungtr_scratchpad_size` function.

**events** List of events to wait for before starting computation. Defaults to empty list.

## Output Parameters

**a** Overwritten by the unitary matrix  $Q$ .

**scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info=-i`, the  $i$ -th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** *LAPACK Singular Value and Eigenvalue Problem Routines*

### 13.2.1.3.2.64 `ungtr_scratchpad_size`

Computes size of scratchpad memory required for `ungtr` function.

`ungtr_scratchpad_size` supports the following precisions.

$T$
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

Computes the number of elements of type T the scratchpad memory to be passed to [ungtr](#) function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.2.65 ungtr\_scratchpad\_size

#### Syntax

```
template<typename T>
std::int64_t onemkl::lapack::ungtr_scratchpad_size(cl::sycl::queue &queue, onemkl::uplo upper_lower, std::int64_t n, std::int64_t lda)
```

#### Input Parameters

**queue** Device queue where calculations by [ungtr](#) function will be performed.

**upper\_lower** Must be `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to [hetrd](#).

**n** The order of the matrix Q ( $0 \leq n$ ).

**lda** The leading dimension of a ( $n \leq lda$ ).

#### Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by `get_info()` method of exception object.

#### Return Value

The number of elements of type T the scratchpad memory to be passed to [ungtr](#) function should be able to hold.

**Parent topic:** [LAPACK Singular Value and Eigenvalue Problem Routines](#)

### 13.2.1.3.2.66 unmtr

Multiplies a complex matrix by the complex unitary matrix Q determined by [hetrd](#).

`unmtr` supports the following precisions.

T
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The routine multiplies a complex matrix  $C$  by  $Q$  or  $Q^H$ , where  $Q$  is the unitary matrix  $Q$  formed by [hetrd](#) when reducing a complex Hermitian matrix  $A$  to tridiagonal form:  $A = Q \cdot T \cdot Q^H$ . Use this routine after a call to [hetrd](#).

Depending on the parameters `left_right` and `trans`, the routine can form one of the matrix products  $Q \cdot C$ ,  $Q^H \cdot C$ ,  $C \cdot Q$ , or  $C \cdot Q^H$  (overwriting the result on  $C$ ).

### 13.2.1.3.2.67 `unmtr` (BUFFER Version)

#### Syntax

```
void onemkl::lapack::unmtr(cl::sycl::queue &queue, onemkl::side left_right, onemkl::uplo upper_lower, onemkl::transpose trans, std::int64_t m, std::int64_t n, cl::sycl::buffer<T, 1> &a, std::int64_t lda, cl::sycl::buffer<T, 1> &tau, cl::sycl::buffer<T, 1> &c, std::int64_t ldc, cl::sycl::buffer<T, 1> &scratchpad, std::int64_t scratchpad_size)
```

#### Input Parameters

In the descriptions below,  $r$  denotes the order of  $Q$ :

$r=m$	if <code>left_right</code> = <code>side::left</code>
$r=n$	if <code>left_right</code> = <code>side::right</code>

**queue** The queue where the routine should be executed.

**left\_right** Must be either `side::left` or `side::right`.

If `left_right`=`side::left`,  $Q$  or  $Q^H$  is applied to  $C$  from the left.

If `left_right`=`side::right`,  $Q$  or  $Q^H$  is applied to  $C$  from the right.

**upper\_lower** Must be either `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to [hetrd](#).

**trans** Must be either `transpose::nontrans` or `transpose::conjtrans`.

If `trans`=`transpose::nontrans`, the routine multiplies  $C$  by  $Q$ .

If `trans`=`transpose::conjtrans`, the routine multiplies  $C$  by  $Q^H$ .

**m** The number of rows in the matrix  $C$  ( $m \geq 0$ ).

**n** The number of columns the matrix  $C$  ( $n \geq 0$ ).

**k** The number of elementary reflectors whose product defines the matrix  $Q$  ( $0 \leq k \leq n$ ).

**a** The buffer  $a$  as returned by [hetrd](#).

**lda** The leading dimension of  $a$  ( $\max(1, r) \leq \text{lda}$ ).

**tau** The buffer  $\tau$  as returned by [hetrd](#). The dimension of  $\tau$  must be at least  $\max(1, r-1)$ .

**c** The buffer  $c$  contains the matrix  $C$ . The second dimension of  $c$  must be at least  $\max(1, n)$ .

**ldc** The leading dimension of  $c$  ( $\max(1, n) \leq \text{ldc}$ ).

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type  $T$ . Size should not be less than the value returned by [unmtr\\_scratchpad\\_size](#) function.

## Output Parameters

**c** Overwritten by the product  $Q \times C$ ,  $Q^H \times C$ ,  $C \times Q$ , or  $C \times Q^H$  (as specified by `left_right` and `trans`).

**scratchpad** Buffer holding scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The `info` code of the problem can be obtained by `get_info()` method of exception object:

If `info==i`, the `i`-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

### 13.2.1.3.2.68 unmtr (USM Version)

#### Syntax

```
cl::sycl::event onemkl::lapack::unmtr(cl::sycl::queue &queue, onemkl::side left_right, onemkl::uplo upper_lower, onemkl::transpose trans, std::int64_t m, std::int64_t n, T *a, std::int64_t lda, T *tau, T *c, std::int64_t ldc, T *scratchpad, std::int64_t scratchpad_size, const cl::sycl::vector_class<cl::sycl::event> &events = {})
```

## Input Parameters

In the descriptions below, `r` denotes the order of  $Q$ :

<code>r=m</code>	<code>if left_right = side:::left</code>
<code>r=n</code>	<code>if left_right = side:::right</code>

**queue** The queue where the routine should be executed.

**left\_right** Must be either `side:::left` or `side:::right`.

If `left_right=side:::left`,  $Q$  or  $Q^H$  is applied to  $C$  from the left.

If `left_right=side:::right`,  $Q$  or  $Q^H$  is applied to  $C$  from the right.

**upper\_lower** Must be either `uplo:::upper` or `uplo:::lower`. Uses the same `upper_lower` as supplied to `hetrd`.

**trans** Must be either `transpose:::nontrans` or `transpose:::conjtrans`.

If `trans=transpose:::nontrans`, the routine multiplies  $C$  by  $Q$ .

If `trans=transpose:::conjtrans`, the routine multiplies  $C$  by  $Q^H$ .

**m** The number of rows in the matrix  $C$  ( $m \geq 0$ ).

**n** The number of columns the matrix  $C$  ( $n \geq 0$ ).

**k** The number of elementary reflectors whose product defines the matrix  $Q$  ( $0 \leq k \leq n$ ).

**a** The pointer to `a` as returned by `hetrd`.

**lda** The leading dimension of a ( $\max(1, r) \leq lda$ ).

**tau** The pointer to tau as returned by [hetrd](#). The dimension of tau must be at least  $\max(1, r-1)$ .

**c** The array c contains the matrix C. The second dimension of c must be at least  $\max(1, n)$ .

**ldc** The leading dimension of c ( $\max(1, n) \leq ldc$ ).

**scratchpad\_size** Size of scratchpad memory as a number of floating point elements of type T. Size should not be less than the value returned by [unmtr\\_scratchpad\\_size](#) function.

**events** List of events to wait for before starting computation. Defaults to empty list.

## Output Parameters

**c** Overwritten by the product  $Q * C$ ,  $Q^H * C$ ,  $C * Q$ , or  $C * Q^H$  (as specified by left\_right and trans).

**scratchpad** Pointer to scratchpad memory to be used by routine for storing intermediate results.

## Throws

**onemkl::lapack::exception** Exception is thrown in case of problems happened during calculations. The info code of the problem can be obtained by `get_info()` method of exception object:

If `info=-i`, the i-th parameter had an illegal value.

If `info` equals to value passed as scratchpad size, and `get_detail()` returns non zero, then passed scratchpad is of insufficient size, and required size should not be less than value return by `get_detail()` method of exception object.

## Return Values

Output event to wait on to ensure computation is complete.

**Parent topic:** [LAPACK Singular Value and Eigenvalue Problem Routines](#)

### 13.2.1.3.2.69 unmtr\_scratchpad\_size

Computes size of scratchpad memory required for [unmtr](#) function.

`unmtr_scratchpad_size` supports the following precisions.

T
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

Computes the number of elements of type T the scratchpad memory to be passed to `unmtr` function should be able to hold. Calls to this routine must specify the template parameter explicitly.

### 13.2.1.3.2.70 `unmtr_scratchpad_size`

#### Syntax

```
template<typename T>
std::int64_t onemkl::lapack::unmtr_scratchpad_size(cl::sycl::queue &queue, onemkl::side
left_right, onemkl::uplo upper_lower,
onemkl::transpose trans, std::int64_t m,
std::int64_t n, std::int64_t lda, std::int64_t
ldc)
```

#### Input Parameters

**queue** Device queue where calculations by `unmtr` function will be performed.

**left\_right** Must be either `side::left` or `side::right`.

If `left_right=side::left`,  $Q$  or  $Q^H$  is applied to  $C$  from the left.

If `left_right=side::right`,  $Q$  or  $Q^H$  is applied to  $C$  from the right.

**upper\_lower** Must be either `uplo::upper` or `uplo::lower`. Uses the same `upper_lower` as supplied to `hetrd`.

**trans** Must be either `transpose::nontrans` or `transpose::conjtrans`.

If `trans=transpose::nontrans`, the routine multiplies  $C$  by  $Q$ .

If `trans=transpose::conjtrans`, the routine multiplies  $C$  by  $Q^H$ .

**m** The number of rows in the matrix  $C$  ( $m \geq 0$ ).

**n** The number of columns the matrix  $C$  ( $n \geq 0$ ).

**k** The number of elementary reflectors whose product defines the matrix  $Q$  ( $0 \leq k \leq n$ ).

**lda** The leading dimension of  $a$  ( $\max(1, r) \leq lda$ ).

**ldc** The leading dimension of  $c$  ( $\max(1, n) \leq ldc$ ).

#### Throws

**onemkl::lapack::exception** Exception is thrown in case of incorrect argument value is supplied. Position of wrong argument can be determined by `get_info()` method of exception object.

## Return Value

The number of elements of type T the scratchpad memory to be passed to `unmtr` function should be able to hold.

**Parent topic:** [LAPACK Singular Value and Eigenvalue Problem Routines](#)

### 13.2.1.3.3 LAPACK-like Extensions Routines

oneAPI Math Kernel Library DPC++ provides additional routines to extend the functionality of the LAPACK routines. These include routines to compute many independent factorizations, linear equation solutions, and similar. The following table lists the LAPACK-like Extensions routine groups.

Routines	Description
<code>geqrf_batch</code>	Computes the QR factorizations of a batch of general matrices.
<code>getrf_batch</code>	Computes the LU factorizations of a batch of general matrices.
<code>getri_batch</code>	Computes the inverses of a batch of LU-factored general matrices.
<code>getrs_batch</code>	Solves systems of linear equations with a batch of LU-factored square coefficient matrices, with multiple right-hand sides.
<code>orgqr_batch</code>	Generates the real orthogonal/complex unitary matrix $Q_i$ of the QR factorization formed by <code>geqrf_batch</code> .
<code>potrf_batch</code>	Computes the Cholesky factorization of a batch of symmetric (Hermitian) positive-definite matrices.
<code>potrs_batch</code>	Solves systems of linear equations with a batch of Cholesky-factored symmetric (Hermitian) positive-definite coefficient matrices, with multiple right-hand sides.

#### 13.2.1.3.3.1 `geqrf_batch`

Computes the QR factorizations of a batch of general matrices.

`geqrf_batch` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

#### Description

The routine forms the QR factorizations of a batch of general matrices  $A_1, A_2, \dots, A_{\text{batch\_size}}$ . No pivoting is performed.

The routine does not form the matrices  $Q_i$  explicitly. Instead,  $Q_i$  is represented as a product of  $\min(m_i, n_i)$  elementary reflectors. Routines are provided to work with  $Q_i$  in this representation.

### 13.2.1.3.3.2 `geqrf_batch` (BUFFER Version)

#### Syntax

```
void onemkl::lapack::geqrf_batch(cl::sycl::queue      &queue,           std::vector<std::int64_t>
                                  const &m,      std::vector<std::int64_t>  const &n,
                                  std::vector<cl::sycl::buffer<T, 1>> &a, std::vector<std::int64_t>
                                  const &lda,    std::vector<cl::sycl::buffer<T, 1>> &tau,
                                  std::vector<cl::sycl::buffer<std::int64_t, 1>> &info)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**m** A vector,  $m[i]$  is the number of rows of the batch matrix  $A_i$  ( $0 \leq m[i]$ ).

**n** A vector,  $n[i]$  is the number of columns of the batch matrix  $A_i$  ( $0 \leq n[i]$ ).

**a** A vector of buffers,  $a[i]$  stores the matrix  $A_i$ .  $a[i]$  must be of size at least  $l_{da}[i] * \max(1, n[i])$ .

**lda** A vector,  $l_{da}[i]$  is the leading dimension of  $a[i]$  ( $m[i] \leq l_{da}[i]$ ).

#### Output Parameters

**a** Overwritten by the factorization data as follows:

The elements on and above the diagonal of the buffer  $a[i]$  contain the  $\min(m[i], n[i])$ -by- $n[i]$  upper trapezoidal matrix  $R_i$  ( $R_i$  is upper triangular if  $m[i] \geq n[i]$ ); the elements below the diagonal, with the array  $\tau_{au}[i]$ , present the orthogonal matrix  $Q_i$  as a product of  $\min(m[i], n[i])$  elementary reflectors.

**tau** Vector of buffers, where  $\tau_{au}[i]$  must have size at least  $\max(1, \min(m[i], n[i]))$ . Contains scalars that define elementary reflectors for the matrix  $Q_i$  in its decomposition in a product of elementary reflectors.

**info** Vector of buffers containing error information.

If  $\text{info}[i]=0$ , the execution is successful.

If  $\text{info}[i]=-k$ , the  $k$ -th parameter had an illegal value.

**Parent topic:** *LAPACK-like Extensions Routines*

### 13.2.1.3.3.3 `getrf_batch`

Computes the LU factorizations of a batch of general matrices.

`getrf_batch` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The routine computes the LU factorizations of a batch of general m-by-n matrices  $A_1, A_2, \dots, A_{\text{batch\_size}}$  as

$$A_i = P_i \times L_i \times U_i$$

Where  $P_i$  is a permutation matrix,  $L_i$  is lower triangular with unit diagonal elements (lower trapezoidal if  $m > n$ ) and  $U$  is upper triangular (upper trapezoidal if  $m < n$ ). The routine uses partial pivoting with row interchanges.

### 13.2.1.3.3.4 `getrf_batch` (BUFFER Version)

#### Syntax

```
void onemkl::lapack::getrf_batch(cl::sycl::queue      &queue,           std::vector<std::int64_t>
                                 const  &m,   std::vector<std::int64_t>  const  &n,
                                 std::vector<cl::sycl::buffer<T, 1>> &a, std::vector<std::int64_t>
                                 const  &lda, std::vector<cl::sycl::buffer<std::int64_t, 1>>
                                 &ipiv, std::vector<cl::sycl::buffer<std::int64_t, 1>> &info)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**m** A vector,  $m[i]$  is the number of rows of the batch matrix  $A_i$  ( $0 \leq m[i]$ ).

**n** A vector,  $n[i]$  is the number of columns of the batch matrix  $A_i$  ( $0 \leq n[i]$ ).

**a** A vector of buffers,  $a[i]$  contains the matrix  $A_i$ .  $a[i]$  must be of size at least  $l\text{da}[i] * \max(1, n[i])$ .

**lda** A vector,  $l\text{da}[i]$  is the leading dimension of  $a[i]$  ( $m[i] \leq l\text{da}[i]$ ).

#### Output Parameters

**a**  $a[i]$  is overwritten by  $L_i$  and  $U_i$ . The unit diagonal elements of  $L_i$  are not stored.

**ipiv** A vector of buffers,  $ipiv[i]$  stores the pivot indices. The dimension of  $ipiv[i]$  must be at least  $\min(m[i], n[i])$ .

**info** Vector of buffers containing error information.

If  $info[i]=0$ , the execution is successful.

If  $info[i]=k$ ,  $U_i(k, k)$  is 0. The factorization has been completed, but  $U_i$  is exactly singular. Division by 0 will occur if you use the factor  $U_i$  for solving a system of linear equations.

**Parent topic:** *LAPACK-like Extensions Routines*

### 13.2.1.3.3.5 `getri_batch`

Computes the inverses of a batch of LU-factored matrices determined by `getrf_batch`.

`getri_batch` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

### Description

The routine computes the inverses  $A_i^{-1}$  of a batch of general matrices  $A_1, A_2, \dots, A_{\text{batch\_size}}$ . Before calling this routine, call `getrf_batch` to compute the LU factorization of  $A_1, A_2, \dots, A_{\text{batch\_size}}$ .

### 13.2.1.3.3.6 `getri_batch` (BUFFER Version)

#### Syntax

```
void onemkl::lapack::getri_batch(cl::sycl::queue &queue, std::vector<std::int64_t> const &n,
                                  std::vector<cl::sycl::buffer<T, 1>> &a, std::vector<std::int64_t>
const &lda, std::vector<cl::sycl::buffer<std::int64_t, 1>>
&ipiv, std::vector<cl::sycl::buffer<std::int64_t, 1>> &info)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**n** A vector,  $n[i]$  is order of the matrix  $A_i$  ( $0 \leq n[i]$ ).

**a** A vector of buffers returned by `getrf_batch`.  $a[i]$  must be of size at least  $\text{lda}[i] * \max(1, n[i])$ .

**lda** A vector,  $\text{lda}[i]$  is the leading dimension of  $a[i]$  ( $n[i] \leq \text{lda}[i]$ ).

**ipiv** A vector of buffers returned by `getrf_batch`. The dimension of  $\text{ipiv}[i]$  must be at least  $\max(1, n[i])$ .

#### Output Parameters

**a**  $a[i]$  is overwritten by the  $n[i]$ -by- $n[i]$  inverse matrix  $A_i^{-1}$ .

**info** Vector of buffers containing error information.

If  $\text{info}[i]=0$ , the execution is successful.

If  $\text{info}[i]=-k$ , the  $k$ -th parameter had an illegal value.

**Parent topic:** *LAPACK-like Extensions Routines*

### 13.2.1.3.3.7 `getrs_batch`

Solves a system of linear equations with a batch of LU-factored square coefficient matrices, with multiple right-hand sides.

`getrs_batch` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

#### Description

The routine solves for  $X_i$  the following systems of linear equations for a batch of general square matrices  $A_1, A_2, \dots, A_{\text{sub}`batch\_size`}$ :

$A_i * X_i = B_i$  If `trans[i] = onemkl::transpose::notrans`

$A_i^T * X_i = B_i$  If `trans[i] = onemkl::transpose::trans`

$A_i^H * X_i = B_i$  If `trans[i] = onemkl::transpose::conjtrans`

Before calling this routine you must call `getrf_batch` to compute the LU factorization of  $A_1, A_2, \dots, A_{\text{sub}`batch\_size`}$ .

### 13.2.1.3.3.8 `getrs_batch` (BUFFER Version)

#### Syntax

```
void onemkl::lapack::getrs_batch(cl::sycl::queue &queue, std::vector<onemkl::transpose>
                                    const &trans, std::vector<std::int64_t> const
                                    &n, std::vector<std::int64_t> const &nrhs,
                                    std::vector<cl::sycl::buffer<T, 1>> &a, std::vector<std::int64_t>
                                    const &lpa, std::vector<cl::sycl::buffer<std::int64_t,
                                    1>> &ipiv, std::vector<cl::sycl::buffer<T, 1>>
                                    &b, std::vector<std::int64_t> const &ldb,
                                    std::vector<cl::sycl::buffer<std::int64_t, 1>> &info)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**trans** A vector, `trans[i]` indicates the form of the linear equations.

**n** A vector, `n[i]` is the number of columns of the batch matrix  $A_i$  ( $0 \leq n[i]$ ).

**nrhs** A vector, the number of right hand sides ( $0 \leq nrhs[i]$ ).

**a** A vector of buffers returned by `getrf_batch`. `a[i]` must be of size at least `lda[i] * max(1, n[i])`.

**lda** A vector, `lda[i]` is the leading dimension of `a[i]` ( $n[i] \leq lda[i]$ ).

**ipiv** A vector of buffers, `ipiv` is the batch of pivots returned by `getrf_batch`.

**b** A vector of buffers,  $b[i]$  contains the matrix  $B_i$  whose columns are the right-hand sides for the systems of equations. The second dimension of  $b_i$  must be at least  $\max(1, nrhs[i])$ .

**ldb** A vector,  $ldb[i]$  is the leading dimension of  $b[i]$ .

## Output Parameters

**b** A vector of buffers,  $b[i]$  is overwritten by the solution matrix  $X_i$ .

**info** Vector of buffers containing error information.

If  $info[i]=0$ , the execution is successful.

If  $info[i]=k$ , the  $k$ -th diagonal element of  $U$  is zero, and the solve could not be completed.

If  $info[i]=-k$ , the  $k$ -th parameter had an illegal value.

**Parent topic:** *LAPACK-like Extensions Routines*

### 13.2.1.3.3.9 orgqr\_batch

Generates the orthogonal/unitary matrix  $Q_i$  of the QR factorizations for a group of general matrices.

`orgqr_batch` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The routine generates the whole or part of the orthogonal/unitary matrices  $Q_1, Q_2, \dots, Q_{batch\_size}$  of the QR factorizations formed by the routine `geqrf_batch`. Use this routine after a call to `geqrf_batch`.

Usually  $Q_i$  is determined from the QR factorization of an  $m_i$ -by- $p_i$  matrix  $A_i$  with  $m_i \geq p_i$ . To compute the whole matrix  $Q_i$ , use:

$m[i]$	$m_i$
$n[i]$	$m_i$
$k[i]$	$p_i$

To compute the leading  $p_i$  columns of  $Q_i$  (which form an orthonormal basis in the space spanned by the columns of  $A_i$ ):

$m[i]$	$m_i$
$n[i]$	$p_i$
$k[i]$	$p_i$

To compute the matrix  $Q_k$  of the QR factorization of the leading  $k$  columns of the matrix  $A_i$ :

m [ i ]	m <sub>i</sub>
n [ i ]	m <sub>i</sub>
k [ i ]	k <sub>i</sub>

To compute the leading k<sub>i</sub> columns of Qk<sub>i</sub> (which form an orthonormal basis in the space spanned by the leading k<sub>i</sub> columns of the matrix A<sub>i</sub>):

m [ i ]	m <sub>i</sub>
n [ i ]	k <sub>i</sub>
k [ i ]	k <sub>i</sub>

### 13.2.1.3.3.10 orgqr\_batch (BUFFER Version)

#### Syntax

```
void onemkl::lapack::orgqr_batch(cl::sycl::queue      &queue,           std::vector<std::int64_t>
                                  const      &m,           std::vector<std::int64_t>  const
                                  &n,           std::vector<std::int64_t>  const      &k,
                                  std::vector<cl::sycl::buffer<T, 1>> &a, std::vector<std::int64_t>
                                  const      &lpa,          std::vector<cl::sycl::buffer<T, 1>> &tau,
                                  std::vector<cl::sycl::buffer<std::int64_t, 1>> &info)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**m** A vector, m [ i ] is the order of the unitary matrix Q<sub>i</sub> (0≤m [ i ]).

**n** A vector, n [ i ] is the number of columns of Q<sub>i</sub> to be computed (0≤n [ i ]≤m [ i ]).

**k** A vector, k [ i ] is the number of elementary reflectors whose product defines the matrix Q<sub>i</sub> (0≤k [ i ]≤n [ i ]).

**a** A vector of buffers as returned by [geqrf\\_batch](#). a [ i ] must be of size at least lda [ i ] \*max (1, n [ i ]).

**lda** A vector, lda [ i ] is the leading dimension of a [ i ] (m [ i ]≤lda [ i ]).

**tau** A vector of buffers tau for storing scalars defining elementary reflectors, as returned by [geqrf\\_batch](#).

#### Output Parameters

**a** a [ i ] is overwritten by n [ i ] leading columns of the m [ i ]-by-m [ i ] orthogonal matrix Q<sub>i</sub>.

**info** Vector of buffers containing error information.

If info [ i ]=0, the execution is successful.

If info [ i ]=-k, the k-th parameter had an illegal value.

**Parent topic:** [LAPACK-like Extensions Routines](#)

### 13.2.1.3.3.11 potrf\_batch

Computes the Cholesky factorizations of a batch of symmetric (Hermitian) positive-definite matrices.

`potrf_batch` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

### Description

The routine forms the Cholesky factorizations of a batch of symmetric positive-definite or, for complex data, Hermitian positive-definite matrices  $A_1, A_2, \dots, A_{\text{batch\_size}}$

$A_i = U_i^T * U_i$  for real data, If `uplo[i] = onemkl::uplo::upper`

$A_i = U_i^H * U_i$  for complex data.

$A_i = L_i^T * L_i$  for real data, If `uplo[i] = onemkl::uplo::lower`

$A_i = L_i^H * L_i$  for complex data.

Where  $L_i$  is a lower triangular matrix and  $U_i$  is an upper triangular matrix.

### 13.2.1.3.3.12 potrf\_batch (BUFFER Version)

#### Syntax

```
void onemkl::lapack::potrf_batch(cl::sycl::queue      &queue,      std::vector<onemkl::uplo>
                                const  &uplo,      std::vector<std::int64_t>  const  &n,
                                std::vector<cl::sycl::buffer<T, 1>> &a, std::vector<std::int64_t>
                                const  &lida,     std::vector<cl::sycl::buffer<std::int64_t,  1>>
                                &info)
```

#### Input Parameters

**queue** The queue where the routine should be executed.

**uplo** A vector, `uplo[i]` indicates whether the upper or lower triangular part of the matrix  $A_i$  is stored and how  $A_i$  is factored:

If `uplo = onemkl::upper`, then buffer `a[i]` stores the upper triangular part of  $A_i$  and the strictly lower triangular part of the matrix is not referenced.

If `uplo = onemkl::lower`, then buffer `a[i]` stores the lower triangular part of  $A_i$  and the strictly upper triangular part of the matrix is not referenced.

**n** A vector, `n[i]` is the number of columns of the batch matrix  $A_i$  ( $0 \leq n[i]$ ).

**a** A vector of buffers, `a[i]` stores the matrix  $A_i$ . `a[i]` must be of size at least `lida[i] * max(1, n[i])`.

**lida** A vector, `lida[i]` is the leading dimension of `a[i]` ( $n[i] \leq lida[i]$ ).

## Output Parameters

**a**  $a[i]$  is overwritten by the Cholesky factor  $U_i$  or  $L_i$ , as specified by `uplo[i]`.

**info** Vector of buffers containing error information.

If `info[i]=0`, the execution is successful.

If `info[i]=k`, the leading minor of order  $k$  (and therefore the matrix  $A_i$  itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix  $A_i$ .

If `info[i]=-k`, the  $k$ -th parameter had an illegal value.

**Parent topic:** [LAPACK-like Extensions Routines](#)

### 13.2.1.3.3.13 `potrs_batch`

Solves a system of linear equations with a batch of Cholesky-factored symmetric (Hermitian) positive-definite coefficient matrices.

`potrs_batch` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The routine solves for  $X_i$ , in batch fashion, the system of linear equations  $A_i^*X_i = B_i$  with a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix  $A$ , given the Cholesky factorization of  $A$ :

$A_i = U_i^T * U_i$  for real data, If `uplo[i] = onemkl::uplo::upper`

$A_i = U_i^H * U_i$  for complex data.

$A_i = L_i^T * L_i$  for real data, If `uplo[i] = onemkl::uplo::lower`

$A_i = L_i^H * L_i$  for complex data.

Where  $L_i$  is a lower triangular matrix and  $U_i$  is an upper triangular matrix.

### 13.2.1.3.3.14 `potrs_batch` (BUFFER Version)

## Syntax

```
void onemkl::lapack::potrs_batch(cl::sycl::queue      &queue,      std::vector<onemkl::uplo>
                                const    &uplo,      std::vector<std::int64_t>  const
                                &n,      std::vector<std::int64_t>  const    &nrhs,
                                std::vector<cl::sycl::buffer<T, 1>> &a, std::vector<std::int64_t>
                                const    &lda,      std::vector<cl::sycl::buffer<T, 1>>
                                &b,      std::vector<std::int64_t>  const    &ldb,
                                std::vector<cl::sycl::buffer<std::int64_t, 1>> &info)
```

## Input Parameters

**queue** The queue where the routine should be executed.

**uplo** A vector,  $\text{uplo}[i]$  indicates whether the upper or lower triangular part of the matrix  $A_i$  is stored and how  $A_i$  is factored:

If  $\text{uplo} = \text{onemkl::upper}$ , then buffer  $a[i]$  stores the upper triangular part of  $A_i$  and the strictly lower triangular part of the matrix is not referenced.

If  $\text{uplo} = \text{onemkl::lower}$ , then buffer  $a[i]$  stores the lower triangular part of  $A_i$  and the strictly upper triangular part of the matrix is not referenced.

**n** A vector,  $n[i]$  is the number of columns of the batch matrix  $A_i$  ( $0 \leq n[i]$ ).

**nrhs** A vector,  $\text{nrhs}[i]$  is the number of right-hand sides ( $0 \leq \text{nrhs}[i]$ ).

**a** A vector of buffers returned by *potrf\_batch*.  $a[i]$  must be of size at least  $\text{lda}[i] * \max(1, n[i])$ .

**lda** A vector,  $\text{lda}[i]$  is the leading dimension of  $a[i]$  ( $n[i] \leq \text{lda}[i]$ ).

**b** A vector of buffers,  $b[i]$  contains the matrix  $B_i$  whose columns are the right-hand sides for the systems of equations. The second dimension of  $b[i]$  must be at least  $\max(1, \text{nrhs}[i])$ .

**ldb** A vector,  $\text{ldb}[i]$  is the leading dimension of  $b[i]$ .

## Output Parameters

**b**  $b[i]$  is overwritten by the solution matrix  $X[i]$ .

**info** Vector of buffers containing error information.

If  $\text{info}[i]=0$ , the execution is successful.

If  $\text{info}[i]=k$ , the  $k$ -th diagonal element of the Cholesky factor is zero and the solve could not be completed.

If  $\text{info}[i]=-k$ , the  $k$ -th parameter had an illegal value.

**Parent topic:** *LAPACK-like Extensions Routines*

## Note

Different arrays used as parameters to oneMKL LAPACK routines must not overlap.

## Warning

LAPACK routines assume that input matrices do not contain IEEE 754 special values such as INF or NaN values. Using these special values may cause LAPACK to return unexpected results or become unstable.

**Parent topic:** *Dense Linear Algebra*

## 13.2.2 Sparse Linear Algebra

The oneAPI Math Kernel Library provides a Data Parallel C++ interface to some of the Sparse Linear Algebra routines. *Sparse BLAS Routines* provide basic operations on sparse vectors and matrices

### 13.2.2.1 Sparse BLAS Routines

Sparse BLAS Routines provide basic operations on sparse vectors and matrices

- *onemkl::sparse::matrixInit*
- *onemkl::sparse::setCSRstructure*
- *onemkl::sparse::gemm*
- *onemkl::sparse::gemv*
- *onemkl::sparse::gemvdot*
- *onemkl::sparse::gemvOptimize*
- *onemkl::sparse::symv*
- *onemkl::sparse::trmv*
- *onemkl::sparse::trmvOptimize*
- *onemkl::sparse::trsv*
- *onemkl::sparse::trsvOptimize*
- *Exceptions*

#### 13.2.2.1.1 `onemkl::sparse::matrixInit`

Initializes a `matrixHandle_t` object to default values.

##### Syntax

```
void onemkl::sparse::matrixInit (matrixHandle_t hMatrix)
```

##### Include Files

`mkl_spblas_sycl.hpp`

##### Description

The `onemkl::sparse::matrixInit` function initializes the `matrixHandle_t` object with default values, otherwise it throws an exception.

## Note

Refer to [Exceptions](#) for a detailed description of the exceptions thrown.

**Parent topic:** Sparse BLAS Routines

### 13.2.2.1.2 onemkl::sparse::setCSRstructure

Creates a handle for a CSR matrix.

## Syntax

**Using SYCL buffers:**

```
void onemkl::sparse::setCSRstructure(matrixHandle_t handle, intType nRows, intType nCols,
                                      onemkl::index_base index, cl::sycl::buffer<intType, 1>
                                      &rowIndex, cl::sycl::buffer<intType, 1> &colIndex,
                                      cl::sycl::buffer<fp, 1> &values)
```

**Using USM pointers:**

```
void onemkl::sparse::setCSRstructure(matrixHandle_t handle, intType nRows, intType nCols,
                                      onemkl::index_base index, intType *rowIndex, intType
                                      *colIndex, fp *values)
```

## Include Files

- mkl\_spblas\_sycl.hpp

## Description

The onemkl::sparse::setCSRstructure routine creates a handle for a sparse matrix of dimensions nRows-by-nCols represented in the CSR format..

## Note

Refer to [Supported Types](#) for a list of supported <fp> and <intType>, and refer to [Exceptions](#) for a detailed description of the exceptions thrown.

## Input Parameters

**handle** Handle to object containing sparse matrix and other internal data for subsequent SYCL Sparse BLAS operations.

**nRows** Number of rows of the input matrix .

**nCols** Number of columns of the input matrix .

**Index** Indicates how input arrays are indexed.

onemkl::index_base::zero	Zero-based (C-style) indexing: indices start at 0.
onemkl::index_base::one	One-based (Fortran-style) indexing: indices start at 1.

**rowIndex** SYCL or USM memory object containing an array of length  $m+1$ . This array contains row indices, such that  $\text{rowsIndex}[i]$  - indexing is the first index of row  $i$  in the arrays `values` and `colIndex`. indexing takes 0 for zero-based indexing and 1 for one-based indexing. Refer to `pointerB` and `pointerE` array description in [Sparse BLAS CSR Matrix Storage Format](#) for more details.

## Note

Refer to [Three Array Variation of CSR Format](#) for more details.

**colIndex** SYCL or USM memory object which stores an array containing the column indices in index-based numbering (`index` takes 0 for zero-based indexing and 1 for one-based indexing) for each non-zero element of the input matrix. Its length is at least `nrows`.

**values** SYCL or USM memory object which stores an array containing non-zero elements of the input matrix. Its length is equal to length of the `colIndex` array. Refer to the `values` array description in [Sparse BLAS CSR Matrix Storage Format](#) for more details.

## Output Parameters

**handle** Handle to object containing sparse matrix and other internal data for subsequent Sycl Sparse BLAS operations.

**Parent topic:** [Sparse BLAS Routines](#)

### 13.2.2.1.3 `onemkl::sparse::gemm`

Computes a sparse matrix-dense matrix product. Currently, only ROW-MAJOR layout for dense matrix storage in Data Parallel C++ `onemkl::sparse::gemm` functionality is supported.

## Syntax

### Note

Currently, complex types are not supported.

#### Using SYCL buffers:

```
void onemkl::sparse::gemm(cl::sycl::queue &queue, onemkl::transpose transpose_val, const fp alpha, matrixHandle_t handle, cl::sycl::buffer<fp, 1> &b, const std::int64_t columns, const std::int64_t ldb, const fp beta, cl::sycl::buffer<fp, 1> &c, const std::int64_t ldc)
```

#### Using USM pointers:

```
void onemkl::sparse::gemm(cl::sycl::queue &queue, onemkl::transpose transpose_val, const fp alpha, matrixHandle_t handle, const fp *b, const std::int64_t columns, const std::int64_t ldb, const fp beta, fp *c, const std::int64_t ldc)
```

## Include Files

- mkl\_spblas\_sycl.hpp

## Description

### Note

Refer to [Supported Types](#) for a list of supported `<fp>` and `<intType>` and refer to [Exceptions](#) for a detailed description of the exceptions thrown. The `onemkl::sparse::gemm` routine computes a sparse matrix-dense matrix defined as

```
c := alpha*op(A)*b + beta*c
```

where:

`alpha` and `beta` are scalars, `b` and `c` are dense matrices.

## Input Parameters

**queue** Specifies the SYCL command queue which will be used for SYCL kernels execution.

**transpose\_val** Specifies operation `op()` on input matrix.

<code>onemkl::transpose::nontrans</code>	Non-transpose, $\text{op}(A) = A$ .
<code>onemkl::transpose::trans</code>	Transpose, $\text{op}(A) = A^T$ .
<code>onemkl::transpose::conjtrans</code>	Conjugate transpose, $\text{op}(A) = A^H$ .

### Note

Currently, the only supported case for operation is `onemkl::transpose::nontrans`.

**alpha** Specifies the scalar `alpha`.

**handle** Handle to object containing sparse matrix and other internal data. Created using one of the `onemkl::sparse::set<sparse_matrix_type>`structure routines.

### Note

Currently, the only supported case for `<sparse_matrix_type>` is CSR.

**b** SYCL or USM memory object containing an array of size at least `rows*ldb`, where `rows` = the number of columns of matrix `A` if `op = onemkl::transpose::nontrans`, or `rows` = the number of rows of matrix `A` otherwise.

**columns** Number of columns of matrix `c`.

**ldb** Specifies the leading dimension of matrix `b`.

**beta** Specifies the scalar `beta`.

**c** SYCL or USM memory object containing an array of size at least `rows*ldc`, where `rows` = the number of columns of matrix `A` if `op = onemkl::transpose::nontrans`, or `rows` = the number of columns of matrix `A` otherwise.

## Output Parameters

**c** Overwritten by the updated matrix  $c$ .

**Parent topic:** Sparse BLAS Routines

### 13.2.2.1.4 onemkl::sparse::gemv

Computes a sparse matrix-dense vector product.

## Syntax

### Note

Complex types are not currently supported.

### Using SYCL buffers:

```
void onemkl::sparse::gemv(cl::sycl::queue &queue, onemkl::transpose transpose_val, fp alpha, matrixHandle_t handle, cl::sycl::buffer<fp, 1> &x, fp beta, cl::sycl::buffer<fp, 1> &y)
```

### Using USM pointers:

```
void onemkl::sparse::gemv(cl::sycl::queue &queue, onemkl::transpose transpose_val, fp alpha, matrixHandle_t handle, fp *x, fp beta, fp *y)
```

## Include Files

- mkl\_spblas\_sycl.hpp

## Description

### Note

Refer to [Supported Types](#) for a list of supported `<fp>` and `<intType>` and refer to [Exceptions](#) for a detailed description of the exceptions thrown. The `onemkl::sparse::gemv` routine computes a sparse matrix-vector product defined as

$$y := \alpha \cdot op(A) \cdot x + \beta \cdot y$$

where:

`alpha` and `beta` are scalars, `x` and `y` are vectors.

## Input Parameters

**queue** Specifies the SYCL command queue which will be used for SYCL kernels execution.

**transpose\_val** Specifies operation `op()` on input matrix.

<code>onemkl::transpose::nontrans</code>	Non-transpose, $\text{op}(A) = A$ .
<code>onemkl::transpose::trans</code>	Transpose, $\text{op}(A) = A^T$ .
<code>onemkl::transpose::conjtrans</code>	Conjugate transpose, $\text{op}(A) = A^H$ .

### Note

Currently, the only supported case for `operation` is `onemkl::transpose::nontrans`.

**alpha** Specifies the scalar `alpha`.

**handle** Handle to object containing sparse matrix and other internal data. Created using one of the `onemkl::sparse::set<sparse_matrix_type>`structure routines.

### Note

Currently, the only supported case for `<sparse_matrix_type>` is CSR.

**x** SYCL or USM memory object containing an array of size at least equal to the number of columns of input matrix if `op = onemkl::transpose::nontrans` and at least the number of rows of input matrix otherwise.

**beta** Specifies the scalar `beta`.

**y** SYCL or USM memory object containing an array of size at least equal to the number of columns of input matrix if `op = onemkl::transpose::nontrans` and at least the number of rows of input matrix otherwise.

## Output Parameters

**y** Overwritten by the updated vector `y`.

## Return Values

None

## Example

An example of how to use `onemkl::sparse::gemv` with SYCL buffers or USM can be found in the oneMKL installation directory, under:

```
examples/sycl/spblas/sparse_gemv.cpp
```

```
examples/sycl/spblas/sparse_gemv_usm.cpp
```

**Parent topic:** Sparse BLAS Routines

### 13.2.2.1.5 onemkl::sparse::gemvdot

Computes a sparse matrix-vector product with dot product.

#### Syntax

##### Note

Currently, complex types are not supported.

##### Using SYCL buffers:

```
void onemkl::sparse::gemvdot (cl::sycl::queue &queue, onemkl::transpose transpose_val, fp alpha, matrixHandle_t handle, cl::sycl::buffer<fp, 1> &x, fp beta, cl::sycl::buffer<fp, 1> &y, cl::sycl::buffer<fp, 1> &d)
```

##### Using USM pointers:

```
void onemkl::sparse::gemvdot (cl::sycl::queue &queue, onemkl::transpose transpose_val, fp alpha, matrixHandle_t handle, fp *x, fp beta, fp *y, fp *d)
```

#### Include Files

- mkl\_spblas\_sycl.hpp

#### Description

##### Note

Refer to [Supported Types](#) for a list of supported `<fp>` and `<intType>` and refer to [Exceptions](#) for a detailed description of the exceptions thrown. The `onemkl::sparse::gemvdot` routine computes a sparse matrix-vector product and dot product defined as

```
y := alpha*op(A)*x + beta*y
```

```
d := x * y
```

where:

$A$  is a general sparse matrix,  $\alpha$ ,  $\beta$ , and  $d$  are scalars,  $x$  and  $y$  are vectors.

#### Input Parameters

**queue** Specifies the SYCL command queue which will be used for SYCL kernels execution.

**transpose\_val** Specifies operation `op()` on input matrix.

<code>onemkl::transpose::nontrans</code>	Non-transpose, $\text{op}(A) = A$ .
<code>onemkl::transpose::trans</code>	Transpose, $\text{op}(A) = A^T$ .
<code>onemkl::transpose::conjtrans</code>	Conjugate transpose, $\text{op}(A) = A^H$ .

## Note

Currently, the only supported case for operation is onemkl::transpose::nontrans.

**alpha** Specifies the scalar alpha.

**handle** Handle to object containing sparse matrix and other internal data. Created using one of the onemkl::sparse::set<sparse\_matrix\_type>structure routines.

## Note

Currently, the only supported case for <sparse\_matrix\_type> is CSR.

**x** SYCL or USM memory object containing an array of size at least equal to the number of columns of input matrix if  $op = \text{onemkl::transpose::nontrans}$  and at least the number of rows of input matrix otherwise.

**beta** Specifies the scalar beta.

**y** SYCL or USM memory object containing an array of size at least equal to the number of columns of input matrix if  $op = \text{onemkl::transpose::nontrans}$  and at least the number of rows of input matrix otherwise.

## Output Parameters

**y** Overwritten by the updated vector y.

**d** Overwritten by the dot product of x and y.

**Parent topic:** Sparse BLAS Routines

### 13.2.2.1.6 onemkl::sparse::gemvOptimize

Performs internal optimizations for onemkl::sparse::gemv by analyzing the matrix structure.

## Syntax

### Note

Currently, complex types are not supported.

The API is the same when using SYCL buffers or USM pointers.

```
void onemkl::sparse::gemvOptimize(cl::sycl::queue &queue, onemkl::transpose transpose_val, ma-
trixHandle_t handle)
```

## Include Files

- mkl\_spblas\_sycl.hpp

## Description

### Note

Refer to [Exceptions](#) for a detailed description of the exceptions thrown. The `onemkl::sparse::gemvOptimize` routine analyzes matrix structure and performs optimizations. Optimized data is then stored in the handle.

## Input Parameters

**queue** Specifies the SYCL command queue which will be used for SYCL kernels execution.

**transpose\_val** Specifies operation `op()` on input matrix.

<code>onemkl::transpose::nontrans</code>	Non-transpose, $\text{op}(A) = A$ .
<code>onemkl::transpose::trans</code>	Transpose, $\text{op}(A) = A^T$ .
<code>onemkl::transpose::conjtrans</code>	Conjugate transpose, $\text{op}(A) = A^H$ .

### Note

Currently, the only supported case for operation is `onemkl::transpose::nontrans`.

**handle** Handle to object containing sparse matrix and other internal data. Created using one of the `onemkl::sparse::set<sparse_matrix_type>`structure routines.

### Note

Currently, the only supported case for `<sparse_matrix_type>` is CSR.

**Parent topic:** Sparse BLAS Routines

### 13.2.2.1.7 `onemkl::sparse::symv`

Computes a sparse matrix-vector product for a symmetric matrix.

## Syntax

### Note

Currently, complex types are not supported.

#### Using SYCL buffers:

```
void onemkl::sparse::symv(cl::sycl::queue &queue, onemkl::uplo uplo_val, fp alpha, matrixHandle_t
                           handle, cl::sycl::buffer<fp, 1> &x, fp beta, cl::sycl::buffer<fp, 1> &y)
```

#### Using USM pointers:

```
void onemkl::sparse::symv(cl::sycl::queue &queue, onemkl::uplo uplo_val, fp alpha, matrixHandle_t
                           handle, fp *x, fp beta, fp *y)
```

## Include Files

- mkl\_spblas\_sycl.hpp

## Description

### Note

Refer to [Exceptions](#) for a detailed description of the exceptions thrown. The onemkl::sparse::symv routine computes a sparse matrix-vector product over a symmetric part defined as

```
y := alpha*A*x + beta*y
```

where:

**alpha** and **beta** are scalars, and **x** and **y** are vectors.

## Input Parameters

**queue** Specifies the SYCL command queue which will be used for SYCL kernels execution.

**uplo\_val** Specifies which symmetric part is to be processed.

onemkl::uplo::lower	The lower symmetric part is processed.
onemkl::uplo::upper	The upper symmetric part is processed.

**alpha** Specifies the scalar **alpha**.

**handle** Handle to object containing sparse matrix and other internal data. Created using one of the onemkl::sparse::set<sparse\_matrix\_type>structure routines.

### Note

Currently, the only supported case for <sparse\_matrix\_type> is CSR.

**x** SYCL or USM memory object containing an array of size at least equal to the number of columns of input matrix.

**beta** Specifies the scalar **beta**.

**y** SYCL or USM memory object containing an array of size at least equal to the number of columns of input matrix.

## Output Parameters

**y** Overwritten by the updated vector **y**.

## Example

An example of how to use `onemkl::sparse::symv` with SYCL buffers or USM can be found in the oneMKL installation directory, under:

```
examples/sycl/spblas/sparse_symv_1.cpp
```

```
examples/sycl/spblas/sparse_symv_1_usm.cpp
```

**Parent topic:** Sparse BLAS Routines

### 13.2.2.1.8 `onemkl::sparse::trmv`

Computes a sparse matrix-dense vector product over upper or lower triangular parts of the matrix.

#### Syntax

#### Note

Currently, complex types are not supported.

#### Using SYCL buffers:

```
void onemkl::sparse::trmv(cl::sycl::queue &queue, onemkl::uplo uplo_val, onemkl::transpose transpose_val, onemkl::diag diag_val, fp alpha, matrixHandle_t handle, cl::sycl::buffer<fp, 1> &x, fp beta, cl::sycl::buffer<fp, 1> &y)
```

#### Using USM pointers:

```
void onemkl::sparse::trmv(cl::sycl::queue &queue, onemkl::uplo uplo_val, onemkl::transpose transpose_val, onemkl::diag diag_val, fp alpha, matrixHandle_t handle, fp *x, fp beta, fp *y)
```

#### Include Files

- `mkl_spblas_sycl.hpp`

#### Description

#### Note

Refer to [Supported Types](#) for a list of supported `<fp>` and `<intType>` and refer to [Exceptions](#) for a detailed description of the exceptions thrown. The `onemkl::sparse::trmv` routine computes a sparse matrix-vector product over a triangular part defined as

```
y := alpha * (op)A*x + beta*y
```

where:

`alpha` and `beta` are scalars, and `x` and `y` are vectors.

## Input Parameters

**queue** Specifies the SYCL command queue which will be used for SYCL kernels execution.

**uplo\_val** Specifies which triangular matrix is to be processed.

onemkl::uplo::lower	The lower triangular matrix part is processed.
onemkl::uplo::upper	The upper triangular matrix part is processed.

**transpose\_val** Specifies operation `op()` on input matrix.

onemkl::transpose::nontrans	Non-transpose, $\text{op}(A) = A$ .
onemkl::transpose::trans	Transpose, $\text{op}(A) = A^T$ .
onemkl::transpose::conjtrans	Conjugate transpose, $\text{op}(A) = A^H$ .

## Note

Currently, the only supported case for `operation` is `onemkl::transpose::nontrans`.

**diag\_val** Specifies if the input matrix has a unit diagonal or not.

onemkl::diag::nonunit	Diagonal elements might not be equal to one.
onemkl::diag::unit	Diagonal elements are equal to one.

**alpha** Specifies the scalar `alpha`.

**handle** Handle to object containing sparse matrix and other internal data. Created using one of the `onemkl::sparse::set<sparse_matrix_type>`structure routines.

## Note

Currently, the only supported case for `<sparse_matrix_type>` is CSR.

**x** SYCL or USM memory object containing an array of size at least equal to the number of columns of input matrix if `op = onemkl::transpose::nontrans` and at least the number of rows of input matrix otherwise.

**beta** Specifies the scalar `beta`.

**y** SYCL or USM memory object containing an array of size at least equal to the number of columns of input matrix if `op = onemkl::transpose::nontrans` and at least the number of rows of input matrix otherwise.

## Output Parameters

**y** Overwritten by the updated vector `y`.

**Parent topic:** Sparse BLAS Routines

### 13.2.2.1.9 onemkl::sparse::trmvOptimize

Performs internal optimizations for onemkl::sparse::trmv by analyzing the matrix structure.

#### Syntax

##### Note

Currently, complex types are not supported.

The API is the same when using SYCL buffers or USM pointers.

```
void onemkl::sparse::trmvOptimize(cl::sycl::queue &queue, onemkl::uplo uplo_val,
                                   onemkl::transpose transpose_val, onemkl::diag diag_val,
                                   matrixHandle_t handle)
```

#### Include Files

- mkl\_spblas\_sycl.hpp

#### Description

##### Note

Refer to [Exceptions](#) for a detailed description of the exceptions thrown. The onemkl::sparse::trmvOptimize routine analyzes matrix structure and performs optimizations. Optimized data is then stored in the handle.

#### Input Parameters

**queue** Specifies the SYCL command queue which will be used for SYCL kernels execution.

**uplo\_val** Specifies the triangular matrix part for the input matrix.

onemkl::uplo::lower	The lower triangular matrix part is processed.
onemkl::uplo::upper	The upper triangular matrix part is processed.

**transpose\_val** Specifies operation `op()` on input matrix.

onemkl::transpose::nontrans	Non-transpose, $\text{op}(A) = A$ .
onemkl::transpose::trans	Transpose, $\text{op}(A) = A^T$ .
onemkl::transpose::conjtrans	Conjugate transpose, $\text{op}(A) = A^H$ .

**Note**

Currently, the only supported case for operation is onemkl::transpose::nontrans.

**diag\_val** Specifies if the input matrix has a unit diagonal or not.

onemkl::diag::nonunit	Diagonal elements might not be equal to one.
onemkl::diag::unit	Diagonal elements are equal to one.

**handle** Handle to object containing sparse matrix and other internal data. Created using one of the onemkl::sparse::set<sparse\_matrix\_type>structure routines.

**Note**

Currently, the only supported case for <sparse\_matrix\_type> is CSR.

**Parent topic:** Sparse BLAS Routines

**13.2.2.1.10 onemkl::sparse::trsv**

Solves a system of linear equations for a triangular sparse matrix.

**Syntax****Note**

Currently, complex types are not supported.

**Using SYCL buffers:**

```
void onemkl::sparse::trsv(cl::sycl::queue &queue, onemkl::uplo uplo_val, onemkl::transpose transpose_val, onemkl::diag diag_val, matrixHandle_t handle, cl::sycl::buffer<fp, 1> &x, cl::sycl::buffer<fp, 1> &y)
```

**Using USM pointers:**

```
void onemkl::sparse::trsv(cl::sycl::queue &queue, onemkl::uplo uplo_val, onemkl::transpose transpose_val, onemkl::diag diag_val, matrixHandle_t handle, fp *x, fp *y)
```

**Include Files**

- mkl\_spblas\_sycl.hpp

**Description****Note**

Refer to [Supported Types](#) for a list of supported <fp> and <intType>, and refer to [Exceptions](#) for a detailed description of the exceptions thrown. The onemkl::sparse::trsv routine solves a system of linear equations for a square matrix:

op(A) * y = x
---------------

where  $A$  is a triangular sparse matrix of size  $m$  rows by  $m$  columns,  $op$  is a matrix modifier for matrix  $A$ ,  $\alpha$  is a scalar, and  $x$  and  $y$  are vectors of length at least  $m$ .

## Input Parameters

**queue** Specifies the SYCL command queue which will be used for SYCL kernels execution.

**uplo\_val** Specifies if the input matrix is an upper triangular or a lower triangular matrix.

onemkl::uplo::lower	The lower triangular matrix part is processed.
onemkl::uplo::upper	The upper triangular matrix part is processed.

**transpose\_val** Specifies operation  $op()$  on input matrix.

onemkl::transpose::nontrans	Non-transpose, $op(A) = A$ .
onemkl::transpose::trans	Transpose, $op(A) = A^T$ .
onemkl::transpose::conjtrans	Conjugate transpose, $op(A) = A^H$ .

## Note

Currently, the only supported case for operation is onemkl::transpose::nontrans.

**diag\_val** Specifies if the input matrix has a unit diagonal or not.

onemkl::diag::nonunit	Diagonal elements might not be equal to one.
onemkl::diag::unit	Diagonal elements are equal to one.

**handle** Handle to object containing sparse matrix and other internal data. Created using one of the onemkl::sparse::set<sparse\_matrix\_type>structure routines.

## Note

Currently, the only supported case for <sparse\_matrix\_type> is CSR.

- x** SYCL or USM memory object containing an array of size at least equal to the number of columns of input matrix if  $op = \text{onemkl::transpose::nontrans}$  and at least the number of rows of input matrix otherwise. It is the input vector  $x$
- y** SYCL or USM memory object containing an array of size at least equal to the number of columns of input matrix if  $op = \text{onemkl::transpose::nontrans}$  and at least the number of rows of input matrix otherwise.

## Output Parameters

**y** SYCL or USM memory object containing an array of size at least  $nRows$  filled with the solution to the system of linear equations.

**Parent topic:** Sparse BLAS Routines

### 13.2.2.1.11 onemkl::sparse::trsvOptimize

Performs internal optimizations for onemkl::sparse::trsv by analyzing the matrix structure.

#### Syntax

##### Note

Currently, complex types are not supported.

The API is the same when using SYCL buffers or USM pointers.

```
void onemkl::sparse::trsvOptimize(cl::sycl::queue &queue, onemkl::uplo uplo_val,
                                   onemkl::transpose transpose_val, onemkl::diag diag_val,
                                   matrixHandle_t handle)
```

#### Include Files

- mkl\_spblas\_sycl.hpp

#### Description

##### Note

Refer to [Exceptions](#) for a detailed description of the exceptions thrown. The onemkl::sparse::trsvOptimize routine analyzes matrix structure and performs optimizations. Optimized data is then stored in the handle.

#### Input Parameters

**queue** Specifies the SYCL command queue which will be used for SYCL kernels execution.

**uplo\_val** Specifies the triangular matrix part for the input matrix.

onemkl::uplo::lower	The lower triangular matrix part is processed.
onemkl::uplo::upper	The upper triangular matrix part is processed.

**transpose\_val** Specifies operation `op()` on input matrix.

onemkl::transpose::nontrans	Non-transpose, $\text{op}(A) = A$ .
onemkl::transpose::trans	Transpose, $\text{op}(A) = A^T$ .
onemkl::transpose::conjtrans	Conjugate transpose, $\text{op}(A) = A^H$ .

**Note**

Currently, the only supported case for operation is `onemkl::transpose::nontrans`.

**diag\_val** Specifies if the input matrix has a unit diagonal or not.

<code>onemkl::diag::nonunit</code>	Diagonal elements might not be equal to one.
<code>onemkl::diag::unit</code>	Diagonal elements are equal to one.

**handle** Handle to object containing sparse matrix and other internal data. Created using one of the `onemkl::sparse::set<sparse_matrix_type>`structure routines.

**Note**

Currently, the only supported case for `<sparse_matrix_type>` is CSR.

**Parent topic:** Sparse BLAS Routines

### 13.2.2.1.12 Supported Types

Data Types <fp>	Integer Types <intType>
<code>float</code>	<code>int</code>
<code>double</code>	<code>std::int64_t</code>
<code>std::complex&lt;float&gt;</code>	
<code>std::complex&lt;double&gt;</code>	

### 13.2.2.1.13 Exceptions

Exception	Description
<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

**Parent topic:** *Sparse Linear Algebra*

## 13.2.3 Discrete Fourier Transforms

The *Fourier Transform Functions* offer several options for computing Discrete Fourier Transforms (DFTs).

### 13.2.3.1 Fourier Transform Functions

- `onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>` Creates an empty descriptor for the templated precision and forward domain.
- `onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::Init` Allocates the descriptor data structure and initializes it with default configuration values.
- `onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::setValue` Sets one particular configuration parameter with the specified configuration value.
- `onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::getValue` Gets the configuration value of one particular configuration parameter.
- `onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::commit` Performs all initialization for the actual FFT computation.
- `onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::computeForward<typename IOType>` Computes the forward FFT.
- `onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::computeBackward<typename IOType>` Computes the backward FFT.

**Parent topic:** *oneMKL Domains*

#### 13.2.3.1.1 `onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>`

Creates an empty descriptor for the templated precision and forward domain.

##### Syntax

```
onemkl::dft::Descriptor<PRECISION, DOMAIN> descriptor
```

##### Include Files

- `mkl_dfti_sycl.hpp`

##### Description

This constructor initializes members to default values and does not throw exceptions. Note that the precision and domain are determined via templating.

##### Template Parameters

Name	Type	Description
PRECISION	<code>onemkl::dft::Precision</code>	<code>Precision::SINGLE</code> or <code>Precision::DOUBLE</code> are supported precisions. Double precision has limited GPU support and full CPU and Host support.
DOMAIN	<code>onemkl::dft::Domain</code>	<code>Domain::REAL</code> or <code>Domain::COMPLEX</code> are supported forward domains.

**Parent topic:** Fourier Transform Functions

### 13.2.3.1.2 `onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::Init`

Allocates the descriptor data structure and initializes it with default configuration values.

#### Syntax

`onemkl::dft::ErrCode init (dimension)`

#### Include Files

- `mkl_dfti_sycl.hpp`

#### Description

This function allocates memory for the descriptor data structure and instantiates it with all the default configuration settings for the precision, forward domain, and dimensions of the transform. This function does not perform any significant computational work, such as computation of twiddle factors. The function `onemkl::dft::Descriptor::commit` does this work after the function `onemkl::dft::Descriptor::setValue` has set values of all necessary parameters.

The interface supports a single `std::int64_t` input for 1-D transforms, and an `std::vector` for N-D transforms.

The function returns `onemkl::dft::ErrCode::NO_ERROR` when it completes successfully.

#### Input Parameters: 1-Dimensional

Name	Type	Description
<code>dimension</code>	<code>std::int64_t</code>	Dimension of the transform 1-D transform.

#### Input Parameters: N-Dimensional

Name	Type	Description
<code>dimensions</code>	<code>std::vector&lt;std::int64_t&gt;</code>	Dimensions of the transform.

#### Output Parameters

Name	Type	Description
<code>status</code>	<code>onemkl::dft::ErrCode</code>	Function completion status.

## Return Values

The function returns a value indicating whether the operation was successful or not, and why.

Return Value	Description
onemkl::dft::ErrCode::NO_ERROR	The operation was successful.
onemkl::dft::ErrCode::INCONSISTENT_CONFIGURATION/ onemkl::dft::ErrCode::INVALID_CONFIGURATION	An input value provided is invalid.
onemkl::dft::ErrCode::UNIMPLEMENTED	Functionality requested is not implemented.
onemkl::dft::ErrCode::MEMORY_ERROR	Internal memory allocation failed.

**Parent topic:** Fourier Transform Functions

### 13.2.3.1.3 onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::setValue

Sets one particular configuration parameter with the specified configuration value.

#### Syntax

```
onemkl::dft::ErrCode setValue (onemkl::dft::ConfigParam param, ...)
```

#### Include Files

- mkl\_dfti\_sycl.hpp

#### Description

This function sets one particular configuration parameter with the specified configuration value. Each configuration parameter is a named constant, and the configuration value must have the corresponding type, which can be a named constant or a native type. For available configuration parameters and the corresponding configuration values, see [Config Params](#).

#### Note

An important addition to the configuration options for DPC++ FFT interface is onemkl::dft::FWD\_DISTANCE, onemkl::dft::BWD\_DISTANCE. The FWD\_DISTANCE describes the distance between different FFTs for the forward domain while BWD\_DISTANCE describes the distance between different FFTs for the backward domain. It is required for all R2C or C2R transforms to use FWD\_DISTANCE and BWD\_DISTANCE instead of INPUT\_STRIDE and OUTPUT\_STRIDE, respectively.

The onemkl::dft::setValue function cannot be used to change configuration parameters onemkl::dft::ConfigParam::FORWARD\_DOMAIN, onemkl::dft::ConfigParam::PRECISION, DFTI\_DIMENSION since these are a part of the template. Likewise, onemkl::dft::ConfigParam::LENGTHS is set with the function call onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::init.

All calls to setValue must be done after init, and before commit. This is because the handle may have been moved to an offloaded device after commit.

Function calls needed to configure an FFT descriptor for a particular call to an FFT computation function are summarized in [DftiSetValue](#).

The function returns `onemkl::dft::ErrCode::NO_ERROR` when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

## Input Parameters

Name	Type	Description
<code>param</code>	<code>onemkl::dft::ConfigParam</code>	Configuration parameter.
<code>value</code>	Depends on the configuration parameter	Configuration value.

## Output Parameters

Name	Type	Description
<code>status</code>	<code>onemkl::dft::ErrCode</code>	Function completion status.

## Return Values

The function returns a value indicating whether the operation was successful or not, and why.

Return Value	Description
<code>onemkl::dft::ErrCode::NO_ERROR</code>	The operation was successful.
<code>onemkl::dft::ErrCode::BAD_DESCRIPTOR</code>	DFTI handle provided is invalid.
<code>onemkl::dft::ErrCode::INCONSISTENT_CONFIGURATION/</code> <code>onemkl::dft::ErrCode::INVALID_CONFIGURATION</code>	An input value provided is invalid.
<code>onemkl::dft::ErrCode::UNIMPLEMENTED</code>	Functionality requested is not implemented.

**Parent topic:** Fourier Transform Functions

### 13.2.3.1.4 `onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::getValue`

Gets the configuration value of one particular configuration parameter.

## Syntax

`onemkl::dft::ErrCode getValue (onemkl::dft::ConfigParam param, ...)`

## Include Files

- mkl\_dfti\_sycl.hpp

## Description

This function gets the configuration value of one particular parameter. Each configuration parameter is a named constant, and the configuration value must have the corresponding type, which can be a named constant or a native type. For available configuration parameters and the corresponding configuration values, see [DftiGetValue](#)

## Note

All calls to getValue must be done after init, and before commit. This is because the handle may have been moved to an offloaded device after commit.

The function returns onemkl::dft::ErrCode::NO\_ERROR when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

## Input Parameters

Name	Type	Description
param	enum DFTI_CONFIG_PARAM	Configuration parameter.
value_ptr	Depends on the configuration parameter	Pointer to configuration value.

## Output Parameters

Name	Type	Description
value	Depends on the configuration parameter	Configuration value.
status	std::int64_t	Function completion status.

## Return Values

The function returns a value indicating whether the operation was successful or not, and why.

Return Value	Description
onemkl::dft::ErrCode::NO_ERROR	The operation was successful.
onemkl::dft::ErrCode::BAD_DESCRIPTOR	DFTI handle provided is invalid.
onemkl::dft::ErrCode::INCONSISTENT_CONFIGURATION/ onemkl::dft::ErrCode::INVALID_CONFIGURATION	An input value provided is invalid.
onemkl::dft::ErrCode::UNIMPLEMENTED	Functionality requested is not implemented.

**Parent topic:** Fourier Transform Functions

### 13.2.3.1.5 `onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::commit`

Performs all initialization for the actual FFT computation.

#### Syntax

```
onemkl::dft::ErrCode Commit (cl::sycl::queue &in)
```

#### Include Files

- mkl\_dfti\_sycl.hpp

#### Description

This function completes initialization of a previously created descriptor, which is required before the descriptor can be used for FFT computations. The cl::sycl::queue may be associated with a host, CPU, or GPU device. Typically, committing the descriptor performs all initialization that is required for the actual FFT computation on the given device. The initialization done by the function may involve exploring different factorizations of the input length to find the optimal computation method.

#### Note

All calls to the `onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::setValue` function to change configuration parameters of a descriptor need to happen after `onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::init` and before `onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::commit`. Typically, a committal function call is immediately followed by a computation function call (see [FFT Compute Functions](#)).

The function returns `onemkl::dft::ErrCode::NO_ERROR` when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

#### Input Parameters

Name	Type	Description
<i>deviceQueue</i>	cl::sycl::queue	Sycl queue for a host, CPU, or GPU device.

#### Output Parameters

Name	Type	Description
<i>status</i>	<code>onemkl::dft::ErrCode</code>	Function completion status.

## Return Values

The function returns a value indicating whether the operation was successful or not, and why.

Return Value	Description
onemkl::dft::ErrCode::NO_ERROR	The operation was successful.
onemkl::dft::ErrCode::BAD_DESCRIPTOR	DFTI handle provided is invalid.
onemkl::dft::ErrCode::INCONSISTENT_CONFIGURATION/ onemkl::dft::ErrCode::INVALID_CONFIGURATION	An input value provided is invalid.
onemkl::dft::ErrCode::UNIMPLEMENTED	Functionality requested is not implemented.
onemkl::dft::ErrCode::MKL_INTERNAL_ERROR	Internal MKL error.

**Parent topic:** Fourier Transform Functions

### 13.2.3.1.6 onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::computeForward<typename IOType>

Computes the forward FFT.

#### Syntax

```
onemkl::dft::ErrCode computeForward(cl::sycl::buffer<IOType, 1> &inout, cl::sycl::event *event = nullptr)
onemkl::dft::ErrCode computeForward(cl::sycl::buffer<IOType, 1> &in, cl::sycl::buffer<IOType, 1> &out,
                                         cl::sycl::event *event = nullptr)
```

#### Include Files

- mkl\_dfti\_sycl.hpp

#### Description

The `onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::computeForward<typename IOType>` function accepts one or more data parameters. With a successfully configured and committed descriptor, this function computes the forward FFT, that is, the `transform` with the minus sign in the exponent,  $\delta = -1$ .

The FFT descriptor must be properly configured prior to the function call.

The number of the data parameters that the function requires may vary depending on the configuration of the descriptor. This variation is accommodated by variable parameters.

Function calls needed to configure an FFT descriptor for a particular call to an FFT computation function are summarized in [Configuring and Computing an FFT in C/C++](#).

The function returns `onemkl::dft::ErrCode::NO_ERROR` when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

## Input Parameters

Name	Type	Description
in-out, in	cl::sycl::buffer<IOType>	buffer containing an array of length no less than specified at onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::init call.

The suffix in parameter names corresponds to the value of the configuration parameter onemkl::dft::ConfigParam::PLACEMENT as follows:

- inout to DFTI\_INPLACE
- in to DFTI\_NOT\_INPLACE

## Output Parameters

Name	Type	Description
in-out, in	cl::sycl::buffer<IOType>	buffer containing an array of length no less than specified at onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::init call.
Event	cl::sycl::event*	If no event pointer is passed, then it defaults to nullptr. If a non-nullptr value is passed, it is set to the event associated with the enqueued computation job.
Status	onemkl::dft::ErrCode	Function completion status.

The suffix in parameter names corresponds to the value of the configuration parameter onemkl::dft::PLACEMENT as follows:

- inout to DFTI\_INPLACE
- out to DFTI\_NOT\_INPLACE

## Return Values

The function returns a value indicating whether the operation was successful or not, and why.

Return Value	Description
onemkl::dft::ErrCode::NO_ERROR	The operation was successful.
onemkl::dft::ErrCode::BAD_DESCRIPTOR	DFTI handle provided is invalid.
onemkl::dft::ErrCode::INCONSISTENT_CONFIGURATION/ onemkl::dft::ErrCode::INVALID_CONFIGURATION	An input value provided is invalid.
onemkl::dft::ErrCode::UNIMPLEMENTED	Functionality requested is not implemented.
onemkl::dft::ErrCode::MEMORY_ERROR	Internal memory allocation failed.

**Parent topic:** Fourier Transform Functions

### 13.2.3.1.7 `onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::computeBackward<typename IOType>`

Computes the backward FFT.

#### Syntax

```
onemkl::dft::ErrCode computeBackward (cl::sycl::buffer<IOType, 1> &inout, cl::sycl::event *event = nullptr)
onemkl::dft::ErrCode computeBackward (cl::sycl::buffer<IOType, 1> &in, cl::sycl::buffer<IOType, 1> &out, cl::sycl::event *event = nullptr)
```

#### Include Files

- mkl\_dfti\_sycl.hpp

#### Description

The `onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::computeBackward<typename IOType>` function accepts the descriptor handle parameter and one or more data parameters. Given a successfully configured and committed descriptor, this function computes the backward FFT, that is, the `transform` with the minus sign in the exponent,  $\delta = -1$ .

The FFT descriptor must be properly configured prior to the function call.

The number of the data parameters that the function requires may vary depending on the configuration of the descriptor. This variation is accommodated by variable parameters.

Function calls needed to configure an FFT descriptor for a particular call to an FFT computation function are summarized in [Configuring and Computing an FFT in C/C++](#).

The function returns `onemkl::dft::ErrCode::NO_ERROR` when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

#### Input Parameters

Name	Type	Description
<code>in-out</code> , <code>in</code>	<code>cl::sycl::buffer&lt;IOType, 1&gt;</code>	<code>cl::sycl::buffer</code> containing an array of length no less than specified at <code>onemkl::dft::Descriptor&lt;onemkl::dft::Precision, onemkl::dft::Domain&gt;::init</code> call.

The suffix `in` parameter names corresponds to the value of the configuration parameter `onemkl::dft::ConfigParam::PLACEMENT` as follows:

- `inout` to DFTI\_INPLACE
- `in` to DFTI\_NOT\_INPLACE

## Output Parameters

Name	Type	Description
in-out, out	cl::sycl::buffer<IOType, 1>	buffer containing an array of length no less than specified at onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::init call.
Event	cl::sycl::event*	If no event pointer is passed, then it defaults to nullptr. If a non-nullptr value is passed, it is set to the event associated with the enqueued computation job.
Status	onemkl::dft::ErrCode	Completion status.

The suffix in parameter names corresponds to the value of the configuration parameter onemkl::dft::PLACEMENT as follows:

- inout to DFTI\_INPLACE
- out to DFTI\_NOT\_INPLACE

## Return Values

The function returns a value indicating whether the operation was successful or not, and why.

Return Value	Description
onemkl::dft::ErrCode::NO_ERROR	The operation was successful.
onemkl::dft::ErrCode::BAD_DESCRIPTOR	DFTI handle provided is invalid.
onemkl::dft::ErrCode::INCONSISTENT_CONFIGURATION/ onemkl::dft::ErrCode::INVALID_CONFIGURATION	An input value provided is invalid.
onemkl::dft::ErrCode::UNIMPLEMENTED	Functionality requested is not implemented.
onemkl::dft::ErrCode::MEMORY_ERROR	Internal memory allocation failed.

**Parent topic:** Fourier Transform Functions

### 13.2.4 Random Number Generators

Statistics Random Number Generators provide a set of routines implementing commonly used pseudorandom, quasi-random, and non-deterministic generators with continuous and discrete distributions.

#### Definitions

Pseudo-random number generator is defined by a structure  $(S, \mu, f, U, g)$ , where:

- $S$  is a finite set of states (the state space)
- $\mu$  is a probability distribution on  $S$  for the initial state (or seed)  $s_0$
- $f : S \rightarrow S$  is the transition function
- $U$  – a finite set of output symbols
- $g : S \rightarrow U$  an output function

The generation of random numbers is as follows:

1. Generate the initial state (called the seed)  $s_0$  according to  $\mu$  and compute  $u_0 = g(s_0)$ .
2. Iterate for  $i=1, \dots, s_i = f(s_{i-1})$  and  $u_i = g(s_i)$ . Output values  $u_i$  are the so-called random numbers produced by the PRNG.

In computational statistics, random variate generation is usually made in two steps:

1. Generating imitations of independent and identically distributed (i.i.d.) random variables having the uniform distribution over the interval  $(0, 1)$
2. Applying transformations to these i.i.d.  $U(0, 1)$  random variates in order to generate (or imitate) random variates and random vectors from arbitrary distributions.

All RNG routines can be classified into several categories:

- Engines (Basic random number generators) classes, which holds state of generator and is a source of i.i.d. random. Refer to [Engines \(Basic Random Number Generators\)](#) for a detailed description.
- Transformation classes for different types of statistical distributions, for example, uniform, normal (Gaussian), binomial, etc. These classes contain all of the distribution's parameters (including generation method). Refer to [Distribution Generators](#) for a detailed description of the distributions.
- Generate function. The current routine is used to obtain random numbers from a given engine with proper statistics defined by a given distribution. Refer to the [Generate Routine](#) section for a detailed description.
- Service routines to modify the engine state: skip ahead and leapfrog functions. Refer to [Service Routines](#) for a description of these routines.
- [oneMKL RNG Usage Model](#)

#### 13.2.4.1 oneMKL RNG Usage Model

A typical algorithm for random number generators is as follows:

1. Create and initialize the object for basic random number generator.
  - Use the skip\_ahead or leapfrog function if it is required (used in parallel with random number generation for Host and CPU devices).
2. Create and initialize the object for distribution generator.
3. Call the generate routine to get random numbers with appropriate statistical distribution.

The following example demonstrates generation of random numbers that is output of basic generator (engine) PHILOX4X32X10. The seed is equal to 777. The generator is used to generate 10,000 normally distributed random numbers with parameters  $a = 5$  and  $\sigma = 2$ . The purpose of the example is to calculate the sample mean for normal distribution with the given parameters.

#### Example of RNG Usage

##### Buffer API

```
#include <iostream>
#include <vector>

#include "CL/sycl.hpp"
#include "mkl_rng_sycl.hpp"
```

(continues on next page)

(continued from previous page)

```

int main() {
    cl::sycl::queue queue;

    const size_t n = 10000;
    std::vector<double> r(n);

    onemkl::rng::philox4x32x10 engine(queue, SEED); // basic random number generator
    ↪object
    onemkl::rng::gaussian<double, onemkl::rng::box_muller2> distr(5.0, 2.0); // ↪
    ↪distribution object

    {
        //create buffer for random numbers
        cl::sycl::buffer<double, 1> r_buf(r.data(), cl::sycl::range<1>{n});

        onemkl::rng::generate(distr, engine, n, r_buf); // perform generation
    }

    double s = 0.0;
    for(int i = 0; i < n; i++) {
        s += r[i];
    }
    s /= n;

    std::cout << "Average = " << s << std::endl;

    return 0;
}

```

## USM API

```

#include <iostream>
#include <vector>
#include "CL/sycl.hpp"
#include "mkl_rng_sycl.hpp"

int main() {
    cl::sycl::queue queue;

    const size_t n = 10000;

    // create USM allocator
    cl::sycl::usm_allocator<double, cl::sycl::usm::alloc::shared> allocator(queue.get_
    ↪context(), queue.get_device());                                         (continues on next page)

```

(continued from previous page)

```

// create vector woth USM allocator
std::vector<double, cl::sycl::usm_allocator<double, cl::sycl::usm::alloc::shared>> r;
r(n, allocator);

onemkl::rng::philox4x32x10 engine(queue, SEED); // basic random number generator
object
onemkl::rng::gaussian<double, onemkl::rng::box_muller2> distr(5.0, 2.0); // distribution object

auto event = onemkl::rng::generate(distr, engine, n, r.data()); // perform generation
// cl::sycl::event object is returned by generate function for synchronisation
event.wait(); // synchronization can be also done by queue.wait()

double s = 0.0;
for(int i = 0; i < n; i++) {
    s += r[i];

    std::cout << "Average = " << s << std::endl;
}

return 0;
}

```

You can also use USM with raw pointers by using the `cl::sycl::malloc_shared` function.

**Parent topic:** *Random Number Generators*

#### 13.2.4.2 Generate Routine

- `onemkl::rng::generate` Entry point to obtain random numbers from a given engine with proper statistics of a given distribution.

**Parent topic:** *Random Number Generators*

### 13.2.4.2.1 onemkl::rng::generate

Entry point to obtain random numbers from a given engine with proper statistics of a given distribution.

#### Syntax

Buffer API

```
template<typename T, method Method, template<typename, method> class Distr, typename EngineType>
void generate(const Distr<T, Method> &distr, EngineType &engine, const std::int64_t n,
              cl::sycl::buffer<T, 1> &r)
```

USM API

```
template<typename T, method Method, template<typename, method> class Distr, typename EngineType>
cl::sycl::event generate(const Distr<T, Method> &distr, EngineType &engine, const std::int64_t n, T
                        *r, const cl::sycl::vector_class<cl::sycl::event> &dependencies)
```

#### Include Files

- mkl\_sycl.hpp

#### Input Parameters

Name	Type	Description
distr	const Distr<T, Method>	Distribution object. See <a href="#">Distributions</a> for details.
engine	EngineType	Engine object. See <a href="#">Engines</a> for details.
n	const std::int64_t	Number of random values to be generated.

#### Output Parameters

Buffer API

Name	Type	Description
r	cl::sycl::buffer<T, 1>	cl::sycl::buffer r to the output vector.

USM API

Name	Type	Description
r	T*	Pointer r to the output vector.
event	cl::sycl::event	Function return event after submitting task in cl::sycl::queue from the engine.

onemkl::rng::generate submits a kernel into a queue that is held by the engine and fills cl::sycl::buffer/T\* vector with n random numbers.

**Parent topic:** Generate Routine

### 13.2.4.3 Engines (Basic Random Number Generators)

oneMKL RNG provides pseudorandom, quasi-random, and non-deterministic random number generators for Data Parallel C++:

Routine	Description
onemkl::rng::mrg32k3a	The combined multiple recursive pseudorandom number generator MRG32k3a [L'Ecuyer99a]
onemkl::rng::philox4x32x10	Philox4x32-10 counter-based pseudorandom number generator with a period of $2^{128}$ PHILOX4X32X10 [Salmon11]
onemkl::rng::mcg31m1	The 31-bit multiplicative congruential pseudorandom number generator MCG(1132489760, 231 -1) [L'Ecuyer99]
onemkl::rng::r250	The 32-bit generalized feedback shift register pseudorandom number generator GFSR(250, 103) [Kirkpatrick81]
onemkl::rng::mcg59	The 59-bit multiplicative congruential pseudorandom number generator MCG(13 <sup>13</sup> , 2 <sup>59</sup> ) from NAG Numerical Libraries [NAG]
onemkl::rng::wichmannHill	Wichmann-Hill pseudorandom number generator (a set of 273 basic generators) from NAG Numerical Libraries [NAG]
onemkl::rng::mt19937	Mersenne Twister pseudorandom number generator MT19937 [Matsumoto98] with period length $2^{19937}-1$ of the produced sequence
onemkl::rng::mt2203	Set of 6024 Mersenne Twister pseudorandom number generators MT2203 [Matsumoto98], [Matsumoto00]. Each of them generates a sequence of period length equal to $2^{2203}-1$ . Parameters of the generators provide mutual independence of the corresponding sequences.
onemkl::rng::sfmt19937	SIMD-oriented Fast Mersenne Twister pseudorandom number generator SFMT19937 [Saito08] with a period length equal to $2^{19937}-1$ of the produced sequence.
onemkl::rng::sobol	Sobol quasi-random number generator [Sobol76], [Bratley88], which works in arbitrary dimension.
onemkl::rng::niederreiter	Niederreiter quasi-random number generator [Bratley92], which works in arbitrary dimension.
onemkl::rng::ars5	ARS-5 counter-based pseudorandom number generator with a period of $2^{128}$ , which uses instructions from the AES-NI set ARS5 [Salmon11].
onemkl::rng::ndrnd	Non-deterministic random number generator (RDRAND-based) [AVX][IntelSWMan]

For some basic generators, oneMKL RNG provides two methods of creating independent states in multiprocessor computations, which are the leapfrog method and the block-splitting method. These sequence splitting methods are also useful in sequential Monte Carlo. The description of these functions can be found in the [Service Routines](#) section.

In addition, MT2203 pseudorandom number generator is a set of 6024 generators designed to create up to 6024 independent random sequences, which might be used in parallel Monte Carlo simulations. Another generator that has the same feature is Wichmann-Hill. It allows creating up to 273 independent random streams. The properties of the generators designed for parallel computations are discussed in detail in [Coddington94].

See [VS Notes](#) for the detailed description.

**Parent topic:** [Random Number Generators](#)

- [onemkl::rng::mrg32k3a](#) The combined multiple recursive pseudorandom number generator MRG32k3a [L'Ecuyer99a]
- [onemkl::rng::philox4x32x10](#) A Philox4x32-10 counter-based pseudorandom number generator. [Salmon11].
- [onemkl::rng::mcg31m1](#) The 31-bit multiplicative congruential pseudorandom number generator MCG(1132489760, 231 -1) [L'Ecuyer99]
- [onemkl::rng::mcg59](#) The 59-bit multiplicative congruential pseudorandom number generator MCG(13<sup>13</sup>, 2<sup>59</sup>) from NAG Numerical Libraries [NAG].
- [onemkl::rng::r250](#) The 32-bit generalized feedback shift register pseudorandom number generator GFSR(250,103)[Kirkpatrick81].

- `onemkl::rng::wichmann_hill` Wichmann-Hill pseudorandom number generator (a set of 273 basic generators) from NAG Numerical Libraries [NAG].
- `onemkl::rng::mt19937` Mersenne Twister pseudorandom number generator MT19937 [Matsumoto98] with period length  $2^{19937}-1$  of the produced sequence.
- `onemkl::rng::sfmt19937` SIMD-oriented Fast Mersenne Twister pseudorandom number generator SFMT19937 [Saito08] with a period length equal to  $2^{19937}-1$  of the produced sequence.
- `onemkl::rng::mt2203` Set of 6024 Mersenne Twister pseudorandom number generators MT2203 [Matsumoto98], [Matsumoto00]. Each of them generates a sequence of period length equal to  $2^{2203}-1$ . Parameters of the generators provide mutual independence of the corresponding sequences..
- `onemkl::rng::ars5` ARS-5 counter-based pseudorandom number generator with a period of  $2^{128}$ , which uses instructions from the AES-NI set ARS5[Salmon11].
- `onemkl::rng::sobol` Sobol quasi-random number generator [Sobol76], [Bratley88], which works in arbitrary dimension.
- `onemkl::rng::niederreiter` Niederreiter quasi-random number generator [Bratley92], which works in arbitrary dimension.
- `onemkl::rng::nondeterministic` Non-deterministic random number generator (RDRAND-based) [AVX][IntelSWMan].

### 13.2.4.3.1 `onemkl::rng::mrg32k3a`

The combined multiple recursive pseudorandom number generator MRG32k3a [ L'Ecuyer99a]

#### Syntax

```
class mrg32k3a: public internal::engine_base<mrg32k3a>{
    mrg32k3a (cl::sycl::queue& queue, std::initializer_list<std::uint32_t> seed)
    mrg32k3a (const mrg32k3a& other)
    mrg32k3a& operator=(const mrg32k3a& other)
    mrg32k3a()
}
```

#### Include Files

- `mkl_sycl.hpp`

#### Description

The combined multiple recursive pseudorandom number generator MRG32k3a [L'Ecuyer99a].

## Input Parameters

Name	Type	Description
queue	cl::sycl::queue	Valid cl::sycl::queue, calls of the onemkl::rng::generate() routine submits kernels in this queue.
seed	std::uint32_t / std::initializer_list<std::uint32_t>	Initial conditions of the generator state or engine state.

See [VS Notes](#) for detailed descriptions.

**Parent topic:** Engines (Basic Random Number Generators)

### 13.2.4.3.2 onemkl::rng::philox4x32x10

A Philox4x32-10 counter-based pseudorandom number generator. [Salmon11].

## Syntax

```
class philox4x32x10 : internal::engine_base<philox4x32x10>{
public:
    philox4x32x10 (cl::sycl::queue& queue,           std::uint64_t seed)
    philox4x32x10 (cl::sycl::queue& queue,           std::initializer_list<std::uint64_t>_
    ↵seed)
    philox4x32x10 (const philox4x32x10& other)
    philox4x32x10& operator=(const philox4x32x10& other)
    ~philox4x32x10()
}
```

## Include Files

- mkl\_sycl.hpp

## Description

A Philox4x32-10 counter-based pseudorandom number generator. [Salmon11].

## Input Parameters

Name	Type	Description
queue	cl::sycl::queue	Valid cl::sycl::queue, calls of the onemkl::rng::generate() routine submits kernels in this queue.
seed	std::uint64_t / std::initializer_list<std::uint64_t>	Initial conditions of the generator state or engine state.

See [VS Notes](#) for detailed descriptions.

**Parent topic:** Engines (Basic Random Number Generators)

### 13.2.4.3.3 onemkl::rng::mcg31m1

The 31-bit multiplicative congruential pseudorandom number generator MCG(1132489760, 231 -1) [L'Ecuyer99]

#### Syntax

```
class mcg31m1 : internal::engine_base<mcg31m1>{
public:
    mcg31m1 (cl::sycl::queue& queue, std::uint32_t seed)
    mcg31m1 (const mcg31m1& other)
    mcg31m1& operator=(const mcg31m1& other)
    ~mcg31m1()
}
```

#### Include Files

- mkl\_sycl.hpp

#### Description

The 31-bit multiplicative congruential pseudorandom number generator MCG(1132489760, 231 -1) [[L'Ecuyer99]].

#### Input Parameters

Name	Type	Description
queue	cl::sycl::queue	Valid cl::sycl::queue, calls of the onemkl::rng::generate() routine submits kernels in this queue.
seed	std::uint32_t	Initial conditions of the engine.

See VS Notes for detailed descriptions.

**Parent topic:** Engines (Basic Random Number Generators)

### 13.2.4.3.4 onemkl::rng::mcg59

The 59-bit multiplicative congruential pseudorandom number generator MCG(1313, 259) from NAG Numerical Libraries [NAG].

#### Syntax

```
class mcg59 : internal::engine_base<mcg59>{
public:
    mcg59 (cl::sycl::queue& queue, std::uint64_t seed)
    mcg59 (const mcg59& other)
    mcg59& operator=(const mcg59& other)
    ~mcg59()
}
```

## Include Files

- mkl\_sycl.hpp

## Description

The 59-bit multiplicative congruential pseudorandom number generator MCG(1313, 259) from NAG Numerical Libraries [NAG].

## Input Parameters

Name	Type	Description
queue	cl::sycl::queue	Valid cl::sycl::queue, calls of onemkl::rng::generate() routine submit kernels in this queue.
seed	std::uint64_t	Initial conditions of the engine.

See [VS Notes](#) for detailed descriptions.

**Parent topic:** Engines (Basic Random Number Generators)

### 13.2.4.3.5 onemkl::rng::r250

The 32-bit generalized feedback shift register pseudorandom number generator GFSR(250,103)[Kirkpatrick81].

## Syntax

```
class r250 : internal::engine_base<r250>{
public:
    r250 (cl::sycl::queue& queue, std::uint32_t           seed)
    r250 (cl::sycl::queue& queue,           std::initializer_list<std::uint32_t> seed)
    r250 (const r250& other)
    r250& operator=(const r250& other)
    ~r250 ()
}
```

## Include Files

- mkl\_sycl.hpp

## Description

The 32-bit generalized feedback shift register pseudorandom number generator GFSR(250,103) [Kirkpatrick81].

## Input Parameters

Name	Type	Description
queue	cl::sycl::queue	Valid cl::sycl::queue, calls of onemkl::rng::generate() routine submit kernels in this queue.
seed	std::uint32_t / std::initializer_list<std::uint32_t>	Initial conditions of the engine.

See [VS Notes](#) for detailed descriptions.

**Parent topic:** Engines (Basic Random Number Generators)

### 13.2.4.3.6 onemkl::rng::wichmann\_hill

Wichmann-Hill pseudorandom number generator (a set of 273 basic generators) from NAG Numerical Libraries [NAG].

## Syntax

```
class wichmann_hill : internal::engine_base<wichmann_hill>{
public:
    wichmann_hill (cl::sycl::queue& queue,           std::uint32_t seed, std::uint32_t_
    ↵engine_idx)
    wichmann_hill (cl::sycl::queue& queue,           std::initializer_list<std::uint32_t>_
    ↵seed, std::uint32_t      engine_idx)
    wichmann_hill (const wichmann_hill& other)
    wichmann_hill& operator=(const wichmann_hill&     other)
    ~wichmann_hill()
}
```

## Include Files

- mkl\_sycl.hpp

## Description

Wichmann-Hill pseudorandom number generator (a set of 273 basic generators) from NAG Numerical Libraries [NAG].

## Input Parameters

Name	Type	Description
queue	cl::sycl::queue	Valid cl::sycl::queue, calls of onemkl::rng::generate() routine submit kernels in this queue.
seed	std::uint32_t / std::initializer_list<std::uint32_t>	Initial conditions of the engine.
en- gine_idx	std::uint32_t	Index of the engine from the set (set contains 273 basic generators)

See [VS Notes](#) for detailed descriptions.

**Parent topic:** Engines (Basic Random Number Generators)

### 13.2.4.3.7 onemkl::rng::mt19937

Mersenne Twister pseudorandom number generator MT19937 [Matsumoto98] with period length  $2^{19937}-1$  of the produced sequence.

#### Syntax

```
class mt19937 : internal::engine_base<mt19937>{
public:
    mt19937 (cl::sycl::queue& queue, std::uint32_t seed)
    mt19937 (cl::sycl::queue& queue, std::initializer_list<std::uint32_t> seed)
    mt19937 (const mt19937& other)
    mt19937& operator=(const mt19937& other)
    ~mt19937()
}
```

#### Include Files

- mkl\_sycl.hpp

#### Description

Mersenne Twister pseudorandom number generator MT19937 [Matsumoto98] with period length  $2^{19937}-1$  of the produced sequence.

#### Input Parameters

Name	Type	Description
queue	cl::sycl::queue	Valid cl::sycl::queue, calls of onemkl::rng::generate() routine submit kernels in this queue.
seed	std::uint32_t / std::initializer_list<std::uint32_t>	Initial conditions of the engine.

See [VS Notes](#) for detailed descriptions.

**Parent topic:** Engines (Basic Random Number Generators)

### 13.2.4.3.8 onemkl::rng::sfmt19937

SIMD-oriented Fast Mersenne Twister pseudorandom number generator SFMT19937 [Saito08] with a period length equal to  $2^{19937}-1$  of the produced sequence.

#### Syntax

```
class sfmt19937 : public internal::engine_base<sfmt19937>{
    sfmt19937 (cl::sycl::queue& queue, std::uint32_t seed)
    sfmt19937 (cl::sycl::queue& queue, std::initializer_list<std::uint32_t> seed)
    sfmt19937 (const sfmt19937& other)
    sfmt19937& operator=(const sfmt19937& other)
    ~sfmt19937 ()
}
```

#### Include Files

- mkl\_sycl.hpp

#### Description

SIMD-oriented Fast Mersenne Twister pseudorandom number generator SFMT19937 [Saito08] with a period length equal to  $2^{19937}-1$  of the produced sequence.

#### Input Parameters

Name	Type	Description
queue	cl::sycl::queue	Valid sycl queue, calls of onemkl::rng::generate() routine submit kernels in this queue.
seed	std::uint32_t / std::initializer_list<std::uint32_t>	Initial conditions of the engine.

See [VS Notes](#) for detailed descriptions.

**Parent topic:** Engines (Basic Random Number Generators)

### 13.2.4.3.9 onemkl::rng::mt2203

Set of 6024 Mersenne Twister pseudorandom number generators MT2203 [Matsumoto98], [Matsumoto00]. Each of them generates a sequence of period length equal to  $2^{2203}-1$ . Parameters of the generators provide mutual independence of the corresponding sequences..

## Syntax

```
class mt2203 : internal::engine_base<mt2203>{
public:
    mt2203 (cl::sycl::queue& queue, std::uint32_t           seed, std::uint32_t engine_
             ↪idx)
    mt2203 (cl::sycl::queue& queue,           std::initializer_list<std::uint32_t> seed, ↪
             ↪std::uint32_t           engine_idx)
    mt2203 (const mt2203& other)
    mt2203& operator=(const mt2203& other)
    ~mt2203()
}
```

## Include Files

- mkl\_sycl.hpp

## Description

Set of 6024 Mersenne Twister pseudorandom number generators MT2203 [Matsumoto98], [Matsumoto00]. Each of them generates a sequence of period length equal to  $2^{2203}-1$ . Parameters of the generators provide mutual independence of the corresponding sequences..

## Input Parameters

Name	Type	Description
queue	cl::sycl::queue	Valid cl::sycl::queue, calls of onemkl::rng::generate() routine submit kernels in this queue.
seed	std::uint32_t / std::initializer_list<std::uint32_t>	Initial conditions of the engine.
en- gine_idx	std::uint32_t	Index of the engine from the set (set contains 6024 basic generators).

See [VS Notes](#) for detailed descriptions.

**Parent topic:** Engines (Basic Random Number Generators)

### 13.2.4.3.10 onemkl::rng::ars5

ARS-5 counter-based pseudorandom number generator with a period of  $2^{128}$ , which uses instructions from the AES-NI set ARS5[Salmon11].

## Syntax

```
class ars5 : internal::engine_base<ars5>{
public:
    ars5 (cl::sycl::queue& queue, std::uint64_t seed)
    ars5 (cl::sycl::queue& queue, std::initializer_list<std::uint64_t> seed)
    ars5 (const ars5& other)
    ars5& operator=(const ars5& other)
    ~ars5()
}
```

## Include Files

- mkl\_sycl.hpp

## Description

ARS-5 counter-based pseudorandom number generator with a period of  $2^{128}$ , which uses instructions from the AES-NI set ARS5[Salmon11].

## Input Parameters

Name	Type	Description
queue	cl::sycl::queue	Valid cl::sycl::queue, calls of onemkl::rng::generate() routine submit kernels in this queue.
seed	std::uint64_t / std::initializer_list<std::uint64_t>	Initial conditions of the engine.

See [VS Notes](#) for detailed descriptions.

**Parent topic:** Engines (Basic Random Number Generators)

### 13.2.4.3.11 onemkl::rng::sobol

Sobol quasi-random number generator [Sobol76], [Bratley88], which works in arbitrary dimension.

## Syntax

```
class sobol : internal::engine_base<sobol>{
public:
    sobol (cl::sycl::queue& queue, std::uint32_t dimensions)
    sobol (cl::sycl::queue& queue, std::vector<std::uint32_t> direction_numbers)
    sobol (const sobol& other)
    sobol& operator=(const sobol& other)
    ~sobol()
}
```

## Include Files

- mkl\_sycl.hpp

## Description

Sobol quasi-random number generator [Sobol76], [Bratley88], which works in arbitrary dimension.

## Input Parameters

Name	Type	Description
queue	cl::sycl::queue	Valid cl::sycl::queue, calls of onemkl::rng::generate() routine submit kernels in this queue.
dimensions	std::uint32_t	Number of dimensions.
direction_numbers	std::vector<std::uint32_t>	User-defined direction numbers.

See [VS Notes](#) for detailed descriptions.

**Parent topic:** Engines (Basic Random Number Generators)

### 13.2.4.3.12 onemkl::rng::niederreiter

Niederreiter quasi-random number generator [Bratley92], which works in arbitrary dimension.

## Syntax

```
class niederreiter : public internal::engine_base<niederreiter>{
    niederreiter (cl::sycl::queue& queue, std::uint32_t dimensions)
    niederreiter (cl::sycl::queue& queue, std::vector<std::uint32_t> irred_
    ↵polynomials)
    niederreiter (const niederreiter& other)
    niederreiter& operator=(const niederreiter& other)
    ~niederreiter()
}
```

## Include Files

- mkl\_sycl.hpp

## Description

Niederreiter quasi-random number generator [Bratley92], which works in arbitrary dimension.

## Input Parameters

Name	Type	Description
queue	cl::sycl::queue	Valid cl::sycl::queue, calls of onemkl::rng::generate() routine submit kernels in this queue.
dimensions	std::uint32_t	Number of dimensions.
ir-red_polynomials	std::vector<std::uint32_t>	User-defined direction numbers.

See [VS Notes](#) for detailed descriptions.

**Parent topic:** Engines (Basic Random Number Generators)

### 13.2.4.3.13 onemkl::rng::nondeterministic

Non-deterministic random number generator (RDRAND-based) [AVX][IntelSWMan].

## Syntax

```
class nondeterministic : public internal::engine_base<nondeterministic>{
    nondeterministic(cl::sycl::queue& queue)
    nondeterministic(const nondeterministic& other)
    nondeterministic& operator=(const nondeterministic& other)
    ~nondeterministic()
}
```

## Include Files

- mkl\_sycl.hpp

## Description

Non-deterministic random number generator (RDRAND-based) [AVX][IntelSWMan].

## Input Parameters

Name	Type	Description
queue	cl::sycl::queue	Valid cl::sycl::queue, calls of onemkl::rng::generate() routine submits kernels in this queue.

See [VS Notes](#) for detailed descriptions.

**Parent topic:** Engines (Basic Random Number Generators)

### 13.2.4.4 Service Routines

Routine	Description
<code>onemkl::rng::leapfrog</code>	Proceed state of engine by the leapfrog method to generate a subsequence of the original sequence
<code>onemkl::rng::skip_ahead</code>	Proceed state of engine by the skip-ahead method to skip a given number of elements from the original sequence

**Parent topic:** *Random Number Generators*

#### 13.2.4.4.1 `onemkl::rng::leapfrog`

Proceed state of engine using the leapfrog method.

##### Syntax

```
template<typename EngineType>
void leapfrog (EngineType &engine, std::uint64_t idx, std::uint64_t stride)
```

##### Include Files

- `mkl_sycl.hpp`

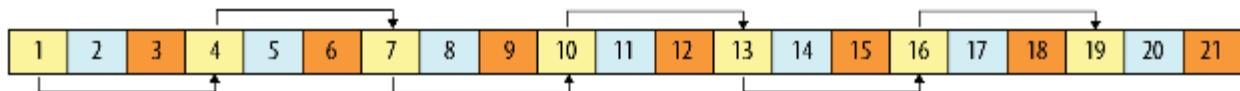
##### Input Parameters

Name	Type	Description
<code>engine</code>	<code>EngineType</code>	Object of engine class, which supports leapfrog.
<code>idx</code>	<code>std::uint64_t</code>	Index of the computational node.
<code>stride</code>	<code>std::uint64_t</code>	Largest number of computational nodes, or stride.

##### Description

The `onemkl::rng::leapfrog` function generates random numbers in an engine with non-unit stride. This feature is particularly useful in distributing random numbers from the original stream across the stride buffers without generating the original random sequence with subsequent manual distribution.

One of the important applications of the leapfrog method is splitting the original sequence into non-overlapping subsequences across stride computational nodes. The function initializes the original random stream (see *Figure “Leapfrog Method”*) to generate random numbers for the computational node `idx`,  $0 \leq \text{idx} < \text{stride}$ , where `stride` is the largest number of computational nodes used.



At the 1st node, the engine generates: 1, 4, 7, 10, 13, 16, 19.

At the 2nd node, the engine generates: 2, 5, 8, 11, 14, 17, 20.

At the 3rd node, the engine generates: 3, 6, 9, 12, 15, 18, 21.

#### Legend:

- 1st node engine
- 2nd node engine
- 3rd node engine

#### Leapfrog Method

The leapfrog method is supported only for those basic generators that allow splitting elements by the leapfrog method, which is more efficient than simply generating them by a generator with subsequent manual distribution across computational nodes. See [VS Notes](#) for details.

The following code illustrates the initialization of three independent streams using the leapfrog method:

#### Code for Leapfrog Method

```
...
// Creating 3 identical engines
onemkl::rng::mcg31ml engine_1(queue, seed);

onemkl::rng::mcg31ml engine_2(queue, engine_1);
onemkl::rng::mcg31ml engine_3(queue, engine_1);

// Leapfrogging the states of engines
onemkl::rng::leapfrog(engine_1, 0, 3);
onemkl::rng::leapfrog(engine_2, 1, 3);
onemkl::rng::leapfrog(engine_3, 2, 3);
// Generating random numbers
...
```

**Parent topic:** Service Routines

#### 13.2.4.4.2 onemkl::rng::skip\_ahead

Proceed state of engine by the skip-ahead method.

## Syntax

The onemkl::rng::skip\_ahead function supports the following interfaces to apply the skip-ahead method:

- Common interface
- Interface with a partitioned number of skipped elements

### Common Interface

```
template<typename EngineType>
void skip_ahead(EngineType &engine, std::uint64_t num_to_skip)
```

### Interface with Partitioned Number of Skipped Elements

```
template<typename EngineType>
void skip_ahead(EngineType &engine, std::initializer_list<std::uint64_t> num_to_skip)
```

## Include Files

- mkl\_sycl.hpp

## Input Parameters

### Common Interface

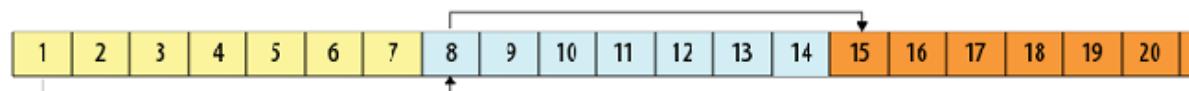
Name	Type	Description
engine	EngineType	Object of engine class, which supports the block-splitting method.
num_to_skip	std::uint64_t	Number of skipped elements.

### Interface with Partitioned Number of Skipped Elements

Name	Type	Description
engine	EngineType	Object of engine class, which supports the block-splitting method.
num_to_skip	std::initializer_list<std::uint64_t>	Partitioned number of skipped elements.

## Description

The onemkl::rng::skip\_ahead function skips a given number of elements in a random sequence provided by engine. This feature is particularly useful in distributing random numbers from original engine across different computational nodes. If the largest number of random numbers used by a computational node is num\_to\_skip, then the original random sequence may be split by onemkl::rng::skip\_ahead into non-overlapping blocks of num\_to\_skip size so that each block corresponds to the respective computational node. The number of computational nodes is unlimited. This method is known as the block-splitting method or as the skip-ahead method. (see *Figure “Block-Splitting Method”*).



At the 1st node, the engine generates: 1, 4, 7, 10, 13, 16, 19.

At the 2nd node, the engine generates: 2, 5, 8, 11, 14, 17, 20.

At the 3rd node, the engine generates: 3, 6, 9, 12, 15, 18, 21.

#### Legend:

- 1st node engine
- 2nd node engine
- 3rd node engine

#### Block-Splitting Method

The skip-ahead method is supported only for those basic generators that allow skipping elements by the skip-ahead method, which is more efficient than simply generating them by generator with subsequent manual skipping. See [VS Notes](#) for details.

Please note that for quasi-random basic generators the skip-ahead method works with components of quasi-random vectors rather than with whole quasi-random vectors. Therefore, to skip NS quasi-random vectors, set the num\_to\_skip parameter equal to the num\_to\_skip \*dim, where dim is the dimension of the quasi-random vector.

When the number of skipped elements is greater than  $2^{63}$  the interface with the partitioned number of skipped elements is used. Prior calls to the function represent the number of skipped elements with the list of size n as shown below:

```
num_to_skip[0]+ num_to_skip[1]*264+ num_to_skip[2]* 2128+ ... +num_to_skip[n-1]*264*(n-1);
```

When the number of skipped elements is less than  $2^{63}$  both interfaces can be used.

The following code illustrates how to initialize three independent streams using the onemkl::rng::skip\_ahead function:

#### Code for Block-Splitting Method

```
...
// Creating 3 identical engines
onemkl::rng::mcg31m1 engine_1(queue, seed);
onemkl::rng::mcg31m1 engine_2(queue, engine_1);
onemkl::rng::mcg31m1 engine_3(queue, engine_2);

// Skipping ahead by 7 elements the 2nd engine
onemkl::rng::skip_ahead(engine_2, 7);

// Skipping ahead by 14 elements the 3rd engine
onemkl::rng::skip_ahead(engine_3, 14);
...
```

## Code for Block-Splitting Method with Partitioned Number of Elements

```
// Creating first engine
onemkl::rng::mrg32k3a engine_1(queue, seed);

// To skip  $2^{64}$  elements in the random stream number of skipped elements should be
//represented as num_to_skip =  $2^{64} = 0 + 1 * 2^{64}$ 
std::initializer_list<std::uint64_t> num_to_skip = {0, 1};

// Creating the 2nd engine based on 1st. Skipping by  $2^{64}$ 
onemkl::rng::mrg32k3a engine_2(queue, engine_1);
onemkl::rng::skip_ahead(engine_2, num_to_skip);

// Creating the 3rd engine based on 2nd. Skipping by  $2^{64}$ 
onemkl::rng::mrg32k3a engine_3(queue, engine_2);
onemkl::rng::skip_ahead(engine_3, num_to_skip);
...
```

**Parent topic:** Service Routines

### 13.2.4.5 Distributions

oneAPI Math Kernel Library RNG routines are used to generate random numbers with different types of distribution. Each function group is introduced below by the type of underlying distribution and contains a short description of its functionality, as well as specifications of the call sequence and the explanation of input and output parameters. *Table “Continuous Distribution Generators”* and *Table “Discrete Distribution Generators”* list the random number generator routines with data types and output distributions, and sets correspondence between data types of the generator routines and the basic random number generators.

Type of Distribution	Data Types	BRNG Type	Data	Description
onemkl::rng::uniform	s, d	s, d		Uniform continuous distribution on the interval [a, b)
onemkl::rng::gaussian	s, d	s, d		Normal (Gaussian) distribution
onemkl::rng::exponential	s, d	s, d		Exponential distribution
onemkl::rng::laplace	s, d	s, d		Laplace distribution (double exponential distribution)
onemkl::rng::weibull	s, d	s, d		Weibull distribution
onemkl::rng::cauchy	s, d	s, d		Cauchy distribution
onemkl::rng::rayleigh	s, d	s, d		Rayleigh distribution
onemkl::rng::lognormal	s, d	s, d		Lognormal distribution
onemkl::rng::gumbel	s, d	s, d		Gumbel (extreme value) distribution
onemkl::rng::gamma	s, d	s, d		Gamma distribution
onemkl::rng::beta	s, d	s, d		Beta distribution
onemkl::rng::chi_square	s, d	s, d		Chi-Square distribution

Type of Distribution	Data Types	BRNG Data Type	Description
onemkl::rng::uniform	d		Uniform discrete distribution on the interval $[a, b)$
onemkl::rng::uniform_bits	i		Uniformly distributed bits in 32-bit chunks
	i	i	Uniformly distributed bits in 64-bit chunks
onemkl::rng::bits	i	i	Bits of underlying BRNG integer recurrence
onemkl::rng::bernoulli	s		Bernoulli distribution
onemkl::rng::geometric	s		Geometric distribution
onemkl::rng::binomial	d		Binomial distribution
onemkl::rng::hypergeometric	dc		Hypergeometric distribution
onemkl::rng::poisson		s (for $\lambda \geq 27$ ) onemkl::rng::gaussian_inverse s (for distribution parameter $\lambda < 27$ ) and d (for $\lambda < 27$ ) (for onemkl::rng::ptpe)	Poisson distribution
onemkl::rng::poisson_v	s		Poisson distribution with varying mean
onemkl::rng::negbinomial	d		Negative binomial distribution, or Pascal distribution
onemkl::rng::multinomial	d		Multinomial distribution

## Modes of random number generation

The library provides two modes of random number generation, accurate and fast. Accurate generation mode is intended for the applications that are highly demanding to accuracy of calculations. When used in this mode, the generators produce random numbers lying completely within definitional domain for all values of the distribution parameters. For example, random numbers obtained from the generator of continuous distribution that is uniform on interval  $[a,b]$  belong to this interval irrespective of what  $a$  and  $b$  values may be. Fast mode provides high performance of generation and also guarantees that generated random numbers belong to the definitional domain except for some specific values of distribution parameters. The generation mode is set by specifying relevant value of the method parameter in generator routines. List of distributions that support accurate mode of generation is given in the table below.

**Parent topic:** *Random Number Generators*



### 13.2.4.5.1 Distributions Template Parameter `onemkl::rng::method` Values

<code>onemkl::rng::method</code>	<code>cu- racy Flag</code>	Distributions	Math Description
standard	Yes No No No	uniform(s, d) uniform(i) uniform_bits bits	Standard method. Currently there is only one method for these functions.
box_muller	No	gaussian	Generates normally distributed random number x thru the pair of uniformly distributed numbers u1 and u2 according to the formula: $x = \sqrt{-2 \ln u_1} \sin 2\pi u_2$
box_muller2	Yes	gaussian lognormal	Generates normally distributed random numbers x1 and x2 thru the pair of uniformly distributed numbers u1 and u2 according to the formulas: $x_1 = \sqrt{-2 \ln u_1} \sin 2\pi u_2$ $x_2 = \sqrt{-2 \ln u_1} \cos 2\pi u_2$
inverse_fung	No Yes Yes No Yes Yes No No	gaussian exponential weibull cauchy rayleigh lognormal gumbel bernoulli geometric	Inverse cumulative distribution function method.
marsaglia	Yes	a	gamma For $\alpha > 1$ , a gamma distributed random number is generated as a cube of properly scaled normal random number; for $0.6 \leq \alpha < 1$ , a gamma distributed random number is generated using rejection from Weibull distribution; for $\alpha < 0.6$ , a gamma distributed random number is obtained using transformation of exponential power distribution; for $\alpha = 1$ , gamma distribution is reduced to exponential distribution.
cheng	Yes	johnk atkinson	For $\min(p, q) > 1$ , Cheng method is used; for $\min(p, q) < 1$ , Johnk method is used, if $q + K \cdot p^2 + C \leq 0$ ( $K = 0.852\dots$ , $C = -0.956\dots$ ) otherwise, Atkinson switching algorithm is used; for $\max(p, q) < 1$ , method of Johnk is used; for $\min(p, q) < 1$ , $\max(p, q) > 1$ , Atkinson switching algorithm is used (CJA stands for Cheng, Johnk, Atkinson); for $p = 1$ or $q = 1$ , inverse cumulative distribution function method is used; for $p = 1$ and $q = 1$ , beta distribution is reduced to uniform distribution.
gamma	No	arsaghia square	(most common): If $\nu \geq 17$ or $\nu$ is odd and $5 \leq \nu \leq 15$ , a chi-square distribution is reduced to a Gamma distribution with these parameters: Shape $\alpha = \nu / 2$ Offset $a = 0$ Scale factor $\beta = 2$ The random numbers of the Gamma distribution are generated.
btpe	No	binomial	Acceptance/rejection method for $n_{trial} \cdot \min(p, 1 - p) \geq 30$ with decomposition into four regions:
ptpe	No	poisson	Acceptance/rejection method for $\lambda \geq 27$ with decomposition into four regions:
gauss	No_inversion No	poisson poisson_v	for $\lambda \geq 1$ , method based on Poisson inverse CDF approximation by Gaussian inverse CDF; for $\lambda < 1$ , table lookup method is used.
hypergeometric	No	oneMKL Domains	Acceptance/rejection method for large mode of distribution with decomposition into three regions: $\frac{(a-1) \cdot (1-p)}{n} \geq 100$

## Note

Accuracy flag represented as a method: `onemkl::rng::<method>` | `onemkl::rng::accurate`

**Parent topic:** Distributions

### 13.2.4.5.2 `onemkl::rng::uniform` (Continuous)

Generates random numbers with uniform distribution.

## Syntax

```
template<typename T = float, method Method = standard>
class uniform {
public:
    uniform(): uniform((T)0.0, (T)1.0){}
    uniform(Ta, T b)
    uniform(const uniform<T, Method>& other)
    T a() const
    T b() const
    uniform<T, Method>& operator=(const uniform<T, Method>& other)
}
```

## Include Files

- `mkl_sycl.hpp`

## Description

The class object is used in `onemkl::rng::generate` function to provide random numbers uniformly distributed over the interval  $[a, b]$ , where  $a, b$  are the left and right bounds of the interval, respectively, and  $a, b \in \mathbb{R} ; a < b$ .

The probability density function is given by:

$$f_{a, b}(x) = \begin{cases} \frac{1}{b - a}, & x \in [a, b] \\ 0, & x \notin [a, b] \end{cases}, \quad -\infty < x < +\infty$$

The cumulative distribution function is as follows:

$$f_{a,b}(x) = \begin{cases} 0, & x < a \\ \frac{x-a}{b-a}, & a \leq x < b, -\infty < x < +\infty \\ 1, & x \geq b \end{cases}$$

## Input Parameters

Name	Type	Description
a	T (float, double)	Left bound a
b	T (float, double)	Right bound b

**Parent topic:** Distributions

### 13.2.4.5.3 onemkl::rng::gaussian

Generates normally distributed random numbers.

## Syntax

```
template<typename T = float, method Method = box_muller2>
class gaussian {
public:
    gaussian(): gaussian((T)0.0, (T)1.0){}
    gaussian(T mean, T stddev)
    gaussian(const gaussian<T, Method>& other)
    T mean() const
    T stddev() const
    gaussian<T, Method>& operator=(const gaussian<T, Method>& other)
}
```

## Include Files

- mkl\_sycl.hpp

## Description

The class object is used in `onemkl::rng::generate` function to provide random numbers with normal (Gaussian) distribution with mean ( $a$ ) and standard deviation ( $\text{stddev}$ ,  $\sigma$ ), where  $a, \sigma \in \mathbb{R}; \sigma > 0$ .

The probability density function is given by:

$$f_{a, \sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x - a)^2}{2\sigma^2}\right), \quad -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a, \sigma}(x) = \int_{-\infty}^x \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y - a)^2}{2\sigma^2}\right) dy, \quad -\infty < x < +\infty.$$

The cumulative distribution function  $F_{a, \sigma}(x)$  can be expressed in terms of standard normal distribution  $\phi(x)$  as

F
$(x) = \phi((x - a) / \sigma)$

## Input Parameters

Name	Type	Description
method	<code>onemkl::rng::method</code>	Generation method. The specific values are as follows: <code>onemkl::rng::box_muller</code> <code>onemkl::rng::box_muller2</code> <code>onemkl::rng::inverse_function</code> See brief descriptions of the methods in <a href="#">Distributions Template Parameter <code>onemkl::rng::method</code> Values</a> .
mean	T ( <code>float</code> , <code>double</code> )	Mean value $a$ .
std-dev	T ( <code>float</code> , <code>double</code> )	Standard deviation $\sigma$ .

**Parent topic:** Distributions

### 13.2.4.5.4 onemkl::rng::exponential

Generates exponentially distributed random numbers.

#### Syntax

```
template<typename T = float, method Method =      inverse_function>
class exponential {
public:
    exponential(): exponential((T)0.0, (T)1.0){}
    exponential(T a, T beta)
    exponential(const exponential<T, Method>& other)
    T a() const
    T beta() const
    exponential<T, Method>& operator=(const      exponential<T, Method>& other)
}
```

#### Include Files

- mkl\_sycl.hpp

#### Description

The onemkl::rng::exponential class object is used in onemkl::rng::generate function to provide random numbers with exponential distribution that has displacement  $a$  and scalefactor  $\beta$ , where  $a, \beta \in \mathbb{R} ; \beta > 0$ .

The probability density function is given by:

$$f_{a,\beta}(x) = \begin{cases} \frac{1}{\beta} \exp(-(x - a)/\beta), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} 1 - \exp(-(x - a)/\beta), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

## Input Parameters

Name	Type	Description
method	onemkl::Generation method.	The specific values are as follows: onemkl::rng::inverse_function   onemkl::rng::accurate See brief descriptions of the methods in <a href="#">Distributions Template Parameter onemkl::rng::method Values</a> .
a	T (float, double)	Displacement $a$ .
beta	T (float, double)	Scalefactor $\beta$ .

**Parent topic:** Distributions

### 13.2.4.5.5 onemkl::rng::laplace

Generates random numbers with Laplace distribution.

#### Syntax

```
template<typename T = float, method Method = inverse_function>
class laplace {
public:
    laplace(): laplace((T)0.0, (T)1.0){}
    laplace(T a, T b)
    laplace(const laplace<T, Method>& other)
    T a() const
    T b() const
    laplace<T, Method>& operator=(const laplace<T,           Method>& other)
}
```

#### Include Files

- mkl\_sycl.hpp

#### Description

The onemkl::rng::laplace class object is used in the onemkl::rng::generate function to provide random numbers with Laplace distribution with mean value (or average)  $a$  and scalefactor ( $b, \beta$ ), where  $a, \beta \in \mathbb{R} ; \beta > 0$ . The scale-factor value determines the standard deviation as

$$\sigma = \beta\sqrt{2}$$

The probability density function is given by:

$$f_{\alpha,\beta}(x) = \frac{1}{\sqrt{2\beta}} \exp\left(-\frac{|x - \alpha|}{\beta}\right), \quad -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{\alpha,\beta}(x) = \begin{cases} \frac{1}{2} \exp\left(-\frac{|x - \alpha|}{\beta}\right), & x \geq \alpha \\ 1 - \frac{1}{2} \exp\left(-\frac{|x - \alpha|}{\beta}\right), & x < \alpha \end{cases}, \quad -\infty < x < +\infty.$$

## Input Parameters

Name	Type	Description
method	<code>onemkl::rng::Generation</code> method.	The specific values are as follows: <code>onemkl::rng::inverse_function</code> See brief descriptions of the methods in <a href="#">Distributions Template Parameter onemkl::rng::method Values</a> .
a	T (float, double)	Mean value a.
b	T (float, double)	Scalefactor b.

**Parent topic:** Distributions

### 13.2.4.5.6 onemkl::rng::weibull

Generates Weibull distributed random numbers.

#### Syntax

```
template<typename T = float, method Method =           inverse_function>
class weibull {
public:
    weibull(): weibull((T)0.0, (T)1.0){}
    weibull(T alpha, T a, T beta)
    weibull(const weibull<T, Method>& other)
    T alpha() const
    T a() const
    T beta() const
    weibull<T, Method>& operator=(const weibull<T,           Method>& other)
}
```

#### Include Files

- mkl\_sycl.hpp

#### Description

The onemkl::rng::weibull class object is used in the onemkl::rng::generate function to provide Weibull distributed random numbers with displacement  $a$ , scalefactor  $\beta$ , and shape  $\alpha$ , where  $\alpha, \beta, a \in \mathbb{R} ; \alpha > 0, \beta > 0$ .

The probability density function is given by:

$$f_{\alpha, \beta}(x) = \begin{cases} \frac{\alpha}{\beta^\alpha} (x - a)^{\alpha-1} \exp\left(-\left(\frac{x-a}{\beta}\right)^\alpha\right), & x \geq a \\ 0, & x < a \end{cases}$$

The cumulative distribution function is as follows:

$$F_{\alpha, \beta}(x) = \begin{cases} 1 - \exp\left(-\left(\frac{x-a}{\beta}\right)^\alpha\right), & x \geq a, -\infty < x < +\infty. \\ 0, & x < a \end{cases}$$

## Input Parameters

Name	Type	Description
method	<code>onemkl::rng::method</code> .	The specific values are as follows: <code>onemkl::rng::inverse_function</code>   <code>onemkl::rng::accurate</code> See brief descriptions of the methods in <a href="#">Distributions Template Parameter onemkl::rng::method Values</a> .
al-pha	<code>T (float, double)</code>	Shape $\alpha$
a	<code>T (float, double)</code>	Displacement $a$ .
beta	<code>T (float, double)</code>	Scalefactor $\beta$ .

**Parent topic:** Distributions

### 13.2.4.5.7 onemkl::rng::cauchy

Generates Cauchy distributed random values.

#### Syntax

```
template<typename T = float, method Method = inverse_function>
class cauchy {
public:
    cauchy(): cauchy((T)0.0, (T)1.0){}
    cauchy(T a, T b)
    cauchy(const cauchy<T, Method>& other)
    T a() const
    T b() const
    cauchy<T, Method>& operator=(const cauchy<T, Method>& other)
}
```

#### Include Files

- mkl\_sycl.hpp

## Description

The `onemkl::rng::cauchy` class object is used in the `onemkl::rng::generate` function to provide Cauchy distributed random numbers with displacement ( $a$ ) and scalefactor ( $b, \beta$ ), where  $a, \beta \in \mathbb{R}$ ;  $\beta > 0$ .

The probability density function is given by:

$$f_{a,\beta}(x) = \frac{1}{\pi\beta \left( 1 + \left( \frac{x - a}{\beta} \right)^2 \right)}, \quad -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \frac{1}{2} + \frac{1}{\pi} \arctan \left( \frac{x - a}{\beta} \right), \quad -\infty < x < +\infty.$$

## Input Parameters

Name	Type	Description
method	<code>onemkl::rng::method</code>	Generation method. The specific values are as follows: <code>onemkl::rng::inverse_function</code> . See brief descriptions of the methods in <a href="#">Distributions Template Parameter <code>onemkl::rng::method</code> Values</a> .
a	T ( <code>float</code> , <code>double</code> )	Displacement a.
b	T ( <code>float</code> , <code>double</code> )	Scalefactor b.

**Parent topic:** Distributions

### 13.2.4.5.8 onemkl::rng::rayleigh

Generates Rayleigh distributed random values.

#### Syntax

```
template<typename T = float, method Method = inverse_function>
class rayleigh {
public:
    rayleigh(): rayleigh((T)0.0, (T)1.0){}
    rayleigh(T a, T b)
    rayleigh(const rayleigh<T, Method>& other)
    T a() const
    T b() const
    rayleigh<T, Method>& operator=(const rayleigh<T, Method>& other)
}
```

#### Include Files

- mkl\_sycl.hpp

#### Description

The onemkl::rng::rayleigh class object is used by the onemkl::rng::generate function to provide Rayleigh distributed random numbers with displacement ( $a$ ) and scalefactor ( $b, \beta$ ), where  $a, \beta \in \mathbb{R} ; \beta > 0$ .

The Rayleigh distribution is a special case of the [Weibull](#) distribution, where the shape parameter  $\alpha = 2$ .

The probability density function is given by:

$$f_{a,\beta}(x) = \begin{cases} \frac{2(x - a)}{\beta^2} \exp\left(-\frac{(x - a)^2}{\beta^2}\right), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} 1 - \exp\left(-\frac{(x - a)^2}{\beta^2}\right), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

## Input Parameters

Name	Type	Description
method	onemkl::Generation method.	The specific values are as follows: onemkl::rng::inverse_function   onemkl::rng::accurate See brief descriptions of the methods in <a href="#">Distributions Template Parameter onemkl::rng::method Values</a> .
a	T (float, double)	Displacement a.
b	T (float, double)	Scalefactor b.

**Parent topic:** Distributions

### 13.2.4.5.9 onemkl::rng::lognormal

Generates log-normally distributed random numbers.

#### Syntax

```
template<typename T = float, method Method = box_muller2>
class lognormal {
public:
    lognormal(): lognormal((T)0.0, (T)1.0, (T) 0.0, (T)1.0){}
    lognormal(Tm, T s, T displ, T scale)
    lognormal(const lognormal<T, Method>& other)
    T m() const
    T s() const
    T displ() const
    T scale() const
    lognormal<T, Method>& operator=(const lognormal<T, Method>& other)
}
```

#### Include Files

- mkl\_sycl.hpp

## Description

The `onemkl::rng::lognormal` class object is used in the `onemkl::rng::generate` function to provide random numbers with average of distribution ( $a$ ,  $a$ ) and standard deviation ( $s$ ,  $\sigma$ ) of subject normal distribution, displacement (`displ`,  $b$ ), and scalefactor (`scale`,  $\beta$ ), where  $a, \sigma, b, \beta \in \mathbb{R}$ ;  $\sigma > 0$ ,  $\beta > 0$ .

The probability density function is given by:

$$f_{a,\sigma,b,\beta}(x) = \begin{cases} \frac{1}{\sigma(x - b)\sqrt{2\pi}} \exp\left(-\frac{[\ln((x - b)/\beta) - a]^2}{2\sigma^2}\right), & x > b \\ 0, & x \leq b \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\sigma,b,\beta}(x) = \begin{cases} \Phi(\ln((x - b)/\beta) - a)/\sigma, & x > b \\ 0, & x \leq b \end{cases}$$

## Input Parameters

Name	Type	Description
method	<code>onemkl::rng::method</code>	Generation method. The specific values are as follows: <code>onemkl::rng::box_muller2</code> or <code>onemkl::rng::inverse_function</code> . See brief descriptions of the methods in <a href="#">Distributions Template Parameter <code>onemkl::rng::method</code> Values</a> .
m	T ( <code>float</code> , <code>double</code> )	Average $a$ of the subject normal distribution.
s	T ( <code>float</code> , <code>double</code> )	Standard deviation $\sigma$ of the subject normal distribution.
displ	T ( <code>float</code> , <code>double</code> )	Displacement <code>displ</code> .
scale	T ( <code>float</code> , <code>double</code> )	Scalefactor <code>scale</code> .

**Parent topic:** [Distributions](#)

### 13.2.4.5.10 onemkl::rng::gumbel

Generates Gumbel distributed random values.

#### Syntax

```
template<typename T = float, method Method =           inverse_function>
class gumbel {
public:
    gumbel(): gumbel((T)0.0, (T)1.0) {}
    gumbel(T a, T b)
    gumbel(const gumbel<T, Method>& other)
    T a() const
    T b() const
    gumbel<T, Method>& operator=(const gumbel<T,           Method>& other)
}
```

#### Include Files

- mkl\_sycl.hpp

#### Description

The onemkl::rng::gumbel class object is used in the onemkl::rng::generate function to provide Gumbel distributed random numbers with displacement ( $a$ ) and scalefactor ( $b, \beta$ ), where  $a, \beta \in \mathbb{R}; \beta > 0$ .

The probability density function is given by:

$$f_{a,\beta}(x) = \left\{ \frac{1}{\beta} \exp\left(\frac{x - a}{\beta}\right) \exp(-\exp((x - a)/\beta)), -\infty < x < +\infty. \right.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = 1 - \exp(-\exp((x - a)/\beta)), -\infty < x < +\infty$$

## Input Parameters

Name	Type	Description
method	onemkl::rng::Generation method.	The specific values are as follows: onemkl::rng::inverse_function See brief descriptions of the methods in <a href="#">Distributions Template Parameter onemkl::rng::method Values</a> .
a	T (float, double)	Displacement a.
b	T (float, double)	Scalefactor b.

**Parent topic:** Distributions

### 13.2.4.5.11 onemkl::rng::gamma

Generates gamma distributed random values.

#### Syntax

```
template<typename T = float, method Method = marsaglia>
class gamma {
public:
    gamma(): gamma((T)1.0, (T)0.0, (T)1.0) {}
    gamma(T alpha, T a, T beta)
    gamma(const gamma<T, Method>& other)
    T alpha() const
    T a() const
    T beta() const
    gamma<T, Method>& operator=(const gamma<T,
                                    Method>& other)
}
```

#### Include Files

- mkl\_sycl.hpp

#### Description

The onemkl::rng::gamma class object is used in the onemkl::rng::generate function to provide random numbers with gamma distribution that has shape parameter  $\alpha$ , displacement  $a$ , and scale parameter  $\beta$ , where  $\alpha, \beta, a \in \mathbb{R}$  ;  $\alpha > 0, \beta > 0$ .

The probability density function is given by:

where  $\gamma(\alpha)$  is the complete gamma function.

The cumulative distribution function is as follows:

## Input Parameters

Name	Type	Description
method	onemkl::rng::method	Generation method. The specific values are as follows: onemkl::rng::marsaglia   onemkl::rng::marsaglia + onemkl::rng::accurate See brief descriptions of the methods in <a href="#">Distributions Template Parameter onemkl::rng::method Values</a> .
alpha	T (float, double)	Shape $\alpha$
a	T (float, double)	Displacement a.
beta	T (float, double)	Scalefactor $\beta$ .

**Parent topic:** [Distributions](#)

### 13.2.4.5.12 onemkl::rng::beta

Generates beta distributed random values.

#### Syntax

```
template<typename T = float, method Method = cheng_johnk_atkinson>
class beta {
public:
    beta(): beta((T)1.0, (T)1.0, (T)(0.0), (T)(1.0)) {}
    beta(T p, T q, T a, T b)
    beta(const beta<T, Method>& other)
    T p() const
    T q() const
    T a() const
    T b() const
    beta<T, Method>& operator=(const beta<T, Method>& other)
}
```

#### Include Files

- mkl\_sycl.hpp

## Description

The onemkl::rng::beta class object is used in the onemkl::rng::generate function to provide random numbers with beta distribution that has shape parameters  $p$  and  $q$ , displacement  $a$ , and scale parameter  $(b, \beta)$ , where  $p, q, a$ , and  $\beta \in \mathbb{R}$ ;  $p > 0, q > 0, \beta > 0$ .

The probability density function is given by:

$$f_{,p,q,a,\beta}(x) = \begin{cases} \frac{1}{B(p,q)\beta^{p+q-1}}(x-a)^{p-1}(\beta+a-x)^{q-1}, & a \leq x < a+\beta, -\infty < x < \infty, \\ 0, & x < a, x \geq a+\beta \end{cases}$$

where  $B(p, q)$  is the complete beta function.

The cumulative distribution function is as follows:

$$F_{,p,q,a,\beta}(x) = \begin{cases} 0, & x < a \\ \int_a^x \frac{1}{B(p,q)\beta^{p+q-1}}(y-a)^{p-1}(\beta+a-y)^{q-1} dy, & a \leq x < a+\beta, -\infty < x < \infty, \\ 1, & x \geq a+\beta \end{cases}$$

## Input Parameters

Name	Type	Description
method	onemkl::rng::method	The specific values are as follows: onemkl::rng::cheng_johnk_atkinson   onemkl::rng::accurate See brief descriptions of the methods in <a href="#">Distributions Template Parameter onemkl::rng::method Values</a> .
p	T (float, double)	Shape p
q	T (float, double)	Shape q
a	T (float, double)	Displacement a.
b	T (float, double)	Scalefactor b.

**Parent topic:** Distributions

### 13.2.4.5.13 onemkl::rng::chi\_square

Generates chi-square distributed random values.

#### Syntax

```
template<typename T = float, method Method = gamma_marsaglia>
class chi_square {
public:
    chi_square(): chi_square(5) {}
    chi_square(std::int32_t n)
    chi_square(const chi_square<T, Method>& other)
    std::int32_t n() const
    chi_square<T, Method>& operator=(const chi_square<T, Method>& other)
}
```

#### Include Files

- mkl\_sycl.hpp

#### Description

The onemkl::rng::chi\_square class object is used in the onemkl::rng::generate function to provide random numbers with chi-square distribution and  $\nu$  degrees of freedom,  $n \in N$ ,  $n > 0$ .

The probability density function is:

$$f_v(x) = \begin{cases} \frac{x^{\frac{n-2}{2}} e^{-\frac{x}{2}}}{2^{n/2} \Gamma\left(\frac{n}{2}\right)}, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

The cumulative distribution function is:

$$F_v(x) = \begin{cases} \int_0^x \frac{y^{\frac{n-2}{2}} e^{-\frac{y}{2}}}{2^{n/2} \Gamma\left(\frac{n}{2}\right)} dy, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

## Input Parameters

Name	Type	Description
method	onemkl::rng::Generation method.	The specific values are as follows: onemkl::rng::gamma_marsaglia See brief descriptions of the methods in <a href="#">Distributions Template Parameter onemkl::rng::method Values</a> .
n	std::int32_t	Degrees of freedom.

**Parent topic:** Distributions

### 13.2.4.5.14 onemkl::rng::uniform (Discrete)

Generates random numbers uniformly distributed over the interval  $[a, b]$ .

## Syntax

```
template<typename T = float, method Method = standard>
class uniform {
public:
    uniform(): uniform((T)0.0, (T)1.0){}
    uniform(T a, T b)
    uniform(const uniform<T, Method>& other)
    T a() const
    T b() const
    uniform<T, Method>& operator=(const uniform<T, Method>& other)
}
```

## Include Files

- mkl\_sycl.hpp

## Description

The onemkl::rng::uniform class object is used in onemkl::rng::generate functions to provide random numbers uniformly distributed over the interval  $[a, b]$ , where  $a, b$  are the left and right bounds of the interval respectively, and  $a, b \in \mathbb{Z}; a < b$ .

The probability distribution is given by:

$$P(X = k) = \frac{1}{b - a}, k \in \{a, a + 1, \dots, b - 1\}.$$

The cumulative distribution function is as follows:

$$F_{a,b}(x) = \begin{cases} 0, & x < a \\ \frac{\lfloor x - a + 1 \rfloor}{b - a}, & a \leq x < b, x \in R \\ 1, & x \geq b \end{cases}$$

## Input Parameters

Name	Type	Description
a	T (std::int32_t)	Left bound a
b	T (std::int32_t)	Right bound b

**Parent topic:** Distributions

### 13.2.4.5.15 onemkl::rng::uniform\_bits

Generates uniformly distributed bits in 32/64-bit chunks.

## Syntax

```
template<typename T = std::uint32_t, method Method = standard>
class uniform_bits {}
```

## Include Files

- mkl\_sycl.hpp

## Description

The onemkl::rng::uniform\_bits class object is used to generate uniformly distributed bits in 32/64-bit chunks. It is designed to ensure each bit in the 32/64-bit chunk is uniformly distributed. It is not supported not for all engines. See [VS Notes](#) for details.

## Input Parameters

Name	Type	Description
T	std::uint32_t / std::uint64_t	Chunk size

**Parent topic:** Distributions

### 13.2.4.5.16 onemkl::rng::bits

Generates bits of underlying engine (BRNG) integer reccurrents.

## Syntax

```
template<typename T = std::uint32_t, method Method = standard>
class bits {}
```

## Include Files

- mkl\_sycl.hpp

## Description

The onemkl::rng::bits class object is used to generate integer random values. Each integer can be treated as a vector of several bits. In a truly random generator, these bits are random, while in pseudorandom generators this randomness can be violated. For example, a drawback of linear congruential generators is that lower bits are less random than higher bits (for example, see [Knuth81]). For this reason, exercise care when using this function. Typically, in a 32-bit LCG only 24 higher bits of an integer value can be considered random. See [VS Notes](#) for details.

## Input Parameters

Name	Type	Description
T	std::uint32_t	Integer type

**Parent topic:** Distributions

### 13.2.4.5.17 onemkl::rng::bernoulli

Generates Bernoulli distributed random values.

## Syntax

```
template<typename T = std::int32_t, method Method = inverse_function>
class bernoulli {
public:
    bernoulli(): bernoulli(0.5){}
    bernoulli(double p)
    bernoulli(const bernoulli<T, Method>& other)
    double p() const
    bernoulli<T, Method>& operator=(const bernoulli<T, Method>& other)
}
```

## Include Files

- mkl\_sycl.hpp

## Description

The onemkl::rng::bernoulli class object is used in the onemkl::rng::generate function to provide Bernoulli distributed random numbers with probability  $p$  of a single trial success, where

$$p \in R; 0 \leq p \leq 1.$$

A variate is called Bernoulli distributed, if after a trial it is equal to 1 with probability of success  $p$ , and to 0 with probability  $1 - p$ .

The probability distribution is given by:

$$P(X = 1) = p$$

$$P(X = 0) = 1 - p$$

The cumulative distribution function is as follows:

$$F_p(x) = \begin{cases} 0, & x < 0 \\ 1 - p, & 0 \leq x < 1, x \in R. \\ 1, & x \geq 1 \end{cases}$$

## Input Parameters

Name	Type	Description
method	onemkl::rng::Generation method.	The specific values are as follows: onemkl::rng::inverse_function See brief descriptions of the methods in <a href="#">Distributions Template Parameter onemkl::rng::method Values</a> .
p	double	Success probability p of a trial.

**Parent topic:** Distributions

### 13.2.4.5.18 onemkl::rng::geometric

Generates geometrically distributed random values.

#### Syntax

```
template<typename T = std::int32_t, method Method = inverse_function>
class geometric {
public:
    geometric(): geometric(0.5){}
    geometric(double p)
    geometric(const geometric<T, Method>& other)
    double p() const
    geometric<T, Method>& operator=(const geometric<T, Method>& other)
}
```

#### Include Files

- mkl\_sycl.hpp

#### Description

The onemkl::rng::geometric class object is used in the onemkl::rng::generate function to provide geometrically distributed random numbers with probability p of a single trial success, where  $p \in \mathbb{R}$ ;  $0 < p < 1$ .

A geometrically distributed variate represents the number of independent Bernoulli trials preceding the first success. The probability of a single Bernoulli trial success is p.

The probability distribution is given by:

$$P(X = k) = p \cdot (1 - p)^k, k \in \{0, 1, 2, \dots\}.$$

The cumulative distribution function is as follows:

$$F_p(x) = \begin{cases} 0, & x < 0 \\ 1 - (1 - p)^{\lfloor x + 1 \rfloor}, & 0 \geq x \in \mathbb{R} \end{cases}$$

## Input Parameters

Name	Type	Description
method	onemkl::rng::Generation method.	The specific values are as follows: onemkl::rng::inverse_function See brief descriptions of the methods in <a href="#">Distributions Template Parameter onemkl::rng::method Values</a> .
p	double	Success probability p of a trial.

**Parent topic:** Distributions

### 13.2.4.5.19 onemkl::rng::binomial

Generates binomially distributed random numbers.

#### Syntax

```
template<typename T = std::int32_t, method Method = btpe>
class binomial {
public:
    binomial(): binomial(5, 0.5){}
    binomial(std::int32_t ntrial, double p)
    binomial(const binomial<T, Method>& other)
    std::int32_t ntrial() const
    double p() const
    binomial<T, Method>& operator=(const binomial<T, Method>& other)
}
```

#### Include Files

- mkl\_sycl.hpp

#### Description

The onemkl::rng::binomial class object is used in the onemkl::rng::generate function to provide binomially distributed random numbers with number of independent Bernoulli trials m, and with probability p of a single trial success, where  $p \in \mathbb{R}; 0 \leq p \leq 1, m \in \mathbb{N}$ .

A binomially distributed variate represents the number of successes in m independent Bernoulli trials with probability of a single trial success p.

The probability distribution is given by:

$$P(X = k) = C_m^k p^k (1 - p)^{m-k}, k \in \{0, 1, \dots, m\}.$$

The cumulative distribution function is as follows:

$$F_{m,p}(x) = \begin{cases} 0, & x < 0 \\ \sum_{k=0}^{\lfloor x \rfloor} C_m^k p^k (1-p)^{m-k}, & 0 \leq x < m, x \in R \\ 1, & x \geq m \end{cases}$$

## Input Parameters

Name	Type	Description
method	<code>onemkl::rng</code>	Generation method. The specific values are as follows: <code>onemkl::rng::btpe</code> See brief descriptions of the methods in Distributions Template Parameter <code>onemkl::rng::method Values</code> .
ntrials	<code>std::int32_t</code>	Number of independent trials.
p	<code>double</code>	Success probability $p$ of a single trial.

**Parent topic:** Distributions

### 13.2.4.5.20 `onemkl::rng::hypergeometric`

Generates hypergeometrically distributed random values.

## Syntax

```
template<typename T = std::int32_t, method Method = h2pe>
class hypergeometric {
public:
    hypergeometric(): hypergeometric(1, 1, 1){}
    hypergeometric(std::int32_t l, std::int32_t s, std::int32_t m)
    hypergeometric(const hypergeometric<T, Method>& other)
    std::int32_t s() const
    std::int32_t m() const
    std::int32_t l() const
    hypergeometric<T, Method>& operator=(const laplace<T, Method>& other)
}
```

## Include Files

- mkl\_sycl.hpp

## Description

The onemkl::rng::hypergeometric class object is used in the onemkl::rng::generate function to provide hypergeometrically distributed random values with lot size  $l$ , size of sampling  $s$ , and number of marked elements in the lot  $m$ , where  $l, m, s \in \{0\} ; l \geq \max(s, m)$ .

Consider a lot of  $l$  elements comprising  $m$  “marked” and  $l-m$  “unmarked” elements. A trial sampling without replacement of exactly  $s$  elements from this lot helps to define the hypergeometric distribution, which is the probability that the group of  $s$  elements contains exactly  $k$  marked elements.

The probability distribution is given by:

$$P(X = k) = \frac{\binom{k}{m} \binom{s-k}{l-m}}{\binom{s}{l}}$$

,  $k \in \{\max(0, s + m - l), \dots, \min(s, m)\}$

The cumulative distribution function is as follows:

$$F_{l,s,m}(x) = \begin{cases} 0, & x < \max(0, s + m - l) \\ \sum_{k=\max(0,s+m-l)}^{\lfloor x \rfloor} \frac{\binom{k}{m} \binom{s-k}{l-m}}{\binom{s}{l}}, & \max(0, s + m - l) \leq x \leq \min(s, m) \\ 1, & x > \min(s, m) \end{cases}$$

## Input Parameters

Name	Type	Description
method	onemkl::rng	Generation method. The specific values are as follows: onemkl::rng::h2pe See brief descriptions of the methods in <a href="#">Distributions Template Parameter onemkl::rng::method Values</a> .
$l$	std::int32_t	Lot size of $l$ .
$s$	std::int32_t	Size of sampling without replacement .
$m$	std::int32_t	Number of marked elements $m$ .

**Parent topic:** Distributions

### 13.2.4.5.21 onemkl::rng::poisson

Generates Poisson distributed random values.

#### Syntax

```
template<typename T = std::int32_t, method Method = ptpe>
class poisson {
public:
    poisson(): poisson(0.5){}
    poisson(double lambda)
    poisson(const poisson<T, Method>& other)
    double lambda() const
    poisson<T, Method>& operator=(const poisson<T, Method>& other)
}
```

#### Include Files

- mkl\_sycl.hpp

#### Description

The onemkl::rng::poisson class object is used in the onemkl::rng::generate function to provide Poisson distributed random numbers with distribution parameter  $\lambda$ , where  $\lambda \in \mathbb{R}; \lambda > 0$ .

The probability distribution is given by:

$$P(X = k) = \frac{\lambda^k e^{-\lambda}}{k!},$$

$k \in \{0, 1, 2, \dots\}$ .

The cumulative distribution function is as follows:

$$F_\lambda(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} \frac{\lambda^k e^{-\lambda}}{k!}, & x \geq 0, x \in \mathbb{R} \\ 0, & x < 0 \end{cases}$$

## Input Parameters

Name	Type	Description
method	onemkl::rng::method	Generation method. The specific values are as follows: onemkl::rng::ptpe onemkl::rng::gaussian_inverse See brief descriptions of the methods in Distributions Template Parameter onemkl::rng::method Values.
lambda	double	Distribution parameter $\lambda$ .

**Parent topic:** Distributions

### 13.2.4.5.22 onemkl::rng::poisson\_v

Generates Poisson distributed random values with varying mean.

#### Syntax

```
template<typename T = std::int32_t, method Method = gaussian_inverse>
class poisson_v {
public:
    poisson_v(std::vector<double> lambda)
    poisson_v(const poisson_v<T, Method>& other)
    std::vector<double> lambda() const
    poisson_v<T, Method>& operator=(const poisson_v<T, Method>& other)
}
```

#### Include Files

- mkl\_sycl.hpp

#### Description

The onemkl::rng::poisson\_v class object is used in the onemkl::rng::generate function to provide n Poisson distributed random numbers  $x_i$  ( $i = 1, \dots, n$ ) with distribution parameter  $\lambda_i$ , where  $\lambda_i \in \mathbb{R}; \lambda_i > 0$ .

The probability distribution is given by:

$$P(X_i = k) = \frac{\lambda_i^k \exp(-\lambda_i)}{k!}, \quad k \in \{0, 1, 2, \dots\}.$$

The cumulative distribution function is as follows:

$$F_{\lambda_i}(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} \frac{\lambda_i^k e^{-\lambda_i}}{k!}, & x \geq 0, x \in R \\ 0, & x < 0 \end{cases}$$

## Input Parameters

Name	Type	Description
method	<code>onemkl::rng::method</code>	Generation method. The specific values are as follows: <code>onemkl::rng::gaussian_inverse</code> See brief descriptions of the methods in <a href="#">Distributions Template Parameter <code>onemkl::rng::method</code> Values</a> .
lambda	<code>std::vector&lt;double&gt;</code>	Array of the distribution parameters $\lambda$ .

**Parent topic:** Distributions

### 13.2.4.5.23 `onemkl::rng::negbinomial`

Generates random numbers with negative binomial distribution.

## Syntax

```
template<typename T = std::int32_t, method Method = nbar>
class negbinomial {
public:
    negbinomial(): negbinomial(0.1, 0.5){}
    negbinomial(double a, double p)
    negbinomial(const negbinomial<T, Method>& other)
    double a() const
    double p() const
    negbinomial<T, Method>& operator=(const negbinomial<T, Method>& other)
}
```

## Include Files

- `mkl_sycl.hpp`

## Description

The onemkl::rng::negbinomial class object is used in the onemkl::rng::generate function to provide random numbers with negative binomial distribution and distribution parameters  $a$  and  $p$ , where  $p, a \in \mathbb{R}; 0 < p < 1; a > 0$ .

If the first distribution parameter  $a \in \mathbb{N}$ , this distribution is the same as Pascal distribution. If  $a \in \mathbb{N}$ , the distribution can be interpreted as the expected time of  $a$ -th success in a sequence of Bernoulli trials, when the probability of success is  $p$ .

The probability distribution is given by:

$$P(X = k) = C_{a+k-1}^k p^a (1 - p)^k, k \in \{0, 1, 2, \dots\}.$$

The cumulative distribution function is as follows:

$$F_{a,p}(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} C_{a+k-1}^k p^a (1 - p)^k, & x \geq 0 \\ 0, & x < 0 \end{cases}, x \in \mathbb{R}$$

## Input Parameters

Name	Type	Description
method	onemkl::rng	Generation method. The specific values are as follows: onemkl::rng::nbar See brief descriptions of the methods in Distributions Template Parameter onemkl::rng::method Values.
a	double	The first distribution parameter $a$ .
p	double	The second distribution parameter $p$ .

**Parent topic:** Distributions

### 13.2.4.5.24 onemkl::rng::multinomial

Generates multinomially distributed random numbers.

## Syntax

```
template<typename T = std::int32_t, method Method = poisson_inverse>
class multinomial {
public:
    multinomial(double ntrial, std::vector<double> p)
    multinomial(const multinomial<T, Method>& other)
    std::int32_t ntrial() const
    std::vector<double> p() const
    multinomial<T, Method>& operator=(const multinomial<T, Method>& other)
}
```

## Include Files

- mkl\_sycl.hpp

## Description

The onemkl::rng::multinomial class object is used in the onemkl::rng::generate function to provide multinomially distributed random numbers with `ntrial` independent trials and `k` possible mutually exclusive outcomes, with corresponding probabilities  $p_i$ , where  $p_i \in \mathbb{R}; 0 \leq p_i \leq 1, m \in \mathbb{N}, k \in \mathbb{N}$ .

The probability distribution is given by:

$$P(X_1 = x_1, \dots, X_k = x_k) = \frac{m!}{\prod_{i=1}^k x_i!} \prod_{i=1}^k p_i^{x_i}, \quad 0 \leq x_i \leq m, \sum_{i=1}^k x_i = m$$

## Input Parameters

Name	Type	Description
method	onemkl::rng::Generation	method. The specific values are as follows: onemkl::rng::poisson_inverse See brief descriptions of the methods in <a href="#">Distributions Template Parameter onemkl::rng::method Values</a> .
ntrial	std::int32_t	Number of independent trials $m$ .
p	std::vector<double>	Probability vector of possible outcomes ( $k$ length).

**Parent topic:** Distributions

### 13.2.4.6 Bibliography

For more information about the VS RNG functionality, refer to the following publications:

- **VS RNG**

- [AVX] Intel. *Intel® Advanced Vector Extensions Programming Reference*. (<http://software.intel.com/file/36945>)
- [Bratley88] Bratley P. and Fox B.L. *Implementing Sobol's Quasirandom Sequence Generator*, ACM Transactions on Mathematical Software, Vol. 14, No. 1, Pages 88-100, March 1988.
- [Bratley92] Bratley P., Fox B.L., and Niederreiter H. *Implementation and Tests of Low-Discrepancy Sequences*, ACM Transactions on Modeling and Computer Simulation, Vol. 2, No. 3, Pages 195-213, July 1992.
- [Coddington94] Coddington, P. D. *Analysis of Random Number Generators Using Monte Carlo Simulation*. Int. J. Mod. Phys. C-5, 547, 1994.
- [IntelSWMan] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 3 vols. (<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>)
- [L'Ecuyer99] L'Ecuyer, Pierre. *Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure*. Mathematics of Computation, 68, 225, 249-260, 1999.
- [L'Ecuyer99a] L'Ecuyer, Pierre. *Good Parameter Sets for Combined Multiple Recursive Random Number Generators*. Operations Research, 47, 1, 159-164, 1999.
- [Kirkpatrick81] Kirkpatrick, S., and Stoll, E. *A Very Fast Shift-Register Sequence Random Number Generator*. Journal of Computational Physics, V. 40. 517-526, 1981.
- [Matsumoto98] Matsumoto, M., and Nishimura, T. *Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator*, ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 1, Pages 3-30, January 1998.
- [Matsumoto00] Matsumoto, M., and Nishimura, T. *Dynamic Creation of Pseudorandom Number Generators*, 56-69, in: Monte Carlo and Quasi-Monte Carlo Methods 1998, Ed. Niederreiter, H. and Spanier, J., Springer 2000, <http://www.math.sci.hiroshima-u.ac.jp/%7Em-mat/MT/DC/dc.html>.
- [NAG] NAG Numerical Libraries. [http://www.nag.co.uk/numeric/numerical\\_libraries.asp](http://www.nag.co.uk/numeric/numerical_libraries.asp)
- [Saito08] Saito, M., and Matsumoto, M. *SIMD-oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator*. Monte Carlo and Quasi-Monte Carlo Methods 2006, Springer, Pages 607 – 622, 2008.  
<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/earticles.html>
- [Salmon11] Salmon, John K., Morales, Mark A., Dror, Ron O., and Shaw, David E., *Parallel Random Numbers: As Easy as 1, 2, 3*. SC '11 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011.
- [Sobol76] Sobol, I.M., and Levitan, Yu.L. *The production of points uniformly distributed in a multidimensional cube*. Preprint 40, Institute of Applied Mathematics, USSR Academy of Sciences, 1976 (In Russian).
- [VS Notes] *oneMKL Vector Statistics Notes*, a document present on the oneMKL product at <http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation/>

### 13.2.5 Vector Math

oneMKL Vector Mathematics functions (VM) compute a mathematical function of each of the vector elements. VM includes a set of functions (arithmetic, power, trigonometric, exponential, hyperbolic, special, and rounding) that operate on vectors of real and complex numbers.

Application programs that improve performance with VM include nonlinear programming software, computation of integrals, financial calculations, computer graphics, and many others.

VM functions fall into the following groups according to the operations they perform:

- **VM Mathematical Functions** compute values of mathematical functions, such as sine, cosine, exponential, or logarithm, on vectors stored contiguously in memory.
- **VM Service Functions** set/get the accuracy modes and the error codes, and create error handlers for mathematical functions.

The VM mathematical functions take an input vector as an argument, compute values of the respective function element-wise, and return the results in an output vector. All the VM mathematical functions can perform in-place operations, where the input and output arrays are at the same memory locations.

- *Special Value Notations*
- *Miscellaneous VM Functions*

#### 13.2.5.1 Special Value Notations

This defines notations of special values for complex functions. The definitions are provided in text, tables, or formulas.

- $z, z_1, z_2, \dots$  denote complex numbers.
- $i, i^2=-1$  is the imaginary unit.
- $x, X, x_1, x_2, \dots$  denote real imaginary parts.
- $y, Y, y_1, y_2, \dots$  denote imaginary parts.
- $X$  and  $Y$  represent any finite positive IEEE-754 floating point values, if not stated otherwise.
- Quiet NaN and signaling NaN are denoted with QNaN and SNaN, respectively.
- The IEEE-754 positive infinities or floating-point numbers are denoted with a + sign before  $X, Y, \dots$ .
- The IEEE-754 negative infinities or floating-point numbers are denoted with a - sign before  $X, Y, \dots$ .

$\text{CONJ}(z)$  and  $\text{CIS}(z)$  are defined as follows:

$$\text{CONJ}(x+i \cdot y) = x - i \cdot y$$

$$\text{CIS}(y) = \cos(y) + i \cdot \sin(y).$$

The special value tables show the result of the function for the  $z$  argument at the intersection of the  $\text{RE}(z)$  column and the  $i^*\text{IM}(z)$  row. If the function raises an exception on the argument  $z$ , the lower part of this cell shows the raised exception and the VM Error Status. An empty cell indicates that this argument is normal and the result is defined mathematically.

**Parent topic:** *Vector Math*

### 13.2.5.2 VM Mathematical Functions

This section describes VM functions that compute values of mathematical functions on real and complex vector arguments with unit increment.

Each function is introduced by its short name, a brief description of its purpose, and the calling sequence for each type of data, as well as a description of the input/output arguments.

The input range of parameters is equal to the mathematical range of the input data type, unless the function description specifies input threshold values, which mark off the precision overflow, as follows:

- `FLT_MAX` denotes the maximum number representable in single precision real data type
- `DBL_MAX` denotes the maximum number representable in double precision real data type

*Table “VM Mathematical Functions” lists available mathematical functions and associated data types.*

Function	Data Types	Description
<b>Arithmetic Functions:</b>		
<code>add</code>	<code>s, d, c, z</code>	Adds vector elements
<code>sub</code>	<code>s, d, c, z</code>	Subtracts vector elements
<code>sqr</code>	<code>s, d</code>	Squares vector elements
<code>mul</code>	<code>s, d, c, z</code>	Multiplies vector elements
<code>mulbyconj</code>	<code>c, z</code>	Multiplies elements of one vector by conjugated elements of the second vector
<code>conj</code>	<code>c, z</code>	Conjugates vector elements
<code>abs</code>	<code>s, d, c, z</code>	Computes the absolute value of vector elements
<code>arg</code>	<code>c, z</code>	Computes the argument of vector elements
<code>linearfrac</code>	<code>s, d</code>	Performs linear fraction transformation of vectors
<code>fmod</code>	<code>s, d</code>	Performs element by element computation of the modulus function of vectors
<code>remainder</code>	<code>s, d</code>	Performs element by element computation of the remainder function on the vectors
<b>Power and Root Functions:</b>		
<code>inv</code>	<code>s, d</code>	Inverts vector elements
<code>div</code>	<code>s, d, c, z</code>	Divides elements of one vector by elements of the second vector
<code>sqrt</code>	<code>s, d, c, z</code>	Computes the square root of vector elements
<code>invsqrt</code>	<code>s, d</code>	Computes the inverse square root of vector elements
<code>cbrt</code>	<code>s, d</code>	Computes the cube root of vector elements
<code>invcbrt</code>	<code>s, d</code>	Computes the inverse cube root of vector elements
<code>pow2o3</code>	<code>s, d</code>	Computes the cube root of the square of each vector element
<code>pow3o2</code>	<code>s, d</code>	Computes the square root of the cube of each vector element
<code>pow</code>	<code>s, d, c, z</code>	Raises each vector element to the specified power
<code>powx</code>	<code>s, d, c, z</code>	Raises each vector element to the constant power
<code>powr</code>	<code>s, d</code>	Computes $a$ to the power $b$ for elements of two vectors, where the elements of $b$ are the powers of $a$
<code>hypot</code>	<code>s, d</code>	Computes the square root of sum of squares
<b>Exponential and Logarithmic Functions:</b>		
<code>exp</code>	<code>s, d, c, z</code>	Computes the base $e$ exponential of vector elements
<code>exp2</code>	<code>s, d</code>	Computes the base 2 exponential of vector elements
<code>exp10</code>	<code>s, d</code>	Computes the base 10 exponential of vector elements
<code>expm1</code>	<code>s, d</code>	Computes the base $e$ exponential of vector elements decreased by 1
<code>ln</code>	<code>s, d, c, z</code>	Computes the natural logarithm of vector elements
<code>log2</code>	<code>s, d</code>	Computes the base 2 logarithm of vector elements
<code>log10</code>	<code>s, d, c, z</code>	Computes the base 10 logarithm of vector elements
<code>log1p</code>	<code>s, d</code>	Computes the natural logarithm of vector elements that are increased by 1
<code>logb</code>	<code>s, d</code>	Computes the exponents of the elements of input vector $a$
<b>Trigonometric Functions:</b>		

Table 1 – continued from

Function	Data Types	Description
cos	s, d, c, z	Computes the cosine of vector elements
sin	s, d, c, z	Computes the sine of vector elements
sincos	s, d	Computes the sine and cosine of vector elements
cis	c, z	Computes the complex exponent of vector elements (cosine and sine combined)
tan	s, d, c, z	Computes the tangent of vector elements
acos	s, d, c, z	Computes the inverse cosine of vector elements
asin	s, d, c, z	Computes the inverse sine of vector elements
atan	s, d, c, z	Computes the inverse tangent of vector elements
atan2	s, d	Computes the four-quadrant inverse tangent of ratios of the elements of two vectors
cospi	s, d	Computes the cosine of vector elements multiplied by $\pi$
sinpi	s, d	Computes the sine of vector elements multiplied by $\pi$
tanpi	s, d	Computes the tangent of vector elements multiplied by $\pi$
acospi	s, d	Computes the inverse cosine of vector elements divided by $\pi$
asinpi	s, d	Computes the inverse sine of vector elements divided by $\pi$
atanpi	s, d	Computes the inverse tangent of vector elements divided by $\pi$
atan2pi	s, d	Computes the four-quadrant inverse tangent of the ratios of the corresponding elements of two vectors multiplied by $\pi$
cosd	s, d	Computes the cosine of vector elements multiplied by $\pi/180$
sind	s, d	Computes the sine of vector elements multiplied by $\pi/180$
tand	s, d	Computes the tangent of vector elements multiplied by $\pi/180$
<b>Hyperbolic Functions:</b>		
cosh	s, d, c, z	Computes the hyperbolic cosine of vector elements
sinh	s, d, c, z	Computes the hyperbolic sine of vector elements
tanh	s, d, c, z	Computes the hyperbolic tangent of vector elements
acosh	s, d, c, z	Computes the inverse hyperbolic cosine of vector elements
asinh	s, d, c, z	Computes the inverse hyperbolic sine of vector elements
atanh	s, d, c, z	Computes the inverse hyperbolic tangent of vector elements.
<b>Special Functions:</b>		
erf	s, d	Computes the error function value of vector elements
erfc	s, d	Computes the complementary error function value of vector elements
cdfnorm	s, d	Computes the cumulative normal distribution function value of vector elements
erfinv	s, d	Computes the inverse error function value of vector elements
erfcinv	s, d	Computes the inverse complementary error function value of vector elements
cdfnorminv	s, d	Computes the inverse cumulative normal distribution function value of vector elements
lgamma	s, d	Computes the natural logarithm for the absolute value of the gamma function
tgamma	s, d	Computes the gamma function of vector elements
expint1	s, d	Computes the exponential integral of vector elements
<b>Rounding Functions:</b>		
floor	s, d	Rounds towards minus infinity
ceil	s, d	Rounds towards plus infinity
trunc	s, d	Rounds towards zero infinity
round	s, d	Rounds to nearest integer
nearbyint	s, d	Rounds according to current mode
rint	s, d	Rounds according to current mode and raising inexact result exception
modf	s, d	Computes the integer and fractional parts
frac	s, d	Computes the fractional part
<b>Miscellaneous Functions:</b>		
copysign	s, d	Returns vector of elements of one argument with signs changed to match other
nextafter	s, d	Returns vector of elements containing the next representable floating-point number
fdim	s, d	Returns vector containing the differences of the corresponding elements of two vectors

Table 1 – continued from

Function	Data Types	Description
fmax	s, d	Returns the larger of each pair of elements of the two vector arguments
fmin	s, d	Returns the smaller of each pair of elements of the two vector arguments
maxmag	s, d	Returns the element with the larger magnitude between each pair of elements
minmag	s, d	Returns the element with the smaller magnitude between each pair of elements

**Parent topic:** *Vector Math*

### 13.2.5.2.1 Arithmetic Functions

Arithmetic functions perform the basic mathematical operations like addition, subtraction, multiplication or computation of the absolute value of the vector elements.

**Parent topic:** *VM Mathematical Functions*

- **add** Performs element by element addition of vector *a* and vector *b*.
- **sub** Performs element by element subtraction of vector *b* from vector *a*.
- **sqr** Performs element by element squaring of the vector.
- **mul** Performs element by element multiplication of vector *a* and vector *b*.
- **mulbyconj** Performs element by element multiplication of vector *a* element and conjugated vector *b* element.
- **conj** Performs element by element conjugation of the vector.
- **abs** Computes absolute value of vector elements.
- **arg** Computes argument of vector elements.
- **linearfrac** Performs linear fraction transformation of vectors *a* and *b* with scalar parameters.
- **fmod** The fmod function performs element by element computation of the modulus function of vector *a* with respect to vector *b*.
- **remainder** Performs element by element computation of the remainder function on the elements of vector *a* and the corresponding elements of vector *b*.

#### 13.2.5.2.1.1 add

Performs element by element addition of vector *a* and vector *b*.

##### Syntax

Buffer API:

```
void add(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &b, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event add(queue &exec_queue, int64_t n, T *a, T *b, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

`add` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The `add(a, b)` function performs element by element addition of vector `a` and vector `b`.

Argument 1	Argument 2	Result	Error Code
+0	+0	+0	
+0	-0	+0	
-0	+0	+0	
-0	-0	-0	
+∞	+∞	+∞	
+∞	-∞	QNAN	
-∞	+∞	QNAN	
-∞	-∞	-∞	
SNAN	any value	QNAN	
any value	SNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

$$\text{add}(x_1 + i \cdot y_1, x_2 + i \cdot y_2) = (x_1 + x_2) + i \cdot (y_1 + y_2)$$

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when all `RE(x)`, `RE(y)`, `IM(x)`, `IM(y)` arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns  $\infty$  in that part of the result, and sets the VM Error Status to `status::overflow` (overriding any possible `status::accuracy_warning` status).

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer `a` containing 1st input vector of size `n`.

**b** The buffer `b` containing 2nd input vector of size `n`.

**mode** Overrides the global VM mode setting for this function call. See `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the 1st input vector of size *n*.

**b** Pointer *b* to the 2nd input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use `add` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vadd.cpp
```

**Parent topic:** Arithmetic Functions

### 13.2.5.2.1.2 sub

Performs element by element subtraction of vector *b* from vector *a*.

#### Syntax

Buffer API:

```
void sub(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &b, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event sub( queue& exec_queue, int64_t n, T* a, T* b, T* y, vector_class<event>* depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {} )
```

`sub` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The sub(a, b) function performs element by element subtraction of vector a and vector b.

Argument 1	Argument 2	Result	Error Code
+0	+0	+0	
+0	-0	+0	
-0	+0	+0	
-0	-0	-0	
$+\infty$	$+\infty$	QNAN	
$+\infty$	$-\infty$	$+\infty$	
$-\infty$	$+\infty$	$-\infty$	
$-\infty$	$-\infty$	QNAN	
SNAN	any value	QNAN	
any value	SNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

$$\text{sub}(x_1 + i \cdot y_1, x_2 + i \cdot y_2) = (x_1 - x_2) + i \cdot (y_1 - y_2)$$

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when all RE(x), RE(y), IM(x), IM(y) arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns  $\infty$  in that part of the result, and sets the VM Error Status to `status::overflow` (overriding any possible `status::accuracy_warning` status).

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer a containing 1st input vector of size n.

**b** The buffer b containing 2nd input vector of size n.

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer a to the 1st input vector of size n.

**b** Pointer b to the 2nd input vector of size n.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use sub can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vsub.cpp
```

**Parent topic:** Arithmetic Functions

### 13.2.5.2.1.3 `sqr`

Performs element by element squaring of the vector.

#### Syntax

Buffer API:

```
void sqr(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined)
```

USM API:

```
event sqr(queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined)
```

`sqr` supports the following precisions.

T
float
double

#### Description

The `sqr()` function performs element by element squaring of the vector.

Argument	Result	Error Code
+0	+0	
-0	+0	
+∞	+∞	
-∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

The `sqr` function does not generate any errors.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing the input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use `sqr` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vsqr.cpp
```

**Parent topic:** Arithmetic Functions

### 13.2.5.2.1.4 mul

Performs element by element multiplication of vector *a* and vector *b*.

## Syntax

Buffer API:

```
void mul (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &b, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event mul (queue &exec_queue, int64_t n, T *a, T *b, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

**mul** supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

## Description

The **mul(a, b)** function performs element by element multiplication of vector a and vector b.

Argument 1	Argument 2	Result	Error Code
+0	+0	+0	
+0	-0	-0	
-0	+0	-0	
-0	-0	+0	
+0	+∞	QNAN	
+0	-∞	QNAN	
-0	+∞	QNAN	
-0	-∞	QNAN	
+∞	+0	QNAN	
+∞	-0	QNAN	
-∞	+0	QNAN	
-∞	-0	QNAN	
+∞	+∞	+∞	
+∞	-∞	-∞	
-∞	+∞	-∞	
-∞	-∞	+∞	
SNAN	any value	QNAN	
any value	SNAN	QNAN	
QNAN	non-SNAN	QNAN	
non-SNAN	QNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

$$\text{mul}(x_1 + i \cdot y_1, x_2 + i \cdot y_2) = (x_1 \cdot x_2 - y_1 \cdot y_2) + i \cdot (x_1 \cdot y_2 + y_1 \cdot x_2)$$

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when all RE(x), RE(y), IM(x), IM(y) arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns  $\infty$  in that part of the result, and sets the VM Error Status to `status::overflow` (overriding any possible `status::accuracy_warning` status).

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing 1st input vector of size *n*.

**b** The buffer *b* containing 2nd input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the 1st input vector of size *n*.

**b** Pointer *b* to the 2nd input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use `mul` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vmulp.cpp
```

**Parent topic:** Arithmetic Functions

### 13.2.5.2.1.5 `mulbyconj`

Performs element by element multiplication of vector **a** element and conjugated vector **b** element.

#### Syntax

Buffer API:

```
void mulbyconj (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &b, buffer<T, 1> &y,  
                 uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event mulbyconj (queue &exec_queue, int64_t n, T *a, T *b, T *y, vector_class<event> *depends, uint64_t  
                  mode = mode::not_defined, error_handler<T> errhandler = {})
```

`mulbyconj` supports the following precisions.

T
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

#### Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer **a** containing 1st input vector of size **n**.

**b** The buffer **b** containing 2nd input vector of size **n**.

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer **a** to the 1st input vector of size **n**.

**b** Pointer **b** to the 2nd input vector of size **n**.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use `mulbyconj` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vmulbyconj.cpp
```

**Parent topic:** Arithmetic Functions

### 13.2.5.2.1.6 conj

Performs element by element conjugation of the vector.

#### Syntax

Buffer API:

```
void conj (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined)
```

USM API:

```
event conj (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined)
```

`conj` supports the following precisions.

T
std::complex<float>
std::complex<double>

#### Description

The `conj` function performs element by element conjugation of the vector.

No special values are specified. The `conj` function does not generate any errors.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use conj can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vconj.cpp
```

**Parent topic:** Arithmetic Functions

### 13.2.5.2.1.7 abs

Computes absolute value of vector elements.

## Syntax

Buffer API:

```
void abs (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode =
    mode::not_defined)
```

USM API:

```
event abs (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode =
    mode::not_defined)
```

**abs** supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

## Description

The **abs(a)** function computes an absolute value of vector elements.

Argument	Result	Error Code
+0	+0	
-0	+0	
+∞	+∞	
-∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

$\text{abs}(a) = \text{hypot}(\text{RE}(a), \text{IM}(a))$ .

The **abs** function does not generate any errors.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer **a** containing input vector of size **n**.

**mode** Overrides the global VM mode setting for this function call. See [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer **a** to the input vector of size **n**.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

**y** The buffer `y` containing the output vector of size `n`.

USM API:

**y** Pointer `y` to the output vector of size `n`.

**return value (event)** Function end event.

## Example

An example of how to use `abs` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vabs.cpp
```

**Parent topic:** Arithmetic Functions

### 13.2.5.2.1.8 arg

Computes argument of vector elements.

## Syntax

Buffer API:

```
void arg(queue &exec_queue, int64_t n, buffer<A, 1> &a, buffer<R, 1> &y, uint64_t mode = mode::not_defined)
```

USM API:

```
event arg(queue &exec_queue, int64_t n, A *a, R *y, vector_class<event> *depends, uint64_t mode = mode::not_defined)
```

`arg` supports the following precisions.

T
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The `arg(a)` function computes argument of vector elements.

See [Special Value Notations](#) for the conventions used in the table below.

<code>RE(a) i·IM(a)</code>	$-\infty$	$-X$	$-0$	$+0$	$+X$	$+\infty$	NAN
$+i\cdot\infty$	$+3\cdot\pi/4$	$+\pi/2$	$+\pi/2$	$+\pi/2$	$+\pi/2$	$+\pi/4$	NAN
$+i\cdot Y$	$+ \pi$		$+\pi/2$	$+\pi/2$		$+0$	NAN
$+i\cdot 0$	$+ \pi$	$+ \pi$	$+ \pi$	$+0$	$+0$	$+0$	NAN
$-i\cdot 0$	$- \pi$	$- \pi$	$- \pi$	$-0$	$-0$	$-0$	NAN
$-i\cdot Y$	$- \pi$		$-\pi/2$	$-\pi/2$		$-0$	NAN
$-i\cdot\infty$	$-3\cdot\pi/4$	$-\pi/2$	$-\pi/2$	$-\pi/2$	$-\pi/2$	$-\pi/4$	NAN
$+i\cdot\text{NAN}$	NAN	NAN	NAN	NAN	NAN	NAN	NAN

## Note

$\arg(a) = \text{atan2}(\text{IM}(a), \text{RE}(a))$

The `arg` function does not generate any errors.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer `a` containing input vector of size `n`.

**mode** Overrides the global VM mode setting for this function call. See [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer `a` to the input vector of size `n`.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

**y** The buffer `y` containing the output vector of size `n`.

USM API:

**y** Pointer `y` to the output vector of size `n`.

**return value (event)** Function end event.

## Example

An example of how to use arg can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/varg.cpp
```

**Parent topic:** Arithmetic Functions

### 13.2.5.2.1.9 linearfrac

Performs linear fraction transformation of vectors  $a$  and  $b$  with scalar parameters.

#### Syntax

Buffer API:

```
void linearfrac (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &b, T scalea, T shifta, T scaleb, T shiftb, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event linearfrac (queue &exec_queue, int64_t n, T *a, T *b, T scalea, T shifta, T scaleb, T shiftb, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

linearfrac supports the following precisions.

T
float
double

#### Description

The linearfrac( $a, b, scalea, shifta, scaleb, shiftb$ ) function performs a linear fraction transformation of vector  $a$  by vector  $b$  with scalar parameters: scaling multipliers  $scalea$ ,  $scaleb$  and shifting addends  $shifta$ ,  $shiftb$ :

$$y[i] = (scalea \cdot a[i] + shifta) / (scaleb \cdot b[i] + shiftb), i=1,2 \dots n$$

The linearfrac function is implemented in the EP accuracy mode only, therefore no special values are defined for this function. If used in HA or LA mode, linearfrac sets the VM Error Status to status::accuracy\_warning. Correctness is guaranteed within the threshold limitations defined for each input parameter (see the table below); otherwise, the behavior is unspecified.

Threshold Limitations on Input Parameters
$2EMIN/2 \leq  scalea  \leq 2(EMAX-2)/2$
$2EMIN/2 \leq  scaleb  \leq 2(EMAX-2)/2$
$ shifta  \leq 2EMAX-2$
$ shiftb  \leq 2EMAX-2$
$2EMIN/2 \leq a[i] \leq 2(EMAX-2)/2$
$2EMIN/2 \leq b[i] \leq 2(EMAX-2)/2$
$a[i] \neq - (shifta/scalea) * (1-\delta_1),  \delta_1  \leq 21-(p-1)/2$
$b[i] \neq - (shiftb/scaleb) * (1-\delta_2),  \delta_2  \leq 21-(p-1)/2$

`EMIN` and `EMAX` are the minimum and maximum exponents and `p` is the number of significant bits (precision) for the corresponding data type according to the ANSI/IEEE Standard 754-2008 ([[IEEE754](#)]):

- for single precision  $EMIN = -126$ ,  $EMAX = 127$ ,  $p = 24$
- for double precision  $EMIN = -1022$ ,  $EMAX = 1023$ ,  $p = 53$

The thresholds become less strict for common cases with `scalea=0` and/or `scaleb=0`:

- if `scalea=0`, there are no limitations for the values of `a[i]` and `shifta`.
- if `scaleb=0`, there are no limitations for the values of `b[i]` and `shiftb`.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer `a` containing 1st input vector of size `n`.

**b** The buffer `b` containing 2nd input vector of size `n`.

**scalea** Constant value for scaling multipliers of vector `a`

**shifta** Constant value for shifting addend of vector `a`

**scaleb** Constant value for scaling multipliers of vector `b`

**shiftb** Constant value for shifting addend of vector `b`

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The pointer `a` to the 1st input vector of size `n`.

**b** The pointer `b` to the 2nd input vector of size `n`.

**scalea** Constant value for scaling multipliers of vector `a`

**shifta** Constant value for shifting addend of vector `a`

**scaleb** Constant value for scaling multipliers of vector `b`

**shiftb** Constant value for shifting addend of vector `b`

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer  $y$  containing the output vector of size  $n$ .

USM API:

**y** Pointer  $y$  to the output vector of size  $n$ .

**return value (event)** Function end event.

## Example

An example of how to use linearfrac can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vllinearfrac.cpp
```

**Parent topic:** Arithmetic Functions

### 13.2.5.2.1.10 fmod

The fmod function performs element by element computation of the modulus function of vector  $a$  with respect to vector  $b$ .

## Syntax

Buffer API:

```
void fmod(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &b, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event fmod(queue &exec_queue, int64_t n, T *a, T *b, T *y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

fmod supports the following precisions.

T
float
double

## Description

The fmod ( $a, b$ ) function computes the modulus function of each element of vector  $a$ , with respect to the corresponding elements of vector  $b$ :

$$a_i - b_i \cdot \text{trunc}(a_i/b_i)$$

In general, the modulus function  $\text{fmod } (a_i, b_i)$  returns the value  $a_i - n \cdot b_i$  for some integer  $n$  such that if  $b_i$  is nonzero, the result has the same sign as  $a_i$  and a magnitude less than the magnitude of  $b_i$ .

Argument 1	Argument 2	Result	Error Code
a not NAN	$\pm 0$	NAN	status::sing
$\pm\infty$	b not NAN	NAN	status::sing
$\pm 0$	$b \neq 0$ , not NAN	$\pm 0$	
a finite	$\pm\infty$	a	
NAN	b		
a	NAN	NAN	

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer a containing 1st input vector of size n.

**b** The buffer b containing 2nd input vector of size n.

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer a to the 1st input vector of size n.

**b** Pointer b to the 2nd input vector of size n.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer y containing the output vector of size n.

USM API:

**y** Pointer y to the output vector of size n.

**return value (event)** Function end event.

## Example

An example of how to use fmod can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vfmod.cpp
```

**Parent topic:** Arithmetic Functions

### 13.2.5.2.1.11 remainder

Performs element by element computation of the remainder function on the elements of vector  $a$  and the corresponding elements of vector  $b$ .

#### Syntax

Buffer API:

```
void remainder (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &b, buffer<T, 1> &y,  
    uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event remainder (queue &exec_queue, int64_t n, T *a, T *b, T *y, vector_class<event> *depends, uint64_t  
    mode = mode::not_defined, error_handler<T> errhandler = {})
```

remainder supports the following precisions.

T
float
double

#### Description

The `remainder (a)` function computes the remainder of each element of vector  $a$ , with respect to the corresponding elements of vector  $b$ : compute the values of  $n$  such that

$$n = a_i - n \cdot b_i$$

where  $n$  is the integer nearest to the exact value of  $a_i/b_i$ . If two integers are equally close to  $a_i/b_i$ ,  $n$  is the even one. If  $n$  is zero, it has the same sign as  $a_i$ .

Argument 1	Argument 2	Result	VM Error Status
a not NAN	$\pm 0$	NAN	status::errdom
$\pm\infty$	b not NAN	NAN	
$\pm 0$	$b \neq 0$ , not NAN	$\pm 0$	
a finite	$\pm\infty$	a	
NAN	b	NAN	
a	NAN	NAN	

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing 1st input vector of size *n*.

**b** The buffer *b* containing 2nd input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the 1st input vector of size *n*.

**b** Pointer *b* to the 2nd input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use remainder can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vremainder.cpp
```

**Parent topic:** Arithmetic Functions

### 13.2.5.2.2 Power and Root Functions

**Parent topic:** VM Mathematical Functions

- `inv` Performs element by element inversion of the vector.
- `div` Performs element by element division of vector `a` by vector `b`
- `sqrt` Computes a square root of vector elements.
- `invsqrt` Computes an inverse square root of vector elements.
- `cbrt` Computes a cube root of vector elements.
- `invcbrt` Computes an inverse cube root of vector elements.
- `pow2o3` Computes the cube root of the square of each vector element.
- `pow3o2` Computes the square root of the cube of each vector element.
- `pow` Computes `a` to the power `b` for elements of two vectors.
- `powx` Computes vector `a` to the scalar power `b`.
- `powr` Computes `a` to the power `b` for elements of two vectors, where the elements of vector argument `a` are all non-negative.
- `hypot` Computes a square root of sum of two squared elements.

#### 13.2.5.2.2.1 `inv`

Performs element by element inversion of the vector.

##### Syntax

Buffer API:

```
void inv(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event inv(queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

`inv` supports the following precisions.

T
float
double

## Description

The inv(a) function performs element by element inversion of the vector.

Argument	Result	VM Error Status
+0	$+\infty$	status::sing
-0	$-\infty$	status::sing
$+\infty$	+0	
$-\infty$	-0	
QNAN	QNAN	
SNAN	QNAN	

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use inv can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vinv.cpp
```

**Parent topic:** Power and Root Functions

### 13.2.5.2.2.2 div

Performs element by element division of vector  $a$  by vector  $b$

#### Syntax

Buffer API:

```
void div(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &b, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event div(queue &exec_queue, int64_t n, T *a, T *b, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

div supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

#### Description

The div( $a,b$ ) function performs element by element division of vector  $a$  by vector  $b$ .

Argument 1	Argument 2	Result	VM Error Status
X > +0	+0	+∞	status::sing
X > +0	-0	-∞	status::sing
X < +0	+0	-∞	status::sing
X < +0	-0	+∞	status::sing
+0	+0	QNAN	status::sing
-0	-0	QNAN	status::sing
X > +0	+∞	+0	
X > +0	-∞	-0	
+∞	+∞	QNAN	
-∞	-∞	QNAN	
QNAN	QNAN	QNAN	
SNAN	SNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Div}(x_1+i*y_1, x_2+i*y_2) = (x_1+i*y_1) * (x_2-i*y_2) / (x_2*x_2+y_2*y_2).$$

Overflow in a complex function occurs when  $x2+i*y2$  is not zero,  $x1, x2, y1, y2$  are finite numbers, but the real or imaginary part of the exact result is so large that it does not fit the target precision. In that case, the function returns  $\infty$  in that part of the result, and sets the VM Error Status to status::overflow.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer  $a$  containing 1st input vector of size  $n$ .

**b** The buffer  $b$  containing 2nd input vector of size  $n$ .

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer  $a$  to the 1st input vector of size  $n$ .

**b** Pointer  $b$  to the 2nd input vector of size  $n$ .

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer  $y$  containing the output vector of size  $n$ .

USM API:

**y** Pointer  $y$  to the output vector of size  $n$ .

**return value (event)** Function end event.

## Example

An example of how to use div can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vdiv.cpp
```

**Parent topic:** Power and Root Functions

### 13.2.5.2.2.3 sqrt

Computes a square root of vector elements.

#### Syntax

Buffer API:

```
void sqrt (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event sqrt (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

sqrt supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

#### Description

The sqrt function computes a square root of vector elements.

Argument	Result	VM Error Status
a < +0	QNAN	status::errdom
+0	+0	
-0	-0	
-∞	QNAN	status::errdom
+∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

+i·∞	+∞+i·∞							
+i·Y	+0+i·∞						+∞+i·0	
+i·0	+0+i·∞		+0+i·0	+0+i·0			+∞+i·0	
-i·0	+0-i·∞		+0-i·0	+0-i·0			+∞-i·0	
-i·Y	+0-i·∞						+∞-i·0	
-i·∞	+∞-i·∞							
+i·NAN								

Notes:

- $\text{Sqrt}(\text{CONJ}(z)) = \text{CONJ}(\text{Sqrt}(z))$ .

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the 1st input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use sqrt can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vsqrt.cpp
```

**Parent topic:** Power and Root Functions

### 13.2.5.2.2.4 invsqrt

Computes an inverse square root of vector elements.

#### Syntax

Buffer API:

```
void invsqrt (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event invsqrt (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

invsqrt supports the following precisions.

T
float
double

#### Description

The invsqrt(a) function computes an inverse square root of vector elements.

Argument	Result	VM Error Status
a < +0	QNAN	status::errdom
+0	+∞	status::sing
-0	-∞	status::sing
-∞	QNAN	status::errdom
+∞	+0	
QNAN	QNAN	
SNAN	QNAN	

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the 1st input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use `invsqrt` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vinvsqrt.cpp
```

**Parent topic:** Power and Root Functions

### 13.2.5.2.2.5 cbrt

Computes a cube root of vector elements.

#### Syntax

Buffer API:

```
void cbrt (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined)
```

USM API:

```
event cbrt (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined)
```

`cbrt` supports the following precisions.

T
float
double

#### Description

The `cbrt(a)` function computes a cube root of vector elements.

Argument	Result	Error Code
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
QNAN	QNAN	
SNAN	QNAN	
+0	+0	

#### Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer `a` containing input vector of size `n`.

**mode** Overrides the global VM mode setting for this function call. See `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer `a` to the input vector of size `n`.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

`y` The buffer `y` containing the output vector of size `n`.

USM API:

`y` Pointer `y` to the output vector of size `n`.

**return value (event)** Function end event.

## Example

An example of how to use `cbrt` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vcbrt.cpp
```

**Parent topic:** Power and Root Functions

### 13.2.5.2.2.6 `invcbrt`

Computes an inverse cube root of vector elements.

## Syntax

Buffer API:

```
void invcbrt (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode =  
        mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event invcbrt (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode =  
        mode::not_defined, error_handler<T> errhandler = {})
```

`invcbrt` supports the following precisions.

T
float
double

## Description

The `invcbrt(a)` function computes an inverse cube root of vector elements.

Argument	Result	Error Code
+0	+∞	<code>status::sing</code>
-0	-∞	<code>status::sing</code>
+∞	+0	
-∞	-0	
QNAN	QNAN	
SNAN	QNAN	

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use invcbrt can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vinvcbrt.cpp
```

**Parent topic:** Power and Root Functions

### 13.2.5.2.2.7 pow2o3

Computes the cube root of the square of each vector element.

#### Syntax

Buffer API:

```
void pow2o3 (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined)
```

USM API:

```
event pow2o3 (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined)
```

pow2o3 supports the following precisions.

T
float
double

#### Description

The pow2o3(a)function computes the cube root of the square of each vector element.

Argument	Result	Error Code
+0	+0	
-0	+0	
$+\infty$	$+\infty$	
$-\infty$	$+\infty$	
QNAN	QNAN	
SNAN	QNAN	

#### Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer a containing input vector of size n.

**mode** Overrides the global VM mode setting for this function call. See `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use `pow2o3` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vpow2o3.cpp
```

**Parent topic:** Power and Root Functions

### 13.2.5.2.2.8 pow3o2

Computes the square root of the cube of each vector element.

## Syntax

Buffer API:

```
void pow3o2 (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event pow3o2 (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

`pow3o2` supports the following precisions.

T
float
double

## Description

The `pow3o2(a)` function computes the square root of the cube of each vector element.

Data Type	Threshold Limitations on Input Parameters
single precision	$ a_i  < (\text{FLT\_MAX})^{2/3}$
double precision	$ a_i  < (\text{FLT\_MAX})^{2/3}$

Argument	Result	VM Error Status
$a < +0$	QNAN	<code>status::errdom</code>
$+0$	$+0$	
$-0$	$-0$	
$-\infty$	QNAN	<code>status::errdom</code>
$+\infty$	$+\infty$	
QNAN	QNAN	
SNAN	QNAN	

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer **y** containing the output vector of size **n**.

USM API:

**y** Pointer **y** to the output vector of size **n**.

**return value (event)** Function end event.

## Example

An example of how to use `pow3o2` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vpow3o2.cpp
```

**Parent topic:** Power and Root Functions

### 13.2.5.2.2.9 pow

Computes **a** to the power **b** for elements of two vectors.

#### Syntax

Buffer API:

```
void pow (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &b, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event pow (queue &exec_queue, int64_t n, T *a, T *b, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

`pow` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

#### Description

The `pow(a,b)` function computes **a** to the power **b** for elements of two vectors.

The real function `pow` has certain limitations on the input range of **a** and **b** parameters. Specifically, if **a[i]** is positive, then **b[i]** may be arbitrary. For negative **a[i]**, the value of **b[i]** must be an integer (either positive or negative).

The complex function `pow` has no input range limitations.

Argument 1	Argument 2	Result	Error Code
+0	neg. odd integer	+∞	status::errdom
-0	neg. odd integer	-∞	status::errdom
+0	neg. even integer	+∞	status::errdom
-0	neg. even integer	+∞	status::errdom
+0	neg. non-integer	+∞	status::errdom
-0	neg. non-integer	+∞	status::errdom
-0	pos. odd integer	+0	
-0	pos. odd integer	-0	
+0	pos. even integer	+0	
-0	pos. even integer	+0	
+0	pos. non-integer	+0	
-0	pos. non-integer	+0	
-1	+∞	+1	
-1	-∞	+1	
+1	any value	+1	
+1	+0	+1	
+1	-0	+1	
+1	+∞	+1	
+1	-∞	+1	
+1	QNAN	+1	
any value	+0	+1	
+0	+0	+1	
-0	+0	+1	
+∞	+0	+1	
-∞	+0	+1	
QNAN	+0	+1	
any value	-0	+1	
+0	-0	+1	
-0	-0	+1	
+∞	-0	+1	
-∞	-0	+1	
QNAN	-0	+1	
a < +0	non-integer	QNAN	status::errdom
a  < 1	-∞	+∞	
+0	-∞	+∞	status::errdom
-0	-∞	+∞	status::errdom
a  > 1	-∞	+0	
+∞	-∞	+0	
-∞	-∞	+0	
a  < 1	+∞	+0	
+0	+∞	+0	
-0	+∞	+0	
a  > 1	+∞	+∞	
+∞	+∞	+∞	
-∞	+∞	+∞	
-∞	neg. odd integer	-0	
-∞	neg. even integer	+0	
-∞	neg. non-integer	+0	
-∞	pos. odd integer	-∞	
-∞	pos. even integer	+∞	

continues on next page

Table 2 – continued from previous page

Argument 1	Argument 2	Result	Error Code
$-\infty$	pos. non-integer	$+\infty$	
$+\infty$	$b < +0$	$+0$	
$+\infty$	$b > +0$	$+\infty$	
Big finite value*	Big finite value*	$+/-\infty$	status::overflow
QNAN	QNAN	QNAN	
QNAN	SNAN	QNAN	
SNAN	QNAN	QNAN	
SNAN	SNAN	QNAN	

\* Overflow in a real function is supported only in the HA/LA accuracy modes. The overflow occurs when x and y are finite numbers, but the result is too large to fit the target precision. In this case, the function:

1. Returns  $\infty$  in the result.
2. Sets the VM Error Status to status::overflow.

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when all RE(x), RE(y), IM(x), IM(y) arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns  $\infty$  in that part of the result, and sets the VM Error Status to status::overflow (overriding any possible status::accuracy\_warning status).

The complex double precision versions of this function are implemented in the EP accuracy mode only. If used in HA or LA mode, the functions set the VM Error Status to status::accuracy\_warning.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer a containing 1st input vector of size n.

**b** The buffer b containing 2nd input vector of size n.

**mode** Overrides the global VM mode setting for this function call. See [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is mode::not\_defined.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer a to the 1st input vector of size n.

**b** Pointer b to the 2nd input vector of size n.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is mode::not\_defined.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer **y** containing the output vector of size **n**.

USM API:

**y** Pointer **y** to the output vector of size **n**.

**return value (event)** Function end event.

## Example

An example of how to use pow can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vpow.cpp
```

**Parent topic:** Power and Root Functions

### 13.2.5.2.2.10 powx

Computes vector **a** to the scalar power **b**.

#### Syntax

Buffer API:

```
void powx(queue &exec_queue, int64_t n, buffer<T, 1> &a, T b, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event powx(queue &exec_queue, int64_t n, T *a, T b, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

**powx** supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

#### Description

The **powx** function computes **a** to the power **b** for a vector **a** and a scalar **b**.

The real function **powx** has certain limitations on the input range of **a** and **b** parameters. Specifically, if **a[i]** is positive, then **b** may be arbitrary. For negative **a[i]**, the value of **b** must be an integer (either positive or negative).

The complex function **powx** has no input range limitations.

Special values and VM Error Status treatment are the same as for the **pow** function.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing 1st input vector of size *n*.

**b** Fixed value of power *b*.

**mode** Overrides the global VM mode setting for this function call. See [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the 1st input vector of size *n*.

**b** Fixed value of power *b*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use `powx` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vpowx.cpp
```

**Parent topic:** Power and Root Functions

### 13.2.5.2.2.11 powr

Computes  $a$  to the power  $b$  for elements of two vectors, where the elements of vector argument  $a$  are all non-negative.

#### Syntax

Buffer API:

```
void powr (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &b, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event powr (queue &exec_queue, int64_t n, T *a, T *b, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

`powr` supports the following precisions.

T
float
double

#### Description

The `powr(a,b)` function raises each element of vector  $a$  by the corresponding element of vector  $b$ . The elements of  $a$  are all nonnegative ( $a_i \geq 0$ ).

Data Type	Threshold Limitations on Input Parameters
single precision	$a_i < (\text{FLT\_MAX})^{1/b}$
double precision	$a_i < (\text{DBL\_MAX})^{1/b}$

Special values and VM Error Status treatment for v?Powr function are the same as for pow, unless otherwise indicated in this table:

Argument 1	Argument 2	Result	Error Code
$a < 0$	any value $b$	NAN	status::errdom
$0 < a < \infty$	$\pm 0$	1	
$\pm 0$	$-\infty < b < 0$	$+\infty$	
$\pm 0$	$-\infty$	$+\infty$	
$\pm 0$	$b > 0$	$+0$	
1	$-\infty < b < \infty$	1	
$\pm 0$	$\pm 0$	NAN	
$+\infty$	$\pm 0$	NAN	
1	$+\infty$	NAN	
$a \geq 0$	NAN	NAN	
NAN	any value $b$	NAN	
$0 < a < 1$	$-\infty$	$+\infty$	
$a > 1$	$-\infty$	$+0$	
$0 \leq a < 1$	$+\infty$	$+0$	
$a > 1$	$+\infty$	$+\infty$	
$+\infty$	$b < +0$	$+0$	
$+\infty$	$b > +0$	$+\infty$	
QNAN	QNAN	QNAN	status::errdom
QNAN	SNAN	QNAN	status::errdom
SNAN	QNAN	QNAN	status::errdom
SNAN	SNAN	QNAN	status::errdom

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer  $a$  containing 1st input vector of size  $n$ .

**b** The buffer  $b$  containing 2nd input vector of size  $n$ .

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer  $a$  to the 1st input vector of size  $n$ .

**b** Pointer  $b$  to the 2nd input vector of size  $n$ .

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer  $y$  containing the output vector of size  $n$ .

USM API:

**y** Pointer  $y$  to the output vector of size  $n$ .

**return value (event)** Function end event.

## Example

An example of how to use `powr` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vpowl.cpp
```

**Parent topic:** Power and Root Functions

### 13.2.5.2.2.12 hypot

Computes a square root of sum of two squared elements.

#### Syntax

Buffer API:

```
void hypot (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &b, buffer<T, 1> &y, uint64_t mode = mode::not_defined)
```

USM API:

```
event hypot (queue &exec_queue, int64_t n, T *a, T *b, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined)
```

`hypot` supports the following precisions.

T
float
double

#### Description

The function `hypot(a,b)` computes a square root of sum of two squared elements.

Argument 1	Argument 2	Result	Error Code
+0	+0	+0	
-0	-0	+0	
$+\infty$	any value	$+\infty$	
any value	$+\infty$	$+\infty$	
SNAN	any value	QNAN	INVALID
any value	SNAN	QNAN	INVALID
QNAN	any value	QNAN	
any value	QNAN	QNAN	

Data Type	Threshold Limitations on Input Parameters
single precision	<code>abs(a[i]) &lt; sqrt(FLT_MAX)</code> <code>abs(b[i]) &lt; sqrt(FLT_MAX)</code>
double precision	<code>abs(a[i]) &lt; sqrt(DBL_MAX)</code> <code>abs(b[i]) &lt; sqrt(DBL_MAX)</code>

The `hypot(a,b)` function does not generate any errors.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

- n** Specifies the number of elements to be calculated.
- a** The buffer `a` containing 1st input vector of size `n`.
- b** The buffer `b` containing 2nd input vector of size `n`.

**mode** Overrides the global VM mode setting for this function call. See `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

USM API:

**exec\_queue** The queue where the routine should be executed.

- n** Specifies the number of elements to be calculated.
- a** Pointer `a` to the 1st input vector of size `n`.
- b** Pointer `b` to the 2nd input vector of size `n`.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

**y** The buffer `y` containing the output vector of size `n`.

USM API:

**y** Pointer `y` to the output vector of size `n`.

**return value (event)** Function end event.

## Example

An example of how to use `hypot` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vhypot.cpp
```

**Parent topic:** Power and Root Functions

### 13.2.5.2.3 Exponential and Logarithmic Functions

**Parent topic:** VM Mathematical Functions

- `exp` Computes an exponential of vector elements.
- `exp2` Computes the base 2 exponential of vector elements.
- `exp10` Computes the base 10 exponential of vector elements.
- `expm1` Computes an exponential of vector elements decreased by 1.
- `ln` Computes natural logarithm of vector elements.
- `log2` Computes the base 2 logarithm of vector elements.
- `log10` Computes the base 10 logarithm of vector elements.
- `log1p` Computes a natural logarithm of vector elements that are increased by 1.
- `logb` Computes the exponents of the elements of input vector `a`.

#### 13.2.5.2.3.1 `exp`

Computes an exponential of vector elements.

##### Syntax

Buffer API:

```
void exp (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event exp (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

`exp` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

##### Description

The `exp(a)` function computes an exponential of vector elements.

Data Type	Threshold Limitations on Input Parameters
single precision	<code>a[i] &lt; Log( FLT_MAX )</code>
double precision	<code>a[i] &lt; Log( DBL_MAX )</code>

Argument	Result	Error Code
+0	+1	
-0	+1	
$a > \text{overflow}$	$+\infty$	<code>status::overflow</code>
$a < \text{underflow}$	$+0$	<code>status::overflow</code>
$+\infty$	$+\infty$	
$-\infty$	$+0$	
QNAN	QNAN	
SNAN	QNAN	

$+i\cdot\infty$					
$+i\cdot Y$					
$+i\cdot 0$					
$-i\cdot 0$					
$-i\cdot Y$					
$-i\cdot\infty$					
$+i\cdot\text{NAN}$					

Notes:

- The complex  $\exp(z)$  function sets the VM Error Status to `status::overflow` in the case of overflow, that is, when both  $\text{RE}(z)$  and  $\text{IM}(z)$  are finite non-zero numbers, but the real or imaginary part of the exact result is so large that it does not meet the target precision.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer  $y$  containing the output vector of size  $n$ .

USM API:

**y** Pointer  $y$  to the output vector of size  $n$ .

**return value (event)** Function end event.

## Example

An example of how to use `exp` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vexp.cpp
```

**Parent topic:** Exponential and Logarithmic Functions

### 13.2.5.2.3.2 `exp2`

Computes the base 2 exponential of vector elements.

#### Syntax

Buffer API:

```
void exp2 (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event exp2 (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

`exp2` supports the following precisions.

T
float
double

#### Description

The `exp2` function computes the base 2 exponential of vector elements.

Data Type	Threshold Limitations on Input Parameters
single precision	$a_i < \log_2(\text{FLT\_MAX})$
double precision	$a_i < \log_2(\text{DBL\_MAX})$

Argument	Result	Error Code
+0	+1	
-0	+1	
a > overflow	+∞	status::overflow
a < underflow	+0	status::underflow
+∞	+∞	
-∞	+0	
QNAN	QNAN	
SNAN	QNAN	

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use `exp2` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vexp2.cpp
```

**Parent topic:** Exponential and Logarithmic Functions

### 13.2.5.2.3.3 exp10

Computes the base 10 exponential of vector elements.

#### Syntax

Buffer API:

```
void exp10 (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event exp10 (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

`exp10` supports the following precisions.

T
float
double

#### Description

The `exp10(a)` function computes the base 10 exponential of vector elements.

Data Type	Threshold Limitations on Input Parameters
single precision	$a_i < \log_{10}(\text{FLT\_MAX})$
double precision	$a_i < \log_{10}(\text{DBL\_MAX})$

Argument	Result	VM Error Status
+0	+1	
-0	+1	
$a > \text{overflow}$	$+\infty$	<code>status::overflow</code>
$a < \text{underflow}$	+0	<code>status::underflow</code>
$+\infty$	$+\infty$	
$-\infty$	+0	
QNAN	QNAN	
SNAN	QNAN	

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use `exp10` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vexp10.cpp
```

**Parent topic:** Exponential and Logarithmic Functions

### 13.2.5.2.3.4 `expm1`

Computes an exponential of vector elements decreased by 1.

#### Syntax

Buffer API:

```
void expm1 (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event expm1 (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

`expm1` supports the following precisions.

T
float
double

#### Description

The `expm1(a)` function computes an exponential of vector elements decreased by 1.

Argument	Result	Error Code
+0	+1	
-0	+1	
a > overflow	+∞	status::overflow
+∞	+∞	
-∞	-0	
QNAN	QNAN	
SNAN	QNAN	

Data Type	Threshold Limitations on Input Parameters
single precision	a[i] < Log( FLT_MAX )
double precision	a[i] < Log( DBL_MAX )

#### Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer **a** containing input vector of size **n**.

**mode** Overrides the global VM mode setting for this function call. See `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer **a** to the input vector of size **n**.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer **y** containing the output vector of size **n**.

USM API:

**y** Pointer **y** to the output vector of size **n**.

**return value (event)** Function end event.

## Example

An example of how to use `expm1` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vexpm1.cpp
```

**Parent topic:** Exponential and Logarithmic Functions

### 13.2.5.2.3.5 ln

Computes natural logarithm of vector elements.

## Syntax

Buffer API:

```
void ln(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined,  
error_handler<T> errhandler = {})
```

USM API:

```
event ln(queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode =  
mode::not_defined, error_handler<T> errhandler = {})
```

`ln` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The `ln(a)` function computes natural logarithm of vector elements.

Argument	Result	Error Code
+1	+0	
a <+0	QNAN	<code>status::errdom</code>
+0	-∞	<code>status::sing</code>
-0	-∞	<code>status::sing</code>
-∞	QNAN	<code>status::errdom</code>
+∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

RE(a) i·IM(a)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	$+\infty + i \cdot \frac{3\pi}{4}$	$+\infty + i \cdot \pi/2$	$+\infty + i \cdot \pi/4$	$+\infty + i \cdot \text{QNAN}$			
+i·Y	$+\infty - i \cdot \pi$					$+\infty + i \cdot 0$	$\text{QNAN} + i \cdot \text{QNAN}$
+i·0	$+\infty - i \cdot \pi$		$-\infty + i \cdot \pi$	$-\infty - i \cdot 0$		$+\infty + i \cdot 0$	$\text{QNAN} + i \cdot \text{QNAN}$
-i·0	$+\infty - i \cdot \pi$		$-\infty + i \cdot \pi$	$-\infty - i \cdot 0$		$+\infty - i \cdot 0$	$\text{QNAN} + i \cdot \text{QNAN}$
-i·Y	$+\infty - i \cdot \pi$					$+\infty - i \cdot 0$	$\text{QNAN} + i \cdot \text{QNAN}$
-i·∞	$+\infty - i \cdot \frac{3\pi}{4}$	$+\infty - i \cdot \pi/2$	$+\infty - i \cdot \pi/4$	$+\infty + i \cdot \text{QNAN}$			
+i·NAN	$+\infty + i \cdot \text{QNAN}$	$\text{QNAN} + i \cdot \text{QNAN}$	$\text{QNAN} + i \cdot \text{QNAN}$	$\text{QNAN} + i \cdot \text{QNAN}$	$\text{QNAN} + i \cdot \text{QNAN}$	$\text{QNAN} + i \cdot \text{QNAN}$	$\text{QNAN} + i \cdot \text{QNAN}$

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer `a` containing input vector of size `n`.

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer `a` to the input vector of size `n`.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer  $y$  containing the output vector of size  $n$ .

USM API:

**y** Pointer  $y$  to the output vector of size  $n$ .

**return value (event)** Function end event.

## Example

An example of how to use  $\ln$  can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vln.cpp
```

**Parent topic:** Exponential and Logarithmic Functions

### 13.2.5.2.3.6 log2

Computes the base 2 logarithm of vector elements.

#### Syntax

Buffer API:

```
void log2 (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event log2 (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

$\log_2$  supports the following precisions.

T
float
double

#### Description

The  $\log_2(a)$  function computes the base 2 logarithm of vector elements.

Argument	Result	Error Code
+1	+0	
$a < +0$	QNAN	status::errdom
+0	$-\infty$	status::sing
-0	$-\infty$	status::sing
$-\infty$	QNAN	status::errdom
$+\infty$	$+\infty$	
QNAN	QNAN	
SNAN	QNAN	

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use `log2` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vlog2.cpp
```

**Parent topic:** Exponential and Logarithmic Functions

### 13.2.5.2.3.7 log10

Computes the base 10 logarithm of vector elements.

#### Syntax

Buffer API:

```
void log10 (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event log10 (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

`log10` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

#### Description

The `log10(a)` function computes the base 10 logarithm of vector elements.

Argument	Result	Error Code
+1	+0	
a <+0	QNAN	status::errdom
+0	-∞	status::sing
-0	-∞	status::sing
-∞	QNAN	status::errdom
+∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

$\text{RE}(a)$ $i\cdot\text{IM}(a)$	$-\infty$	$-X$	$-0$	$+0$	$+X$	$+\infty$	NAN
$+i\cdot\infty$	$+\infty + i \frac{3}{4} \frac{\pi}{\ln(10)}$	$+\infty + i \frac{\pi}{2} \frac{1}{\ln(10)}$	$+\infty + i \frac{\pi}{4} \frac{1}{\ln(10)}$	$+\infty+i\cdot\text{QNAN}$			
$+i\cdot Y$	$+\infty + i \frac{\pi}{\ln(10)}$					$+\infty+i\cdot 0$	$\text{QNAN}+i\cdot\text{QNAN}$
$+i\cdot 0$	$+\infty + i \frac{\pi}{\ln(10)}$		$-\infty + i \frac{\pi}{\ln(10)}$	$-\infty+i\cdot 0$		$+\infty+i\cdot 0$	$\text{QNAN}+i\cdot\text{QNAN}$
$-i\cdot 0$	$+\infty - i \frac{\pi}{\ln(10)}$		$-\infty - i \frac{\pi}{\ln(10)}$	$-\infty-i\cdot 0$		$+\infty-i\cdot 0$	$\text{QNAN}-i\cdot\text{QNAN}$
$-i\cdot Y$	$+\infty - i \frac{\pi}{\ln(10)}$					$+\infty-i\cdot 0$	$\text{QNAN}+i\cdot\text{QNAN}$
$-i\cdot\infty$	$+\infty + i \frac{3}{4} \frac{\pi}{\ln(10)}$	$+\infty - i \frac{\pi}{2} \frac{1}{\ln(10)}$	$+\infty - i \frac{\pi}{4} \frac{1}{\ln(10)}$	$+\infty+i\cdot\text{QNAN}$			
$+i\cdot\text{NAN}$	$+\infty+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use `log10` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vlog10.cpp
```

**Parent topic:** Exponential and Logarithmic Functions

### 13.2.5.2.3.8 `log1p`

Computes a natural logarithm of vector elements that are increased by 1.

#### Syntax

Buffer API:

```
void log1p(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode =  
    mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event log1p(queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode =  
    mode::not_defined, error_handler<T> errhandler = {})
```

`log1p` supports the following precisions.

T
float
double

#### Description

The `log1p(a)` function computes a natural logarithm of vector elements that are increased by 1.

Argument	Result	VM Error Status
-1	-∞	<code>status::sing</code>
<i>a</i> < -1	QNAN	<code>status::errdom</code>
+0	+0	
-0	-0	
-∞	QNAN	<code>status::errdom</code>
+∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use `log1p` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vlog1p.cpp
```

**Parent topic:** Exponential and Logarithmic Functions

### 13.2.5.2.3.9 logb

Computes the exponents of the elements of input vector a.

#### Syntax

Buffer API:

```
void logb (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event logb (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

`logb` supports the following precisions.

T
float
double

#### Description

The `logb(a)` function computes the exponents of the elements of the input vector a. For each element  $a_i$  of vector a, this is the integral part of  $\log_2|a_i|$ . The returned value is exact and is independent of the current rounding direction mode.

Argument	Result	VM Error Status
+0	$+\infty$	status::errdom
-0	$-\infty$	status::errdom
$-\infty$	$+\infty$	
$+\infty$	$+\infty$	
QNAN	QNAN	
SNAN	QNAN	

#### Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer a containing input vector of size n.

**mode** Overrides the global VM mode setting for this function call. See `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the 1st input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use `logb` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vlogb.cpp
```

**Parent topic:** Exponential and Logarithmic Functions

### 13.2.5.2.4 Trigonometric Functions

**Parent topic:** VM Mathematical Functions

- `cos` Computes cosine of vector elements.
- `sin` Computes sine of vector elements.
- `sincos` Computes sine and cosine of vector elements.
- `cis` Computes complex exponent of real vector elements (cosine and sine of real vector elements combined to complex value).
- `tan` Computes tangent of vector elements.
- `acos` Computes inverse cosine of vector elements.
- `asin` Computes inverse sine of vector elements.
- `atan` Computes inverse tangent of vector elements.
- `atan2` Computes four-quadrant inverse tangent of elements of two vectors.
- `cospi` Computes the cosine of vector elements multiplied by  $\pi$ .
- `sinpi` Computes the sine of vector elements multiplied by  $\pi$ .
- `tanpi` Computes the tangent of vector elements multiplied by  $\pi$ .
- `acospi` Computes the inverse cosine of vector elements divided by  $\pi$ .
- `asinpi` Computes the inverse sine of vector elements divided by  $\pi$ .

- `atanpi` Computes the inverse tangent of vector elements divided by  $\pi$ .
- `atan2pi` Computes the four-quadrant inverse tangent of the ratios of the corresponding elements of two vectors divided by  $\pi$ .
- `cosd` Computes the cosine of vector elements multiplied by  $\pi/180$ .
- `sind` Computes the sine of vector elements multiplied by  $\pi/180$ .
- `tand` Computes the tangent of vector elements multiplied by  $\pi/180$ .

### 13.2.5.2.4.1 cos

Computes cosine of vector elements.

#### Syntax

Buffer API:

```
void cos (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode =
    mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event cos (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode =
    mode::not_defined, error_handler<T> errhandler = {})
```

`cos` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

#### Description

The `cos(a)` function computes cosine of vector elements.

Note that arguments  $\text{abs}(a[i]) \leq 213$  and  $\text{abs}(a[i]) \leq 216$  for single and double precisions, respectively, are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Argument	Result	VM Error Status
+0	+1	
-0	+1	
$+\infty$	QNAN	status::errdom
$-\infty$	QNAN	status::errdom
QNAN	QNAN	
SNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Cos}(z) = \text{Cosh}(i*z).$$

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use cos can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vcos.cpp
```

**Parent topic:** Trigonometric Functions

### 13.2.5.2.4.2 sin

Computes sine of vector elements.

#### Syntax

Buffer API:

```
void sin(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event sin(queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

`sin` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

#### Description

The `sin(a)` function computes sine of vector elements.

Note that arguments  $\text{abs}(a[i]) \leq 213$  and  $\text{abs}(a[i]) \leq 216$  for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Argument	Result	VM Error Status
+0	+0	
-0	-0	
$+\infty$	QNAN	status::errdom
$-\infty$	QNAN	status::errdom
QNAN	QNAN	
SNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Sin}(z) = -i * \text{Sinh}(i * z).$$

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use sin can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vsin.cpp
```

**Parent topic:** Trigonometric Functions

### 13.2.5.2.4.3 sincos

Computes sine and cosine of vector elements.

#### Syntax

Buffer API:

```
void sincos (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, buffer<T, 1> &z, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event sincos (queue &exec_queue, int64_t n, T *a, T *y, T *z, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

*sincos* supports the following precisions.

T
float
double

#### Description

The sincos(a) function computes sine and cosine of vector elements.

Note that arguments  $\text{abs}(a[i]) \leq 213$  and  $\text{abs}(a[i]) \leq 216$  for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Argument	Result 1	Result 2	Error Code
+0	+0	+1	
-0	-0	+1	
+∞	QNAN	QNAN	status::errdom
-∞	QNAN	QNAN	status::errdom
QNAN	QNAN	QNAN	
SNAN	QNAN	QNAN	

#### Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is *mode*::not\_defined.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output sine vector of size *n*.

**z** The buffer *z* containing the output cosine vector of size *n*.

USM API:

**y** Pointer *y* to the output sine vector of size *n*.

**z** The buffer *z* containing the output cosine vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use sincos can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vsincos.cpp
```

**Parent topic:** Trigonometric Functions

### 13.2.5.2.4.4 cis

Computes complex exponent of real vector elements (cosine and sine of real vector elements combined to complex value).

## Syntax

Buffer API:

```
void cis(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event cis(queue &exec_queue, int64_t n, A *a, R *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

`cis` supports the following precisions.

T	R
<code>float</code>	<code>std::complex&lt;float&gt;</code>
<code>double</code>	<code>std::complex&lt;double&gt;</code>

## Description

The `cis(a)` function computes complex exponent of real vector elements (cosine and sine of real vector elements combined to complex value).

Argument	Result	Error Code
• 0	$+1+i\cdot 0$	
• 0	$+1-i\cdot 0$	
• $\infty$	$\text{QNAN}+i\cdot \text{QNAN}$	<code>status::errdom</code>
• $\infty$	$\text{QNAN}+i\cdot \text{QNAN}$	<code>status::errdom</code>
<code>QNAN</code>	$\text{QNAN}+i\cdot \text{QNAN}$	
<code>SNAN</code>	$\text{QNAN}+i\cdot \text{QNAN}$	

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer `a` containing input vector of size `n`.

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer `a` to the input vector of size `n`.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use cis can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vcis.cpp
```

**Parent topic:** Trigonometric Functions

### 13.2.5.2.4.5 tan

Computes tangent of vector elements.

#### Syntax

Buffer API:

```
void tan (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode =  
    mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event tan (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode =  
    mode::not_defined, error_handler<T> errhandler = {})
```

*tan* supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

#### Description

The *tan(a)* function computes tangent of vector elements.

Note that arguments  $\text{abs}(a[i]) \leq 213$  and  $\text{abs}(a[i]) \leq 216$  for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Argument	Result	Error Code
+0	+0	
-0	-0	
+∞	QNAN	status::errdom
-∞	QNAN	status::errdom
QNAN	QNAN	
SNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Tan}(z) = -i \cdot \text{Tanh}(i \cdot z).$$

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use tan can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vtan.cpp
```

**Parent topic:** Trigonometric Functions

### 13.2.5.2.4.6 acos

Computes inverse cosine of vector elements.

#### Syntax

Buffer API:

```
void acos (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event acos (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

acos supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

#### Description

The acos(a) function computes inverse cosine of vector elements.

Argument	Result	Error Code
+0	$+\pi/2$	
-0	$+\pi/2$	
+1	+0	
-1	$+\pi$	
$ a  > 1$	QNAN	status::errdom
$+\infty$	QNAN	status::errdom
$-\infty$	QNAN	status::errdom
QNAN	QNAN	
SNAN	QNAN	

$\text{RE}(a) + i\text{IM}(a)$	$-\infty$	$-X$	$-0$	$+0$	$+X$	$+\infty$	NAN
$+i\cdot\infty$	$+3\cdot\pi/4 - i\cdot\infty$	$+\pi/2 - i\cdot\infty$	$+\pi/2 - i\cdot\infty$	$+\pi/2 - i\cdot\infty$	$+\pi/2 - i\cdot\infty$	$+\pi/4 - i\cdot\infty$	QNAN - $i\cdot\infty$
$+i\cdot Y$	$+i\cdot\infty$					$+0 - i\cdot\infty$	QNAN + $i\cdot$ QNAN
$+i\cdot 0$	$+i\cdot\infty$		$+i\cdot 0$	$+i\cdot 0$		$+0 - i\cdot\infty$	QNAN + $i\cdot$ QNAN
$-i\cdot 0$	$+i\cdot\infty$		$+i\cdot 0$	$+i\cdot 0$		$+0 + i\cdot\infty$	QNAN + $i\cdot$ QNAN
$-i\cdot Y$	$+i\cdot\infty$					$+0 + i\cdot\infty$	QNAN + $i\cdot$ QNAN
$-i\cdot\infty$	$+3\pi/4 + i\cdot\infty$	$+\pi/2 + i\cdot\infty$	$+\pi/2 + i\cdot\infty$	$+\pi/2 + i\cdot\infty$	$+\pi/2 + i\cdot\infty$	$+\pi/4 + i\cdot\infty$	QNAN + $i\cdot\infty$
$+i\cdot\text{NAN}$	QNAN + $i\cdot\infty$	QNAN + $i\cdot$ QNAN	QNAN + $i\cdot 2 + i\cdot\text{QNA}$	QNAN + $i\cdot 2 + i\cdot$ QNAN	QNAN + $i\cdot$ QNAN	QNAN + $i\cdot\infty$	QNAN + $i\cdot$ QNAN

Notes:

- $\text{acos}(\text{CONJ}(a)) = \text{CONJ}(\text{acos}(a))$ .

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer  $a$  containing input vector of size  $n$ .

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer  $a$  to the input vector of size  $n$ .

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer  $y$  containing the output vector of size  $n$ .

USM API:

**y** Pointer  $y$  to the output vector of size  $n$ .

**return value (event)** Function end event.

## Example

An example of how to use acos can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vacos.cpp
```

**Parent topic:** Trigonometric Functions

### 13.2.5.2.4.7 asin

Computes inverse sine of vector elements.

#### Syntax

Buffer API:

```
void asin(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event asin(queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

asin supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

#### Description

The asin(a) function computes inverse sine of vector elements.

Argument	Result	Error Code
+0	+0	
-0	-0	
+1	$+\pi/2$	
-1	$-\pi/2$	
$ a  > 1$	QNAN	status::errdom
$+\infty$	QNAN	status::errdom
$-\infty$	QNAN	status::errdom
QNAN	QNAN	
SNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

$$\text{asin}(a) = -i \cdot \text{asinh}(i \cdot z).$$

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use asin can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vasin.cpp
```

**Parent topic:** Trigonometric Functions

### 13.2.5.2.4.8 atan

Computes inverse tangent of vector elements.

#### Syntax

Buffer API:

```
void atan(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode =
mode::not_defined)
```

USM API:

```
event atan(queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode =
mode::not_defined)
```

atan supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

#### Description

The atan(a) function computes inverse tangent of vector elements.

Argument	Result	Error Code
+0	+0	
-0	-0	
+∞	+π/2	
-∞	-π/2	
QNAN	QNAN	
SNAN	QNAN	

The atan function does not generate any errors.

Specifications for special values of the complex functions are defined according to the following formula

$$\text{atan}(a) = -i \cdot \text{atanh}(i \cdot a).$$

#### Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer a containing input vector of size n.

**mode** Overrides the global VM mode setting for this function call. See `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use atan can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vatan.cpp
```

**Parent topic:** Trigonometric Functions

### 13.2.5.2.4.9 atan2

Computes four-quadrant inverse tangent of elements of two vectors.

## Syntax

Buffer API:

```
void atan2 (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &b, buffer<T, 1> &y, uint64_t  
mode = mode::not_defined)
```

USM API:

```
event atan2 (queue &exec_queue, int64_t n, T *a, T *b, T *y, vector_class<event> *depends, uint64_t mode  
= mode::not_defined)
```

ad2d supports the following precisions.

T
float
double

## Description

The atan2(a,b) function computes four-quadrant inverse tangent of elements of two vectors.

The elements of the output vector are computed as the four-quadrant arctangent of  $a[i] / b[i]$ .

Argument 1	Argument 2	Result	Error Code
$-\infty$	$-\infty$	$-3\pi/4$	
$-\infty$	$b < +0$	$-\pi/2$	
$-\infty$	$-0$	$-\pi/2$	
$-\infty$	$+0$	$-\pi/2$	
$-\infty$	$b > +0$	$-\pi/2$	
$-\infty$	$+\infty$	$-\pi/4$	
$a < +0$	$-\infty$	$-\pi$	
$a < +0$	$-0$	$-\pi/2$	
$a < +0$	$+0$	$-\pi/2$	
$a < +0$	$+\infty$	$-0$	
$-0$	$-\infty$	$-\pi$	
$-0$	$b < +0$	$-\pi$	
$-0$	$-0$	$-\pi$	
$-0$	$+0$	$-0$	
$-0$	$b > +0$	$-0$	
$-0$	$+\infty$	$-0$	
$+0$	$-\infty$	$+\pi$	
$+0$	$b < +0$	$+\pi$	
$+0$	$-0$	$+\pi$	
$+0$	$+0$	$+0$	
$+0$	$b > +0$	$+0$	
$+0$	$+\infty$	$+0$	
$a > +0$	$-\infty$	$+\pi$	
$a > +0$	$-0$	$+\pi/2$	
$a > +0$	$+0$	$+\pi/2$	
$a > +0$	$+\infty$	$+0$	
$+\infty$	$-\infty$	$+3\pi/4$	
$+\infty$	$b < +0$	$+\pi/2$	
$+\infty$	$-0$	$+\pi/2$	
$+\infty$	$+0$	$+\pi/2$	
$+\infty$	$b > +0$	$+\pi/2$	
$+\infty$	$+\infty$	$+\pi/4$	
$a > +0$	QNAN	QNAN	
$a > +0$	SNAN	QNAN	
QNAN	$b > +0$	QNAN	
SNAN	$b > +0$	QNAN	
QNAN	QNAN	QNAN	
QNAN	SNAN	QNAN	
SNAN	QNAN	QNAN	
SNAN	SNAN	QNAN	

The atan2(a,b) function does not generate any errors.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing 1st input vector of size *n*.

**b** The buffer *b* containing 2nd input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the 1st input vector of size *n*.

**b** Pointer *b* to the 2nd input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use atan2 can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vatan2.cpp
```

**Parent topic:** Trigonometric Functions

### 13.2.5.2.4.10 cospi

Computes the cosine of vector elements multiplied by  $\pi$ .

## Syntax

Buffer API:

```
void cospi (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode =
    mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event cospi (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode =
    mode::not_defined, error_handler<T> errhandler = {})
```

**cospi** supports the following precisions.:

T
float
double

## Description

The **cospi(a)** function computes the cosine of vector elements multiplied by  $\pi$ . For an argument  $a$ , the function computes  $\cos(\pi \cdot a)$ .

Argument	Result	Error Code
+0	+1	
-0	+1	
$n + 0.5$ , for any integer $n$ where $n + 0.5$ is representable	+0	
$+\infty$	QNAN	status::errdom
$-\infty$	QNAN	status::errdom
QNAN	QNAN	
SNAN	QNAN	

If arguments  $\text{abs}(a_i) \leq 2^{22}$  for single precision or  $\text{abs}(a_i) \leq 2^{51}$  for double precision, they belong to the *fast computational path*: arguments for which VM provides the best possible performance. Avoid arguments which do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer  $a$  containing input vector of size  $n$ .

**mode** Overrides the global VM mode setting for this function call. See [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use cospi can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vcospis.cpp
```

**Parent topic:** Trigonometric Functions

### 13.2.5.2.4.11 sinpi

Computes the sine of vector elements multiplied by  $\pi$ .

## Syntax

Buffer API:

```
void sinpi (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode =
    mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event sinpi (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode =
    mode::not_defined, error_handler<T> errhandler = {})
```

`sinpi` supports the following precisions.

T
float
double

## Description

The `sinpi(a)` function computes the sine of vector elements multiplied by  $\pi$ . For an argument  $a$ , the function computes  $\sin(\pi^*a)$ .

Argument	Result	Error Code
+0	+0	
-0	-0	
+n, positive integer	+0	
-n, negative integer	-0	
+∞	QNAN	<code>status::errdom</code>
-∞	QNAN	<code>status::errdom</code>
QNAN	QNAN	
SNAN	QNAN	

If arguments  $\text{abs}(a_i) \leq 2^{22}$  for single precision or  $\text{abs}(a_i) \leq 2^{51}$  for double precision, they belong to the *fast computational path*: arguments for which VM provides the best possible performance. Avoid arguments which do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer  $a$  containing input vector of size  $n$ .

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer  $a$  to the input vector of size  $n$ .

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer  $y$  containing the output vector of size  $n$ .

USM API:

**y** Pointer  $y$  to the output vector of size  $n$ .

**return value (event)** Function end event.

## Example

An example of how to use sinpi can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vsinpi.cpp
```

**Parent topic:** Trigonometric Functions

### 13.2.5.2.4.12 tanpi

Computes the tangent of vector elements multiplied by  $\pi$ .

#### Syntax

Buffer API:

```
void tanpi (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event tanpi (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

`tanpi` supports the following precisions.

T
float
double

#### Description

The `tanpi(a)` function computes the tangent of vector elements multiplied by  $\pi$ . For an argument  $a$ , the function computes  $\tan(\pi^*a)$ .

Argument	Result	Error Code
+0	+0	
-0	+0	
n, even integer	*copysign(0.0, n)	
n, odd integer	*copysign(0.0, -n)	
n + 0.5, for n even integer and n + 0.5 representable	+∞	
n + 0.5, for n odd integer and n + 0.5 representable	-∞	
+∞	QNAN	status::errdom
-∞	QNAN	status::errdom
QNAN	QNAN	
SNAN	QNAN	

The `copysign(x, y)` function returns the first vector argument `x` with the sign changed to match that of the second argument `y`.

If arguments  $\text{abs}(a_i) \leq 2^{13}$  for single precision or  $\text{abs}(a_i) \leq 2^{67}$  for double precision, they belong to the *fast computational path*: arguments for which VM provides the best possible performance. Avoid arguments with do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer `a` containing input vector of size `n`.

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer `a` to the input vector of size `n`.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer  $y$  containing the output vector of size  $n$ .

USM API:

**y** Pointer  $y$  to the output vector of size  $n$ .

**return value (event)** Function end event.

## Example

An example of how to use tanpi can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vtanpi.cpp
```

**Parent topic:** Trigonometric Functions

### 13.2.5.2.4.13 `acospi`

Computes the inverse cosine of vector elements divided by  $\pi$ .

#### Syntax

Buffer API:

```
void acospi (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event acospi (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

`acospi` supports the following precisions.

T
float
double

#### Description

The `acospi(a)` function computes the inverse cosine of vector elements divided by  $\pi$ . For an argument  $a$ , the function computes  $\text{acos}(a)/\pi$ .

Argument	Result	Error Code
+0	+1/2	
-0	+1/2	
+1	+0	
-1	+1	
$ a  > 1$	QNAN	status::errdom
$+\infty$	QNAN	status::errdom
$-\infty$	QNAN	status::errdom
QNAN	QNAN	
SNAN	QNAN	

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use acospi can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vacospi.cpp
```

**Parent topic:** Trigonometric Functions

### 13.2.5.2.4.14 asinpi

Computes the inverse sine of vector elements divided by  $\pi$ .

#### Syntax

Buffer API:

```
void asinpi (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event asinpi (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

asinpi supports the following precisions.

T
float
double

#### Description

The asinpi(a) function computes the inverse sine of vector elements divided by  $\pi$ . For an argument a, the function computes  $\text{asinpi}(a)/\pi$ .

Argument	Result	Error Code
+0	+0	
-0	-0	
+1	+1/2	
-1	-1/2	
a  > 1	QNAN	status::errdom
+∞	QNAN	status::errdom
-∞	QNAN	status::errdom
QNAN	QNAN	
SNAN	QNAN	

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use asinpi can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vasinpi.cpp
```

**Parent topic:** Trigonometric Functions

### 13.2.5.2.4.15 atanpi

Computes the inverse tangent of vector elements divided by  $\pi$ .

#### Syntax

Buffer API:

```
void atanpi (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined)
```

USM API:

```
event atanpi (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined)
```

atanpi supports the following precisions.

T
float
double

#### Description

The atanpi(a) function computes the inverse tangent of vector elements divided by  $\pi$ . For an argument a, the function computes  $\text{atan}(a)/\pi$ .

Argument	Result	Error Code
+0	+0	
-0	-0	
$+\infty$	$+1/2$	
$-\infty$	$-1/2$	
QNaN	QNaN	
SNaN	QNaN	

The atanpi function does not generate any errors.

#### Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer a containing input vector of size n.

**mode** Overrides the global VM mode setting for this function call. See [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is mode::not\_defined.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer a to the input vector of size n.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

**y** The buffer **y** containing the output vector of size **n**.

USM API:

**y** Pointer **y** to the output vector of size **n**.

**return value (event)** Function end event.

## Example

An example of how to use atanpi can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vatanpi.cpp
```

**Parent topic:** Trigonometric Functions

### 13.2.5.2.4.16 atan2pi

Computes the four-quadrant inverse tangent of the ratios of the corresponding elements of two vectors divided by  $\pi$ .

## Syntax

Buffer API:

```
void atan2pi (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &b, buffer<T, 1> &y, uint64_t  
mode = mode::not_defined)
```

USM API:

```
event atan2pi (queue &exec_queue, int64_t n, T *a, T *b, T *y, vector_class<event> *depends, uint64_t  
mode = mode::not_defined)
```

atan2pi supports the following precisions.

T
float
double

## Description

The atan2pi(a,b) function computes the four-quadrant inverse tangent of the ratios of the corresponding elements of two vectors divided by  $\pi$ .

For the elements of the output vector  $y$ , the function computers the four-quadrant arctangent of  $a_i/b_i$ , with the result divided by  $\pi$ .

Argument 1	Argument 2	Result	Error Code
$-\infty$	$-\infty$	$-3/4$	
$-\infty$	$b < +0$	$-1/2$	
$-\infty$	$-0$	$+1/2$	
$-\infty$	$+0$	$-1/2$	
$-\infty$	$x > +0$	$-1/2$	
$-\infty$	$+\infty$	$-1/4$	
$a < +0$	$-\infty$	$-1$	
$a < +0$	$-0$	$-1/2$	
$a < +0$	$+0$	$-1/2$	
$a < +0$	$+\infty$	$-0$	
$-0$	$-\infty$	$-1$	
$-0$	$b < +0$	$-1$	
$-0$	$-0$	$-1$	
$-0$	$+0$	$-0$	
$-0$	$b > +0$	$-0$	
$-0$	$+\infty$	$-0$	
$+0$	$-\infty$	$+1$	
$+0$	$b < +0$	$+1$	
$+0$	$-0$	$+1$	
$+0$	$+0$	$+0$	
$+0$	$b > +0$	$+0$	
$+0$	$+\infty$	$+0$	
$a > +0$	$-\infty$	$+1$	
$a > +0$	$-0$	$+1/2$	
$x > +0$	$+0$	$+1/2$	
$a > +0$	$+\infty$	$+1/4$	
$+\infty$	$-\infty$	$+3/4$	
$+\infty$	$b < +0$	$+1/2$	
$+\infty$	$-0$	$+1/2$	
$+\infty$	$+0$	$+1/2$	
$+\infty$	$b > +0$	$+1/2$	
$+\infty$	$+\infty$	$+1/4$	
$a > +0$	QNAN	QNAN	
$a > +0$	SNAN	QNAN	
QNAN	$b > +0$	QNAN	
SNAN	$x > +0$	QNAN	
QNAN	QNAN	QNAN	
QNAN	SNAN	QNAN	
SNAN	QNAN	QNAN	
SNAN	SNAN	QNAN	

The atan2pi(a,b) function does not generate any errors.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing 1st input vector of size *n*.

**b** The buffer *b* containing 2nd input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the 1st input vector of size *n*.

**b** Pointer *b* to the 2nd input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use atan2pi can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vatan2pi.cpp
```

**Parent topic:** Trigonometric Functions

### 13.2.5.2.4.17 cosd

Computes the cosine of vector elements multiplied by  $\pi/180$ .

## Syntax

Buffer API:

```
void cosd(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event cosd(queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

`cosd` supports the following precisions.

T
float
double

## Description

The `cosd(a)` function is a degree argument trigonometric function. It computes the cosine of vector elements multiplied by  $\pi/180$ . For an argument  $a$ , the function computes  $\cos(\pi*a/180)$ .

Note that arguments  $\text{abs}(a_i) \leq 2^{24}$  for single precision or  $\text{abs}(a_i) \leq 2^{52}$  for double precision, they belong to the *fast computational path*: trigonometric function arguments for which VM provides the best possible performance. Avoid arguments with do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

Argument	Result	Error Code
+0	+1	
-0	+1	
$+\infty$	QNAN	status::errdom
$-\infty$	QNAN	status::errdom
QNAN	QNAN	
SNAN	QNAN	

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer  $a$  containing input vector of size  $n$ .

**mode** Overrides the global VM mode setting for this function call. See `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use `cosd` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vcosd.cpp
```

**Parent topic:** Trigonometric Functions

### 13.2.5.2.4.18 sind

Computes the sine of vector elements multiplied by  $\pi/180$ .

## Syntax

Buffer API:

```
void sind(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event sind(queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

`sind` supports the following precisions.

T
float
double

## Description

The  $\text{sind}(a)$  function is a degree argument trigonometric function. It computes the sine of vector elements multiplied by  $\pi/180$ . For an argument  $a$ , the function computes  $\sin(\pi*a/180)$ .

Note that arguments  $\text{abs}(a_i) \leq 2^{24}$  for single precision or  $\text{abs}(a_i) \leq 2^{52}$  for double precision, they belong to the *fast computational path*: trigonometric function arguments for which VM provides the best possible performance. Avoid arguments with do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

Argument	Result	Error Code
+0	+0	
-0	-0	
$+\infty$	QNAN	<code>status::errdom</code>
$-\infty$	QNAN	<code>status::errdom</code>
QNAN	QNAN	
SNAN	QNAN	

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer  $a$  containing input vector of size  $n$ .

**mode** Overrides the global VM mode setting for this function call. See [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer  $a$  to the input vector of size  $n$ .

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer  $y$  containing the output vector of size  $n$ .

USM API:

**y** Pointer  $y$  to the output vector of size  $n$ .

**return value (event)** Function end event.

## Example

An example of how to use sind can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vsind.cpp
```

**Parent topic:** Trigonometric Functions

### 13.2.5.2.4.19 tand

Computes the tangent of vector elements multiplied by  $\pi/180$ .

#### Syntax

Buffer API:

```
void tand(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event tand(queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

tand supports the following precisions.

T
float
double

#### Description

The tand(a) function computes the tangent of vector elements multiplied by  $\pi/180$ . For an argument  $x$ , the function computes  $\tan(\pi^*x/180)$ .

Note that arguments  $\text{abs}(a_i) \leq 2^{38}$  for single precision or  $\text{abs}(a_i) \leq 2^{67}$  for double precision, they belong to the *fast computational path*: trigonometric function arguments for which VM provides the best possible performance. Avoid arguments with do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

Argument	Result	Error Code
+0	+1	
-0	+1	
$\pm\infty$	QNAN	status::errdom
$\pm\infty$	QNAN	status::errdom
QNAN	QNAN	
SNAN	QNAN	

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use `tand` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vtand.cpp
```

**Parent topic:** Trigonometric Functions

### 13.2.5.2.5 Hyperbolic Functions

**Parent topic:** VM Mathematical Functions

- `cosh` Computes hyperbolic cosine of vector elements.
- `sinh` Computes hyperbolic sine of vector elements.
- `tanh` Computes hyperbolic tangent of vector elements.
- `acosh` Computes inverse hyperbolic cosine (nonnegative) of vector elements.
- `asinh` Computes inverse hyperbolic sine of vector elements.
- `atanh` Computes inverse hyperbolic tangent of vector elements.

#### 13.2.5.2.5.1 `cosh`

Computes hyperbolic cosine of vector elements.

### Syntax

Buffer API:

```
void cosh (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode =  
    mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event cosh (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode =  
    mode::not_defined, error_handler<T> errhandler = {})
```

`cosh` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

## Description

The  $\cosh(a)$  function computes hyperbolic cosine of vector elements.

Data Type	Threshold Limitations on Input Parameters
single precision	$-\text{Log}(\text{FLT\_MAX}) - \text{Log}_2 < a[i] < \text{Log}(\text{FLT\_MAX}) + \text{Log}_2$
double precision	$-\text{Log}(\text{DBL\_MAX}) - \text{Log}_2 < a[i] < \text{Log}(\text{DBL\_MAX}) + \text{Log}_2$

Argument	Result	Error Code
+0	+1	
-0	+1	
$X > \text{overflow}$	$+\infty$	<code>status::overflow</code>
$X < -\text{overflow}$	$+\infty$	<code>status::overflow</code>
$+\infty$	$+\infty$	
$-\infty$	$+\infty$	
QNAN	QNAN	
SNAN	QNAN	

$+i\infty$	$+\infty+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$
$+i\cdot Y$	$+\infty\cdot\text{Cos}(Y) - i\cdot\infty\cdot\text{Sin}(Y)$				$+\infty\cdot\text{CIS}(Y)$	$\text{QNAN}+i\cdot\text{QNAN}$
$+i\cdot 0$	$+\infty\cdot i\cdot 0$		$+1\cdot i\cdot 0$	$+1+i\cdot 0$		$+1\cdot i\cdot 0$
$-i\cdot 0$	$+\infty+i\cdot 0$		$+1+i\cdot 0$	$+1-i\cdot 0$		$+1\cdot i\cdot 0$
$-i\cdot Y$	$+\infty\cdot\text{Cos}(Y) - i\cdot\infty\cdot\text{Sin}(Y)$				$+\infty\cdot\text{CIS}(Y)$	$\text{QNAN}+i\cdot\text{QNAN}$
$-i\cdot\infty$	$+\infty+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$	$\text{QNAN}-i\cdot 0$	$\text{QNAN}+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$
$+i\cdot\text{NAN}$	$+\infty+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$

Notes:

- The complex  $\cosh(a)$  function sets the VM Error Status to `status::overflow` in the case of overflow, that is, when  $\text{RE}(a), \text{IM}(a)$  are finite non-zero numbers, but the real or imaginary part of the exact result is so large that it does not meet the target precision.
- $\cosh(\text{CONJ}(a)) = \text{CONJ}(\cosh(a))$
- $\cosh(-a) = \cosh(a)$ .

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use `cosh` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vcosh.cpp
```

**Parent topic:** Hyperbolic Functions

### 13.2.5.2.5.2 sinh

Computes hyperbolic sine of vector elements.

## Syntax

Buffer API:

```
void sinh (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event sinh (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

`sinh` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The `sinh(a)` function computes hyperbolic sine of vector elements.

Data Type	Threshold Limitations on Input Parameters
single precision	$-\text{Log}(\text{FLT\_MAX}) - \text{Log}(2) < a[i] < \text{Log}(\text{FLT\_MAX}) + \text{Log}(2)$
double precision	$-\text{Log}(\text{DBL\_MAX}) - \text{Log}(2) < a[i] < \text{Log}(\text{DBL\_MAX}) + \text{Log}(2)$

Argument	Result	Error Code
+0	+0	
-0	-0	
$a > \text{overflow}$	$+\infty$	<code>status::overflow</code>
$a < -\text{overflow}$	$-\infty$	<code>status::overflow</code>
$+\infty$	$+\infty$	
$-\infty$	$-\infty$	
QNAN	QNAN	
SNAN	QNAN	

$+i\cdot\infty$	$-\infty+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$ $0+i\cdot\text{QNAN}$	$+0+i\cdot\text{QNAN}$ $\text{QNAN}+i\cdot\text{QNAN}$	$-\infty+i\cdot\text{QNAN}$ $\text{QNAN}+i\cdot\text{QNAN}$	
$+i\cdot Y$	$-\infty\cdot\text{Cos}(Y)+i\cdot\infty\cdot\text{Sin}(Y)$				$+\infty\cdot\text{CIS}(Y)$ $\text{QNAN}+i\cdot\text{QNAN}$
$+i\cdot 0$	$-\infty+i\cdot 0$		$-0+i\cdot 0$	$+0+i\cdot 0$	$+\infty+i\cdot 0$ $\text{QNAN}+i\cdot 0$
$-i\cdot 0$	$-\infty-i\cdot 0$		$-0-i\cdot 0$	$+0-i\cdot 0$	$+\infty-i\cdot 0$ $\text{QNAN}-i\cdot 0$
$-i\cdot Y$	$-\infty\cdot\text{Cos}(Y)+i\cdot\infty\cdot\text{Sin}(Y)$				$+\infty\cdot\text{CIS}(Y)$ $\text{QNAN}+i\cdot\text{QNAN}$
$-i\cdot\infty$	$-\infty+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$ $0+i\cdot\text{QNAN}$	$+0+i\cdot\text{QNAN}$ $\text{QNAN}+i\cdot\text{QNAN}$	$-\infty+i\cdot\text{QNAN}$ $\text{QNAN}+i\cdot\text{QNAN}$	
$+i\cdot\text{NAN}$	$-\infty+i\cdot\text{QNAN}$	$\text{QNAN}+i\cdot\text{QNAN}$ $0+i\cdot\text{QNAN}$	$+0+i\cdot\text{QNAN}$ $\text{QNAN}+i\cdot\text{QNAN}$	$-\infty+i\cdot\text{QNAN}$ $\text{QNAN}+i\cdot\text{QNAN}$	

Notes:

- The complex `sinh(a)` function sets the VM Error Status to `status::overflow` in the case of overflow, that is, when `RE(a)`, `IM(a)` are finite non-zero numbers, but the real or imaginary part of the exact result is so large that it does not meet the target precision.
- $\text{sinh}(\text{CONJ}(a)) = \text{CONJ}(\text{sinh}(a))$
- $\text{sinh}(-a) = -\text{sinh}(a)$ .

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use sinh can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vsinh.cpp
```

**Parent topic:** Hyperbolic Functions

### 13.2.5.2.5.3 tanh

Computes hyperbolic tangent of vector elements.

#### Syntax

Buffer API:

```
void tanh (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined)
```

USM API:

```
event tanh (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined)
```

`tanh` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

#### Description

The `tanh(a)` function computes hyperbolic tangent of vector elements.

Argument	Result	Error Code
+0	+0	
-0	-0	
+∞	+1	
-∞	-1	
QNAN	QNAN	
SNAN	QNAN	

+i·∞	-1+i·0	QNAN+i·QNANQNAN+i·QNANQNAN+i·QNANQNAN+i·QNANQNAN+i·QNANQNAN+i·QNAN+1+i·0	QNAN+i·QNAN
+i·Y	-		+1+i·0·Tan(Y)QNAN+i·QNAN
	1+i·0·Tan(Y)		
+i·0	-1+i·0	-0+i·0	+0+i·0
-i·0	-1-i·0	-0-i·0	+0-i·0
-i·Y	-		+1+i·0·Tan(Y)QNAN+i·QNAN
	1+i·0·Tan(Y)		
-i·∞	-1-i·0	QNAN+i·QNANQNAN+i·QNANQNAN+i·QNANQNAN+i·QNANQNAN+i·QNAN+1-i·0	QNAN+i·QNAN
+i·NAN	-1+i·0	QNAN+i·QNANQNAN+i·QNANQNAN+i·QNANQNAN+i·QNANQNAN+i·QNAN+1+i·0	QNAN+i·QNAN

Notes:

- $\tanh(\text{CONJ}(a)) = \text{CONJ}(\tanh(a))$
- $\tanh(-a) = -\tanh(a)$ .

The `tanh(a)` function does not generate any errors.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use tanh can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vtanh.cpp
```

**Parent topic:** Hyperbolic Functions

### 13.2.5.2.5.4 acosh

Computes inverse hyperbolic cosine (nonnegative) of vector elements.

## Syntax

Buffer API:

```
void acosh (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode =
    mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event acosh (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode =
    mode::not_defined, error_handler<T> errhandler = {})
```

`acosh` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The `acosh(a)` function computes inverse hyperbolic cosine (nonnegative) of vector elements.

Argument	Result	Error Code
+1	+0	
a < +1	QNAN	<code>status::errdom</code>
-∞	QNAN	<code>status::errdom</code>
+∞	+∞	
QNAN	QNAN	
SNAN	QNAN	

RE(a) i·IM(a)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	$+\infty + i \cdot \frac{3\pi}{4}$	$+\infty + i \cdot \pi/2$	$+\infty + i \cdot \pi/4$	$+\infty + i \cdot \text{QNAN}$			
+i·Y	$+\infty + i \cdot \pi$					$+\infty + i \cdot 0$	$\text{QNAN} + i \cdot \text{QNAN}$
+i·0	$+\infty + i \cdot \pi$		$+0 + i \cdot \pi/2$	$+0 + i \cdot \pi/2$		$+0 + i \cdot 0$	$\text{QNAN} + i \cdot \text{QNAN}$
-i·0	$+\infty + i \cdot \pi$		$+0 + i \cdot \pi/2$	$+0 + i \cdot \pi/2$		$+0 + i \cdot 0$	$\text{QNAN} + i \cdot \text{QNAN}$
-i·Y	$+\infty + i \cdot \pi$					$+0 + i \cdot 0$	$\text{QNAN} + i \cdot \text{QNAN}$
-i·∞	$+\infty - i \cdot \frac{3\pi}{4}$	$+\infty - i \cdot \pi/2$	$+\infty - i \cdot \pi/4$	$+\infty + i \cdot \text{QNAN}$			
+i·NAN	$+\infty + i \cdot \text{QNAN}$	QNAN + i·QNAN					

Notes:

- $\text{acosh}(\text{CONJ}(a)) = \text{CONJ}(\text{acosh}(a))$ .

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use `acosh` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vacosh.cpp
```

**Parent topic:** Hyperbolic Functions

### 13.2.5.2.5.5 asinh

Computes inverse hyperbolic sine of vector elements.

#### Syntax

Buffer API:

```
void asinh (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined)
```

USM API:

```
event asinh (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined)
```

`asinh` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

#### Description

The `asinh(a)` function computes inverse hyperbolic sine of vector elements.

Argument	Result	Error Code
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
QNAN	QNAN	
SNAN	QNAN	

RE(a) i·IM(a)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	-∞+i·π/4	-∞+i·π/2	+∞+i·π/2	+∞+i·π/2	+∞+i·π/2	+∞+i·π/4	+∞+i·QNAN
+i·Y	-∞+i·0					+∞+i·0	QNAN+i·QNAN
+i·0	+∞+i·0		+0+i·0	+0+i·0		+∞+i·0	QNAN+i·QNAN
-i·0	-∞-i·0		-0-i·0	+0-i·0		+∞-i·0	QNAN- i·QNAN
-i·Y	-∞-i·0					+∞-i·0	QNAN+i·QNAN
-i·∞	-∞-i·π/4	-∞-i·π/2	-∞-i·π/2	+∞-i·π/2	+∞-i·π/2	+∞-i·π/4	+∞+i·QNAN
+i·NAN	- ∞+i·QNAN	QNAN+i·QNAN QNAN+i·QNAN	QNAN+i·QNAN QNAN+i·QNAN	QNAN+i·QNAN QNAN+i·QNAN	QNAN+i·QNAN QNAN+i·QNAN	QNAN+i·QNAN QNAN+i·QNAN	QNAN+i·QNAN QNAN+i·QNAN

The `asinh(a)` function does not generate any errors.

Notes:

- `asinh(CONJ(a)) = CONJ(asinh(a))`

- $\text{asinh}(-a) = -\text{asinh}(a)$ .

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use `asinh` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vasinhp.cpp
```

**Parent topic:** Hyperbolic Functions

### 13.2.5.2.5.6 atanh

Computes inverse hyperbolic tangent of vector elements.

## Syntax

Buffer API:

```
void atanh (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode =
    mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event atanh (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode =
    mode::not_defined, error_handler<T> errhandler = {})
```

`atanh` supports the following precisions.

T
float
double
<code>std::complex&lt;float&gt;</code>
<code>std::complex&lt;double&gt;</code>

## Description

The `atanh(a)` function computes inverse hyperbolic tangent of vector elements.

Argument	Result	Error Code
+1	+∞	<code>status::sing</code>
-1	-∞	<code>status::sing</code>
a  > 1	QNAN	<code>status::errdom</code>
-∞	QNAN	<code>status::errdom</code>
+∞	+∞	<code>status::errdom</code>
QNAN	QNAN	
SNAN	QNAN	

RE(a) i·IM(a)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	-0+i·π/2	-0+i·π/2	-0+i·π/2	+0+i·π/2	+0+i·π/2	+0+i·π/2	+0+i·π/2
+i·Y	-0+i·π/2					+0+i·π/2	QNAN+i·QNAN
+i·0	-0+i·π/2		-0+i·0	+0+i·0		+0+i·π/2	QNAN+i·QNAN
-i·0	-0-i·π/2		-0-i·0	+0-i·0		+0-i·π/2	QNAN- i·QNAN
-i·Y	-0-i·π/2					+0-i·π/2	QNAN+i·QNAN
-i·∞	-0-i·π/2	-0-i·π/2	-0-i·π/2	+0-i·π/2	+0-i·π/2	+0-i·π/2	+0-i·π/2
+i·NAN	- 0+i·QNAN	QNAN+i·QNAN- 0+i·QNAN		+0+i·QNAN	QNAN+i·QNAN+0+i·QNAN	QNAN+i·QNAN	

Notes:

- `atanh(±1±i·0) = ±∞±i·0`, and `status::sing` error is generated
- `atanh(CONJ(a)) = CONJ(atanh(a))`
- `atanh(-a) == -atanh(a)`.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use atanh can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vataanh.cpp
```

**Parent topic:** Hyperbolic Functions

### 13.2.5.2.6 Special Functions

**Parent topic:** VM Mathematical Functions

- `erf` Computes the error function value of vector elements.
- `erfc` Computes the complementary error function value of vector elements.
- `cdfnorm` Computes the cumulative normal distribution function values of vector elements.
- `erfinv` Computes inverse error function value of vector elements.
- `erfcinv` Computes the inverse complementary error function value of vector elements.
- `cdfnorminv` Computes the inverse cumulative normal distribution function values of vector elements.
- `lgamma` Computes the natural logarithm of the absolute value of gamma function for vector elements.
- `tgamma` Computes the gamma function of vector elements.
- `expint1` Computes the exponential integral of vector elements.

#### 13.2.5.2.6.1 erf

Computes the error function value of vector elements.

##### Syntax

Buffer API:

```
void erf(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined)
```

USM API:

```
event erf(queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined)
```

`erf` supports the following precisions.

T
float
double

##### Description

The `erf` function computes the error function values for elements of the input vector `a` and writes them to the output vector `y`.

The error function is defined as given by:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt .$$

Useful relations:

$$1. \quad \text{erfc}(x) = 1 - \text{erf}(x),$$

where  $\text{erfc}$  is the complementary error function.

$$2. \quad \Phi(x) = \frac{1}{2} \text{erf}(x/\sqrt{2}),$$

where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp(-t^2/2) dt$$

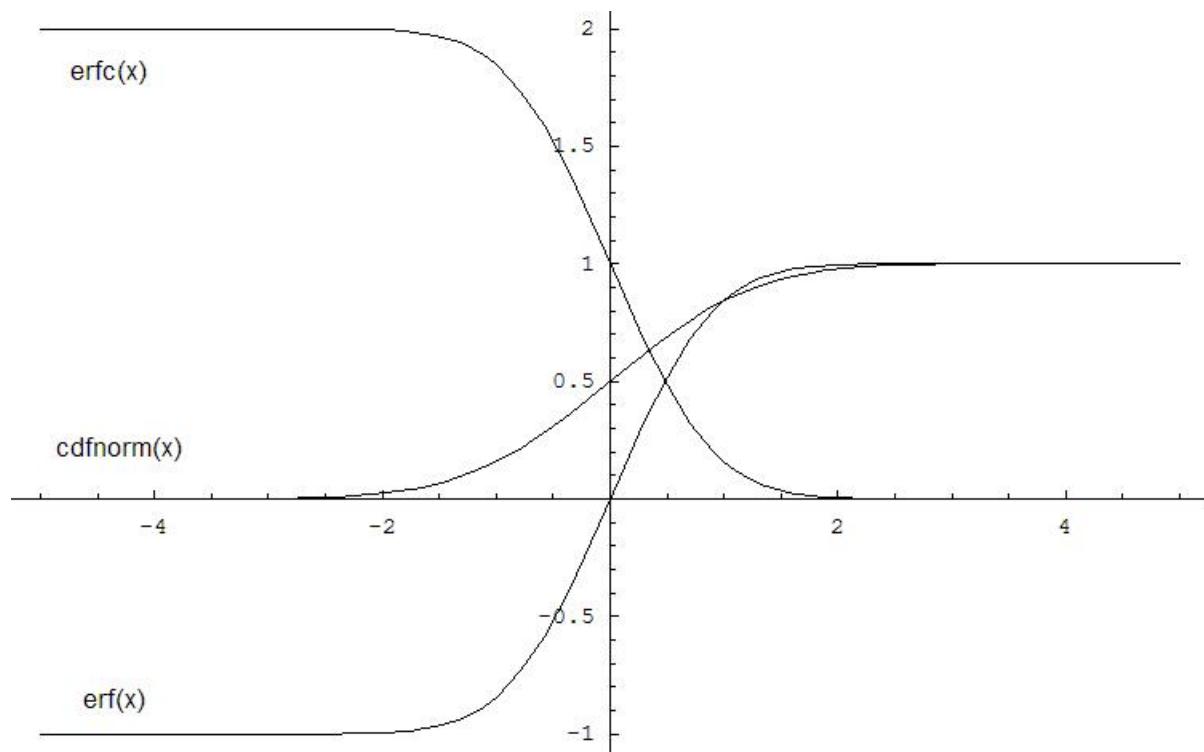
is the cumulative normal distribution function.

$$3. \quad \Phi^{-1}(x) = \sqrt{2} \text{erf}^{-1}(2x - 1),$$

where  $\phi^{-1}(x)$  and  $\text{erf}^{-1}(x)$  are the inverses to  $\phi(x)$  and  $\text{erf}(x)$ , respectively.

The following figure illustrates the relationships among erf family functions (erf, erfc, cdfnorm).

erf Family Functions Relationship |



Useful relations for these functions:

$$\text{erf}(x) + \text{erfc}(x) = 1$$

$$\text{cdfnorm}(x) = \frac{1}{2} \left( 1 + \text{erf} \left( \frac{x}{\sqrt{2}} \right) \right) = 1 - \frac{1}{2} \text{erfc} \left( \frac{x}{\sqrt{2}} \right)$$

Argument	Result	Error Code
$+\infty$	+1	
$-\infty$	-1	
QNAN	QNAN	
SNAN	QNAN	

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use erf can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/verf.cpp
```

**Parent topic:** Special Functions

### 13.2.5.2.6.2 erfc

Computes the complementary error function value of vector elements.

#### Syntax

Buffer API:

```
void erfc (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event erfc (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

`erfc` supports the following precisions.

T
float
double

#### Description

The `erfc` function computes the complementary error function values for elements of the input vector `a` and writes them to the output vector `y`.

The complementary error function is defined as follows:

$$\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt .$$

Useful relations:

1.  $\text{erfc}(x) = 1 - \text{erf}(x).$

2.  $\Phi(x) = \frac{1}{2} \text{erf}(x/\sqrt{2}),$

where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp(-t^2/2) dt$$

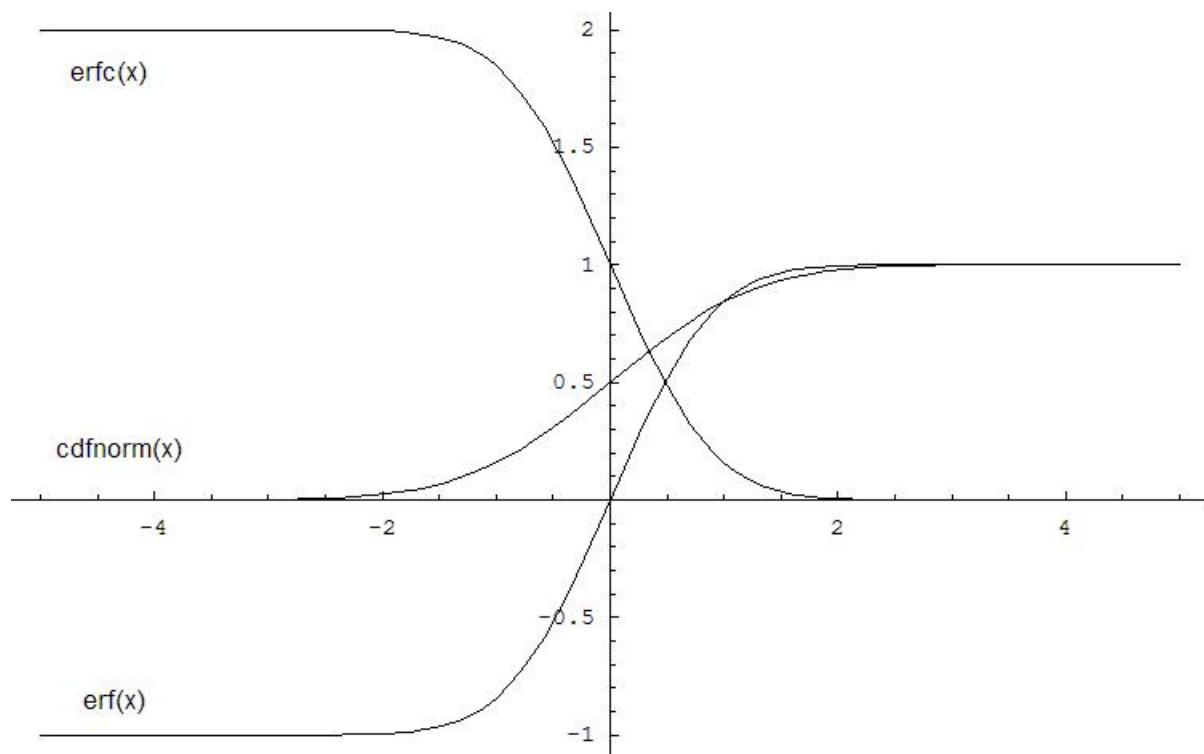
is the cumulative normal distribution function.

$$3. \quad \Phi^{-1}(x) = \sqrt{2} \operatorname{erf}^{-1}(2x - 1),$$

where  $\phi^{-1}(x)$  and  $\operatorname{erf}^{-1}(x)$  are the inverses to  $\phi(x)$  and  $\operatorname{erf}(x)$ , respectively.

The following figure illustrates the relationships among erf family functions (erf, erfc, cdfnorm).

#### erfc Family Functions Relationship I



Useful relations for these functions:

$$\operatorname{erf}(x) + \operatorname{erfc}(x) = 1$$

$$\operatorname{cdfnorm}(x) = \frac{1}{2} \left( 1 + \operatorname{erf} \left( \frac{x}{\sqrt{2}} \right) \right) = 1 - \frac{1}{2} \operatorname{erfc} \left( \frac{x}{\sqrt{2}} \right)$$

Argument	Result	Error Code
a > underflow	+0	status::underflow
+∞	+0	
-∞	+2	
QNAN	QNAN	
SNAN	QNAN	

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer a containing input vector of size n.

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer a to the input vector of size n.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer y containing the output vector of size n.

USM API:

**y** Pointer y to the output vector of size n.

**return value (event)** Function end event.

## Example

An example of how to use erfc can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/verfc.cpp
```

**Parent topic:** Special Functions

### 13.2.5.2.6.3 cdfnorm

Computes the cumulative normal distribution function values of vector elements.

#### Syntax

Buffer API:

```
void cdfnorm(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event cdfnorm(queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

cdfnorm supports the following precisions.

T
float
double

#### Description

The cdfnorm function computes the cumulative normal distribution function values for elements of the input vector *a* and writes them to the output vector *y*.

The cumulative normal distribution function is defined as given by:

$$\text{CdfNorm}(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{t^2}{2}} dt .$$

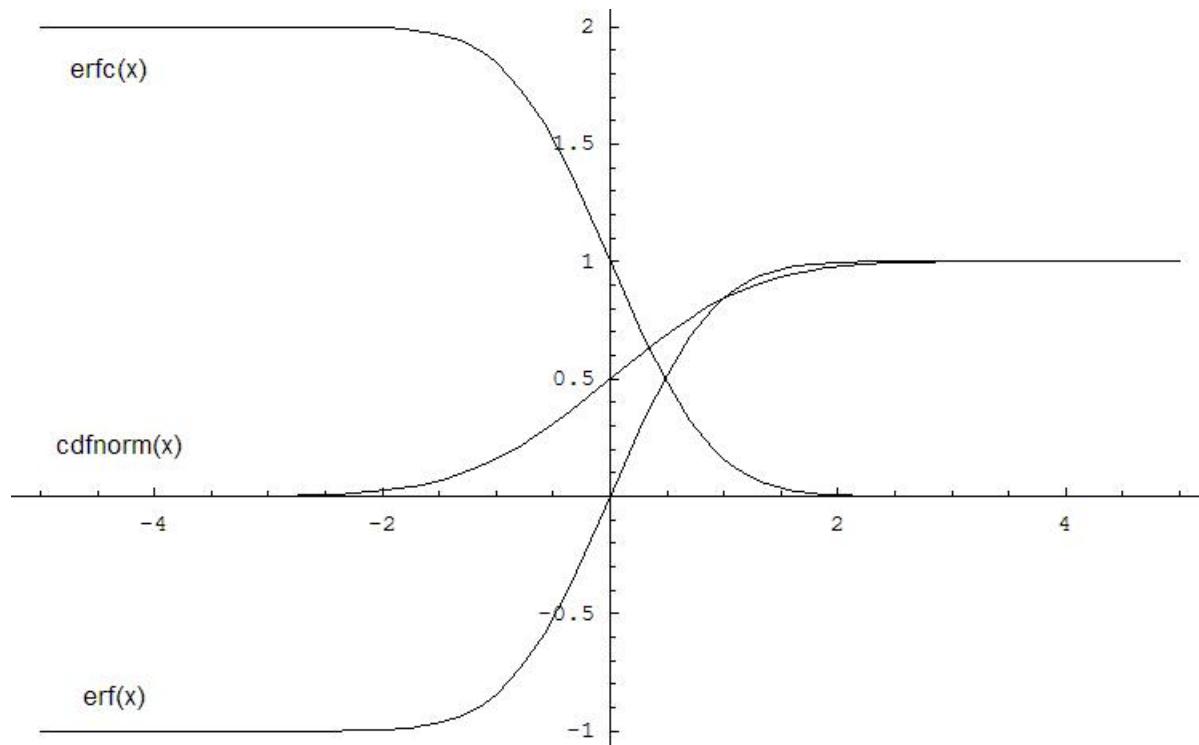
Useful relations:

$$\text{cdfnorm}(x) = \frac{1}{2} \left( 1 + \text{erf} \left( \frac{x}{\sqrt{2}} \right) \right) = 1 - \frac{1}{2} \text{erfc} \left( \frac{x}{\sqrt{2}} \right)$$

where erf and erfc are the error and complementary error functions.

The following figure illustrates the relationships among erf family functions (erf, erfc, cdfnorm).

cdfnorm Family Functions Relationship |



Useful relations for these functions:

$$\text{erf}(x) + \text{erfc}(x) = 1$$

$$\text{cdfnorm}(x) = \frac{1}{2} \left( 1 + \text{erf} \left( \frac{x}{\sqrt{2}} \right) \right) = 1 - \frac{1}{2} \text{erfc} \left( \frac{x}{\sqrt{2}} \right)$$

Argument	Result	Error Code
a < underflow	+0	status::underflow
$+\infty$	+1	
$-\infty$	+0	
QNAN	QNAN	
SNAN	QNAN	

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use `cdfnorm` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vcdfnorm.cpp
```

**Parent topic:** Special Functions

### 13.2.5.2.6.4 erfinv

Computes inverse error function value of vector elements.

#### Syntax

Buffer API:

```
void erfinv(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode =
    mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event erfinv(queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode =
    mode::not_defined, error_handler<T> errhandler = {})
```

`erfinv` supports the following precisions.

T
float
double

#### Description

The `erfinv(a)` function computes the inverse error function values for elements of the input vector `a` and writes them to the output vector `y`

$$y = \text{erf}^{-1}(a),$$

where `erf(x)` is the error function defined as given by:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt .$$

Useful relations:

$$1. \quad \text{erf}^{-1}(x) = \text{erfc}^{-1}(1 - x),$$

where `erfc` is the complementary error function.

$$2. \quad \Phi(x) = \frac{1}{2} \text{erf}(x/\sqrt{2}),$$

where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp(-t^2/2) dt$$

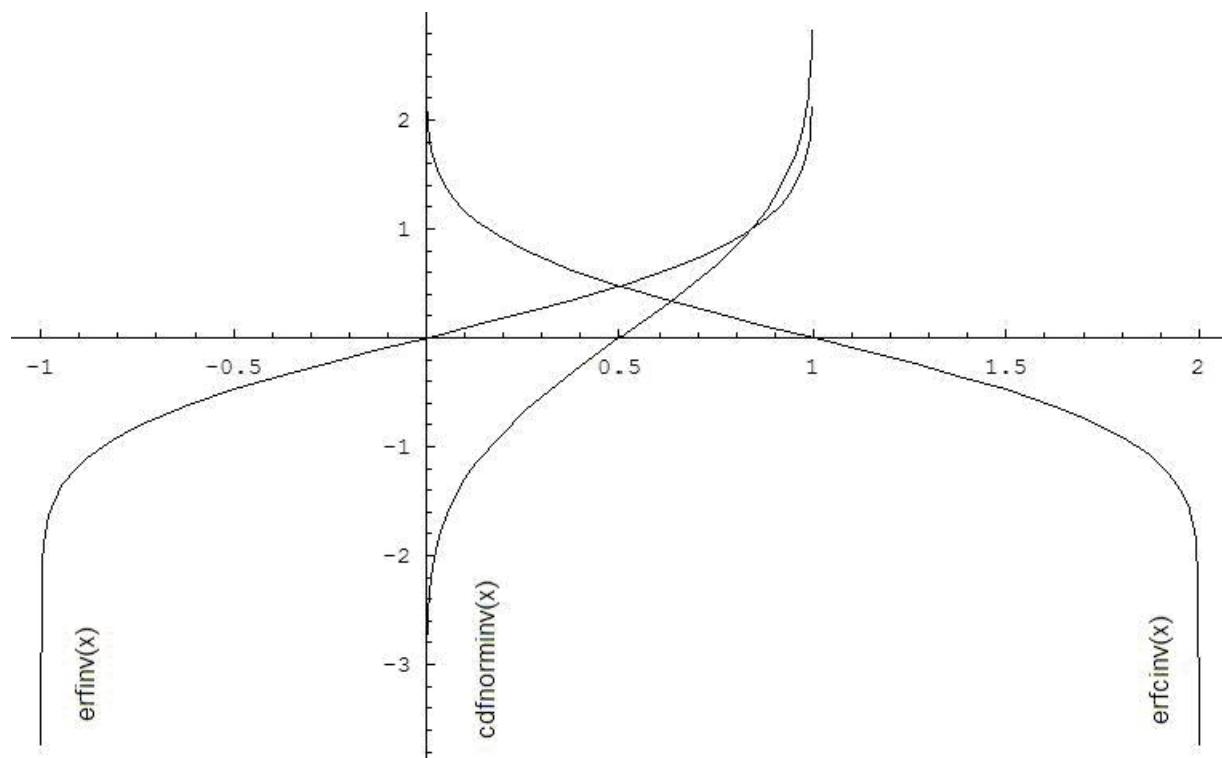
is the cumulative normal distribution function.

$$3. \quad \Phi^{-1}(x) = \sqrt{2} \operatorname{erf}^{-1}(2x - 1),$$

where  $\phi^{-1}(x)$  and  $\operatorname{erf}^{-1}(x)$  are the inverses to  $\phi(x)$  and  $\operatorname{erf}(x)$ , respectively.

The following figure illustrates the relationships among erfinv family functions ( $\operatorname{erfinv}$ ,  $\operatorname{erfcinv}$ ,  $\operatorname{cdfnorminv}$ ).

$\operatorname{erfinv}$  Family Functions Relationship |



Useful relations for these functions:

$$\operatorname{erfcinv}(x) = \operatorname{erfinv}(1 - x)$$

$$\operatorname{cdfnorminv}(x) = \sqrt{2}\operatorname{erfinv}(2x - 1) = \sqrt{2}\operatorname{erfcinv}(2 - 2x)$$

Argument	Result	Error Code
+0	+0	
-0	-0	
+1	$+\infty$	status::sing
-1	$-\infty$	status::sing
$ a  > 1$	QNAN	status::errdom
$+\infty$	QNAN	status::errdom
$-\infty$	QNAN	status::errdom
QNAN	QNAN	
SNAN	QNAN	

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use erfcinv can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/verfcinv.cpp
```

**Parent topic:** Special Functions

### 13.2.5.2.6.5 erfcinv

Computes the inverse complementary error function value of vector elements.

#### Syntax

Buffer API:

```
void erfcinv(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event erfcinv(queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

erfcinv supports the following precisions.

T
float
double

#### Description

The erfcinv(a) function computes the inverse complimentary error function values for elements of the input vector a and writes them to the output vector y.

The inverse complementary error function is defined as given by:

$$\text{erfcinv}(x) = \text{erfinv}(1 - x)$$

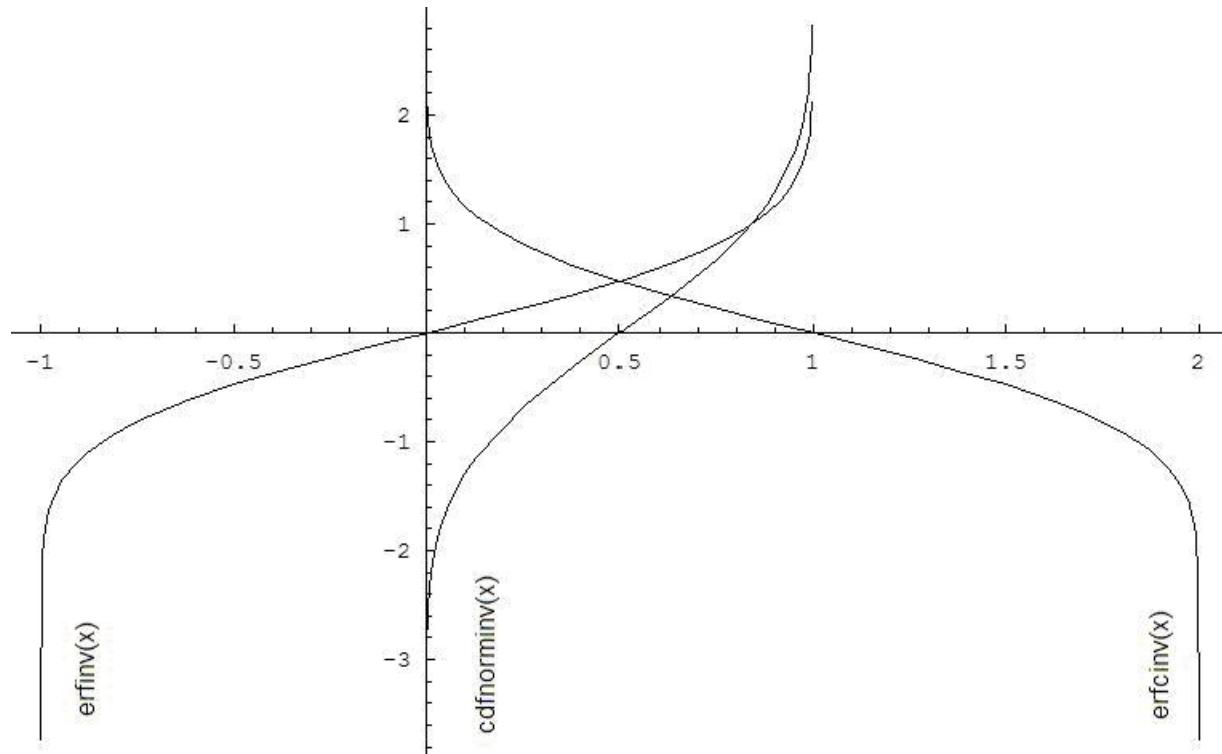
$$\text{erfinv}(x) = \text{erf}^{-1}(x)$$

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

where  $\text{erf}(x)$  denotes the error function and  $\text{erfinv}(x)$  denotes the inverse error function.

The following figure illustrates the relationships among  $\text{erfinv}$  family functions ( $\text{erfinv}$ ,  $\text{erfcinv}$ ,  $\text{cdfnorminv}$ ).

#### erfcinv Family Functions Relationship |



Useful relations for these functions:

$$\text{erfcinv}(x) = \text{erfinv}(1 - x)$$

$$\text{cdfnorminv}(x) = \sqrt{2}\text{erfinv}(2x - 1) = \sqrt{2}\text{erfcinv}(2 - 2x)$$

Argument	Result	Error Code
+1	+0	
+2	-∞	status::sing
-0	+∞	status::sing
+0	+∞	status::sing
$a < -0$	QNAN	status::errdom
$a > +2$	QNAN	status::errdom
$+\infty$	QNAN	status::errdom
$-\infty$	QNAN	status::errdom
QNAN	QNAN	
SNAN	QNAN	

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

- n** Specifies the number of elements to be calculated.
- a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

- n** Specifies the number of elements to be calculated.
- a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use erfcinv can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/verfcinv.cpp
```

**Parent topic:** Special Functions

### 13.2.5.2.6.6 cdfnorminv

Computes the inverse cumulative normal distribution function values of vector elements.

#### Syntax

Buffer API:

```
void cdfnorminv(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event cdfnorminv(queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

cdfnorminv supports the following precisions.

T
float
double

#### Description

The cdfnorminv(a) function computes the inverse cumulative normal distribution function values for elements of the input vector a and writes them to the output vector y.

The inverse cumulative normal distribution function is defined as given by:

$$\text{CdfNormInv}(x) = \text{CdfNorm}^{-1}(x) ,$$

where cdfnorm(x) denotes the cumulative normal distribution function.

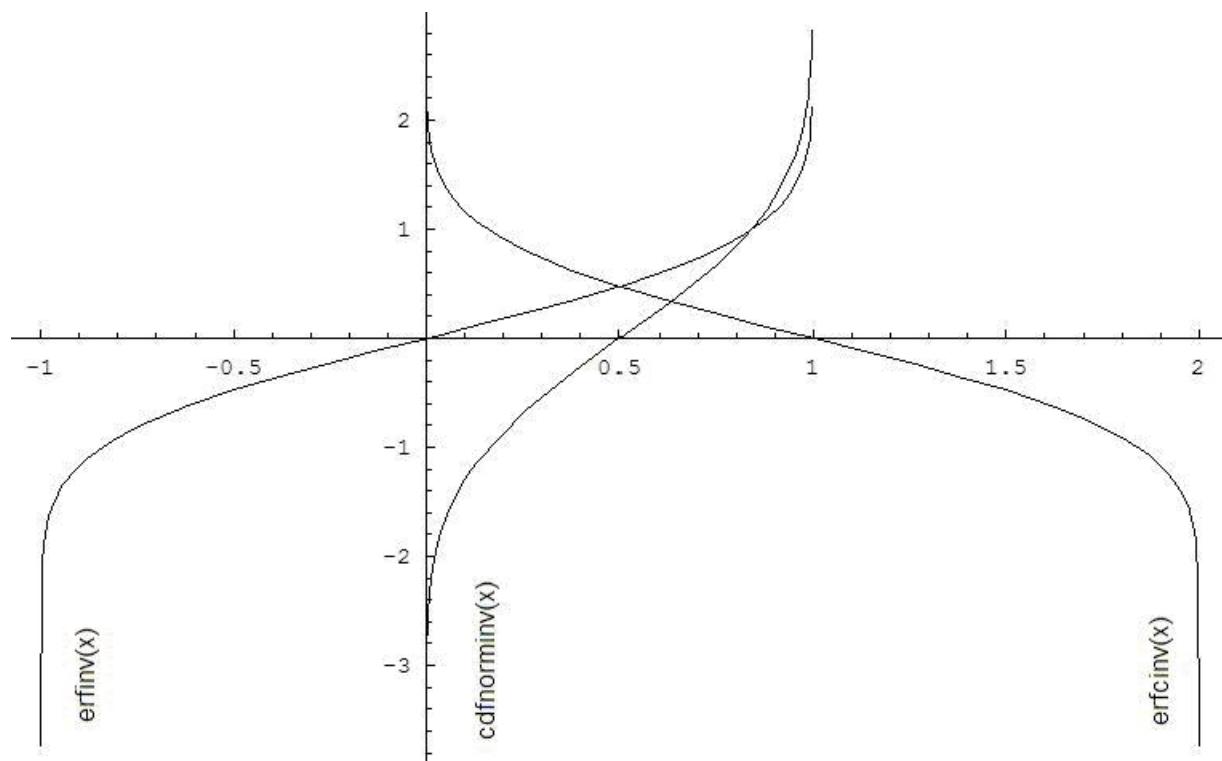
Useful relations:

$$\text{cdfnorminv}(x) = \sqrt{2}\text{erfinv}(2x - 1) = \sqrt{2}\text{erfcinv}(2 - 2x)$$

where erfinv(x)denotes the inverse error function and erfcinv(x)denotes the inverse complementary error functions.

The following figure illustrates the relationships among erfinv family functions (erfinv, erfcinv, cdfnorminv).

### cdfnorminv Family Functions Relationship |



Useful relations for these functions:

$$\text{erfcinv}(x) = \text{erfinv}(1 - x)$$

$$\text{cdfnorminv}(x) = \sqrt{2}\text{erfinv}(2x - 1) = \sqrt{2}\text{erfcinv}(2 - 2x)$$

Argument	Result	Error Code
+0.5	+0	
+1	+∞	status::sing
-0	-∞	status::sing
+0	-∞	status::sing
a < -0	QNAN	status::errdom
a > +1	QNAN	status::errdom
+∞	QNAN	status::errdom
-∞	QNAN	status::errdom
QNAN	QNAN	
SNAN	QNAN	

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use `cdfnorminv` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vcdfnorminv.cpp
```

**Parent topic:** Special Functions

### 13.2.5.2.6.7 Igamma

Computes the natural logarithm of the absolute value of gamma function for vector elements.

#### Syntax

Buffer API:

```
void lgamma (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode =
    mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event lgamma (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode =
    mode::not_defined, error_handler<T> errhandler = {})
```

`lgamma` supports the following precisions.

T
float
double

#### Description

The `Igamma(a)` function computes the natural logarithm of the absolute value of gamma function for elements of the input vector `a` and writes them to the output vector `y`. Precision overflow thresholds for the `Igamma` function are beyond the scope of this document. If the result does not meet the target precision, the function sets the VM Error Status to `status::overflow`.

Argument	Result	Error Code
+1	+0	
+2	+0	
+0	+∞	<code>status::sing</code>
-0	+∞	<code>status::sing</code>
negative integer	+∞	<code>status::sing</code>
-∞	+∞	
+∞	+∞	
<code>a &gt; overflow</code>	+∞	<code>status::overflow</code>
QNaN	QNaN	
SNaN	QNaN	

#### Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer `a` containing input vector of size `n`.

**mode** Overrides the global VM mode setting for this function call. See `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use Igamma can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vlgamma.cpp
```

**Parent topic:** Special Functions

### 13.2.5.2.6.8 tgamma

Computes the gamma function of vector elements.

## Syntax

Buffer API:

```
void tgamma(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode =
    mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event tgamma(queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode =
    mode::not_defined, error_handler<T> errhandler = {})
```

`tgamma` supports the following precisions.

T
float
double

## Description

The `tgamma(a)` function computes the gamma function for elements of the input vector `a` and writes them to the output vector `y`. Precision overflow thresholds for the `tgamma` function are beyond the scope of this document. If the result does not meet the target precision, the function raises sets the VM Error Status to `status::sing`.

Argument	Result	Error Code
+0	+∞	<code>status::sing</code>
-0	-∞	<code>status::sing</code>
negative integer	QNAN	<code>status::errdom</code>
-∞	QNAN	<code>status::errdom</code>
+∞	+∞	
<code>a &gt; overflow</code>	+∞	<code>status::sing</code>
QNAN	QNAN	
SNAN	QNAN	

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer `a` containing input vector of size `n`.

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer `a` to the input vector of size `n`.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use tgamma can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vtgamma.cpp
```

**Parent topic:** Special Functions

### 13.2.5.2.6.9 expint1

Computes the exponential integral of vector elements.

#### Syntax

Buffer API:

```
void expint1(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event expint1(queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

`expint1` supports the following precisions.

T
float
double

#### Description

The `expint1(a)` function computes the exponential integral of vector elements of the input vector *a* and writes them to the output vector *y*.

For positive real values *x*, this can be written as:

$$E_1(x) = \int_x^{\infty} \frac{e^{-t}}{t} dt = \int_1^{\infty} \frac{e^{-xt}}{t} dt$$

For negative real values *x*, the result is defined as NAN.

Argument	Result	Error Code
$x < +0$	QNAN	status::errdom
+0	$+\infty$	status::sing
-0	$+\infty$	status::sing
$+\infty$	+0	
$-\infty$	QNAN	status::errdom
QNAN	QNAN	
SNAN	QNAN	

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use `expint1` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vexpint1.cpp
```

**Parent topic:** Special Functions

### 13.2.5.2.7 Rounding Functions

**Parent topic:** VM Mathematical Functions

- `floor` Computes an integer value rounded towards minus infinity for each vector element.
- `ceil` Computes an integer value rounded towards plus infinity for each vector element.
- `trunc` Computes an integer value rounded towards zero for each vector element.
- `round` Computes a value rounded to the nearest integer for each vector element.
- `nearbyint` Computes a rounded integer value in the current rounding mode for each vector element.
- `rint` Computes a rounded integer value in the current rounding mode.
- `modf` Computes a truncated integer value and the remaining fraction part for each vector element.
- `frac` Computes a signed fractional part for each vector element.

#### 13.2.5.2.7.1 `floor`

Computes an integer value rounded towards minus infinity for each vector element.

#### Syntax

Buffer API:

```
void floor(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined)
```

USM API:

```
event floor(queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined)
```

`floor` supports the following precisions.

T
float
double

## Description

The floor(a)function computes an integer value rounded towards minus infinity for each vector element.

$$y_i = \lfloor a_i \rfloor$$

Argument	Result	Error Code
+0	+0	
-0	-0	
$+\infty$	$+\infty$	
$-\infty$	$-\infty$	
QNAN	QNAN	
SNAN	QNAN	

The floor function does not generate any errors.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use floor can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vfloor.cpp
```

**Parent topic:** Rounding Functions

### 13.2.5.2.7.2 ceil

Computes an integer value rounded towards plus infinity for each vector element.

#### Syntax

Buffer API:

```
void ceil (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined)
```

USM API:

```
event ceil (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined)
```

*ceil* supports the following precisions.

T
float
double

#### Description

The *ceil(a)* function computes an integer value rounded towards plus infinity for each vector element.

$$y_i = \lceil a_i \rceil$$

Argument	Result	Error Code
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
QNAN	QNAN	
SNAN	QNAN	

The ceil function does not generate any errors.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use ceil can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vceil.cpp
```

**Parent topic:** Rounding Functions

### 13.2.5.2.7.3 trunc

Computes an integer value rounded towards zero for each vector element.

#### Syntax

Buffer API:

```
void trunc (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined)
```

USM API:

```
event trunc (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined)
```

`trunc` supports the following precisions.

T
float
double

#### Description

The `trunc(a)` function computes an integer value rounded towards zero for each vector element.

$$a_i \geq 0, y_i = \lfloor a_i \rfloor$$

$$a_i < 0, y_i = \lceil a_i \rceil$$

Argument	Result	Error Code
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
QNAN	QNAN	
SNAN	QNAN	

The `trunc` function does not generate any errors.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use `trunc` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vtrunc.cpp
```

**Parent topic:** Rounding Functions

### 13.2.5.2.7.4 round

Computes a value rounded to the nearest integer for each vector element.

## Syntax

Buffer API:

```
void round(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode =
           mode::not_defined)
```

USM API:

```
event round(queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode =
           mode::not_defined)
```

`round` supports the following precisions.

T
float
double

## Description

The `round(a)` function computes a value rounded to the nearest integer for each vector element. Input elements that are halfway between two consecutive integers are always rounded away from zero regardless of the rounding mode.

Argument	Result	Error Code
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
QNAN	QNAN	
SNAN	QNAN	

The `round(a)` function does not generate any errors.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use round can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vround.cpp
```

**Parent topic:** Rounding Functions

### 13.2.5.2.7.5 `nearbyint`

Computes a rounded integer value in the current rounding mode for each vector element.

## Syntax

Buffer API:

```
void nearbyint (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined)
```

USM API:

```
event nearbyint (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined)
```

`nearbyint` supports the following precisions.

T
float
double

## Description

The `nearbyint(a)` function computes a rounded integer value in a current rounding mode for each vector element.

Argument	Result	Error Code
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
QNAN	QNAN	
SNAN	QNAN	

The `nearbyint` function does not generate any errors.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use `nearbyint` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vnearbyint.cpp
```

**Parent topic:** Rounding Functions

### 13.2.5.2.7.6 `rint`

Computes a rounded integer value in the current rounding mode.

## Syntax

Buffer API:

```
void rint (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined)
```

USM API:

```
event rint (queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined)
```

*rint* supports the following precisions.

T
float
double

## Description

The *rint*(*a*) function computes a rounded floating-point integer value using the current rounding mode for each vector element.

The rounding mode affects the results computed for inputs that fall between consecutive integers. For example:

- $f(0.5) = 0$ , for rounding modes set to round to nearest round toward zero or to minus infinity.
- $f(0.5) = 1$ , for rounding modes set to plus infinity.
- $f(-1.5) = -2$ , for rounding modes set to round to nearest or to minus infinity.
- $f(-1.5) = -1$ , for rounding modes set to round toward zero or to plus infinity.

Argument	Result	Error Code
+0	+0	
-0	-0	
$+\infty$	$+\infty$	
$-\infty$	$-\infty$	
QNAN	QNAN	
SNAN	QNAN	

The *rint* function does not generate any errors.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is *mode*::not\_defined.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use `rint` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vrint.cpp
```

**Parent topic:** Rounding Functions

### 13.2.5.2.7.7 modf

Computes a truncated integer value and the remaining fraction part for each vector element.

## Syntax

Buffer API:

```
void modf (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, buffer<T, 1> &z, uint64_t mode = mode::not_defined)
```

USM API:

```
event modf (queue &exec_queue, int64_t n, T *a, T *y, T *z, vector_class<event> *depends, uint64_t mode = mode::not_defined)
```

`modf` supports the following precisions.

T
float
double

## Description

The modf(a) function computes a truncated integer value and the remaining fraction part for each vector element.

$$a_i \geq 0, \begin{cases} y_i = \lfloor a_i \rfloor \\ z_i = a_i - \lfloor a_i \rfloor \end{cases}$$

$$a_i < 0, \begin{cases} y_i = \lceil a_i \rceil \\ z_i = a_i - \lceil a_i \rceil \end{cases}$$

Argument	Result 1	Result 2	Error Code
+0	+0	+0	
-0	-0	-0	
+∞	+∞	+0	
-∞	-∞	-0	
SNAN	QNAN	QNAN	
QNAN	QNAN	QNAN	

The modf function does not generate any errors.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer a containing input vector of size n.

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer a to the input vector of size n.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

- y** The buffer  $y$  containing the output vector of size  $n$  for truncated integer values.
- z** The buffer  $z$  containing the output vector of size  $n$  for remaining fraction parts.

USM API:

- y** Pointer  $y$  to the output vector of size  $n$  for truncated integer values.
  - z** Pointer  $z$  to the output vector of size  $n$  for remaining fraction parts.
- return value (event)** Function end event.

## Example

An example of how to use modf can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vmodf.cpp
```

**Parent topic:** Rounding Functions

### 13.2.5.2.7.8 frac

Computes a signed fractional part for each vector element.

## Syntax

Buffer API:

```
void frac(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &y, uint64_t mode = mode::not_defined)
```

USM API:

```
event frac(queue &exec_queue, int64_t n, T *a, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined)
```

`frac` supports the following precisions.

T
float
double

## Description

The `frac(a)` function computes a signed fractional part for each vector element.

$$y_i = \begin{cases} a_i - \lfloor a_i \rfloor, & a_i \geq 0 \\ a_i - \lceil a_i \rceil, & a_i < 0 \end{cases}$$

Argument	Result	Error Code
+0	+0	
-0	-0	
+∞	+0	
-∞	-0	
QNAN	QNAN	
SNAN	QNAN	

The `frac` function does not generate any errors.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer `a` containing input vector of size `n`.

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer `a` to the input vector of size `n`.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

**y** The buffer `y` containing the output vector of size `n`.

USM API:

**y** Pointer `y` to the output vector of size `n`.

**return value (event)** Function end event.

## Example

An example of how to use `frac` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vfrac.cpp
```

**Parent topic:** Rounding Functions

### 13.2.5.3 VM Service Functions

The VM Service functions enable you to set/get the accuracy mode and error code. These functions are available both in the Fortran and C interfaces. The table below lists available VM Service functions and their short description.

Function Short Name	Description
<code>set_mode</code>	Sets the VM mode
<code>get_mode</code>	Gets the VM mode
<code>set_status</code>	Sets the VM Error Status
<code>get_status</code>	Gets the VM Error Status
<code>clear_status</code>	Clears the VM Error Status
<code>create_error_handler</code>	Creates the local VM error handler for a function

**Parent topic:** *Vector Math*

#### 13.2.5.3.1 `setmode`

Sets a new mode for VM functions according to the `mode` parameter and returns the previous VM mode.

##### Syntax

```
uint64_t set_mode (queue &exec_queue, uint64_t new_mode)
```

##### Description

The `set_mode` function sets a new mode for VM functions according to the `new_mode` parameter and returns the previous VM mode. The mode change has a global effect on all the VM functions within a thread.

The `mode` parameter is designed to control accuracy and handling of denormalized numbers. You can obtain all other possible values of the `mode` parameter using bitwise OR (`|`) operation to combine one value for handling of denormalized numbers.

The `mode::ftzdzon` is specifically designed to improve the performance of computations that involve denormalized numbers at the cost of reasonable accuracy loss. This mode changes the numeric behavior of the functions: denormalized input values are treated as zeros (DAZ = denormals-are-zero) and denormalized results are flushed to zero (FTZ = flush-to-zero). Accuracy loss may occur if input and/or output values are close to denormal range.

Value of mode	Description
Accuracy Control	
mode::ha	High accuracy versions of VM functions.
mode::la	Low accuracy versions of VM functions.
mode::ep	Enhanced performance accuracy versions of VM functions.
Denormalized Numbers Handling Control	
mode::ftzdazon	Faster processing of denormalized inputs is enabled.
mode::ftzdazoff	Faster processing of denormalized inputs is disabled.
Other	
mode::not_defined	VM status not defined.

The default value of the mode parameter is:

```
mode::ha | mode::ftdazoff
```

## Input Parameters

**exec\_queue** The queue where the routine should be executed.

**new\_mode** Specifies the VM mode to be set.

## Output Parameters

**return value (old\_mode)** Specifies the former VM mode.

## Example

```
oldmode = set_mode (exec_queue , mode::la);
oldmode = set_mode (exec_queue , mode::ep | mode::ftzdazon);
```

**Parent topic:** VM Service Functions

### 13.2.5.3.2 get\_mode

Gets the VM mode.

#### Syntax

```
uint64_t get_mode (queue &exec_queue)
```

## Description

The function `get_mode` function returns the global VM mode parameter that controls accuracy, handling of denormalized numbers, and error handling options. The variable value is a combination by bitwise OR (|) of the values listed in the following table.

Value of mode	Description
Accuracy Control	
<code>mode::ha</code>	High accuracy versions of VM functions.
<code>mode::la</code>	Low accuracy versions of VM functions.
<code>mode::ep</code>	Enhanced performance accuracy versions of VM functions.
Denormalized Numbers Handling Control	
<code>mode::ftzdazon</code>	Faster processing of denormalized inputs is enabled.
<code>mode::ftzdazoff</code>	Faster processing of denormalized inputs is disabled.
Other	
<code>mode::not_defined</code>	VM status not defined.

See example below:

## Input Parameters

**exec\_queue** The queue where the routine should be executed.

## Output Parameters

**return value (old\_mode)** Specifies the global VM mode.

## Example

```
accm = get_mode (exec_queue) & mode::accuracy_mask;
denm = get_mode (exec_queue) & mode::ftzdaz_mask;
```

**Parent topic:** VM Service Functions

### 13.2.5.3.3 set\_status

Sets the global VM Status according to new values and returns the previous VM Status.

## Syntax

```
uint8_t set_status (queue &exec_queue, uint_8 new_status)
```

## Description

The set\_status function sets the global VM Status to new value and returns the previous VM Status.

The global VM Status is a single value and it accumulates via bitwise OR ( | ) all errors that happen inside VM functions. The following table lists the possible error values.

Status	Description
Successful Execution	
status::success	VM function execution completed successfully
status::not_defined	VM status not defined
Warnings	
status::accuracy_warning	VM function execution completed successfully in a different accuracy mode
Computational Errors	
status::errdom	Values are out of a range of definition producing invalid (QNaN) result
status::sing	Values cause divide-by-zero (singularity) errors and produce and invalid (QNaN or Inf) result
status::overflow	An overflow happened during the calculation process
status::underflow	An underflow happened during the calculation process

## Input Parameters

**exec\_queue** The queue where the routine should be executed.

**new\_status** Specifies the VM status to be set.

## Output Parameters

**return value (old\_status)** Specifies the former VM status.

## Example

```
uint8_t olderr = set_status (exec_queue, status::success);

if (olderr & status::errdom)
{
    std::cout << "Errdom status returned" << std::endl;
}

if (olderr & status::sing)
{
    std::cout << "Singularity status returned" << std::endl;
}
```

**Parent topic:** VM Service Functions

### 13.2.5.3.4 get\_status

Gets the VM Status.

#### Syntax

```
uint8_t get_status (queue &exec_queue)
```

#### Description

The `get_status` function gets the VM status.

The global VM Status is a single value and it accumulates via bitwise OR ( | ) all computational errors that happen inside VM functions. The following table lists the possible error values.

Status	Description
Successful Execution	
<code>status::success</code>	VM function execution completed successfully
<code>status::not_defined</code>	VM status not defined
Warnings	
<code>status::accuracy_warning</code>	VM function execution completed successfully in a different accuracy mode
Computational Errors	
<code>status::errdom</code>	Values are out of a range of definition producing invalid (QNaN) result
<code>status::sing</code>	Values cause divide-by-zero (singularity) errors and produce and invalid (QNaN or Inf) result
<code>status::overflow</code>	An overflow happened during the calculation process
<code>status::underflow</code>	An underflow happened during the calculation process

#### Input Parameters

**exec\_queue** The queue where the routine should be executed.

#### Output Parameters

**return value (status)** Specifies the VM status.

#### Example

```
uint8_t err = get_status (exec_queue);

if (err & status::errdom)
{
    std::cout << "Errdom status returned" << std::endl;
}

if (err & status::sing)
{
```

(continues on next page)

(continued from previous page)

```
    std::cout << "Singularity status returned" << std::endl;
}
```

**Parent topic:** VM Service Functions

### 13.2.5.3.5 clear\_status

Sets the VM Status according to `status::success` and returns the previous VM Status.

#### Syntax

```
uint8_t clear_status (queue &exec_queue)
```

#### Description

The `clear_status` function sets the VM status to `status::success` and returns the previous VM status.

The global VM Status is a single value and it accumulates all errors that happen inside VM functions. The following table lists the possible error values.

Status	Description
Successful Execution	
<code>status::success</code>	VM function execution completed successfully
<code>status::not_defined</code>	VM status not defined
Warnings	
<code>status::accuracy_warning</code>	VM function execution completed successfully in a different accuracy mode
Computational Errors	
<code>status::errdom</code>	Values are out of a range of definition producing invalid (QNaN) result
<code>status::sing</code>	Values cause divide-by-zero (singularity) errors and produce and invalid (QNaN or Inf) result
<code>status::overflow</code>	An overflow happened during the calculation process
<code>status::underflow</code>	An underflow happened during the calculation process

#### Input Parameters

**exec\_queue** The queue where the routine should be executed.

#### Output Parameters

**return value (old\_status)** Specifies the former VM status.

## Example

```
uint8_t olderr = clear_status (exec_queue);
```

**Parent topic:** VM Service Functions

### 13.2.5.3.6 create\_error\_handler

Creates the local VM Error Handler for a function..

## Syntax

Buffer API:

```
error_handler<T> create_error_handler (buffer<uint8_t, 1> &errarray, int64_t length = 1, uint8_t errstatus = status::not_defined, T fixup = 0.0, bool copysign = false)
```

USM API:

```
error_handler<T> create_error_handler (uint8_t *errarray, int64_t length = 1, uint8_t errstatus = status::not_defined, T fixup = 0.0, bool copysign = false)
```

`create_error_handler` supports the following precisions.

T
float
double
std::complex<float>
std::complex<double>

## Description

The `create_error_handler` creates the local VM Error handler to be passed to VM functions which support error handling.

The local VM Error Handler supports three modes:

- **Single status mode:** all errors happened during function execution are being written into one status value.  
At the execution end the single value is either un-changed if no errors happened or contained accumulated (merged by bitwise OR) error statuses happened in function execution.  
Set the array pointer to any `status` object and the length equals 1 to enable this mode.
- **Multiple status mode:** error statuses are saved as an array by indices where they happen.  
Notice that only error statuses are being written into the array, the success statuses are not to be written.  
That means the array needs to be allocated and initialized by user before function execution.  
To enable this mode allocate `status` array with the same size as argument and result vectors, set the `errarray` pointer to it and the length to the vector size.
- **Fixup mode:** for all arguments which caused specific error status results to be overwritten by a user-defined value.

To enable this mode set desirable errstatus and fixup values. The fixup value is written to results for each argument which caused the errstatus error.

If the copysign is set to true then fixup value's sign set to the same sign of the argument which caused the errstatus – a suitable option for symmetric math functions.

The following table lists the possible computational error values.

Status	Description
Successful Execution	
status::success	VM function execution completed successfully
status::not_defined	VM status not defined
Warnings	
status::accuracy_warning	VM function execution completed successfully in a different accuracy mode
Computational Errors	
status::errdom	Values are out of a range of definition producing invalid (QNaN) result
status::sing	Values cause divide-by-zero (singularity) errors and produce and invalid (QNaN or Inf) result
status::overflow	An overflow happened during the calculation process
status::underflow	An underflow happened during the calculation process

Notes:

- You must allocate and initialize array errarray before calling VM functions in multiple status error handling mode.

The array should be large enough to contain n error codes, where n is the same as inputoutput vector size for the VM function.

- If no arguments passed to the create\_error\_handler function, then the empty object is created with all of three error handling modes disabled.

In this case, the VM math functions set the global error status only.

## Input Parameters

**errarray** Array to store error statuses (should be a buffer for buffer API).

**length** Length of the errarray. This is an optional argument, default value is 1.

**errcode** Error status to fixup results. This is an optional argument, default value is status::not\_defined.

**fixup** Fixup value for results. This is an optional argument, default value is 0.0.

**copysign** Flag for setting the fixup value's sign the same as the argument's. This is an optional argument, default value false.

## Output Parameters

**return value** Specifies the error handler object to be created.

## Examples

The following examples are possible usage models (USM API).

Single status mode with `create_error_handler()`:

```
error_handler<float> handler = vm::create_error_handler (st);
vm::sin(exec_queue, 1000, a, r, handler);
if ( st[0] & status::errdom)
{
    std::cout << "Errdom status returned" << std::endl;
}
```

Single status mode without `create_error_handler()`:

```
vm::sin(exec_queue, 1000, a, r, {st });
std::cout << status << std::endl;
if ( st[0] & status::errdom)
{
    std::cout << "Errdom status returned" << std::endl;
}
```

The `st` contains either `status::success` or accumulated error statuses if computational errors occurred in `vm::erfinv`.

Multiple status mode with `create_error_handler()`:

```
error_handler<float> handler = vm::create_error_handler (st, 1000);
vm::inv(exec_queue, 1000, a, r, handler);
for(int i=0; i<1000; i++)
    std::cout << st[i] << std::endl;
```

Multiple status mode without `create_error_handler()`:

```
vm::inv(exec_queue, 1000, a, r, {st, 1000});
for(int i=0; i<1000; i++)
    std::cout << st[i] << std::endl;
```

The `st` array contains all codes for computational errors that occur at the same vector indices `i` as the arguments that caused the errors.

Fixup status mode with `create_error_handler()`:

```
float fixup = 1.0;
error_handler<float> handler = vm::create_error_handler (nullptr, 0,
    ↵status::errdom, fixup, true);
vm::erfinv(exec_queue, 1000, a, r, handler);
```

Fixup status mode without `create_error_handler()`:

```
float fixup = 1.0;
vm::erfinv(exec_queue, 1000, a, r, { nullptr, 0, status::errdom, fixup, true });
```

All results in `r` which computation generated `status::errdom` are replaced by `fixup` values.

In the example above all the `erfinv` function's NAN results caused by greater than `|1|` arguments are replaced by 1.0 value with the same sign as the corresponding argument.

Mixed (Single and Fixup) status mode with `create_error_handler()`:

```

float fixup = 1e38;
error_handler<float> handler = vm::create_error_handler (st, 1, status::overflow, fixup);
vm::exp(exec_queue, 1000, a, r, handler);
if ( st & status::underflow)
{
    std::cout << "Underflow status returned" << std::endl;
}

```

Mixed (Single and Fixup) status mode without `create_error_handler()`:

```

float fixup = 1e38;
vm::exp(exec_queue, 1000, a, r, {st, 1, status::overflow, fixup});
if ( st & status::underflow)
{
    std::cout << "Underflow status returned" << std::endl;
}

```

Mixed (Multiple and Fixup) status mode with `create_error_handler()`:

```

float fixup = 1.0;
error_handler<float> handler = vm::create_error_handler (st, 1000, status::errdom, fixup);
vm::acospi(exec_queue, 1000, a, r, handler);
for(int i=0; i<1000; i++)
    std::cout << st[i] << std::endl;

```

Mixed (Multiple and Fixup) status mode without `create_error_handler()`:

```

float fixup = 1.0;
vm::acospi(exec_queue, 1000, a, r, { st, 1000, status::errdom, fixup});
for(int i=0; i<1000; i++)
    std::cout << st[i] << std::endl;

```

The `st` array contains all codes for computational errors that occur at the same vector indices `i` as the arguments that caused the errors. Additionally, all results in `r` which computation generated `status::errdom` are replaced by `fixup` values.

No local error handling mode:

```

vm::pow(exec_queue, n, a, b, r);
uint8_t err = vm::get_status (exec_queue);

if (err & status::errdom)
{
    std::cout << "Errdom status returned" << std::endl;
}

if (err & status::sing)
{
    std::cout << "Singularity status returned" << std::endl;
}

```

Only global accumulated error status `err` is set.

**Parent topic:** VM Service Functions

### 13.2.5.4 Miscellaneous VM Functions

- **copysign** Returns vector of elements of one argument with signs changed to match other argument elements.
- **nextafter** Returns vector of elements containing the next representable floating-point values following the values from the elements of one vector in the direction of the corresponding elements of another vector.
- **fdim** Returns vector containing the differences of the corresponding elements of the vector arguments if the first is larger and +0 otherwise.
- **fmax** Returns the larger of each pair of elements of the two vector arguments.
- **fmin** Returns the smaller of each pair of elements of the two vector arguments.
- **maxmag** Returns the element with the larger magnitude between each pair of elements of the two vector arguments.
- **minmag** Returns the element with the smaller magnitude between each pair of elements of the two vector arguments.

**Parent topic:** *Vector Math*

#### 13.2.5.4.1 copysign

Returns vector of elements of one argument with signs changed to match other argument elements.

##### Syntax

Buffer API:

```
void copysign(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &b, buffer<T, 1> &y, uint64_t
mode = mode::not_defined)
```

USM API:

```
event copysign(queue &exec_queue, int64_t n, T *a, T *b, T *y, vector_class<event> *depends, uint64_t
mode = mode::not_defined)
```

copysign supports the following precisions.

T
float
double

##### Description

The copysign(a,b) function returns the first vector argument elements with the sign changed to match the sign of the second vector argument's corresponding elements.

Argument 1	Argument 2	Result	Error Code
any value	positive value	+any value	
any value	negative value	-any value	

The copysign(a,b) function does not generate any errors.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing 1st input vector of size *n*.

**b** The buffer *b* containing 2nd input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the 1st input vector of size *n*.

**b** Pointer *b* to the 2nd input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use `copysign` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vcopysign.cpp
```

**Parent topic:** [Miscellaneous VM Functions](#)

### 13.2.5.4.2 `nextafter`

Returns vector of elements containing the next representable floating-point values following the values from the elements of one vector in the direction of the corresponding elements of another vector.

## Syntax

Buffer API:

```
void nextafter (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &b, buffer<T, 1> &y,
                uint64_t mode = mode::not_defined, error_handler<T> errhandler = {})
```

USM API:

```
event nextafter (queue &exec_queue, int64_t n, T *a, T *b, T *y, vector_class<event> *depends, uint64_t
                  mode = mode::not_defined, error_handler<T> errhandler = {})
```

nextafter supports the following precisions.

T
float
double

## Description

The nextafter(a,b) function returns a vector containing the next representable floating-point values following the first vector argument elements in the direction of the second vector argument's corresponding elements.

Arguments/Results	Error Code
Input vector argument element is finite and the corresponding result vector element value is infinite	status::overflow
Result vector element value is subnormal or zero, and different from the corresponding input vector argument element	status::underflow

Even though underflow or overflow can occur, the returned value is independent of the current rounding direction mode.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing 1st input vector of size *n*.

**b** The buffer *b* containing 2nd input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is mode::not\_defined.

**errhandler** Sets local error handling mode for this function call. See the [create\\_error\\_handler](#) function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the 1st input vector of size *n*.

**b** Pointer *b* to the 2nd input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

**errhandler** Sets local error handling mode for this function call. See the `create_error_handler` function for arguments and their descriptions. This is an optional parameter. The local error handler is disabled by default.

## Output Parameters

Buffer API:

**y** The buffer **y** containing the output vector of size **n**.

USM API:

**y** Pointer **y** to the output vector of size **n**.

**return value (event)** Function end event.

## Example

An example of how to use `nextafter` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vnnextafter.cpp
```

**Parent topic:** Miscellaneous VM Functions

### 13.2.5.4.3 `fdim`

Returns vector containing the differences of the corresponding elements of the vector arguments if the first is larger and  $+0$  otherwise.

## Syntax

Buffer API:

```
void fdim(queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &b, buffer<T, 1> &y, uint64_t mode = mode::not_defined)
```

USM API:

```
event fdim(queue &exec_queue, int64_t n, T *a, T *b, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined)
```

`fdim` supports the following precisions.

T
float
double

## Description

The `fdim(a,b)` function returns a vector containing the differences of the corresponding elements of the first and second vector arguments if the first element is larger, and `+0` otherwise.

Argument 1	Argument 2	Result	Error Code
any	QNAN	QNAN	
any	SNAN	QNAN	
QNAN	any	QNAN	
SNAN	any	QNAN	

The `fdim(a,b)` function does not generate any errors.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer `a` containing 1st input vector of size `n`.

**b** The buffer `b` containing 2nd input vector of size `n`.

**mode** Overrides the global VM mode setting for this function call. See `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer `a` to the 1st input vector of size `n`.

**b** Pointer `b` to the 2nd input vector of size `n`.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

**y** The buffer `y` containing the output vector of size `n`.

USM API:

**y** Pointer `y` to the output vector of size `n`.

**return value (event)** Function end event.

## Example

An example of how to use fdim can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vfdim.cpp
```

**Parent topic:** Miscellaneous VM Functions

### 13.2.5.4.4 fmax

Returns the larger of each pair of elements of the two vector arguments.

#### Syntax

Buffer API:

```
void fmax (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &b, buffer<T, 1> &y, uint64_t mode = mode::not_defined)
```

USM API:

```
event fmax (queue &exec_queue, int64_t n, T *a, T *b, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined)
```

fmax supports the following precisions.

T
float
double

#### Description

The fmax(*a,b*) function returns a vector with element values equal to the larger value from each pair of corresponding elements of the two vectors *a* and *b*: if *a* < b fmax(*a,b*) returns *b*, otherwise fmax(*a,b*) returns *a*.

Argument 1	Argument 2	Result	Error Code
a not NAN	NAN	a	
NAN	b not NAN	b	
NAN	NAN	NAN	

The fmax(*a,b*) function does not generate any errors.

#### Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing 1st input vector of size *n*.

**b** The buffer *b* containing 2nd input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer `a` to the 1st input vector of size `n`.

**b** Pointer `b` to the 2nd input vector of size `n`.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

**y** The buffer `y` containing the output vector of size `n`.

USM API:

**y** Pointer `y` to the output vector of size `n`.

**return value (event)** Function end event.

## Example

An example of how to use `fmax` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vfmax.cpp
```

**Parent topic:** Miscellaneous VM Functions

### 13.2.5.4.5 fmin

Returns the smaller of each pair of elements of the two vector arguments.

## Syntax

Buffer API:

```
void fmin (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &b, buffer<T, 1> &y, uint64_t mode = mode::not_defined)
```

USM API:

```
event fmin (queue &exec_queue, int64_t n, T *a, T *b, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined)
```

`fmin` supports the following precisions.

T
float
double

## Description

The `fmin(a,b)` function returns a vector with element values equal to the smaller value from each pair of corresponding elements of the two vectors `a` and `b`: if `a > b``fmin(a,b)` returns `b`, otherwise `fmin(a,b)` returns `a`.

Argument 1	Argument 2	Result	Error Code
a not NAN	NAN	a	
NAN	b not NAN	b	
NAN	NAN	NAN	

The `fmin(a,b)` function does not generate any errors.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer `a` containing 1st input vector of size `n`.

**b** The buffer `b` containing 2nd input vector of size `n`.

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer `a` to the 1st input vector of size `n`.

**b** Pointer `b` to the 2nd input vector of size `n`.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

**y** The buffer `y` containing the output vector of size `n`.

USM API:

**y** Pointer `y` to the output vector of size `n`.

**return value (event)** Function end event.

## Example

An example of how to use fmin can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vfmin.cpp
```

**Parent topic:** Miscellaneous VM Functions

### 13.2.5.4.6 maxmag

Returns the element with the larger magnitude between each pair of elements of the two vector arguments.

#### Syntax

Buffer API:

```
void maxmag (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &b, buffer<T, 1> &y, uint64_t mode = mode::not_defined)
```

USM API:

```
event maxmag (queue &exec_queue, int64_t n, T *a, T *b, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined)
```

maxmag supports the following precisions.

T
float
double

#### Description

The maxmag(a,b) function returns a vector with element values equal to the element with the larger magnitude from each pair of corresponding elements of the two vectors a and b:

- If  $|a| > |b|$  maxmag(a,b) returns a, otherwise maxmag(a,b) returns b.
- If  $|b| > |a|$  maxmag(a,b) returns b, otherwise maxmag(a,b) returns a.
- Otherwise maxmag(a,b) behaves like fmax.

Argument 1	Argument 2	Result	Error Code
a not NAN	NAN	a	
NAN	b not NAN	b	
NAN	NAN	NAN	

The maxmag(a,b) function does not generate any errors.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer *a* containing 1st input vector of size *n*.

**b** The buffer *b* containing 2nd input vector of size *n*.

**mode** Overrides the global VM mode setting for this function call. See [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer *a* to the 1st input vector of size *n*.

**b** Pointer *b* to the 2nd input vector of size *n*.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the [set\\_mode](#) function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

**y** The buffer *y* containing the output vector of size *n*.

USM API:

**y** Pointer *y* to the output vector of size *n*.

**return value (event)** Function end event.

## Example

An example of how to use maxmag can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vmaxmag.cpp
```

**Parent topic:** Miscellaneous VM Functions

### 13.2.5.4.7 minmag

Returns the element with the smaller magnitude between each pair of elements of the two vector arguments.

## Syntax

Buffer API:

```
void minmag (queue &exec_queue, int64_t n, buffer<T, 1> &a, buffer<T, 1> &b, buffer<T, 1> &y, uint64_t mode = mode::not_defined)
```

USM API:

```
event minmag (queue &exec_queue, int64_t n, T *a, T *b, T *y, vector_class<event> *depends, uint64_t mode = mode::not_defined)
```

`minmag` supports the following precisions.

T
float
double

## Description

The `minmag(a,b)` function returns a vector with element values equal to the element with the smaller magnitude from each pair of corresponding elements of the two vectors `a` and `b`:

- If  $|a| < |b|$  `minmag(a,b)` returns `a`, otherwise `minmag(a,b)` returns `b`.
- If  $|b| < |a|$  `minmag(a,b)` returns `b`, otherwise `minmag(a,b)` returns `a`.
- Otherwise `minmag` behaves like `fmin`.

Argument 1	Argument 2	Result	Error Code
a not NAN	NAN	a	
NAN	b not NAN	b	
NAN	NAN	NAN	

The `minmag(a,b)` function does not generate any errors.

## Input Parameters

Buffer API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** The buffer `a` containing 1st input vector of size `n`.

**b** The buffer `b` containing 2nd input vector of size `n`.

**mode** Overrides the global VM mode setting for this function call. See `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

USM API:

**exec\_queue** The queue where the routine should be executed.

**n** Specifies the number of elements to be calculated.

**a** Pointer `a` to the 1st input vector of size `n`.

**b** Pointer `b` to the 2nd input vector of size `n`.

**depends** Vector of dependent events (to wait for input data to be ready).

**mode** Overrides the global VM mode setting for this function call. See the `set_mode` function for possible values and their description. This is an optional parameter. The default value is `mode::not_defined`.

## Output Parameters

Buffer API:

**y** The buffer `y` containing the output vector of size `n`.

USM API:

**y** Pointer `y` to the output vector of size `n`.

**return value (event)** Function end event.

## Example

An example of how to use `minmag` can be found in the oneMKL installation directory, under:

```
examples/sycl/vml/vminmag.cpp
```

**Parent topic:** Miscellaneous VM Functions

### 13.2.5.5 Bibliography

For more information about the VM functionality, refer to the following publications:

- **VM**

[IEEE754] IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std 754-2008.

---

**CHAPTER  
FOURTEEN**

---

**CONTRIBUTORS**

Maksim Banin, Intel  
Konstantin Boyarinov, Intel  
Robert Cohn, Intel  
Max Domeika, Intel  
Roman Dubtsov, Intel  
Evarist Fomenko, Intel  
Ruslan Israfilov, Intel  
Alexei Katranov, Intel  
Michael Kinsner, Intel  
Ivan Kochin, Intel  
Maria Kraynyuk, Intel  
Alexey Kukanov, Intel  
Geoff Lowney, Intel  
Ekaterina Mekhnetsova, Intel  
Andrey Nikolaev, Intel  
Nikita Ponomarev, Intel  
Alison Richards, Intel  
Todd Rosenquist, Intel  
Sally Sams, Intel  
Sanjiv Shah, Intel  
Mikhail Smorkalov, Intel  
Peng Tu, Intel  
Zack Waters, Intel

## HTML AND PDF VERSIONS

This section describes the versions that were available at time of publication. See the [latest specification](#) for updates.

Table 1: oneAPI Versions Table

Version	Date	View
<a href="#">0.85</a>	06/29/2020	<a href="#">HTML</a> <a href="#">PDF</a>
<a href="#">0.8</a>	05/29/2020	<a href="#">HTML</a> <a href="#">PDF</a>
<a href="#">0.7</a>	03/26/2020	<a href="#">HTML</a> <a href="#">PDF</a>
<a href="#">0.5</a>	11/17/2019	<a href="#">HTML</a>

## 15.1 Release Notes

### 15.1.1 0.85

- oneVPL
  - High-performance video decode supporting MPEG-2, MPEG-4/H.264/AVC, H.265/HEVC, AV1, VP9, MJPEG;
  - High-performance video encode supporting MPEG-2, MPEG-4/H.264/AVC, H.265/HEVC, AV1, VP9, MJPEG;
  - Video processing for composition, alpha blending, deinterlace, resize, rotate, denoise, procamp, crop, detail, frame rate conversion, and color conversion.
- oneDNN
  - Added individual primitive definitions providing mathematical operation definitions and explaining details of their use
  - Expanded the programming model section explaining device abstraction and its interoperability with DPC++
  - Added the data model section explaining supported data types and memory layouts
  - Added specification for primitive attributes explaining, among other things, low-precision inference and bfloat16 training

### 15.1.2 0.8

- Level Zero
  - Updated to 0.95
- oneMKL
  - Continuing modifications to oneMKL Architecture and BLAS domain
  - Significant refactoring and updating of LAPACK domain API descriptions and structure.
- oneTBB
  - Significant rewrite and reorganization
- oneDAL
  - Extended description of Data Management component, added description of basic elements of algorithms, and error handling mechanism
  - Added description of namespaces and structure of the header files
  - Added specification of kNN algorithm
  - Introduced math notations section, extended glossary section
- oneDNN
  - Detailed descriptions for data model (tensor formats and data types), and execution models

### 15.1.3 0.7

- DPC++: 10 new language extensions including performance features like sub-groups and atomics, as well as features to allow more concise programs.
- oneDNN: Major restructuring of the document, with high-level introduction to the concepts
- Level Zero: Updated to 0.91. Open source release of driver implementing the specification
- oneDAL: Major restructuring of the document, with high-level introduction to the concepts
- oneVPL: Added support for device selection, context sharing, workstream presets and configurations, video processing and encoding APIs to easily construct a video processing pipeline.
- oneMKL: Added USM APIs. Major restructuring of document. Added architecture section with overview of execution model, memory model and API design.

### 15.1.4 0.5

Initial public release

---

CHAPTER  
**SIXTEEN**

---

## **LEGAL NOTICES AND DISCLAIMERS**

The content of this oneAPI Specification is licensed under the [Creative Commons Attribution 4.0 International License](#). Unless stated otherwise, the sample code examples in this document are released to you under the [MIT license](#).

This specification is a continuation of Intel's decades-long history of working with standards groups and industry/academia initiatives such as The Khronos Group\*, to create and define specifications in an open and fair process to achieve interoperability and interchangeability. oneAPI is intended to be an open specification and we encourage you to help us make it better. Your feedback is optional, but to enable Intel to incorporate any feedback you may provide to this specification, and to further upstream your feedback to other standards bodies, including The Khronos Group SYCL\* specification, please submit your feedback under the terms and conditions below. Any contribution of your feedback to the oneAPI Specification does not prohibit you from also contributing your feedback directly to other standard bodies, including The Khronos Group under their respective submission policies.

By opening an issue, providing feedback, or otherwise contributing to the specification, you agree that Intel will be free to use, disclose, reproduce, modify, license, or otherwise distribute your feedback at its sole discretion without any obligations or restrictions of any kind, including without limitation, intellectual property rights or licensing obligations.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice.

Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

\*Other names and brands may be claimed as the property of others.

---

CHAPTER  
**SEVENTEEN**

---

## PAGE NOT FOUND

We cannot find the page. Please try:

- Starting the navigation from [spec.oneapi.com](#)
- Clearing your [browser cache](#) and starting the navigation from [spec.oneapi.com](#)
- Filing an issue in [Github](#)
- Emailing to: [oneAPI@intel.com](mailto:oneAPI@intel.com)

## BIBLIOGRAPHY

- [OpenCLSpec] Khronos OpenCL Working Group, The OpenCL Specification Version:2.1 Document Revision:24 Available from [opencl-2.1.pdf](#)
- [SYCLSpec] Khronos®OpenCL™ Working Group — SYCL™ subgroup, SYCL™ Specification SYCL™ integrates OpenCL™ devices with modern C++, Version 1.2.1 Available from [sycl-1.2.1.pdf](#)
- [Lloyd82] Stuart P Lloyd. *Least squares quantization in PCM*. IEEE Transactions on Information Theory 1982, 28 (2): 1982pp: 129–137.
- [Bro07] Bro, R.; Acar, E.; Kolda, T. *Resolving the sign ambiguity in the singular value decomposition*. SANDIA Report, SAND2007-6422, Unlimited Release, October, 2007.
- [Bentley80] J. L. Bentley. Multidimensional Divide and Conquer. Communications of the ACM, 23(4):214–229, 1980.
- [Friedman17] J. Friedman, T. Hastie, R. Tibshirani. *The Elements of Statistical Learning Data Mining, Inference, and Prediction*. Springer, 2017.
- [Zhang04] T. Zhang. Solving Large Scale Linear Prediction Problems Using Stochastic Gradient Descent Algorithms. ICML 2004: Proceedings Of The Twenty-First International Conference On Machine Learning, 919–926, 2004.

# INDEX

## Symbols

\_mfxExtCencParam (*C++ struct*), 840  
\_mfxExtCencParam::Header (*C++ member*), 841  
\_mfxExtCencParam::StatusReportIndex  
    (*C++ member*), 841  
~combinable (*C++ function*), 619  
~global\_control (*C++ function*), 380  
~graph (*C++ function*), 327  
~null\_mutex (*C++ function*), 644  
~null\_rw\_mutex (*C++ function*), 645  
~overwrite\_node (*C++ function*), 348  
~queuing\_mutex (*C++ function*), 642  
~queuing\_rw\_mutex (*C++ function*), 643  
~speculative\_spin\_mutex (*C++ function*), 640  
~speculative\_spin\_rw\_mutex (*C++ function*),  
    641  
~spin\_mutex (*C++ function*), 638  
~spin\_rw\_mutex (*C++ function*), 639  
~split\_node (*C++ function*), 364  
~task\_arena (*C++ function*), 385  
~task\_group (*C++ function*), 382  
~task\_group\_context (*C++ function*), 379  
~task\_scheduler\_observer (*C++ function*), 388  
~write\_once\_node (*C++ function*), 350

## A

abs (*C++ function*), 1217  
Accessor, 229  
acos (*C++ function*), 1275  
acosh (*C++ function*), 1308  
acospi (*C++ function*), 1288  
activate (*C++ function*), 337  
active\_value (*C++ function*), 380  
add (*C++ function*), 307, 1206  
allocate (*C++ function*), 629, 630, 632  
allocator\_type (*C++ function*), 629  
API, 230, 710  
arg (*C++ function*), 1218  
asin (*C++ function*), 1277  
asinh (*C++ function*), 1310  
asinpi (*C++ function*), 1290  
atan (*C++ function*), 1279

atan2 (*C++ function*), 1280  
atan2pi (*C++ function*), 1293  
atanh (*C++ function*), 1312  
atanpi (*C++ function*), 1292  
attach (*C++ struct*), 384  
automatic (*C++ member*), 384  
AVC, 710  
**B**  
Batch mode, 229  
begin (*C++ function*), 316, 625, 627  
blocked\_range (*C++ function*), 315  
Body::~Body (*C++ function*), 280, 290–292  
Body::assign (*C++ function*), 282  
Body::Body (*C++ function*), 280, 282, 290–292  
Body::join (*C++ function*), 280  
Body::operator() (*C++ function*), 280–282, 284,  
    290–292  
Body::reverse\_join (*C++ function*), 282  
BRC, 710  
broadcast\_node (*C++ function*), 359  
buffer\_node (*C++ function*), 352  
Builder, 229

## C

cache\_aligned\_resource (*C++ function*), 633  
cancel (*C++ function*), 383  
cancel\_group\_execution (*C++ function*), 379  
canceled (*C macro*), 383  
capture\_fp\_settings (*C++ function*), 379  
cast\_to (*C++ function*), 372  
Categorical feature, 228  
cbrt (*C++ function*), 1234  
cdfnorm (*C++ function*), 1320  
cdfnorminv (*C++ function*), 1329  
ceil (*C++ function*), 1339  
cis (*C++ function*), 1271  
Classification, 228  
clear (*C++ function*), 312, 348, 350, 619  
clear\_status (*C++ function*), 1356  
Clustering, 228  
combinable (*C++ function*), 618, 619

combine (*C++ function*), 619, 625  
 Combine::operator () (*C++ function*), 283  
 combine\_each (*C++ function*), 619, 625  
 Commit (*C++ function*), 1143  
 compete (*C macro*), 383  
 composite\_node (*C++ function*), 368  
 computeBackward (*C++ function*), 1146  
 computeForward (*C++ function*), 1144  
 conj (*C++ function*), 1215  
 const\_iterator (*C++ type*), 315  
 Contiguous data, 229  
 Continuous feature, 228  
 copysign (*C++ function*), 1361  
 CORE, 648  
 cos (*C++ function*), 1266  
 cosd (*C++ function*), 1296  
 cosh (*C++ function*), 1301  
 cospi (*C++ function*), 1283  
 CQP, 710  
 CR::begin (*C++ function*), 289  
 CR::const\_reference (*C++ type*), 289  
 CR::difference\_type (*C++ type*), 289  
 CR::end (*C++ function*), 289  
 CR::grainsize (*C++ function*), 289  
 CR::iterator (*C++ type*), 289  
 CR::reference (*C++ type*), 289  
 CR::size\_type (*C++ type*), 289  
 CR::value\_type (*C++ type*), 289  
 create\_error\_handler (*C++ function*), 1357  
 current\_thread\_index (*C++ function*), 387

## D

Data format, 229  
 Data layout, 229  
 Data type, 229  
 Dataset, 228  
 deallocate (*C++ function*), 629, 630, 632  
 DECODE, 648  
 descriptor (*C++ member*), 1138  
 dGPU/dGfx, 710  
 Direct3D, 710  
 Direct3D 11, 710  
 Direct3D 9, 710  
 div (*C++ function*), 1228  
 dnnl::algorithm (*C++ enum*), 52  
 dnnl::algorithm::binary\_add (*C++ enumerator*), 54  
 dnnl::algorithm::binary\_max (*C++ enumerator*), 54  
 dnnl::algorithm::binary\_min (*C++ enumerator*), 54  
 dnnl::algorithm::binary\_mul (*C++ enumerator*), 54

dnnl::algorithm::convolution\_auto (*C++ enumerator*), 52  
 dnnl::algorithm::convolution\_direct (*C++ enumerator*), 52  
 dnnl::algorithm::convolution\_winograd (*C++ enumerator*), 52  
 dnnl::algorithm::deconvolution\_direct (*C++ enumerator*), 53  
 dnnl::algorithm::deconvolution\_winograd (*C++ enumerator*), 53  
 dnnl::algorithm::eltwise\_abs (*C++ enumerator*), 53  
 dnnl::algorithm::eltwise\_bounded\_relu (*C++ enumerator*), 53  
 dnnl::algorithm::eltwise\_clip (*C++ enumerator*), 53  
 dnnl::algorithm::eltwise\_elu (*C++ enumerator*), 53  
 dnnl::algorithm::eltwise\_elu\_use\_dst\_for\_bwd (*C++ enumerator*), 54  
 dnnl::algorithm::eltwise\_exp (*C++ enumerator*), 53  
 dnnl::algorithm::eltwise\_exp\_use\_dst\_for\_bwd (*C++ enumerator*), 54  
 dnnl::algorithm::eltwise\_gelu (*C++ enumerator*), 53  
 dnnl::algorithm::eltwise\_gelu\_erf (*C++ enumerator*), 53  
 dnnl::algorithm::eltwise\_gelu\_tanh (*C++ enumerator*), 53  
 dnnl::algorithm::eltwise\_linear (*C++ enumerator*), 53  
 dnnl::algorithm::eltwise\_log (*C++ enumerator*), 53  
 dnnl::algorithm::eltwise\_logistic (*C++ enumerator*), 53  
 dnnl::algorithm::eltwise\_logistic\_use\_dst\_for\_bwd (*C++ enumerator*), 54  
 dnnl::algorithm::eltwise\_pow (*C++ enumerator*), 53  
 dnnl::algorithm::eltwise\_relu (*C++ enumerator*), 53  
 dnnl::algorithm::eltwise\_relu\_use\_dst\_for\_bwd (*C++ enumerator*), 54  
 dnnl::algorithm::eltwise\_round (*C++ enumerator*), 53  
 dnnl::algorithm::eltwise\_soft\_relu (*C++ enumerator*), 53  
 dnnl::algorithm::eltwise\_sqrt (*C++ enumerator*), 53  
 dnnl::algorithm::eltwise\_sqrt\_use\_dst\_for\_bwd (*C++ enumerator*), 54  
 dnnl::algorithm::eltwise\_square (*C++ enumerator*), 53

dnnl::algorithm::eltwise\_swish (*C++ enumerator*), 53  
dnnl::algorithm::eltwise\_tanh (*C++ enumerator*), 53  
dnnl::algorithm::eltwise\_tanh\_use\_dst\_for\_bwd (*C++ enumerator*), 54  
dnnl::algorithm::lbr\_gru (*C++ enumerator*), 54  
dnnl::algorithm::lrn\_across\_channels (*C++ enumerator*), 54  
dnnl::algorithm::lrn\_within\_channel (*C++ enumerator*), 54  
dnnl::algorithm::pooling\_avg (*C++ enumerator*), 54  
dnnl::algorithm::pooling\_avg\_exclude\_padding (*C++ enumerator*), 54  
dnnl::algorithm::pooling\_avg\_include\_padding (*C++ enumerator*), 54  
dnnl::algorithm::pooling\_max (*C++ enumerator*), 54  
dnnl::algorithm::resampling\_linear (*C++ enumerator*), 55  
dnnl::algorithm::resampling\_nearest (*C++ enumerator*), 54  
dnnl::algorithm::undef (*C++ enumerator*), 52  
dnnl::algorithm::vanilla\_gru (*C++ enumerator*), 54  
dnnl::algorithm::vanilla\_lstm (*C++ enumerator*), 54  
dnnl::algorithm::vanilla\_rnn (*C++ enumerator*), 54  
dnnl::batch\_normalization\_backward (*C++ struct*), 77  
dnnl::batch\_normalization\_backward::batch\_normalization\_backward (*C++ function*), 78  
dnnl::batch\_normalization\_backward::descdnnl::binary (*C++ struct*), 81  
dnnl::batch\_normalization\_backward::descdnnl::binary::desc (*C++ function*), 81  
dnnl::batch\_normalization\_backward::descdnnl::binary::primitive\_desc (*C++ struct*), 81  
dnnl::batch\_normalization\_backward::descdnnl::binary::primitive\_desc::dst\_desc (*C++ function*), 82  
dnnl::batch\_normalization\_backward::primdnnl::binary::primitive\_desc::primitive\_desc (*C++ function*), 82  
dnnl::batch\_normalization\_backward::primdnnl::binary::primitive\_desc::src0\_desc (*C++ function*), 82  
dnnl::batch\_normalization\_backward::primdnnl::binary::primitive\_desc::src1\_desc (*C++ function*), 82  
dnnl::batch\_normalization\_backward::primdnnl::binary::primitive\_desc::src\_desc (*C++ function*), 82  
dnnl::batch\_normalization\_backward::primdnnl::descat (*C++ struct*), 83  
dnnl::batch\_normalization\_backward::primdnnl::descat::concat (*C++ function*), 84  
dnnl::batch\_normalization\_backward::primdnnl::descat::primitive\_desc (*C++ struct*), 84  
dnnl::batch\_normalization\_backward::primdnnl::descat::primitive\_desc::dst\_desc (*C++ function*), 84  
dnnl::batch\_normalization\_backward::primdnnl::descat::primitive\_desc::dst\_desc (*C++ function*), 84

(*C++ function*), 84  
dnnl::concat::primitive\_desc::src\_desc (*C++ function*), 84  
dnnl::convolution\_backward\_data (*C++ struct*), 94  
dnnl::convolution\_backward\_data::convolution\_ba (*C++ function*), 94  
dnnl::convolution\_backward\_data::desc (*C++ struct*), 94  
dnnl::convolution\_backward\_data::desc::desc (*C++ function*), 94, 95  
dnnl::convolution\_backward\_data::primitive\_desd (*C++ struct*), 103  
dnnl::convolution\_backward\_data::primitive\_desd (*C++ function*), 96  
dnnl::convolution\_backward\_data::primitive\_desd (*C++ struct*), 104  
dnnl::convolution\_backward\_data::primitive\_desd (*C++ function*), 96  
dnnl::convolution\_backward\_data::primitive\_desd (*C++ function*), 104  
dnnl::convolution\_backward\_data::primitive\_desd (*C++ function*), 95  
dnnl::convolution\_backward\_data::primitive\_desd (*C++ function*), 103  
dnnl::convolution\_backward\_data::deconvolution\_ba (*C++ function*), 93  
dnnl::convolution\_backward\_data::deconvolution\_backward\_data (*C++ struct*), 102  
dnnl::convolution\_backward\_data::deconvolution\_backward\_data::desc (*C++ function*), 103  
dnnl::convolution\_backward\_data::deconvolution\_backward\_data::desc::desc (*C++ function*), 103  
dnnl::convolution\_backward\_data::deconvolution\_backward\_data::desc::desc (*C++ function*), 104  
dnnl::convolution\_backward\_data::deconvolution\_backward\_data::desc::desc (*C++ function*), 104  
dnnl::convolution\_backward\_data::deconvolution\_backward\_data::desc::desc (*C++ function*), 104  
dnnl::convolution\_backward\_data::deconvolution\_backward\_data::desc::desc (*C++ function*), 104  
dnnl::convolution\_backward\_weights (*C++ struct*), 96  
dnnl::convolution\_backward\_weights::convolution (*C++ function*), 104  
dnnl::convolution\_backward\_weights::desc (*C++ struct*), 96  
dnnl::convolution\_backward\_weights::desc::desc (*C++ function*), 105  
dnnl::convolution\_backward\_weights::desc::desc (*C++ function*), 96–98  
dnnl::convolution\_backward\_weights::primitive\_desd (*C++ struct*), 105  
dnnl::convolution\_backward\_weights::primitive\_desd (*C++ function*), 98  
dnnl::convolution\_backward\_weights::primitive\_desd (*C++ function*), 105  
dnnl::convolution\_backward\_weights::primitive\_desd (*C++ function*), 99  
dnnl::convolution\_backward\_weights::primitive\_desd (*C++ function*), 106  
dnnl::convolution\_backward\_weights::primitive\_desd (*C++ function*), 107  
dnnl::convolution\_backward\_weights::primitive\_desd (*C++ function*), 99  
dnnl::convolution\_backward\_weights::primitive\_desd (*C++ function*), 108  
dnnl::convolution\_backward\_weights::primitive\_desd (*C++ function*), 99  
dnnl::convolution\_backward\_weights::primitive\_desd (*C++ function*), 109  
dnnl::convolution\_backward\_weights::primitive\_desd (*C++ function*), 99  
dnnl::convolution\_backward\_weights::primitive\_desd (*C++ function*), 107  
dnnl::convolution\_forward (*C++ struct*), 90  
dnnl::convolution\_forward::convolution\_fdmwärde (*C++ function*), 107  
dnnl::convolution\_forward::desc (*C++ struct*), 91  
dnnl::convolution\_forward::desc::desc (*C++ function*), 91, 92  
dnnl::convolution\_forward::primitive\_desd  
dnnl::deconvolution\_forward::desc (*C++ struct*), 99  
dnnl::deconvolution\_forward::deconvolution\_forward::desc (*C++ function*), 99  
dnnl::deconvolution\_forward::deconvolution\_forward::desc::desc (*C++ function*), 99  
dnnl::deconvolution\_forward::deconvolution\_forward::desc::desc (*C++ function*), 100, 101  
dnnl::deconvolution\_forward::deconvolution\_forward::desc::desc (*C++ struct*), 102

```

dnnl::deconvolution_forward::primitive_desc::bias_desc
    (C++ function), 102           dnnl::engine::kind::gpu (C++ enumerator),
dnnl::deconvolution_forward::primitive_desc::ds23_desc
    (C++ function), 102           dnnl::error (C++ struct), 20
dnnl::deconvolution_forward::primitive_desc::pgimibàvkwadé (C++ struct), 180
    (C++ function), 102           dnnl::gru_backward::desc (C++ struct), 180
dnnl::deconvolution_forward::primitive_desc::sgcudbackward::desc (C++ func-
    (C++ function), 102           tion), 181
dnnl::deconvolution_forward::primitive_desc::wgrighbacward::gru_backward (C++
    (C++ function), 102           function), 180
dnnl::eltwise_backward (C++ struct), 112       dnnl::gru_backward::primitive_desc (C++
dnnl::eltwise_backward::desc (C++ struct),
    112                           struct), 181
dnnl::eltwise_backward::desc (C++ function),
    113                           dnnl::gru_backward::primitive_desc::bias_desc
dnnl::eltwise_backward::eltwise_backward
    (C++ function), 112           (C++ function), 182
dnnl::eltwise_backward::primitive_desc
    (C++ struct), 113           dnnl::gru_backward::primitive_desc::diff_bias_desc
dnnl::eltwise_backward::primitive_desc::diff_ds (C++ function), 183
    (C++ function), 114           dnnl::gru_backward::primitive_desc::diff_dst_iter_
dnnl::eltwise_backward::primitive_desc::diff_sr (C++ function), 183
    (C++ function), 113           dnnl::gru_backward::primitive_desc::diff_src_layer_
dnnl::eltwise_backward::primitive_desc::primiti(C++ function), 183
    (C++ function), 113           dnnl::gru_backward::primitive_desc::diff_weights_it_
dnnl::eltwise_backward::primitive_desc::src_des(C++ function), 183
    (C++ function), 113           dnnl::gru_backward::primitive_desc::diff_weights_la_
dnnl::eltwise_forward (C++ struct), 111
dnnl::eltwise_forward::desc (C++ struct),
    111                           dnnl::gru_backward::primitive_desc::dst_iter_desc
dnnl::eltwise_forward::desc (C++ function),
    111                           (C++ function), 182
dnnl::eltwise_forward::eltwise_forward
    (C++ function), 111           dnnl::gru_backward::primitive_desc::dst_layer_desc
dnnl::eltwise_forward::primitive_desc
    (C++ struct), 111           (C++ function), 182
dnnl::eltwise_forward::primitive_desc::dshndesgru_backward::primitive_desc::src_layer_desc
    (C++ function), 112           (C++ function), 182
dnnl::eltwise_forward::primitive_desc::pdmiltigéudbackward::primitive_desc::weights_iter_de
    (C++ function), 112           (C++ function), 182
dnnl::eltwise_forward::primitive_desc::sdondesgru_backward::primitive_desc::weights_layer_o
    (C++ function), 112           (C++ function), 182
dnnl::engine (C++ struct), 23
dnnl::engine::engine (C++ function), 23
dnnl::engine::get_count (C++ function), 24
dnnl::engine::get_kind (C++ function), 23
dnnl::engine::get_sycl_context (C++ func-
    tion), 24
dnnl::engine::get_sycl_device (C++ func-
    tion), 24
dnnl::engine::kind (C++ enum), 23
dnnl::engine::kind::any (C++ enumerator),
    23
dnnl::engine::kind::cpu (C++ enumerator),

```



dnnl::layer\_normalization\_forward::desc::desc (C++ function), 187  
(C++ function), 125 dnnl::lbr\_gru\_backward::primitive\_desc::weights\_it...  
dnnl::layer\_normalization\_forward::layer\_norm(C++ function), 187  
(C++ function), 124 dnnl::lbr\_gru\_backward::primitive\_desc::weights\_lay...  
dnnl::layer\_normalization\_forward::primitive\_de(C++ function), 187  
(C++ struct), 125 dnnl::lbr\_gru\_backward::primitive\_desc::workspace\_c...  
dnnl::layer\_normalization\_forward::primitive\_de(C++ function), 188  
(C++ function), 126 dnnl::lbr\_gru\_forward (C++ struct), 183  
dnnl::layer\_normalization\_forward::primitive\_de(C++ function), 188  
(C++ function), 126 183  
dnnl::layer\_normalization\_forward::primitive\_de(C++ function), 184  
(C++ function), 125  
dnnl::layer\_normalization\_forward::primitive\_de(C++ function), 183  
(C++ function), 125  
dnnl::layer\_normalization\_forward::primitive\_de(C++ function), 184  
(C++ function), 126  
dnnl::layer\_normalization\_forward::primitive\_de(C++ function), 183  
(C++ function), 126  
dnnl::layer\_normalization\_forward::primitive\_de(C++ function), 185  
(C++ function), 126  
dnnl::layer\_normalization\_forward::primitive\_de(C++ function), 185  
(C++ function), 126  
dnnl::layer\_normalization\_forward::primitive\_de(C++ function), 185  
(C++ function), 126  
dnnl::lbr\_gru\_backward (C++ struct), 185 dnnl::lbr\_gru\_forward::primitive\_desc::dst\_iter\_des...  
dnnl::lbr\_gru\_backward::desc (C++ struct), 185 (C++ function), 185  
185 dnnl::lbr\_gru\_forward::primitive\_desc::primitive\_de...  
dnnl::lbr\_gru\_backward::desc::desc (C++ function), 186 (C++ function), 184  
dnnl::lbr\_gru\_backward::lbr\_gru\_backward (C++ function), 185 dnnl::lbr\_gru\_forward::primitive\_desc::src\_iter\_des...  
dnnl::lbr\_gru\_backward::primitive\_desc (C++ struct), 186 (C++ function), 185  
dnnl::lbr\_gru\_backward::primitive\_desc (C++ struct), 186 (C++ function), 185  
dnnl::lbr\_gru\_backward::primitive\_desc::bias\_de(C++ function), 185  
(C++ function), 187 dnnl::lbr\_gru\_forward::primitive\_desc::weights\_lay...  
dnnl::lbr\_gru\_backward::primitive\_desc::diff\_bias(C++ function), 185  
(C++ function), 188 dnnl::lbr\_gru\_forward::primitive\_desc::workspace\_d...  
dnnl::lbr\_gru\_backward::primitive\_desc::diff\_desc(C++ function), 185  
(C++ function), 188 dnnl::logsoftmax\_backward (C++ struct), 131  
dnnl::lbr\_gru\_backward::primitive\_desc::diff\_desc(C++ function), 188 dnnl::logsoftmax\_backward::desc (C++ struct), 131  
dnnl::lbr\_gru\_backward::primitive\_desc::diff\_desc(C++ function), 188 (C++ function), 131  
dnnl::lbr\_gru\_backward::primitive\_desc::diff\_desc(C++ function), 188  
(C++ function), 188 dnnl::logsoftmax\_backward::primitive\_desc (C++ struct), 132  
dnnl::lbr\_gru\_backward::primitive\_desc::diff\_desc(C++ function), 188  
(C++ function), 188 dnnl::logsoftmax\_backward::primitive\_desc::diff\_dst...  
dnnl::lbr\_gru\_backward::primitive\_desc::diff\_desc(C++ function), 188 (C++ function), 132  
dnnl::lbr\_gru\_backward::primitive\_desc::diff\_desc(C++ function), 187 dnnl::logsoftmax\_backward::primitive\_desc::diff\_src...  
dnnl::lbr\_gru\_backward::primitive\_desc::diff\_desc(C++ function), 187 (C++ function), 132  
dnnl::lbr\_gru\_backward::primitive\_desc::diff\_desc(C++ function), 187  
(C++ function), 187 dnnl::logsoftmax\_forward (C++ struct), 130  
dnnl::lbr\_gru\_backward::primitive\_desc::src\_layer(C++ struct), 130

dnnl::logsoftmax_forward::desc::desc (C++ function), 130	dnnl::lstm_backward::primitive_desc::diff_bias_desc (C++ function), 178
dnnl::logsoftmax_forward::logsoftmax_forward::lstm_backward::primitive_desc::diff_dst_iter (C++ function), 178	
dnnl::logsoftmax_forward::primitive_descdnnl::lstm_backward::primitive_desc::diff_dst_iter (C++ struct), 130	dnnl::lstm_backward::primitive_desc::diff_dst_iter (C++ function), 178
dnnl::logsoftmax_forward::primitive_descdndst::dnnl::lstm_backward::primitive_desc::diff_dst_layer (C++ function), 131	dnnl::lstm_backward::primitive_desc::diff_dst_layer (C++ function), 178
dnnl::logsoftmax_forward::primitive_descdmpfimistimebackward::primitive_desc::diff_src_iter (C++ function), 130, 131	dnnl::lstm_backward::primitive_desc::diff_src_iter (C++ function), 177
dnnl::logsoftmax_forward::primitive_descdnsitic::dnnl::lstm_backward::primitive_desc::diff_src_iter (C++ function), 131	dnnl::lstm_backward::primitive_desc::diff_src_iter (C++ function), 177
dnnl::lrn_backward (C++ struct), 136	dnnl::lstm_backward::primitive_desc::diff_src_layer (C++ function), 177
dnnl::lrn_backward::desc (C++ struct), 136	dnnl::lstm_backward::primitive_desc::diff_weights_
dnnl::lrn_backward::desc::desc (C++ func- tion), 136	tion), 178
dnnl::lrn_backward::lrn_backward (C++ function), 136	dnnl::lstm_backward::primitive_desc::diff_weights_
dnnl::lrn_backward::primitive_desc (C++ struct), 136	tion), 178
dnnl::lrn_backward::primitive_desc::diffddat::dnnl::lstm_backward::primitive_desc::diff_weights_	function), 178
(C++ function), 137	(C++ function), 178
dnnl::lrn_backward::primitive_desc::diffdst::dnnl::lstm_backward::primitive_desc::dst_iter_c_des- c (C++ function), 137	c (C++ function), 177
dnnl::lrn_backward::primitive_desc::primdimve::dnnl::lstm_backward::primitive_desc::dst_iter_des- c (C++ function), 137	c (C++ function), 177
dnnl::lrn_backward::primitive_desc::workspade::dnnl::lstm_backward::primitive_desc::dst_layer_des- c (C++ function), 137	c (C++ function), 177
dnnl::lrn_forward (C++ struct), 134	dnnl::lstm_backward::primitive_desc::primitive_desc (C++ function), 176
dnnl::lrn_forward::desc (C++ struct), 135	dnnl::lstm_backward::primitive_desc::src_iter_c_des- c (C++ function), 177
dnnl::lrn_forward::desc::desc (C++ func- tion), 135	dnnl::lstm_backward::primitive_desc::src_iter_desc (C++ function), 176
dnnl::lrn_forward::lrn_forward (C++ func- tion), 135	dnnl::lstm_backward::primitive_desc::src_layer_des- c (C++ function), 176
dnnl::lrn_forward::primitive_desc (C++ struct), 135	dnnl::lstm_backward::primitive_desc::src_layer_des- c (C++ function), 176
dnnl::lrn_forward::primitive_desc::dst_densl::lstm_backward::primitive_desc::weights_iter_	c (C++ function), 177
(C++ function), 136	
dnnl::lrn_forward::primitive_desc::primdimve::dnnl::lstm_backward::primitive_desc::weights_layer_	c (C++ function), 177
(C++ function), 135	
dnnl::lrn_forward::primitive_desc::src_densl::lstm_backward::primitive_desc::weights_peephole_	c (C++ function), 177
(C++ function), 136	
dnnl::lrn_forward::primitive_desc::workspane::dnnl::lstm_backward::primitive_desc::weights_proje- ct (C++ function), 136	c (C++ function), 177
dnnl::lstm_backward (C++ struct), 172	dnnl::lstm_backward::primitive_desc::workspace_des- c (C++ function), 177
dnnl::lstm_backward::desc (C++ struct), 172	dnnl::lstm_forward (C++ struct), 168
dnnl::lstm_backward::desc::desc (C++ function), 173–175	dnnl::lstm_forward::desc (C++ struct), 169
dnnl::lstm_backward::lstm_backward (C++ function), 172	dnnl::lstm_forward::desc (C++ func- tion), 169, 170
dnnl::lstm_backward::primitive_desc (C++ struct), 176	dnnl::lstm_forward::lstm_forward (C++ function), 169
dnnl::lstm_backward::primitive_desc::biasdndsc::lstm_forward::primitive_desc (C++ function), 177	dnnl::lstm_forward::lstm_forward (C++ struct), 171

dnnl::lstm\_forward::primitive\_desc::bias\_desc::memory::data\_type::u8 (*C++ enumerator*, 172)  
dnnl::lstm\_forward::primitive\_desc::dst\_desc::memory::data\_type::undef (*C++ enumerator*, 172)  
dnnl::lstm\_forward::primitive\_desc::dst\_desc::memory::desc (*C++ struct*, 36)  
(C++ function), 172 dnnl::memory::desc::data\_type (*C++ function*, 26)  
dnnl::lstm\_forward::primitive\_desc::dst\_layer\_desc, 38  
(C++ function), 172 dnnl::memory::desc::desc (*C++ function*, 36, 37)  
dnnl::lstm\_forward::primitive\_desc::primitive\_desc  
(C++ function), 171 dnnl::memory::desc::dims (*C++ function*, 38)  
dnnl::lstm\_forward::primitive\_desc::src\_desc::memory::desc::get\_size (*C++ function*, 39)  
(C++ function), 171  
dnnl::lstm\_forward::primitive\_desc::src\_desc::is\_zero (*C++ function*, 39)  
(C++ function), 171  
dnnl::lstm\_forward::primitive\_desc::src\_desc::operator!= (*C++ function*, 39)  
(C++ function), 171  
dnnl::lstm\_forward::primitive\_desc::weights\_desc::operator== (*C++ function*, 39)  
(C++ function), 172  
dnnl::lstm\_forward::primitive\_desc::weights\_desc::permute\_axes (*C++ function*, 38)  
(C++ function), 172  
dnnl::lstm\_forward::primitive\_desc::weights\_desc::reshape (*C++ function*, 37)  
(C++ function), 172  
dnnl::lstm\_forward::primitive\_desc::weights\_desc::submemory\_desc (*C++ function*, 37)  
(C++ function), 172  
dnnl::lstm\_forward::primitive\_desc::workspacememory::dim (*C++ type*, 28)  
(C++ function), 172 dnnl::memory::dims (*C++ type*, 28)  
dnnl::matmul (*C++ struct*, 140)  
dnnl::matmul::desc (*C++ struct*, 140)  
dnnl::matmul::desc::desc (*C++ function*, 140)  
dnnl::matmul::matmul (*C++ function*, 140)  
dnnl::matmul::primitive\_desc (*C++ struct*, 140)  
dnnl::matmul::primitive\_desc::bias\_desc  
(C++ function), 141  
dnnl::matmul::primitive\_desc::dst\_desc  
(C++ function), 141  
dnnl::matmul::primitive\_desc::primitive\_desc  
(C++ function), 141  
dnnl::matmul::primitive\_desc::src\_desc  
(C++ function), 141  
dnnl::matmul::primitive\_desc::weights\_desc  
(C++ function), 141  
dnnl::memory (*C++ struct*, 39)  
dnnl::memory::data\_type (*C++ enum*, 26)  
dnnl::memory::data\_type::bf16 (*C++ enumerator*, 26)  
dnnl::memory::data\_type::f16 (*C++ enumerator*, 26)  
dnnl::memory::data\_type::f32 (*C++ enumerator*, 26)  
dnnl::memory::data\_type::s32 (*C++ enumerator*, 26)  
dnnl::memory::data\_type::s8 (*C++ enumerator*, 26)  
dnnl::memory::format\_tag::any (*C++ enumerator*, 32)  
dnnl::memory::format\_tag::ab (*C++ enumerator*, 32)  
dnnl::memory::format\_tag::abc (*C++ enumerator*, 32)  
dnnl::memory::format\_tag::abcd (*C++ enumerator*, 32)  
dnnl::memory::format\_tag::abcde (*C++ enumerator*, 32)  
dnnl::memory::format\_tag::abcdef (*C++ enumerator*, 32)  
dnnl::memory::format\_tag::abdc (*C++ enumerator*, 32)  
dnnl::memory::format\_tag::abdec (*C++ enumerator*, 32)  
dnnl::memory::format\_tag::acb (*C++ enumerator*, 32)  
dnnl::memory::format\_tag::acbde (*C++ enumerator*, 32)  
dnnl::memory::format\_tag::acbdef (*C++ enumerator*, 32)  
dnnl::memory::format\_tag::acdb (*C++ enumerator*, 32)  
dnnl::memory::format\_tag::acdeb (*C++ enumerator*, 32)  
dnnl::memory::format\_tag::any (*C++ enumerator*, 32)

*merator), 32*  
*dnnl::memory::format\_tag::ba (C++ enumerator), 32*  
*dnnl::memory::format\_tag::bac (C++ enumerator), 32*  
*dnnl::memory::format\_tag::bacd (C++ enumerator), 32*  
*dnnl::memory::format\_tag::bacde (C++ enumerator), 33*  
*dnnl::memory::format\_tag::bca (C++ enumerator), 32*  
*dnnl::memory::format\_tag::bcda (C++ enumerator), 32*  
*dnnl::memory::format\_tag::bcdea (C++ enumerator), 33*  
*dnnl::memory::format\_tag::cba (C++ enumerator), 32*  
*dnnl::memory::format\_tag::cdba (C++ enumerator), 33*  
*dnnl::memory::format\_tag::cdeba (C++ enumerator), 33*  
*dnnl::memory::format\_tag::chwn (C++ enumerator), 34*  
*dnnl::memory::format\_tag::cn (C++ enumerator), 33*  
*dnnl::memory::format\_tag::dcab (C++ enumerator), 33*  
*dnnl::memory::format\_tag::decab (C++ enumerator), 33*  
*dnnl::memory::format\_tag::defcab (C++ enumerator), 33*  
*dnnl::memory::format\_tag::dhwigo (C++ enumerator), 35*  
*dnnl::memory::format\_tag::dhwio (C++ enumerator), 34*  
*dnnl::memory::format\_tag::giodhw (C++ enumerator), 35*  
*dnnl::memory::format\_tag::giohw (C++ enumerator), 35*  
*dnnl::memory::format\_tag::goidhw (C++ enumerator), 35*  
*dnnl::memory::format\_tag::goihw (C++ enumerator), 35*  
*dnnl::memory::format\_tag::goiw (C++ enumerator), 34*  
*dnnl::memory::format\_tag::hwigo (C++ enumerator), 35*  
*dnnl::memory::format\_tag::hwio (C++ enumerator), 34*  
*dnnl::memory::format\_tag::idhwo (C++ enumerator), 34*  
*dnnl::memory::format\_tag::ihwo (C++ enumerator), 34*  
*dnnl::memory::format\_tag::io (C++ enumerator), 34*  
*ator), 34*  
*dnnl::memory::format\_tag::iodhw (C++ enumerator), 34*  
*dnnl::memory::format\_tag::iohw (C++ enumerator), 34*  
*dnnl::memory::format\_tag::iwo (C++ enumerator), 34*  
*dnnl::memory::format\_tag::ldgo (C++ enumerator), 35*  
*dnnl::memory::format\_tag::ldgoi (C++ enumerator), 35*  
*dnnl::memory::format\_tag::ldigo (C++ enumerator), 35*  
*dnnl::memory::format\_tag::ldio (C++ enumerator), 35*  
*dnnl::memory::format\_tag::ldnc (C++ enumerator), 35*  
*dnnl::memory::format\_tag::ldoi (C++ enumerator), 35*  
*dnnl::memory::format\_tag::nc (C++ enumerator), 33*  
*dnnl::memory::format\_tag::ncdhw (C++ enumerator), 34*  
*dnnl::memory::format\_tag::nchw (C++ enumerator), 33*  
*dnnl::memory::format\_tag::ncw (C++ enumerator), 33*  
*dnnl::memory::format\_tag::ndhwc (C++ enumerator), 34*  
*dnnl::memory::format\_tag::nhwc (C++ enumerator), 34*  
*dnnl::memory::format\_tag::nt (C++ enumerator), 33*  
*dnnl::memory::format\_tag::ntc (C++ enumerator), 35*  
*dnnl::memory::format\_tag::nwc (C++ enumerator), 33*  
*dnnl::memory::format\_tag::odhwi (C++ enumerator), 34*  
*dnnl::memory::format\_tag::ohwi (C++ enumerator), 34*  
*dnnl::memory::format\_tag::oi (C++ enumerator), 34*  
*dnnl::memory::format\_tag::oidhw (C++ enumerator), 34*  
*dnnl::memory::format\_tag::oihw (C++ enumerator), 34*  
*dnnl::memory::format\_tag::oiw (C++ enumerator), 34*  
*dnnl::memory::format\_tag::owi (C++ enumerator), 34*  
*dnnl::memory::format\_tag::tn (C++ enumerator), 33*  
*dnnl::memory::format\_tag::tnc (C++ enumerator), 34*

*merator), 35*  
 dnnl::memory::format\_tag::undef (*C++ enum*, 32)  
 dnnl::memory::format\_tag::wigo (*C++ enum*, 35)  
 dnnl::memory::format\_tag::wio (*C++ enum*, 34)  
 dnnl::memory::format\_tag::x (*C++ enum*, 33)  
 dnnl::memory::get\_data\_handle (*C++ function*, 40)  
 dnnl::memory::get\_desc (*C++ function*, 40)  
 dnnl::memory::get\_engine (*C++ function*, 40)  
 dnnl::memory::get\_sycl\_buffer (*C++ function*, 41)  
 dnnl::memory::map\_data (*C++ function*, 41)  
 dnnl::memory::memory (*C++ function*, 39, 40)  
 dnnl::memory::set\_data\_handle (*C++ function*, 40, 41)  
 dnnl::memory::set\_sycl\_buffer (*C++ function*, 42)  
 dnnl::memory::unmap\_data (*C++ function*, 41)  
 dnnl::normalization\_flags (*C++ enum*, 55)  
 dnnl::normalization\_flags::fuse\_norm\_reldnnl::primitive (*C++ struct*, 44)  
*(C++ enumerator), 55*  
 dnnl::normalization\_flags::none (*C++ enum*, 55)  
 dnnl::normalization\_flags::use\_global\_stathsl::primitive::get\_kind (*C++ function*, 45)  
*(C++ enumerator), 55*  
 dnnl::normalization\_flags::use\_scale\_shiftnl::primitive::kind::batch\_normalization  
*(C++ enumerator), 55*  
 dnnl::pooling\_backward (*C++ struct*, 145)  
 dnnl::pooling\_backward::desc (*C++ struct*, 145)  
 dnnl::pooling\_backward::desc::desc (*C++ function*, 146)  
 dnnl::pooling\_backward::pooling\_backward (*C++ function*, 145)  
 dnnl::pooling\_backward::primitive\_desc (*C++ struct*, 146)  
 dnnl::pooling\_backward::primitive\_desc::desc (*C++ function*, 147)  
 dnnl::pooling\_backward::primitive\_desc::diff\_differntiator, 45  
*(C++ function), 147*  
 dnnl::pooling\_backward::primitive\_desc::diff\_sr(C+de enumerator, 45)  
*(C++ function), 147*  
 dnnl::pooling\_backward::primitive\_desc::primiti(C+de enumerator, 45)  
*(C++ function), 146*  
 dnnl::pooling\_backward::primitive\_desc::workspacenumerator, 45  
*(C++ function), 147*  
 dnnl::pooling\_forward (*C++ struct*, 143)  
 dnnl::pooling\_forward::desc (*C++ struct*, 144)  
 dnnl::pooling\_forward::desc::desc (*C++ function*, 144)  
 dnnl::pooling\_forward::pooling\_forward (*C++ enum*, 144)  
 dnnl::pooling\_forward::primitive\_desc  
*(C++ struct), 144*  
 dnnl::pooling\_forward::primitive\_desc::dst\_desc  
*(C++ function), 145*  
 dnnl::pooling\_forward::primitive\_desc::primitive\_desc  
*(C++ function), 144, 145*  
 dnnl::pooling\_forward::primitive\_desc::src\_desc  
*(C++ function), 145*  
 dnnl::pooling\_forward::primitive\_desc::workspace\_desc  
*(C++ function), 145*  
 dnnl::post\_ops (*C++ struct*, 61)  
 dnnl::post\_ops::append\_eltwise (*C++ function*, 62)  
 dnnl::post\_ops::append\_sum (*C++ function*, 61)  
 dnnl::post\_ops::get\_params\_eltwise (*C++ function*, 62)  
 dnnl::post\_ops::get\_params\_sum (*C++ function*, 61)  
 dnnl::post\_ops::kind (*C++ function*, 61)  
 dnnl::post\_ops::len (*C++ function*, 61)  
 dnnl::post\_ops::post\_ops (*C++ function*, 61)  
 dnnl::primitive (*C++ struct*, 44)  
 dnnl::primitive::execute (*C++ function*, 45)  
 dnnl::primitive::execute\_sycl (*C++ function*, 46)  
 dnnl::primitive::get\_kind (*C++ function*, 45)  
 dnnl::primitive::kind (*C++ enum*, 44)  
 dnnl::primitive::kind::batch\_normalization  
*(C++ enumerator), 45*  
 dnnl::primitive::kind::binary (*C++ enum*, 45)  
 dnnl::primitive::kind::concat (*C++ enum*, 44)  
 dnnl::primitive::kind::convolution (*C++ enum*, 44)  
 dnnl::primitive::kind::deconvolution  
*(C++ enumerator), 45*  
 dnnl::primitive::kind::eltwise (*C++ enum*, 45)  
 dnnl::primitive::kind::inner\_product  
*(C++ enumerator), 45*  
 dnnl::primitive::kind::layer\_normalization  
*(C++ enumerator), 45*  
 dnnl::primitive::kind::logsoftmax (*C++ enum*, 45)  
 dnnl::primitive::kind::workspacenumerator, 45  
*(C++ function), 145*  
 dnnl::primitive::kind::lrn (*C++ enum*, 45)  
 dnnl::primitive::kind::matmul (*C++ enum*, 45)  
 dnnl::primitive::kind::pooling (*C++ enum*, 45)  
 dnnl::primitive::kind::reorder (*C++ enum*, 45)

merator), 44  
 dnnl::primitive::kind::resampling (C++ enumerator), 45  
 dnnl::primitive::kind::rnn (C++ enumerator), 45  
 dnnl::primitive::kind::shuffle (C++ enumerator), 44  
 dnnl::primitive::kind::softmax (C++ enumerator), 45  
 dnnl::primitive::kind::sum (C++ enumerator), 44  
 dnnl::primitive::kind::undef (C++ enumerator), 44  
 dnnl::primitive::operator= (C++ function), 46  
 dnnl::primitive::primitive (C++ function), 45  
 dnnl::primitive\_attr (C++ struct), 69  
 dnnl::primitive\_attr::get\_output\_scales (C++ function), 69  
 dnnl::primitive\_attr::get\_post\_ops (C++ function), 71  
 dnnl::primitive\_attr::get\_scales (C++ function), 70  
 dnnl::primitive\_attr::get\_scratchpad\_mode  
 dnnl::primitive\_attr::get\_zero\_points (C++ function), 70  
 dnnl::primitive\_attr::primitive\_attr (C++ function), 69  
 dnnl::primitive\_attr::set\_output\_scales (C++ function), 69  
 dnnl::primitive\_attr::set\_post\_ops (C++ function), 71  
 dnnl::primitive\_attr::set\_rnn\_data\_qparams (C++ function), 71  
 dnnl::primitive\_attr::set\_rnn\_weights\_qparams (C++ function), 72  
 dnnl::primitive\_attr::set\_scales (C++ function), 70  
 dnnl::primitive\_attr::set\_scratchpad\_mode (C++ function), 69  
 dnnl::primitive\_attr::set\_zero\_points (C++ function), 71  
 dnnl::primitive\_desc (C++ struct), 49  
 dnnl::primitive\_desc::next\_impl (C++ function), 49  
 dnnl::primitive\_desc::primitive\_desc (C++ function), 49  
 dnnl::primitive\_desc\_base (C++ struct), 46  
 dnnl::primitive\_desc\_base::diff\_dst\_desc  
 dnnl::primitive\_desc\_base::diff\_src\_desc  
 dnnl::primitive\_desc\_base::dst\_desc

dnnl::primitive\_desc\_base::diff\_weights\_desc (C++ function), 48  
 dnnl::primitive\_desc\_base::dst\_desc (C++ function), 47, 48  
 dnnl::primitive\_desc\_base::get\_engine (C++ function), 46  
 dnnl::primitive\_desc\_base::get\_kind (C++ function), 49  
 dnnl::primitive\_desc\_base::get\_primitive\_attr (C++ function), 49  
 dnnl::primitive\_desc\_base::impl\_info\_str (C++ function), 46  
 dnnl::primitive\_desc\_base::primitive\_desc\_base (C++ function), 46  
 dnnl::primitive\_desc\_base::query\_md (C++ function), 47  
 dnnl::primitive\_desc\_base::query\_s64 (C++ function), 46  
 dnnl::primitive\_desc\_base::scratchpad\_desc (C++ function), 49  
 dnnl::primitive\_desc\_base::scratchpad\_engine (C++ function), 49  
 dnnl::primitive\_desc\_base::src\_desc (C++ function), 47, 48  
 dnnl::primitive\_desc\_base::weights\_desc (C++ function), 47, 48  
 dnnl::primitive\_desc\_base::workspace\_desc (C++ function), 49  
 dnnl::prop\_kind (C++ enum), 52  
 dnnl::prop\_kind::backward (C++ enumerator), 52  
 dnnl::prop\_kind::backward\_bias (C++ enumerator), 52  
 dnnl::prop\_kind::backward\_data (C++ enumerator), 52  
 dnnl::prop\_kind::backward\_weights (C++ enumerator), 52  
 dnnl::prop\_kind::forward (C++ enumerator), 52  
 dnnl::prop\_kind::forward\_inference (C++ enumerator), 52  
 dnnl::prop\_kind::forward\_scoring (C++ enumerator), 52  
 dnnl::prop\_kind::forward\_training (C++ enumerator), 52  
 dnnl::prop\_kind::undef (C++ enumerator), 52  
 dnnl::reorder (C++ struct), 149  
 dnnl::reorder::execute (C++ function), 149  
 dnnl::reorder::execute\_sycl (C++ function), 149  
 dnnl::reorder::primitive\_desc (C++ struct), 150  
 dnnl::reorder::primitive\_desc::dst\_desc (C++ function), 151

```

dnnl::reorder::primitive_desc::get_dst_edgmhe:rnn_primitive_desc_base::bias_desc
    (C++ function), 150
    (C++ function), 50
dnnl::reorder::primitive_desc::get_src_edgmhe:rnn_primitive_desc_base::diff_bias_desc
    (C++ function), 150
    (C++ function), 51
dnnl::reorder::primitive_desc::primitiveddes:rnn_primitive_desc_base::diff_dst_iter_c_desc
    (C++ function), 150
    (C++ function), 52
dnnl::reorder::primitive_desc::src_desc dnnl::rnn_primitive_desc_base::diff_dst_iter_desc
    (C++ function), 150
    (C++ function), 52
dnnl::reorder::reorder (C++ function), 149
dnnl::resampling_backward (C++ struct), 155
dnnl::resampling_backward::desc (C++ struct), 155
dnnl::resampling_backward::desc::desc dnnl::rnn_primitive_desc_base::diff_dst_layer_desc
    (C++ function), 156
    (C++ function), 51
dnnl::resampling_backward::desc::desc dnnl::rnn_primitive_desc_base::diff_src_iter_c_desc
    (C++ function), 156
    (C++ function), 51
dnnl::resampling_backward::desc::desc dnnl::rnn_primitive_desc_base::diff_src_layer_desc
    (C++ struct), 156
    (C++ function), 51
dnnl::resampling_backward::desc::desc dnnl::rnn_primitive_desc_base::diff_weights_iter_desc
    (C++ function), 157
    (C++ function), 51
dnnl::resampling_backward::desc::desc dnnl::rnn_primitive_desc_base::diff_weights_layer_desc
    (C++ function), 157
    (C++ function), 51
dnnl::resampling_backward::desc::desc dnnl::rnn_primitive_desc_base::diff_weights_peephole_desc
    (C++ function), 156
    (C++ function), 51
dnnl::resampling_backward::resampling_backward rnn_primitive_desc_base::diff_weights_projective_desc
    (C++ function), 155
    (C++ function), 51
dnnl::resampling_forward (C++ struct), 153
dnnl::resampling_forward::desc (C++ struct), 153
dnnl::resampling_forward::desc::desc dnnl::rnn_primitive_desc_base::dst_iter_c_desc
    (C++ function), 154
    (C++ function), 51
dnnl::resampling_forward::desc::desc dnnl::rnn_primitive_desc_base::dst_iter_desc
    (C++ function), 51
dnnl::resampling_forward::desc::desc dnnl::rnn_primitive_desc_base::dst_layer_desc
    (C++ function), 51
dnnl::resampling_forward::primitive_desc (C++ struct), 154
dnnl::resampling_forward::primitive_desc::dst_desc dnnl::rnn_primitive_desc_base::rnn_primitive_desc_base::dst_desc
    (C++ function), 155
    (C++ function), 50
dnnl::resampling_forward::primitive_desc::dst_desc dnnl::rnn_primitive_desc_base::src_iter_c_desc
    (C++ function), 155
    (C++ function), 50
dnnl::resampling_forward::primitive_desc::src_desc dnnl::rnn_primitive_desc_base::src_iter_desc
    (C++ function), 155
    (C++ function), 50
dnnl::resampling_forward::primitive_desc::src_desc dnnl::rnn_primitive_desc_base::src_layer_desc
    (C++ function), 155
    (C++ function), 50
dnnl::resampling_forward::resampling_forward dnnl::rnn_primitive_desc_base::weights_iter_desc
    (C++ function), 153
    (C++ function), 50
dnnl::rnn_direction (C++ enum), 163
dnnl::rnn_direction::bidirectional_concat dnnl::rnn_primitive_desc_base::weights_layer_desc
    (C++ enumerator), 163
    (C++ function), 50
dnnl::rnn_direction::bidirectional_sum dnnl::rnn_primitive_desc_base::weights_peephole_desc
    (C++ enumerator), 163
    (C++ function), 50
dnnl::rnn_direction::unidirectional (C++ enumerator), 163
dnnl::rnn_primitive_desc_base::weights_projection_desc
    (C++ function), 50
dnnl::rnn_direction::unidirectional_left dnnl::scratchpad_mode (C++ enum), 63
    (C++ enumerator), 163
    (C++ function), 50
dnnl::rnn_direction::unidirectional_right dnnl::scratchpad_mode::library (C++ enumerator), 63
    (C++ enumerator), 163
    (C++ function), 50
dnnl::rnn_flags (C++ enum), 163
dnnl::rnn_flags::undef (C++ enumerator), 163
dnnl::rnn_primitive_desc_base (C++ struct), 50
    (C++ function), 191
    (C++ function), 192

```

dnnl::shuffle\_backward::desc::desc (C++ function), 192  
dnnl::shuffle\_backward::primitive\_desc (C++ struct), 192  
dnnl::shuffle\_backward::primitive\_desc::desc (C++ function), 192  
dnnl::shuffle\_backward::primitive\_desc::diff\_desc (C++ function), 194  
dnnl::shuffle\_backward::primitive\_desc::dst\_desc (C++ function), 194  
dnnl::shuffle\_backward::primitive\_desc::flags (C++ enum), 25  
dnnl::shuffle\_backward::primitive\_desc::primitivemanager (C++ enumerator), 25  
dnnl::shuffle\_backward::primitive\_desc::default\_order (C++ enumerator), 25  
dnnl::shuffle\_backward::stream (C++ struct), 24  
dnnl::shuffle\_backward::shuffle\_backward (C++ function), 192  
dnnl::shuffle\_forward (C++ struct), 190  
dnnl::shuffle\_forward::desc (C++ struct), 191  
dnnl::shuffle\_forward::desc::desc (C++ function), 191  
dnnl::shuffle\_forward::primitive\_desc (C++ struct), 191  
dnnl::shuffle\_forward::primitive\_desc::desc (C++ function), 191  
dnnl::shuffle\_forward::primitive\_desc::primitivemanager (C++ function), 198  
dnnl::shuffle\_forward::sum (C++ struct), 198  
dnnl::shuffle\_forward::sum::primitive\_desc (C++ struct), 198  
dnnl::sum::dst\_desc (C++ function), 199  
dnnl::sum::primitive\_desc (C++ function), 198, 199  
dnnl::sum::src\_desc (C++ function), 199  
dnnl::sum::sum (C++ function), 198  
dnnl::vanilla\_rnn\_backward (C++ struct), 166  
dnnl::vanilla\_rnn\_backward::desc (C++ struct), 166  
dnnl::vanilla\_rnn\_backward::desc (C++ function), 166  
dnnl::vanilla\_rnn\_backward::primitive\_desc (C++ struct), 167  
dnnl::vanilla\_rnn\_backward::primitive\_desc::bias\_desc (C++ function), 168  
dnnl::vanilla\_rnn\_backward::primitive\_desc::diff\_bias\_desc (C++ function), 168  
dnnl::vanilla\_rnn\_backward::primitive\_desc::diff\_desc (C++ function), 168  
dnnl::vanilla\_rnn\_backward::primitive\_desc::dst\_desc (C++ function), 168  
dnnl::vanilla\_rnn\_backward::primitive\_desc::flags (C++ enum), 25  
dnnl::vanilla\_rnn\_backward::primitive\_desc::primitivemanager (C++ enumerator), 25  
dnnl::vanilla\_rnn\_backward::primitive\_desc::default\_order (C++ enumerator), 25  
dnnl::vanilla\_rnn\_backward::stream (C++ struct), 24  
dnnl::vanilla\_rnn\_backward::shuffle\_backward (C++ function), 192  
dnnl::vanilla\_rnn\_backward::softmax\_backward (C++ function), 194  
dnnl::vanilla\_rnn\_backward::softmax\_desc (C++ struct), 194  
dnnl::vanilla\_rnn\_backward::softmax\_desc::desc (C++ function), 194  
dnnl::vanilla\_rnn\_backward::softmax\_desc::dst\_desc (C++ function), 194  
dnnl::vanilla\_rnn\_backward::softmax\_desc::flags (C++ enum), 25  
dnnl::vanilla\_rnn\_backward::softmax\_desc::primitivemanager (C++ enumerator), 25  
dnnl::vanilla\_rnn\_backward::softmax\_desc::default\_order (C++ enumerator), 25  
dnnl::vanilla\_rnn\_backward::stream (C++ struct), 24  
dnnl::vanilla\_rnn\_backward::shuffle\_backward (C++ function), 192  
dnnl::vanilla\_rnn\_backward::softmax\_backward (C++ function), 194  
dnnl::vanilla\_rnn\_backward::softmax\_desc (C++ struct), 194  
dnnl::vanilla\_rnn\_backward::softmax\_desc::desc (C++ function), 194  
dnnl::vanilla\_rnn\_backward::softmax\_desc::dst\_desc (C++ function), 194  
dnnl::vanilla\_rnn\_backward::softmax\_desc::flags (C++ enum), 25  
dnnl::vanilla\_rnn\_backward::softmax\_desc::primitivemanager (C++ enumerator), 25  
dnnl::vanilla\_rnn\_backward::softmax\_desc::default\_order (C++ enumerator), 25  
dnnl::vanilla\_rnn\_backward::stream (C++ struct), 24

dnnl::vanilla\_rnn\_backward::primitive\_de~~NNLdARGIdFFFdS~~<sub>C</sub>\_ITER\_C (C macro), 57  
(C++ function), 168  
DNNL\_ARG\_DIFF\_DST\_LAYER (C macro), 57  
dnnl::vanilla\_rnn\_backward::primitive\_de~~NNLdARGlAyeF\_d6A6E~~<sub>C</sub>\_SHIFT (C macro), 57  
(C++ function), 168  
DNNL\_ARG\_DIFF\_SRC (C macro), 57  
dnnl::vanilla\_rnn\_backward::primitive\_de~~NNLpARGidFIVE\_dR6c0~~<sub>C</sub>, 56  
(C++ function), 167  
DNNL\_ARG\_DIFF\_SRC\_1 (C macro), 57  
dnnl::vanilla\_rnn\_backward::primitive\_de~~NNLsARGIdFFFdS~~<sub>C</sub>\_2 (C macro), 57  
(C++ function), 167  
DNNL\_ARG\_DIFF\_SRC\_ITER (C macro), 57  
dnnl::vanilla\_rnn\_backward::primitive\_de~~NNLsARGlAyeF\_dR6c~~<sub>C</sub>\_ITER\_C (C macro), 57  
(C++ function), 167  
DNNL\_ARG\_DIFF\_SRC\_LAYER (C macro), 57  
dnnl::vanilla\_rnn\_backward::primitive\_de~~NNLwARGhdSFT\_wEIGHeS~~<sub>C</sub> (C macro), 57  
(C++ function), 168  
DNNL\_ARG\_DIFF\_WEIGHTS\_0 (C macro), 57  
dnnl::vanilla\_rnn\_backward::primitive\_de~~NNLwARGhdSFT\_wEIGHeSc1~~<sub>C</sub> (C macro), 57  
(C++ function), 168  
DNNL\_ARG\_DIFF\_WEIGHTS\_ITER (C macro), 58  
dnnl::vanilla\_rnn\_backward::primitive\_de~~NNLwARSpAEE\_wEIGHeS~~<sub>C</sub> (C macro), 57  
(C++ function), 168  
DNNL\_ARG\_DST (C macro), 56  
dnnl::vanilla\_rnn\_backward::vanilla\_rnn\_DNNLARG\_DST\_0 (C macro), 56  
(C++ function), 166  
DNNL\_ARG\_DST\_1 (C macro), 56  
dnnl::vanilla\_rnn\_forward (C++ struct), 163  
dnnl::vanilla\_rnn\_forward::desc (C++ struct), 164  
dnnl::vanilla\_rnn\_forward::desc::desc (C++ function), 164  
dnnl::vanilla\_rnn\_forward::desc::desc (C++ function), 164  
dnnl::vanilla\_rnn\_forward::primitive\_des~~NNLARG\_MEAN~~<sub>C</sub> (C macro), 56  
(C++ struct), 164  
DNNL\_ARG\_MULTIPLE\_DST (C macro), 58  
dnnl::vanilla\_rnn\_forward::primitive\_des~~NNLbiARGdMULTIPLE\_SRC~~<sub>C</sub> (C macro), 58  
(C++ function), 165  
DNNL\_ARG\_SCALE\_SHIFT (C macro), 56  
dnnl::vanilla\_rnn\_forward::primitive\_des~~NNLdsARGtSRCd6CHPAD~~<sub>C</sub> (C macro), 56  
(C++ function), 165  
DNNL\_ARG\_SRC (C macro), 55  
dnnl::vanilla\_rnn\_forward::primitive\_des~~NNLdsARGsSRC\_d6C~~<sub>C</sub> (C macro), 55  
(C++ function), 165  
DNNL\_ARG\_SRC\_1 (C macro), 55  
dnnl::vanilla\_rnn\_forward::primitive\_des~~NNLdpARGtSRC\_d6C~~<sub>C</sub> (C macro), 55  
(C++ function), 165  
DNNL\_ARG\_SRC\_ITER (C macro), 55  
dnnl::vanilla\_rnn\_forward::primitive\_des~~NNLrARGtSRCd6TER~~<sub>C</sub> (C macro), 55  
(C++ function), 165  
DNNL\_ARG\_SRC\_LAYER (C macro), 55  
dnnl::vanilla\_rnn\_forward::primitive\_des~~NNLrARGtay0r~~<sub>C</sub> (C macro), 56  
(C++ function), 165  
DNNL\_ARG\_VARIANCE (C macro), 56  
dnnl::vanilla\_rnn\_forward::primitive\_des~~NNLwARGtWEIGHTS\_d6C~~<sub>C</sub> (C macro), 56  
(C++ function), 165  
DNNL\_ARG\_WEIGHTS\_0 (C macro), 56  
dnnl::vanilla\_rnn\_forward::primitive\_des~~NNLwARGtWEIGHTS\_d6C~~<sub>C</sub> (C macro), 56  
(C++ function), 165  
DNNL\_ARG\_WEIGHTS\_ITER (C macro), 56  
dnnl::vanilla\_rnn\_forward::primitive\_des~~NNLwARGSpAEE\_d6CHES~~<sub>C</sub> (C macro), 56  
(C++ function), 165  
DNNL\_ARG\_WORKSPACE (C macro), 56  
dnnl::vanilla\_rnn\_forward::vanilla\_rnn\_f~~NNLwRMEMORY\_ALLOCATE~~<sub>C</sub> (C macro), 42  
(C++ function), 164  
DNNL\_MEMORY\_NONE (C macro), 42  
DNNL\_ATTR\_OUTPUT\_SCALES (C macro), 58  
DNNL\_ATTR\_ZERO\_POINTS (C macro), 58  
DNNL\_ARG\_BIAS (C macro), 56  
DNNL\_ARG\_DIFF\_BIAS (C macro), 58  
DNNL\_ARG\_DIFF\_DST (C macro), 57  
DNNL\_ARG\_DIFF\_DST\_0 (C macro), 57  
DNNL\_ARG\_DIFF\_DST\_1 (C macro), 57  
DNNL\_ARG\_DIFF\_DST\_2 (C macro), 57  
DNNL\_ARG\_DIFF\_DST\_ITER (C macro), 57  
DNNL\_RUNTIME\_DIM\_VAL (C macro), 58  
DNNL\_RUNTIME\_F32\_VAL (C macro), 58  
DNNL\_RUNTIME\_S32\_VAL (C macro), 58  
DNNL\_RUNTIME\_SIZE\_VAL (C macro), 58  
do\_allocate (C++ function), 633  
do\_deallocate (C++ function), 633  
do\_is\_equal (C++ function), 633  
DPC++, 230  
DRM, 710

DXVA2, **710**

## E

empty (*C++ function*), 316, 624  
 ENCODE, **648**  
 end (*C++ function*), 316, 625, 627  
 enqueue (*C++ function*), 385  
 erf (*C++ function*), 1314  
 erfc (*C++ function*), 1317  
 erfcinv (*C++ function*), 1326  
 erfinv (*C++ function*), 1323  
 ets\_key\_usage\_type::ets\_key\_per\_instance  
     (*C++ enum*), 626  
 ets\_key\_usage\_type::ets\_no\_key         (*C++ enum*), 626  
 ets\_key\_usage\_type::ets\_suspend\_aware  
     (*C++ enum*), 626  
 exception\_thrown (*C++ function*), 327  
 execute (*C++ function*), 386  
 exp (*C++ function*), 1249  
 exp10 (*C++ function*), 1253  
 exp2 (*C++ function*), 1251  
 expint1 (*C++ function*), 1335  
 expml (*C++ function*), 1255

## F

fdim (*C++ function*), 1364  
 Feature, **228**  
 Feature vector, **228**  
 filter (*C++ function*), 312  
 Flat data, **229**  
 flatten2d (*C++ function*), 628  
 flattened2d (*C++ function*), 627, 628  
 floor (*C++ function*), 1337  
 fmax (*C++ function*), 1366  
 fmin (*C++ function*), 1367  
 fmod (*C++ function*), 1222  
 frac (*C++ function*), 1349  
 Func::~Func (*C++ function*), 292  
 Func::Func (*C++ function*), 289, 291  
 Func::operator () (*C++ function*), 280, 289, 292

## G

generate (*C++ function*), 1151  
 get\_mode (*C++ function*), 1352  
 get\_status (*C++ function*), 1355  
 Getter, **229**  
 getValue (*C++ function*), 1141  
 global\_control (*C++ function*), 380  
 GOP, **710**  
 GPB, **710**  
 grainsize (*C++ function*), 316  
 graph (*C++ function*), 327

## H

H.264, **710**  
 H::~H (*C++ function*), 288  
 H::equal (*C++ function*), 288  
 H::H (*C++ function*), 288  
 H::hash (*C++ function*), 288  
 HDR, **710**  
 Heterogeneous data, **230**  
 Homogeneous data, **230**  
 Host/Device, **230**  
 HRD, **710**  
 hypot (*C++ function*), 1247  
 I

I010, **710**  
 IDR, **710**  
 iGPU, **710**  
 Immutability, **230**  
 indexer\_node (*C++ function*), 366  
 Inference, **228**  
 Inference set, **228**  
 init (*C++ function*), 1139  
 initialize (*C++ function*), 385  
 input\_node (*C++ function*), 336  
 input\_ports (*C++ function*), 366, 368  
 Interval feature, **228**  
 inv (*C++ function*), 1226  
 invcbrt (*C++ function*), 1235  
 invsqrt (*C++ function*), 1232  
 is\_a (*C++ function*), 373  
 is\_active (*C++ function*), 385  
 is\_canceling (*C++ function*), 383  
 is\_cancelled (*C++ function*), 327  
 is\_current\_task\_group\_canceling     (*C++ function*), 383  
 is\_divisible (*C++ function*), 316  
 is\_final\_scan (*C++ function*), 304  
 is\_group\_execution\_cancelled     (*C++ function*), 379  
 is\_observing (*C++ function*), 388  
 is\_valid (*C++ function*), 348, 350  
 isolate (*C++ function*), 387  
 IYUV, **710**

## J

JIT, **230**

## K

Kernel, **230**  
 kind\_t::bound (*C++ enum*), 378  
 kind\_t::isolated (*C++ enum*), 378

## L

LA, **711**

- Label, [228](#)  
*leapfrog (C++ function)*, [1164](#)  
*left (C++ function)*, [325](#)  
*lgamma (C++ function)*, [1332](#)  
*limiter\_node (C++ function)*, [358](#)  
*linearfrac (C++ function)*, [1220](#)  
*ln (C++ function)*, [1256](#)  
*local (C++ function)*, [619](#), [624](#)  
*lock (C++ function)*, [638](#), [639](#), [644](#), [645](#)  
*lock\_shared (C++ function)*, [639](#), [645](#)  
*log10 (C++ function)*, [1260](#)  
*log1p (C++ function)*, [1262](#)  
*log2 (C++ function)*, [1258](#)  
*logb (C++ function)*, [1264](#)
- M**
- `M::~scoped_lock (C++ function)`, [285](#)  
`M::is_fair_mutex (C++ member)`, [285](#), [287](#)  
`M::is_recursive_mutex (C++ member)`, [285](#), [287](#)  
`M::is_rw_mutex (C++ member)`, [285](#), [287](#)  
`M::scoped_lock (C++ function)`, [285](#)  
`M::scoped_lock (C++ type)`, [285](#)  
`M::scoped_lock::acquire (C++ function)`, [285](#)  
`M::scoped_lock::release (C++ function)`, [285](#)  
`M::scoped_lock::try_acquire (C++ function)`, [285](#)  
`make_filter (C++ function)`, [312](#)  
`max_concurrency (C++ function)`, [385](#), [387](#)  
`max_size (C++ function)`, [632](#)  
`maxmag (C++ function)`, [1369](#)  
`MCTF`, [711](#)  
`Metadata`, [230](#)  
`MFX_ANGLE_0 (C++ enumerator)`, [758](#)  
`MFX_ANGLE_180 (C++ enumerator)`, [758](#)  
`MFX_ANGLE_270 (C++ enumerator)`, [758](#)  
`MFX_ANGLE_90 (C++ enumerator)`, [758](#)  
`MFX_B_REF_OFF (C++ enumerator)`, [741](#)  
`MFX_B_REF_PYRAMID (C++ enumerator)`, [741](#)  
`MFX_B_REF_UNKNOWN (C++ enumerator)`, [741](#)  
`MFX_BITSTREAM_COMPLETE_FRAME (C++ enumerator)`, [745](#)  
`MFX_BITSTREAM_EOS (C++ enumerator)`, [745](#)  
`MFX_BLOCKSIZE_MIN_16X16 (C++ enumerator)`, [744](#)  
`MFX_BLOCKSIZE_MIN_4X4 (C++ enumerator)`, [744](#)  
`MFX_BLOCKSIZE_MIN_8X8 (C++ enumerator)`, [744](#)  
`MFX_BLOCKSIZE_UNKNOWN (C++ enumerator)`, [744](#)  
`MFX_BPSET_DEFAULT (C++ enumerator)`, [742](#)  
`MFX_BPSET_IFRAME (C++ enumerator)`, [742](#)  
`MFX_BRC_BIG_FRAME (C++ enumerator)`, [761](#)  
`MFX_BRC_OK (C++ enumerator)`, [761](#)  
`MFX_BRC_PANIC_BIG_FRAME (C++ enumerator)`, [761](#)  
`MFX_BRC_PANIC_SMALL_FRAME (C++ enumerator)`, [761](#)  
`MFX_BRC_SMALL_FRAME (C++ enumerator)`, [761](#)  
`MFX_CHROMA_SITING_HORIZONTAL_CENTER (C++ enumerator)`, [759](#)  
`MFX_CHROMA_SITING_HORIZONTAL_LEFT (C++ enumerator)`, [759](#)  
`MFX_CHROMA_SITING_UNKNOWN (C++ enumerator)`, [759](#)  
`MFX_CHROMA_SITING_VERTICAL_BOTTOM (C++ enumerator)`, [759](#)  
`MFX_CHROMA_SITING_VERTICAL_CENTER (C++ enumerator)`, [759](#)  
`MFX_CHROMA_SITING_VERTICAL_TOP (C++ enumerator)`, [759](#)  
`MFX_CHROMAFORMAT_JPEG_SAMPLING (C++ enumerator)`, [732](#)  
`MFX_CHROMAFORMAT_MONOCHROME (C++ enumerator)`, [732](#)  
`MFX_CHROMAFORMAT_RESERVED1 (C++ enumerator)`, [732](#)  
`MFX_CHROMAFORMAT_YUV400 (C++ enumerator)`, [732](#)  
`MFX_CHROMAFORMAT_YUV411 (C++ enumerator)`, [732](#)  
`MFX_CHROMAFORMAT_YUV420 (C++ enumerator)`, [732](#)  
`MFX_CHROMAFORMAT_YUV422 (C++ enumerator)`, [732](#)  
`MFX_CHROMAFORMAT_YUV422H (C++ enumerator)`, [732](#)  
`MFX_CHROMAFORMAT_YUV422V (C++ enumerator)`, [732](#)  
`MFX_CHROMAFORMAT_YUV444 (C++ enumerator)`, [732](#)  
`MFX_CODEC_AV1 (C++ enumerator)`, [735](#)  
`MFX_CODEC_AVC (C++ enumerator)`, [735](#)  
`MFX_CODEC_HEVC (C++ enumerator)`, [735](#)  
`MFX_CODEC_JPEG (C++ enumerator)`, [735](#)  
`MFX_CODEC_MPEG2 (C++ enumerator)`, [735](#)  
`MFX_CODEC_VC1 (C++ enumerator)`, [735](#)  
`MFX_CODEC_VP9 (C++ enumerator)`, [735](#)  
`MFX_CODINGOPTION_ADAPTIVE (C++ enumerator)`, [745](#)  
`MFX_CODINGOPTION_OFF (C++ enumerator)`, [745](#)  
`MFX_CODINGOPTION_ON (C++ enumerator)`, [745](#)  
`MFX_CODINGOPTION_UNKNOWN (C++ enumerator)`, [745](#)  
`MFX_CONTENT_FULL_SCREEN_VIDEO (C++ enumerator)`, [744](#)  
`MFX_CONTENT_NON_VIDEO_SCREEN (C++ enumerator)`, [744](#)  
`MFX_CONTENT_UNKNOWN (C++ enumerator)`, [744](#)  
`MFX_CORRUPTION_ABSENT_BOTTOM_FIELD (C++`

MFX_CORRUPTION_ABSENT_TOP_FIELD ( <i>C++ enumerator</i> ), 734	( <i>C++ enumerator</i> ), 734	MFX_EXTBUFF_CENC_PARAM ( <i>C++ enumerator</i> ), 749
MFX_CORRUPTION_MAJOR ( <i>C++ enumerator</i> ), 734		MFX_EXTBUFF_CHROMA_LOC_INFO ( <i>C++ enumerator</i> ), 747
MFX_CORRUPTION_MINOR ( <i>C++ enumerator</i> ), 734		MFX_EXTBUFF_CODING_OPTION ( <i>C++ enumerator</i> ), 745
MFX_CORRUPTION_REFERENCE_FRAME ( <i>C++ enumerator</i> ), 734		MFX_EXTBUFF_CODING_OPTION2 ( <i>C++ enumerator</i> ), 746
MFX_CORRUPTION_REFERENCE_LIST ( <i>C++ enumerator</i> ), 734		MFX_EXTBUFF_CODING_OPTION3 ( <i>C++ enumerator</i> ), 747
MFX_DECODERDESCRIPTION_VERSION ( <i>C macro</i> ), 841		MFX_EXTBUFF_CODING_OPTION_SPSPPS ( <i>C++ enumerator</i> ), 745
MFX_DEINTERLACING_24FPS_OUT ( <i>C++ enumerator</i> ), 755		MFX_EXTBUFF_CODING_OPTION_VPS ( <i>C++ enumerator</i> ), 748
MFX_DEINTERLACING_30FPS_OUT ( <i>C++ enumerator</i> ), 755		MFX_EXTBUFF_CONTENT_LIGHT_LEVEL_INFO ( <i>C++ enumerator</i> ), 748
MFX_DEINTERLACING_ADVANCED ( <i>C++ enumerator</i> ), 755		MFX_EXTBUFF_DEC_VIDEO_PROCESSING ( <i>C++ enumerator</i> ), 747
MFX_DEINTERLACING_ADVANCED_NOREF ( <i>C++ enumerator</i> ), 755		MFX_EXTBUFF_DECODE_ERROR_REPORT ( <i>C++ enumerator</i> ), 748
MFX_DEINTERLACING_ADVANCED_SCD ( <i>C++ enumerator</i> ), 755		MFX_EXTBUFF_DECODED_FRAME_INFO ( <i>C++ enumerator</i> ), 748
MFX_DEINTERLACING_AUTO_DOUBLE ( <i>C++ enumerator</i> ), 755		MFX_EXTBUFF_DIRTY_RECTANGLES ( <i>C++ enumerator</i> ), 748
MFX_DEINTERLACING_AUTO_SINGLE ( <i>C++ enumerator</i> ), 755		MFX_EXTBUFF_ENCODED_FRAME_INFO ( <i>C++ enumerator</i> ), 747
MFX_DEINTERLACING_BOB ( <i>C++ enumerator</i> ), 755		MFX_EXTBUFF_ENCODED_SLICES_INFO ( <i>C++ enumerator</i> ), 748
MFX_DEINTERLACING_DETECT_INTERLACE ( <i>C++ enumerator</i> ), 755		MFX_EXTBUFF_ENCODED_UNITS_INFO ( <i>C++ enumerator</i> ), 749
MFX_DEINTERLACING_FIELD_WEAVING ( <i>C++ enumerator</i> ), 755		MFX_EXTBUFF_ENCODER_CAPABILITY ( <i>C++ enumerator</i> ), 746
MFX_DEINTERLACING_FIXED_TELECINE_PATTERN ( <i>C++ enumerator</i> ), 755		MFX_EXTBUFF_ENCODER_IPCM_AREA ( <i>C++ enumerator</i> ), 749
MFX_DEINTERLACING_FULL_FR_OUT ( <i>C++ enumerator</i> ), 755		MFX_EXTBUFF_ENCODER_RESET_OPTION ( <i>C++ enumerator</i> ), 746
MFX_DEINTERLACING_HALF_FR_OUT ( <i>C++ enumerator</i> ), 755		MFX_EXTBUFF_ENCODER_ROI ( <i>C++ enumerator</i> ), 747
MFX_DEVICEDESCRIPTION_VERSION ( <i>C macro</i> ), 841		MFX_EXTBUFF_HEVC_PARAM ( <i>C++ enumerator</i> ), 747
MFX_ENCODERDESCRIPTION_VERSION ( <i>C macro</i> ), 841		MFX_EXTBUFF_HEVC_REFLIST_CTRL ( <i>C++ enumerator</i> ), 748
MFX_ERROR_FRAME_GAP ( <i>C++ enumerator</i> ), 757		MFX_EXTBUFF_HEVC_REFLISTS ( <i>C++ enumerator</i> ), 748
MFX_ERROR_PPS ( <i>C++ enumerator</i> ), 757		MFX_EXTBUFF_HEVC_REGION ( <i>C++ enumerator</i> ), 748
MFX_ERROR_SLICEDATA ( <i>C++ enumerator</i> ), 757		MFX_EXTBUFF_HEVC_TEMPORAL_LAYERS ( <i>C++ enumerator</i> ), 748
MFX_ERROR_SLICEHEADER ( <i>C++ enumerator</i> ), 757		MFX_EXTBUFF_HEVC_TILES ( <i>C++ enumerator</i> ), 747
MFX_ERROR_SPS ( <i>C++ enumerator</i> ), 757		MFX_EXTBUFF_INSERT_HEADERS ( <i>C++ enumerator</i> ), 749
MFX_EXTBUFF_AVC_REFLIST_CTRL ( <i>C++ enumerator</i> ), 746		MFX_EXTBUFF_JPEG_HUFFMAN ( <i>C++ enumerator</i> ), 749
MFX_EXTBUFF_AVC_REFLISTS ( <i>C++ enumerator</i> ), 747		MFX_EXTBUFF_JPEG_QT ( <i>C++ enumerator</i> ), 749
MFX_EXTBUFF_AVC_ROUNDING_OFFSET ( <i>C++ enumerator</i> ), 749		MFX_EXTBUFF_MASTERING_DISPLAY_COLOUR_VOLUME ( <i>C++ enumerator</i> ), 749
MFX_EXTBUFF_AVC_TEMPORAL_LAYERS ( <i>C++ enumerator</i> ), 746		
MFX_EXTBUFF_BRC ( <i>C++ enumerator</i> ), 749		

MFX\_EXTBUFF\_MB\_DISABLE\_SKIP\_MAP (C++ enumerator), 747

MFX\_EXTBUFF\_MB\_FORCE\_INTRA (C++ enumerator), 747

MFX\_EXTBUFF\_MBQP (C++ enumerator), 747

MFX\_EXTBUFF\_MOVING\_RECTANGLES (C++ enumerator), 748

MFX\_EXTBUFF\_MULTI\_FRAME\_CONTROL (C++ enumerator), 749

MFX\_EXTBUFF\_MULTI\_FRAME\_PARAM (C++ enumerator), 749

MFX\_EXTBUFF\_MV\_OVER\_PIC\_BOUNDARIES (C++ enumerator), 748

MFX\_EXTBUFF\_MVC\_SEQ\_DESC (C++ enumerator), 749

MFX\_EXTBUFF\_MVC\_TARGET\_VIEWS (C++ enumerator), 749

MFX\_EXTBUFF\_PARTIAL\_BITSTREAM\_PARAM (C++ enumerator), 749

MFX\_EXTBUFF\_PICTURE\_TIMING\_SEI (C++ enumerator), 746

MFX\_EXTBUFF\_PRED\_WEIGHT\_TABLE (C++ enumerator), 748

MFX\_EXTBUFF\_THREADS\_PARAM (C++ enumerator), 745

MFX\_EXTBUFF\_TIME\_CODE (C++ enumerator), 748

MFX\_EXTBUFF\_VIDEO\_SIGNAL\_INFO (C++ enumerator), 746

MFX\_EXTBUFF\_vp8\_CODING\_OPTION (C++ enumerator), 749

MFX\_EXTBUFF\_vp9\_PARAM (C++ enumerator), 749

MFX\_EXTBUFF\_vp9\_SEGMENTATION (C++ enumerator), 749

MFX\_EXTBUFF\_vp9\_TEMPORAL\_LAYERS (C++ enumerator), 749

MFX\_EXTBUFF\_VPP\_AUXDATA (C++ enumerator), 745

MFX\_EXTBUFF\_VPP\_COLOR\_CONVERSION (C++ enumerator), 748

MFX\_EXTBUFF\_VPP\_COLORFILL (C++ enumerator), 748

MFX\_EXTBUFF\_VPP\_COMPOSITE (C++ enumerator), 747

MFX\_EXTBUFF\_VPP\_DEINTERLACING (C++ enumerator), 747

MFX\_EXTBUFF\_VPP\_DENOISE (C++ enumerator), 745

MFX\_EXTBUFF\_VPP\_DETAIL (C++ enumerator), 746

MFX\_EXTBUFF\_VPP\_DONOTUSE (C++ enumerator), 745

MFX\_EXTBUFF\_VPP\_DOUSE (C++ enumerator), 746

MFX\_EXTBUFF\_VPP\_FIELD\_PROCESSING (C++ enumerator), 747

MFX\_EXTBUFF\_VPP\_FRAME\_RATE\_CONVERSION (C++ enumerator), 746

MFX\_EXTBUFF\_VPP\_IMAGE\_STABILIZATION (C++ enumerator), 746

MFX\_EXTBUFF\_VPP\_MCTF (C++ enumerator), 749

MFX\_EXTBUFF\_VPP\_MIRRORING (C++ enumerator), 748

MFX\_EXTBUFF\_VPP\_PROCAMP (C++ enumerator), 746

MFX\_EXTBUFF\_VPP\_ROTATION (C++ enumerator), 748

MFX\_EXTBUFF\_VPP\_SCALING (C++ enumerator), 748

MFX\_EXTBUFF\_VPP\_SCENE\_ANALYSIS (C++ enumerator), 746

MFX\_EXTBUFF\_VPP\_VIDEO\_SIGNAL\_INFO (C++ enumerator), 747

MFX\_FOURCC\_A2RGB10 (C++ enumerator), 731

MFX\_FOURCC\_ABGR16 (C++ enumerator), 731

MFX\_FOURCC\_ARGB16 (C++ enumerator), 731

MFX\_FOURCC\_AYUV (C++ enumerator), 731

MFX\_FOURCC\_AYUV\_RGB4 (C++ enumerator), 732

MFX\_FOURCC\_BGR4 (C++ enumerator), 731

MFX\_FOURCC\_BGRA (C++ enumerator), 731

MFX\_FOURCC\_I010 (C++ enumerator), 731

MFX\_FOURCC\_I420 (C++ enumerator), 730

MFX\_FOURCC\_IYUV (C++ enumerator), 730

MFX\_FOURCC\_NV12 (C++ enumerator), 730

MFX\_FOURCC\_NV16 (C++ enumerator), 730

MFX\_FOURCC\_NV21 (C++ enumerator), 730

MFX\_FOURCC\_P010 (C++ enumerator), 731

MFX\_FOURCC\_P016 (C++ enumerator), 731

MFX\_FOURCC\_P210 (C++ enumerator), 731

MFX\_FOURCC\_P8 (C++ enumerator), 731

MFX\_FOURCC\_P8\_TEXTURE (C++ enumerator), 731

MFX\_FOURCC\_R16 (C++ enumerator), 731

MFX\_FOURCC\_RGB4 (C++ enumerator), 731

MFX\_FOURCC\_RGB565 (C++ enumerator), 730

MFX\_FOURCC\_RGBP (C++ enumerator), 730

MFX\_FOURCC\_UYVY (C++ enumerator), 732

MFX\_FOURCC\_Y210 (C++ enumerator), 732

MFX\_FOURCC\_Y216 (C++ enumerator), 732

MFX\_FOURCC\_Y410 (C++ enumerator), 732

MFX\_FOURCC\_Y416 (C++ enumerator), 732

MFX\_FOURCC\_YUY2 (C++ enumerator), 730

MFX\_FOURCC\_YV12 (C++ enumerator), 730

MFX\_FRAMEDATA\_ORIGINAL\_TIMESTAMP (C++ enumerator), 733

MFX\_FRAMEORDER\_UNKNOWN (C++ enumerator), 733

MFX\_FRAMESURFACE1\_VERSION (C macro), 841

MFX\_FRAMESURFACEINTERFACE\_VERSION (C macro), 841

MFX\_FRAMETYPE\_B (C++ enumerator), 751

MFX\_FRAMETYPE\_I (C++ enumerator), 751

MFX\_FRAMETYPE\_IDR (C++ enumerator), 751

MFX\_FRAMETYPE\_P (*C++ enumerator*), 751  
MFX\_FRAMETYPE\_REF (*C++ enumerator*), 751  
MFX\_FRAMETYPE\_S (*C++ enumerator*), 751  
MFX\_FRAMETYPE\_UNKNOWN (*C++ enumerator*), 751  
MFX\_FRAMETYPE\_xB (*C++ enumerator*), 751  
MFX\_FRAMETYPE\_xI (*C++ enumerator*), 751  
MFX\_FRAMETYPE\_xIDR (*C++ enumerator*), 751  
MFX\_FRAMETYPE\_xP (*C++ enumerator*), 751  
MFX\_FRAMETYPE\_xREF (*C++ enumerator*), 751  
MFX\_FRAMETYPE\_xs (*C++ enumerator*), 751  
MFX\_FRCALGM\_DISTRIBUTED\_TIMESTAMP (*C++ enumerator*), 753  
MFX\_FRCALGM\_FRAME\_INTERPOLATION (*C++ enumerator*), 753  
MFX\_FRCALGM\_PRESERVE\_TIMESTAMP (*C++ enumerator*), 753  
MFX\_GOP\_CLOSED (*C++ enumerator*), 739  
MFX\_GOP\_STRICT (*C++ enumerator*), 739  
MFX\_GPUCOPY\_DEFAULT (*C++ enumerator*), 728  
MFX\_GPUCOPY\_OFF (*C++ enumerator*), 728  
MFX\_GPUCOPY\_ON (*C++ enumerator*), 728  
MFX\_HEVC\_CONSTR\_RECT\_INTRA (*C++ enumerator*), 756  
MFX\_HEVC\_CONSTR\_RECT\_LOWER\_BIT\_RATE (*C++ enumerator*), 757  
MFX\_HEVC\_CONSTR\_RECT\_MAX\_10BIT (*C++ enumerator*), 756  
MFX\_HEVC\_CONSTR\_RECT\_MAX\_12BIT (*C++ enumerator*), 756  
MFX\_HEVC\_CONSTR\_RECT\_MAX\_420CHROMA (*C++ enumerator*), 756  
MFX\_HEVC\_CONSTR\_RECT\_MAX\_422CHROMA (*C++ enumerator*), 756  
MFX\_HEVC\_CONSTR\_RECT\_MAX\_8BIT (*C++ enumerator*), 756  
MFX\_HEVC\_CONSTR\_RECT\_MAX\_MONOCHROME (*C++ enumerator*), 756  
MFX\_HEVC\_CONSTR\_RECT\_ONE\_PICTURE\_ONLY (*C++ enumerator*), 756  
MFX\_HEVC\_NALU\_TYPE\_CRA\_NUT (*C++ enumerator*), 752  
MFX\_HEVC\_NALU\_TYPE\_IDR\_N\_LP (*C++ enumerator*), 752  
MFX\_HEVC\_NALU\_TYPE\_IDR\_W\_RADL (*C++ enumerator*), 752  
MFX\_HEVC\_NALU\_TYPE\_RADL\_N (*C++ enumerator*), 752  
MFX\_HEVC\_NALU\_TYPE\_RADL\_R (*C++ enumerator*), 752  
MFX\_HEVC\_NALU\_TYPE\_RASL\_N (*C++ enumerator*), 752  
MFX\_HEVC\_NALU\_TYPE\_RASL\_R (*C++ enumerator*), 752  
MFX\_HEVC\_NALU\_TYPE\_TRAIL\_N (*C++ enumerator*), 752  
MFX\_HEVC\_NALU\_TYPE\_TRAIL\_R (*C++ enumerator*), 752  
MFX\_HEVC\_NALU\_TYPE\_UNKNOWN (*C++ enumerator*), 752  
MFX\_HEVC\_REGION\_ENCODING\_OFF (*C++ enumerator*), 758  
MFX\_HEVC\_REGION\_ENCODING\_ON (*C++ enumerator*), 758  
MFX\_HEVC\_REGION\_SLICE (*C++ enumerator*), 757  
MFX\_IMAGESTAB\_MODE\_BOXING (*C++ enumerator*), 753  
MFX\_IMAGESTAB\_MODE\_UPSCALE (*C++ enumerator*), 753  
MFX\_IMPL\_AUTO (*C++ enumerator*), 727  
MFX\_IMPL\_AUTO\_ANY (*C++ enumerator*), 727  
MFX\_IMPL\_BASETYPE (*C macro*), 727  
MFX\_IMPL\_HARDWARE (*C++ enumerator*), 727  
MFX\_IMPL\_HARDWARE2 (*C++ enumerator*), 727  
MFX\_IMPL\_HARDWARE3 (*C++ enumerator*), 727  
MFX\_IMPL\_HARDWARE4 (*C++ enumerator*), 727  
MFX\_IMPL\_HARDWARE\_ANY (*C++ enumerator*), 727  
MFX\_IMPL\_NAME\_LEN (*C macro*), 713  
MFX\_IMPL\_RUNTIME (*C++ enumerator*), 727  
MFX\_IMPL\_SOFTWARE (*C++ enumerator*), 727  
MFX\_IMPL\_UNSUPPORTED (*C++ enumerator*), 727  
MFX\_IMPL\_VIA\_ANY (*C++ enumerator*), 727  
MFX\_IMPL\_VIA\_D3D11 (*C++ enumerator*), 727  
MFX\_IMPL\_VIA\_D3D9 (*C++ enumerator*), 727  
MFX\_IMPL\_VIA\_VAAPI (*C++ enumerator*), 727  
MFX\_IMPLDESCRIPTION\_VERSION (*C macro*), 841  
MFX\_INTERPOLATION\_ADVANCED (*C++ enumerator*), 758  
MFX\_INTERPOLATION\_BILINEAR (*C++ enumerator*), 758  
MFX\_INTERPOLATION\_DEFAULT (*C++ enumerator*), 758  
MFX\_INTERPOLATION\_NEAREST\_NEIGHBOR (*C++ enumerator*), 758  
MFX\_IOPATTERN\_IN\_SYSTEM\_MEMORY (*C++ enumerator*), 734  
MFX\_IOPATTERN\_IN\_VIDEO\_MEMORY (*C++ enumerator*), 734  
MFX\_IOPATTERN\_OUT\_SYSTEM\_MEMORY (*C++ enumerator*), 734  
MFX\_IOPATTERN\_OUT\_VIDEO\_MEMORY (*C++ enumerator*), 734  
MFX\_JPEG\_COLORFORMAT\_RGB (*C++ enumerator*), 762  
MFX\_JPEG\_COLORFORMAT\_UNKNOWN (*C++ enumerator*), 762  
MFX\_JPEG\_COLORFORMAT\_YCbCr (*C++ enumerator*), 762  
MFX\_LEGACY\_VERSION (*C macro*), 841

- MFX\_LEVEL\_AVC\_1 (*C++ enumerator*), 737  
 MFX\_LEVEL\_AVC\_11 (*C++ enumerator*), 737  
 MFX\_LEVEL\_AVC\_12 (*C++ enumerator*), 737  
 MFX\_LEVEL\_AVC\_13 (*C++ enumerator*), 737  
 MFX\_LEVEL\_AVC\_1b (*C++ enumerator*), 737  
 MFX\_LEVEL\_AVC\_2 (*C++ enumerator*), 737  
 MFX\_LEVEL\_AVC\_21 (*C++ enumerator*), 737  
 MFX\_LEVEL\_AVC\_22 (*C++ enumerator*), 737  
 MFX\_LEVEL\_AVC\_3 (*C++ enumerator*), 737  
 MFX\_LEVEL\_AVC\_31 (*C++ enumerator*), 737  
 MFX\_LEVEL\_AVC\_32 (*C++ enumerator*), 737  
 MFX\_LEVEL\_AVC\_4 (*C++ enumerator*), 738  
 MFX\_LEVEL\_AVC\_41 (*C++ enumerator*), 738  
 MFX\_LEVEL\_AVC\_42 (*C++ enumerator*), 738  
 MFX\_LEVEL\_AVC\_5 (*C++ enumerator*), 738  
 MFX\_LEVEL\_AVC\_51 (*C++ enumerator*), 738  
 MFX\_LEVEL\_AVC\_52 (*C++ enumerator*), 738  
 MFX\_LEVEL\_HEVC\_1 (*C++ enumerator*), 738  
 MFX\_LEVEL\_HEVC\_2 (*C++ enumerator*), 738  
 MFX\_LEVEL\_HEVC\_21 (*C++ enumerator*), 738  
 MFX\_LEVEL\_HEVC\_3 (*C++ enumerator*), 738  
 MFX\_LEVEL\_HEVC\_31 (*C++ enumerator*), 738  
 MFX\_LEVEL\_HEVC\_4 (*C++ enumerator*), 738  
 MFX\_LEVEL\_HEVC\_41 (*C++ enumerator*), 739  
 MFX\_LEVEL\_HEVC\_5 (*C++ enumerator*), 739  
 MFX\_LEVEL\_HEVC\_51 (*C++ enumerator*), 739  
 MFX\_LEVEL\_HEVC\_52 (*C++ enumerator*), 739  
 MFX\_LEVEL\_HEVC\_6 (*C++ enumerator*), 739  
 MFX\_LEVEL\_HEVC\_61 (*C++ enumerator*), 739  
 MFX\_LEVEL\_HEVC\_62 (*C++ enumerator*), 739  
 MFX\_LEVEL\_MPEG2\_HIGH (*C++ enumerator*), 738  
 MFX\_LEVEL\_MPEG2\_HIGH1440 (*C++ enumerator*), 738  
 MFX\_LEVEL\_MPEG2\_LOW (*C++ enumerator*), 738  
 MFX\_LEVEL\_MPEG2\_MAIN (*C++ enumerator*), 738  
 MFX\_LEVEL\_UNKNOWN (*C++ enumerator*), 737  
 MFX\_LEVEL\_VC1\_0 (*C++ enumerator*), 738  
 MFX\_LEVEL\_VC1\_1 (*C++ enumerator*), 738  
 MFX\_LEVEL\_VC1\_2 (*C++ enumerator*), 738  
 MFX\_LEVEL\_VC1\_3 (*C++ enumerator*), 738  
 MFX\_LEVEL\_VC1\_4 (*C++ enumerator*), 738  
 MFX\_LEVEL\_VC1\_HIGH (*C++ enumerator*), 738  
 MFX\_LEVEL\_VC1\_LOW (*C++ enumerator*), 738  
 MFX\_LEVEL\_VC1\_MEDIAN (*C++ enumerator*), 738  
 MFX\_LONGTERM\_IDX\_NO\_IDX (*C++ enumerator*), 754  
 MFX\_LOOKAHEAD\_DS\_2x (*C++ enumerator*), 742  
 MFX\_LOOKAHEAD\_DS\_4x (*C++ enumerator*), 742  
 MFX\_LOOKAHEAD\_DS\_OFF (*C++ enumerator*), 742  
 MFX\_LOOKAHEAD\_DS\_UNKNOWN (*C++ enumerator*), 742  
 MFX\_MBQP\_MODE\_QP\_ADAPTIVE (*C++ enumerator*), 756  
 MFX\_MBQP\_MODE\_QP\_DELTA (*C++ enumerator*), 756  
 MFX\_MBQP\_MODE\_QP\_VALUE (*C++ enumerator*), 756  
 MFX\_MEMTYPE\_DXVA2\_DECODER\_TARGET (*C++ enumerator*), 750  
 MFX\_MEMTYPE\_DXVA2\_PROCESSOR\_TARGET (*C++ enumerator*), 750  
 MFX\_MEMTYPE\_EXPORT\_FRAME (*C++ enumerator*), 750  
 MFX\_MEMTYPE\_EXTERNAL\_FRAME (*C++ enumerator*), 750  
 MFX\_MEMTYPE\_FROM\_DECODE (*C++ enumerator*), 750  
 MFX\_MEMTYPE\_FROM\_ENC (*C++ enumerator*), 750  
 MFX\_MEMTYPE\_FROM\_ENCODE (*C++ enumerator*), 750  
 MFX\_MEMTYPE\_FROM\_VPPIN (*C++ enumerator*), 750  
 MFX\_MEMTYPE\_FROM\_VPPOUT (*C++ enumerator*), 750  
 MFX\_MEMTYPE\_INTERNAL\_FRAME (*C++ enumerator*), 750  
 MFX\_MEMTYPE\_PERSISTENT\_MEMORY (*C++ enumerator*), 750  
 MFX\_MEMTYPE\_RESERVED1 (*C++ enumerator*), 750  
 MFX\_MEMTYPE\_SHARED\_RESOURCE (*C++ enumerator*), 751  
 MFX\_MEMTYPE\_SYSTEM\_MEMORY (*C++ enumerator*), 750  
 MFX\_MEMTYPE\_VIDEO\_MEMORY\_DECODER\_TARGET (*C++ enumerator*), 750  
 MFX\_MEMTYPE\_VIDEO\_MEMORY\_ENCODER\_TARGET (*C++ enumerator*), 751  
 MFX\_MEMTYPE\_VIDEO\_MEMORY\_PROCESSOR\_TARGET (*C++ enumerator*), 750  
 MFX\_MIRRORING\_DISABLED (*C++ enumerator*), 759  
 MFX\_MIRRORING\_HORIZONTAL (*C++ enumerator*), 759  
 MFX\_MIRRORING\_VERTICAL (*C++ enumerator*), 759  
 MFX\_MVPRECISION\_HALFPEL (*C++ enumerator*), 744  
 MFX\_MVPRECISION\_INTEGER (*C++ enumerator*), 744  
 MFX\_MVPRECISION\_QUARTERPEL (*C++ enumerator*), 744  
 MFX\_MVPRECISION\_UNKNOWN (*C++ enumerator*), 744  
 MFX\_NOMINALRANGE\_0\_255 (*C++ enumerator*), 754  
 MFX\_NOMINALRANGE\_16\_235 (*C++ enumerator*), 754  
 MFX\_NOMINALRANGE\_UNKNOWN (*C++ enumerator*), 754  
 MFX\_P\_REF\_DEFAULT (*C++ enumerator*), 743  
 MFX\_P\_REF\_PYRAMID (*C++ enumerator*), 743  
 MFX\_P\_REF\_SIMPLE (*C++ enumerator*), 743  
 MFX\_PARTIAL\_BITSTREAM\_ANY (*C++ enumerator*), 761

MFX\_PARTIAL\_BITSTREAM\_BLOCK (*C++ enumerator*), 761  
MFX\_PARTIAL\_BITSTREAM\_NONE (*C++ enumerator*), 761  
MFX\_PARTIAL\_BITSTREAM\_SLICE (*C++ enumerator*), 761  
MFX\_PAYLOAD\_CTRL\_SUFFIX (*C++ enumerator*), 750  
MFX\_PAYLOAD\_IDR (*C++ enumerator*), 754  
MFX\_PAYLOAD\_OFF (*C++ enumerator*), 754  
MFX\_PICSTRUCT\_FIELD\_BFF (*C++ enumerator*), 733  
MFX\_PICSTRUCT\_FIELD\_BOTTOM (*C++ enumerator*), 733  
MFX\_PICSTRUCT\_FIELD\_PAIED\_NEXT (*C++ enumerator*), 733  
MFX\_PICSTRUCT\_FIELD\_PAIED\_PREV (*C++ enumerator*), 733  
MFX\_PICSTRUCT\_FIELD\_REPEAT (*C++ enumerator*), 733  
MFX\_PICSTRUCT\_FIELD\_SINGLE (*C++ enumerator*), 733  
MFX\_PICSTRUCT\_FIELD\_TFF (*C++ enumerator*), 733  
MFX\_PICSTRUCT\_FIELD\_TOP (*C++ enumerator*), 733  
MFX\_PICSTRUCT\_FRAME\_DOUBLING (*C++ enumerator*), 733  
MFX\_PICSTRUCT\_FRAME\_TRIPLING (*C++ enumerator*), 733  
MFX\_PICSTRUCT\_PROGRESSIVE (*C++ enumerator*), 733  
MFX\_PICSTRUCT\_UNKNOWN (*C++ enumerator*), 733  
MFX\_PICTYPE\_BOTTOMFIELD (*C++ enumerator*), 756  
MFX\_PICTYPE\_FRAME (*C++ enumerator*), 756  
MFX\_PICTYPE\_TOPFIELD (*C++ enumerator*), 756  
MFX\_PICTYPE\_UNKNOWN (*C++ enumerator*), 756  
MFX\_PLATFORM\_APOLLOLAKE (*C++ enumerator*), 729  
MFX\_PLATFORM\_BAYTRAIL (*C++ enumerator*), 728  
MFX\_PLATFORM\_BROADWELL (*C++ enumerator*), 728  
MFX\_PLATFORM\_CANNONLAKE (*C++ enumerator*), 729  
MFX\_PLATFORM\_CHERRYTRAIL (*C++ enumerator*), 728  
MFX\_PLATFORM\_COFFEEAKE (*C++ enumerator*), 729  
MFX\_PLATFORM\_ELKHARTLAKE (*C++ enumerator*), 729  
MFX\_PLATFORM\_GEMINILAKE (*C++ enumerator*), 729  
MFX\_PLATFORM\_HASWELL (*C++ enumerator*), 728  
MFX\_PLATFORM\_ICELAKE (*C++ enumerator*), 729  
MFX\_PLATFORM\_IVYBRIDGE (*C++ enumerator*), 728  
MFX\_PLATFORM\_JASPERLAKE (*C++ enumerator*), 729  
MFX\_PLATFORM\_KABYLAKE (*C++ enumerator*), 729  
MFX\_PLATFORM\_SANDYBRIDGE (*C++ enumerator*), 728  
MFX\_PLATFORM\_SKYLAKE (*C++ enumerator*), 728  
MFX\_PLATFORM\_TIGERLAKE (*C++ enumerator*), 729  
MFX\_PLATFORM\_UNKNOWN (*C++ enumerator*), 728  
MFX\_PROFILE\_AVC\_BASELINE (*C++ enumerator*), 735  
MFX\_PROFILE\_AVC\_CONSTRAINED\_BASELINE (*C++ enumerator*), 735  
MFX\_PROFILE\_AVC\_CONSTRAINED\_HIGH (*C++ enumerator*), 735  
MFX\_PROFILE\_AVC\_CONSTRAINT\_SET0 (*C++ enumerator*), 735  
MFX\_PROFILE\_AVC\_CONSTRAINT\_SET1 (*C++ enumerator*), 735  
MFX\_PROFILE\_AVC\_CONSTRAINT\_SET2 (*C++ enumerator*), 735  
MFX\_PROFILE\_AVC\_CONSTRAINT\_SET3 (*C++ enumerator*), 735  
MFX\_PROFILE\_AVC\_CONSTRAINT\_SET4 (*C++ enumerator*), 736  
MFX\_PROFILE\_AVC\_CONSTRAINT\_SET5 (*C++ enumerator*), 736  
MFX\_PROFILE\_AVC\_EXTENDED (*C++ enumerator*), 735  
MFX\_PROFILE\_AVC\_HIGH (*C++ enumerator*), 735  
MFX\_PROFILE\_AVC\_HIGH10 (*C++ enumerator*), 735  
MFX\_PROFILE\_AVC\_HIGH\_422 (*C++ enumerator*), 735  
MFX\_PROFILE\_AVC\_MAIN (*C++ enumerator*), 735  
MFX\_PROFILE\_AVC\_MULTIVIEW\_HIGH (*C++ enumerator*), 736  
MFX\_PROFILE\_AVC\_STEREO\_HIGH (*C++ enumerator*), 736  
MFX\_PROFILE\_HEVC\_MAIN (*C++ enumerator*), 736  
MFX\_PROFILE\_HEVC\_MAIN10 (*C++ enumerator*), 736  
MFX\_PROFILE\_HEVC\_MAINSP (*C++ enumerator*), 736  
MFX\_PROFILE\_HEVC\_RECT (*C++ enumerator*), 736  
MFX\_PROFILE\_HEVC\_SCC (*C++ enumerator*), 736  
MFX\_PROFILE\_JPEG\_BASELINE (*C++ enumerator*), 737  
MFX\_PROFILE\_MPEG2\_HIGH (*C++ enumerator*), 736  
MFX\_PROFILE\_MPEG2\_MAIN (*C++ enumerator*), 736  
MFX\_PROFILE\_MPEG2\_SIMPLE (*C++ enumerator*), 736  
MFX\_PROFILE\_UNKNOWN (*C++ enumerator*), 735  
MFX\_PROFILE\_VC1\_ADVANCED (*C++ enumerator*), 736

MFX\_PROFILE\_VC1\_MAIN (*C++ enumerator*), 736  
MFX\_PROFILE\_VC1\_SIMPLE (*C++ enumerator*), 736  
MFX\_PROFILE\_VP8\_0 (*C++ enumerator*), 736  
MFX\_PROFILE\_VP8\_1 (*C++ enumerator*), 736  
MFX\_PROFILE\_VP8\_2 (*C++ enumerator*), 736  
MFX\_PROFILE\_VP8\_3 (*C++ enumerator*), 736  
MFX\_PROFILE\_VP9\_0 (*C++ enumerator*), 737  
MFX\_PROFILE\_VP9\_1 (*C++ enumerator*), 737  
MFX\_PROFILE\_VP9\_2 (*C++ enumerator*), 737  
MFX\_PROFILE\_VP9\_3 (*C++ enumerator*), 737  
MFX\_PROTECTION\_CENC\_WV\_CLASSIC (*C++ enumerator*), 762  
MFX\_PROTECTION\_CENC\_WV\_GOOGLE\_DASH (*C++ enumerator*), 762  
MFX\_RATECONTROL\_AVBR (*C++ enumerator*), 740  
MFX\_RATECONTROL\_CBR (*C++ enumerator*), 740  
MFX\_RATECONTROL\_CQP (*C++ enumerator*), 740  
MFX\_RATECONTROL\_ICQ (*C++ enumerator*), 740  
MFX\_RATECONTROL\_LA (*C++ enumerator*), 740  
MFX\_RATECONTROL\_LA\_HRD (*C++ enumerator*), 741  
MFX\_RATECONTROL\_LA\_ICQ (*C++ enumerator*), 740  
MFX\_RATECONTROL\_QVBR (*C++ enumerator*), 741  
MFX\_RATECONTROL\_VBR (*C++ enumerator*), 740  
MFX\_RATECONTROL\_VCM (*C++ enumerator*), 740  
MFX\_REFRESH\_HORIZONTAL (*C++ enumerator*), 743  
MFX\_REFRESH\_NO (*C++ enumerator*), 743  
MFX\_REFRESH\_SLICE (*C++ enumerator*), 743  
MFX\_REFRESH\_VERTICAL (*C++ enumerator*), 743  
MFX\_ROI\_MODE\_PRIORITY (*C++ enumerator*), 754  
MFX\_ROI\_MODE\_QP\_DELTA (*C++ enumerator*), 754  
MFX\_ROI\_MODE\_QP\_VALUE (*C++ enumerator*), 754  
MFX\_ROTATION\_0 (*C++ enumerator*), 761  
MFX\_ROTATION\_180 (*C++ enumerator*), 761  
MFX\_ROTATION\_270 (*C++ enumerator*), 761  
MFX\_ROTATION\_90 (*C++ enumerator*), 761  
MFX\_SAO\_DISABLE (*C++ enumerator*), 757  
MFX\_SAO\_ENABLE\_CHROMA (*C++ enumerator*), 757  
MFX\_SAO\_ENABLE\_LUMA (*C++ enumerator*), 757  
MFX\_SAO\_UNKNOWN (*C++ enumerator*), 757  
MFX\_SCALING\_MODE\_DEFAULT (*C++ enumerator*), 758  
MFX\_SCALING\_MODE\_LOWPOWER (*C++ enumerator*), 758  
MFX\_SCALING\_MODE\_QUALITY (*C++ enumerator*), 758  
MFX\_SCANTYPE\_INTERLEAVED (*C++ enumerator*), 762  
MFX\_SCANTYPE\_NONINTERLEAVED (*C++ enumerator*), 762  
MFX\_SCANTYPE\_UNKNOWN (*C++ enumerator*), 762  
MFX\_SCENARIO\_ARCHIVE (*C++ enumerator*), 744  
MFX\_SCENARIO\_CAMERA\_CAPTURE (*C++ enumerator*), 744  
MFX\_SCENARIO\_DISPLAY\_Remoting (*C++ enumerator*), 744  
MFX\_SCENARIO\_GAME\_STREAMING (*C++ enumerator*), 744  
MFX\_SCENARIO\_LIVE\_STREAMING (*C++ enumerator*), 744  
MFX\_SCENARIO\_REMOTE\_GAMING (*C++ enumerator*), 744  
MFX\_SCENARIO\_UNKNOWN (*C++ enumerator*), 744  
MFX\_SCENARIO\_VIDEO\_CONFERENCE (*C++ enumerator*), 744  
MFX\_SCENARIO\_VIDEO\_SURVEILLANCE (*C++ enumerator*), 744  
MFX\_SKIPFRAME\_BRC\_ONLY (*C++ enumerator*), 742  
MFX\_SKIPFRAME\_INSERT\_DUMMY (*C++ enumerator*), 742  
MFX\_SKIPFRAME\_INSERT NOTHING (*C++ enumerator*), 742  
MFX\_SKIPFRAME\_NO\_SKIP (*C++ enumerator*), 742  
MFX\_STRFIELD\_LEN (*C macro*), 713  
MFX\_STRUCT\_VERSION (*C macro*), 841  
MFX\_TARGETUSAGE\_1 (*C++ enumerator*), 739  
MFX\_TARGETUSAGE\_2 (*C++ enumerator*), 739  
MFX\_TARGETUSAGE\_3 (*C++ enumerator*), 739  
MFX\_TARGETUSAGE\_4 (*C++ enumerator*), 739  
MFX\_TARGETUSAGE\_5 (*C++ enumerator*), 740  
MFX\_TARGETUSAGE\_6 (*C++ enumerator*), 740  
MFX\_TARGETUSAGE\_7 (*C++ enumerator*), 740  
MFX\_TARGETUSAGE\_BALANCED (*C++ enumerator*), 740  
MFX\_TARGETUSAGE\_BEST\_QUALITY (*C++ enumerator*), 740  
MFX\_TARGETUSAGE\_BEST\_SPEED (*C++ enumerator*), 740  
MFX\_TARGETUSAGE\_UNKNOWN (*C++ enumerator*), 740  
MFX\_TELECINE\_PATTERN\_2332 (*C++ enumerator*), 755  
MFX\_TELECINE\_PATTERN\_32 (*C++ enumerator*), 755  
MFX\_TELECINE\_PATTERN\_41 (*C++ enumerator*), 755  
MFX\_TELECINE\_PATTERN\_FRAME\_REPEAT (*C++ enumerator*), 755  
MFX\_TELECINE\_POSITION\_PROVIDED (*C++ enumerator*), 755  
MFX\_TIER\_HEVC\_HIGH (*C++ enumerator*), 739  
MFX\_TIER\_HEVC\_MAIN (*C++ enumerator*), 739  
MFX\_TIMESTAMP\_UNKNOWN (*C++ enumerator*), 733  
MFX\_TIMESTAMPCALC\_TELECINE (*C++ enumerator*), 734  
MFX\_TIMESTAMPCALC\_UNKNOWN (*C++ enumerator*), 734  
MFX\_TRANSFERMATRIX\_BT601 (*C++ enumerator*),

754  
 MFX\_TRANSFERMATRIX\_BT709 (*C++ enumerator*), 754  
 MFX\_TRANSFERMATRIX\_UNKNOWN (*C++ enumerator*), 754  
 MFX\_TRELLIS\_B (*C++ enumerator*), 741  
 MFX\_TRELLIS\_I (*C++ enumerator*), 741  
 MFX\_TRELLIS\_OFF (*C++ enumerator*), 741  
 MFX\_TRELLIS\_P (*C++ enumerator*), 741  
 MFX\_TRELLIS\_UNKNOWN (*C++ enumerator*), 741  
 MFX\_VARIANT\_VERSION (*C macro*), 841  
 MFX\_VERSION (*C macro*), 841  
 MFX\_VERSION\_MAJOR (*C macro*), 841  
 MFX\_VERSION\_MINOR (*C macro*), 841  
 MFX\_VERSION\_NEXT (*C macro*), 841  
 MFX\_VP9\_REF\_ALTREF (*C++ enumerator*), 759  
 MFX\_VP9\_REF\_GOLDEN (*C++ enumerator*), 759  
 MFX\_VP9\_REF\_INTRA (*C++ enumerator*), 759  
 MFX\_VP9\_REF\_LAST (*C++ enumerator*), 759  
 MFX\_VP9\_SEGMENT\_FEATURE\_LOOP\_FILTER  
     (*C++ enumerator*), 760  
 MFX\_VP9\_SEGMENT\_FEATURE\_QINDEX (*C++ enumerator*), 760  
 MFX\_VP9\_SEGMENT\_FEATURE\_REFERENCE (*C++ enumerator*), 760  
 MFX\_VP9\_SEGMENT\_FEATURE\_SKIP (*C++ enumerator*), 760  
 MFX\_VP9\_SEGMENT\_ID\_BLOCK\_SIZE\_16x16  
     (*C++ enumerator*), 760  
 MFX\_VP9\_SEGMENT\_ID\_BLOCK\_SIZE\_32x32  
     (*C++ enumerator*), 760  
 MFX\_VP9\_SEGMENT\_ID\_BLOCK\_SIZE\_64x64  
     (*C++ enumerator*), 760  
 MFX\_VP9\_SEGMENT\_ID\_BLOCK\_SIZE\_8x8 (*C++ enumerator*), 760  
 MFX\_VP9\_SEGMENT\_ID\_BLOCK\_SIZE\_UNKNOWN  
     (*C++ enumerator*), 760  
 MFX\_VPP\_COPY\_FIELD (*C++ enumerator*), 756  
 MFX\_VPP\_COPY\_FRAME (*C++ enumerator*), 756  
 MFX\_VPP\_SWAP\_FIELDS (*C++ enumerator*), 756  
 MFX\_VPPDESCRIPTION\_VERSION (*C macro*), 841  
 MFX\_WEIGHTED\_PRED\_DEFAULT (*C++ enumerator*), 743  
 MFX\_WEIGHTED\_PRED\_EXPLICIT (*C++ enumerator*), 743  
 MFX\_WEIGHTED\_PRED\_IMPLICIT (*C++ enumerator*), 743  
 MFX\_WEIGHTED\_PRED\_UNKNOWN (*C++ enumerator*), 743  
 mfxA2RGB10 (*C++ struct*), 773  
 mfxA2RGB10:::A (*C++ member*), 773  
 mfxA2RGB10:::B (*C++ member*), 773  
 mfxA2RGB10:::G (*C++ member*), 773  
 mfxA2RGB10:::R (*C++ member*), 773  
 mfxAccelerationMode (*C++ enum*), 713  
 mfxAccelerationMode:::MFX\_ACCEL\_MODE\_NA  
     (*C++ enumerator*), 713  
 mfxAccelerationMode:::MFX\_ACCEL\_MODE\_VIA\_D3D11  
     (*C++ enumerator*), 713  
 mfxAccelerationMode:::MFX\_ACCEL\_MODE\_VIA\_D3D9  
     (*C++ enumerator*), 713  
 mfxAccelerationMode:::MFX\_ACCEL\_MODE\_VIA\_VAAPI  
     (*C++ enumerator*), 713  
 mfxAdapterInfo (*C++ struct*), 785  
 mfxAdapterInfo:::Number (*C++ member*), 785  
 mfxAdapterInfo:::Platform (*C++ member*), 785  
 mfxAdaptersInfo (*C++ struct*), 786  
 mfxAdaptersInfo:::Adapters (*C++ member*),  
     786  
 mfxAdaptersInfo:::NumActual (*C++ member*),  
     786  
 mfxAdaptersInfo:::NumAlloc (*C++ member*),  
     786  
 mfxBitstream (*C++ struct*), 779  
 mfxBitstream:::Data (*C++ member*), 779  
 mfxBitstream:::DataFlag (*C++ member*), 780  
 mfxBitstream:::DataLength (*C++ member*), 780  
 mfxBitstream:::DataOffset (*C++ member*), 779  
 mfxBitstream:::DecodeTimeStamp (*C++ mem-  
     ber*), 779  
 mfxBitstream:::EncryptedData (*C++ member*),  
     779  
 mfxBitstream:::ExtParam (*C++ member*), 779  
 mfxBitstream:::FrameType (*C++ member*), 780  
 mfxBitstream:::MaxLength (*C++ member*), 780  
 mfxBitstream:::NumExtParam (*C++ member*),  
     779  
 mfxBitstream:::PicStruct (*C++ member*), 780  
 mfxBitstream:::reserved2 (*C++ member*), 780  
 mfxBitstream:::TimeStamp (*C++ member*), 779  
 mfxBRCFrmeCtrl (*C++ struct*), 833  
 mfxBRCFrmeCtrl:::DeltaQP (*C++ member*), 834  
 mfxBRCFrmeCtrl:::ExtParam (*C++ member*),  
     834  
 mfxBRCFrmeCtrl:::InitialCpbRemovalDelay  
     (*C++ member*), 833  
 mfxBRCFrmeCtrl:::InitialCpbRemovalOffset  
     (*C++ member*), 833  
 mfxBRCFrmeCtrl:::MaxFrameSize (*C++ mem-  
     ber*), 833  
 mfxBRCFrmeCtrl:::MaxNumRepak (*C++ mem-  
     ber*), 834  
 mfxBRCFrmeCtrl:::NumExtParam (*C++ mem-  
     ber*), 834  
 mfxBRCFrmeCtrl:::QpY (*C++ member*), 833  
 mfxBRCFrmeParam (*C++ struct*), 832  
 mfxBRCFrmeParam:::CodedFrameSize (*C++  
     member*), 833

mfxBRCFrameParam::DisplayOrder (*C++ member*), 833  
 mfxBRCFrameParam::EncodedOrder (*C++ member*), 832  
 mfxBRCFrameParam::ExtParam (*C++ member*), 833  
 mfxBRCFrameParam::FrameCmplx (*C++ member*), 832  
 mfxBRCFrameParam::FrameType (*C++ member*), 833  
 mfxBRCFrameParam::LongTerm (*C++ member*), 832  
 mfxBRCFrameParam::NumExtParam (*C++ member*), 833  
 mfxBRCFrameParam::NumRecode (*C++ member*), 833  
 mfxBRCFrameParam::PyramidLayer (*C++ member*), 833  
 mfxBRCFrameParam::SceneChange (*C++ member*), 832  
 mfxBRCFrameStatus (*C++ struct*), 834  
 mfxBRCFrameStatus::BRCstatus (*C++ member*), 834  
 mfxBRCFrameStatus::MinFrameSize (*C++ member*), 834  
 mfxChar (*C++ type*), 712  
 MFXCloneSession (*C++ function*), 844  
 MFXClose (*C++ function*), 843  
 mfxComponentInfo (*C++ struct*), 785  
 mfxComponentInfo::Requirements (*C++ member*), 785  
 mfxComponentInfo::Type (*C++ member*), 785  
 mfxComponentType (*C++ enum*), 760  
 mfxComponentType::MFX\_COMPONENT\_DECODE (*C++ enumerator*), 760  
 mfxComponentType::MFX\_COMPONENT\_ENCODE (*C++ enumerator*), 760  
 mfxComponentType::MFX\_COMPONENT\_VPP (*C++ enumerator*), 760  
 mfxConfig (*C++ type*), 713  
 MFXCreateConfig (*C++ function*), 721  
 MFXCreateSession (*C++ function*), 724  
 mfxDecoderDescription (*C++ struct*), 715  
 mfxDecoderDescription::Codecs (*C++ member*), 715  
 mfxDecoderDescription::decoder (*C++ struct*), 715  
 mfxDecoderDescription::decoder::CodecID mfxDeviceDescription::SubDevices (*C++ member*), 720  
 mfxDecoderDescription::decoder::decprofile mfxDeviceDescription::subdevices (*C++ struct*), 720  
 mfxDecoderDescription::decoder::decprofile mfxDeviceDescription::subdevices::Index (*C++ member*), 720  
 mfxDecoderDescription::decoder::decprofile mfxDeviceDescription::subdevices::reserved

(C++ member), 720  
`mfxDeviceDescription::subdevices::SubDeviceEncoderDescription::encoder::encprofile::Profile`  
(C++ member), 720  
`(C++ member), 717`  
`mfxDeviceDescription::Version (C++ member), 720`  
`mfxEncoderDescription::encoder::encprofile::reserved`  
(C++ member), 717  
`mfxEncoderDescription::encoder::MaxcodecLevel`  
(C++ member), 717  
`mfxEncoderDescription::encoder::NumProfiles`  
(C++ member), 717  
`mfxEncoderDescription::encoder::Profiles`  
(C++ member), 717  
`mfxEncoderDescription::encoder::reserved`  
(C++ member), 717  
`mfxEncoderDescription::NumCodecs (C++ member), 717`  
`mfxEncoderDescription::reserved (C++ member), 717`  
`mfxEncoderDescription::Version (C++ member), 717`  
`mfxEncodeStat (C++ struct), 780`  
`mfxEncodeStat::NumBit (C++ member), 780`  
`mfxEncodeStat::NumCachedFrame (C++ member), 780`  
`mfxEncodeStat::NumFrame (C++ member), 780`  
`MFXEnumImplementations (C++ function), 723`  
`mfxExtAVCEncodedFrameInfo (C++ struct), 805`  
`mfxExtAVCEncodedFrameInfo::BRCPanicMode`  
(C++ member), 805  
`mfxExtAVCEncodedFrameInfo::FrameOrder`  
(C++ member), 805  
`mfxExtAVCEncodedFrameInfo::Header (C++ member), 805`  
`mfxExtAVCEncodedFrameInfo::LongTermIdx`  
(C++ member), 805  
`mfxExtAVCEncodedFrameInfo::MAD (C++ member), 805`  
`mfxExtAVCEncodedFrameInfo::PicStruct`  
(C++ member), 805  
`mfxExtAVCEncodedFrameInfo::QP (C++ member), 805`  
`mfxExtAVCEncodedFrameInfo::end (C++ member), 806`  
`mfxExtAVCEncodedFrameInfo::ColorFormats`  
`mfxExtAVCEncodedFrameInfo::SecondFieldOffset`  
(C++ member), 806  
`mfxExtAVCEncodedFrameInfo::end (C++ member), 806`  
`mfxExtAVCEncodedFrameInfo::UsedRefListL1`  
(C++ member), 806  
`mfxExtAVCEncodedFrameInfo::end (C++ member), 806`  
`mfxExtAVCRefListCtrl (C++ struct), 800`  
`mfxEncoderDescription::encoder::encprofile::end (C++ member), 806`  
`mfxEncoderDescription::encoder::encprofile::HandleType`  
(C++ member), 806  
`mfxEncoderDescription::encoder::encprofile::end (C++ member), 806`  
`mfxEncoderDescription::encoder::encprofile::nameOrder`  
(C++ member), 800  
`mfxEncoderDescription::encoder::encprofile::end (C++ member), 806`  
`mfxEncoderDescription::encoder::encprofile::Header (C++ member), 800`  
`mfxEncoderDescription::encoder::encprofile::end (C++ member), 806`  
`mfxEncoderDescription::encoder::encprofile::LongTermIdx`  
(C++ member), 800  
`mfxEncoderDescription::encoder::encprofile::end (C++ member), 806`  
`mfxEncoderDescription::encoder::encprofile::LongTermRefList`

(*C++ member*), 800  
*mfxExtAVCRefListCtrl*::NumRefIdxL0Active (*C++ member*), 800  
*mfxExtAVCRefListCtrl*::NumRefIdxL1Active (*C++ member*), 800  
*mfxExtAVCRefListCtrl*::PicStruct (*C++ member*), 800  
*mfxExtAVCRefListCtrl*::PreferredRefList (*C++ member*), 800  
*mfxExtAVCRefListCtrl*::RejectedRefList (*C++ member*), 800  
*mfxExtAVCRefListCtrl*::reserved (*C++ member*), 800  
*mfxExtAVCRefListCtrl*::ViewId (*C++ member*), 800  
*mfxExtAVCRefLists* (*C++ struct*), 807  
*mfxExtAVCRefLists*::Header (*C++ member*), 807  
*mfxExtAVCRefLists*::mfxRefPic (*C++ struct*), 807  
*mfxExtAVCRefLists*::mfxRefPic::FrameOrder (*C++ member*), 808  
*mfxExtAVCRefLists*::mfxRefPic::PicStruct (*C++ member*), 808  
*mfxExtAVCRefLists*::NumRefIdxL0Active (*C++ member*), 807  
*mfxExtAVCRefLists*::NumRefIdxL1Active (*C++ member*), 807  
*mfxExtAVCRefLists*::RefPicList1 (*C++ member*), 807  
*mfxExtAVCRoundingOffset* (*C++ struct*), 813  
*mfxExtAVCRoundingOffset*::EnableRoundingInter (*C++ member*), 813  
*mfxExtAVCRoundingOffset*::EnableRoundingIntra (*C++ member*), 813  
*mfxExtAVCRoundingOffset*::Header (*C++ member*), 813  
*mfxExtAVCRoundingOffset*::RoundingOffsetInter (*C++ member*), 813  
*mfxExtAVCRoundingOffset*::RoundingOffsetIntra (*C++ member*), 813  
*mfxExtAvCTemporalLayers* (*C++ struct*), 803  
*mfxExtAvCTemporalLayers*::BaseLayerPID (*C++ member*), 803  
*mfxExtAvCTemporalLayers*::Header (*C++ member*), 803  
*mfxExtAvCTemporalLayers*::Scale (*C++ member*), 803  
*mfxExtBRC* (*C++ struct*), 834  
*mfxExtBRC*::Close (*C++ member*), 835  
*mfxExtBRC*::GetFrameCtrl (*C++ member*), 835  
*mfxExtBRC*::Header (*C++ member*), 835  
*mfxExtBRC*::Init (*C++ member*), 835  
*mfxExtBRC*::pthis (*C++ member*), 835  
*mfxExtBRC*::Reset (*C++ member*), 835  
*mfxExtBRC*::Update (*C++ member*), 836  
*mfxExtBuffer* (*C++ struct*), 787  
*mfxExtBuffer*::BufferId (*C++ member*), 787  
*mfxExtBuffer*::BufferSz (*C++ member*), 787  
*mfxExtChromaLocInfo* (*C++ struct*), 808  
*mfxExtChromaLocInfo*::ChromaLocInfoPresentFlag (*C++ member*), 808  
*mfxExtChromaLocInfo*::ChromaSampleLocTypeBottomField (*C++ member*), 808  
*mfxExtChromaLocInfo*::ChromaSampleLocTypeTopField (*C++ member*), 808  
*mfxExtChromaLocInfo*::Header (*C++ member*), 808  
*mfxExtChromaLocInfo*::reserved (*C++ member*), 808  
*mfxExtCodingOption* (*C++ struct*), 787  
*mfxExtCodingOption2* (*C++ struct*), 789  
*mfxExtCodingOption2*::AdaptiveB (*C++ member*), 791  
*mfxExtCodingOption2*::AdaptiveI (*C++ member*), 791  
*mfxExtCodingOption2*::BitrateLimit (*C++ member*), 790  
*mfxExtCodingOption2*::BRefType (*C++ member*), 791  
*mfxExtCodingOption2*::BufferingPeriodSEI (*C++ member*), 792  
*mfxExtCodingOption2*::DisableDeblockingIdc (*C++ member*), 792  
*mfxExtCodingOption2*::DisableVUI (*C++ member*), 792  
*mfxExtCodingOption2*::EnableMAD (*C++ member*), 792  
*mfxExtCodingOption2*::ExtBRC (*C++ member*), 790  
*mfxExtCodingOption2*::FixedFrameRate  
*mfxExtCodingOption2*::Header (*C++ member*), 792  
*mfxExtCodingOption2*::Header (*C++ member*), 790  
*mfxExtCodingOption2*::IntRefCycleSize (*C++ member*), 790  
*mfxExtCodingOption2*::IntRefQPDelta (*C++ member*), 790  
*mfxExtCodingOption2*::IntRefType (*C++ member*), 790  
*mfxExtCodingOption2*::LookAheadDepth (*C++ member*), 790  
*mfxExtCodingOption2*::LookAheadDS (*C++ member*), 791  
*mfxExtCodingOption2*::MaxFrameSize (*C++ member*), 790  
*mfxExtCodingOption2*::MaxQPB (*C++ member*), 792

mfxExtCodingOption2::MaxQPI (*C++ member*), 791  
 mfxExtCodingOption2::MaxQPP (*C++ member*), 792  
 mfxExtCodingOption2::MaxSliceSize (*C++ member*), 790  
 mfxExtCodingOption2::MBBRC (*C++ member*), 790  
 mfxExtCodingOption2::MinQPB (*C++ member*), 792  
 mfxExtCodingOption2::MinQPI (*C++ member*), 791  
 mfxExtCodingOption2::MinQPP (*C++ member*), 791  
 mfxExtCodingOption2::NumMbPerSlice (*C++ member*), 791  
 mfxExtCodingOption2::RepeatPPS (*C++ member*), 791  
 mfxExtCodingOption2::SkipFrame (*C++ member*), 791  
 mfxExtCodingOption2::Trellis (*C++ member*), 791  
 mfxExtCodingOption2::UseRawRef (*C++ member*), 792  
 mfxExtCodingOption3 (*C++ struct*), 793  
 mfxExtCodingOption3::AdaptiveMaxFrameSize (*C++ member*), 796  
 mfxExtCodingOption3::AspectRatioInfoPresent (*C++ member*), 794  
 mfxExtCodingOption3::BitstreamRestriction (*C++ member*), 794  
 mfxExtCodingOption3::BRCPanicMode (*C++ member*), 796  
 mfxExtCodingOption3::ContentInfo (*C++ member*), 795  
 mfxExtCodingOption3::DirectBiasAdjustment (*C++ member*), 793  
 mfxExtCodingOption3::EnableMBForceIntra (*C++ member*), 796  
 mfxExtCodingOption3::EnableMBQP (*C++ member*), 793  
 mfxExtCodingOption3::EnableNalUnitType (*C++ member*), 796  
 mfxExtCodingOption3::EnableQPOffset (*C++ member*), 795  
 mfxExtCodingOption3::EncodedUnitsInfo (*C++ member*), 796  
 mfxExtCodingOption3::ExtBrcAdaptiveLTR (*C++ member*), 796  
 mfxExtCodingOption3::FadeDetection (*C++ member*), 795  
 mfxExtCodingOption3::GlobalMotionBiasAdjustment (*C++ member*), 793  
 mfxExtCodingOption3::GPB (*C++ member*), 795  
 mfxExtCodingOption3::Header (*C++ member*), 793  
 mfxExtCodingOption3::IntRefCycleDist (*C++ member*), 793  
 mfxExtCodingOption3::LowDelayBRC (*C++ member*), 796  
 mfxExtCodingOption3::LowDelayHrd (*C++ member*), 794  
 mfxExtCodingOption3::MaxFrameSizeI (*C++ member*), 795  
 mfxExtCodingOption3::MaxFrameSizeP (*C++ member*), 795  
 mfxExtCodingOption3::MBDisableSkipMap (*C++ member*), 794  
 mfxExtCodingOption3::MotionVectorsOverPicBoundaries (*C++ member*), 794  
 mfxExtCodingOption3::MVCostScalingFactor (*C++ member*), 794  
 mfxExtCodingOption3::NumRefActiveBL0 (*C++ member*), 795  
 mfxExtCodingOption3::NumRefActiveBL1 (*C++ member*), 795  
 mfxExtCodingOption3::NumRefActiveP (*C++ member*), 795  
 mfxExtCodingOption3::NumSliceB (*C++ member*), 793  
 mfxExtCodingOption3::NumSliceI (*C++ member*), 793  
 mfxExtCodingOption3::NumSliceP (*C++ member*), 793  
 mfxExtCodingOption3::OverscanAppropriate (*C++ member*), 794  
 mfxExtCodingOption3::OverscanInfoPresent (*C++ member*), 794  
 mfxExtCodingOption3::PRefType (*C++ member*), 795  
 mfxExtCodingOption3::QPOffset (*C++ member*), 795  
 mfxExtCodingOption3::QVBRQuality (*C++ member*), 793  
 mfxExtCodingOption3::RepartitionCheckEnable (*C++ member*), 796  
 mfxExtCodingOption3::reserved (*C++ member*), 796  
 mfxExtCodingOption3::reserved1 (*C++ member*), 794  
 mfxExtCodingOption3::reserved2 (*C++ member*), 795  
 mfxExtCodingOption3::reserved3 (*C++ member*), 795  
 mfxExtCodingOption3::reserved5 (*C++ member*), 796  
 mfxExtCodingOption3::reserved6 (*C++ member*), 795

mfxExtCodingOption3::ScenarioInfo (*C++ member*), 794  
mfxExtCodingOption3::TargetBitDepthChroma (*C++ member*), 796  
mfxExtCodingOption3::TargetBitDepthLuma mfxExtCodingOption3::SingleSeiNalUnit (*C++ member*), 796  
mfxExtCodingOption3::TargetChromaFormatPmfxExtCodingOption3::ResetRefList (*C++ member*), 789  
mfxExtCodingOption3::TransformSkip (*C++ member*), 795  
mfxExtCodingOption3::WeightedBiPred (*C++ member*), 794  
mfxExtCodingOption3::WeightedPred (*C++ member*), 794  
mfxExtCodingOption3::WinBRCMaxAvgKbps (*C++ member*), 793  
mfxExtCodingOption3::WinBRCSize (*C++ member*), 793  
mfxExtCodingOption3::AUDelimiter (*C++ member*), 789  
mfxExtCodingOption3::CAVLC (*C++ member*), 788  
mfxExtCodingOption3::FieldOutput (*C++ member*), 789  
mfxExtCodingOption3::FramePicture (*C++ member*), 788  
mfxExtCodingOption3::Header (*C++ member*), 788  
mfxExtCodingOption3::InterPredBlockSize (*C++ member*), 789  
mfxExtCodingOption3::IntraPredBlockSize (*C++ member*), 789  
mfxExtCodingOption3::MaxDecFrameBuffering (*C++ member*), 789  
mfxExtCodingOption3::MECostType (*C++ member*), 788  
mfxExtCodingOption3::MSESearchType (*C++ member*), 788  
mfxExtCodingOption3::MVPrecision (*C++ member*), 789  
mfxExtCodingOption3::MVSearchWindow (*C++ member*), 788  
mfxExtCodingOption3::NalHrdConformance (*C++ member*), 788  
mfxExtCodingOption3::PicTimingSEI (*C++ member*), 789  
mfxExtCodingOption3::RateDistortionOpt (*C++ member*), 788  
mfxExtCodingOption3::RecoveryPointSEI (*C++ member*), 788  
mfxExtCodingOption3::RefPicListReordering (*C++ member*), 789  
mfxExtCodingOption3::RefPicMarkRep (*C++ member*), 789  
mfxExtCodingOption3::ResetRefList (*C++ member*), 789  
mfxExtCodingOption3::SingleSeiNalUnit (*C++ member*), 788  
mfxExtCodingOption3::ViewOutput (*C++ member*), 788  
mfxExtCodingOption3::VuiNalHrdParameters (*C++ member*), 789  
mfxExtCodingOption3::VuiVclHrdParameters (*C++ member*), 788  
mfxExtCodingOptionSPSPPS (*C++ struct*), 797  
mfxExtCodingOptionSPSPPS::Header (*C++ member*), 797  
mfxExtCodingOptionSPSPPS::PPSBuffer (*C++ member*), 797  
mfxExtCodingOptionSPSPPS::PPSBufSize (*C++ member*), 797  
mfxExtCodingOptionSPSPPS::PPSID (*C++ member*), 797  
mfxExtCodingOptionSPSPPS::SPSBuffer (*C++ member*), 797  
mfxExtCodingOptionSPSPPS::SPSBufSize (*C++ member*), 797  
mfxExtCodingOptionSPSPPS::SPSID (*C++ member*), 797  
mfxExtCodingOptionVPS (*C++ struct*), 798  
mfxExtCodingOptionVPS::Header (*C++ member*), 798  
mfxExtCodingOptionVPS::VPSBuffer (*C++ member*), 798  
mfxExtCodingOptionVPS::VPSBufSize (*C++ member*), 798  
mfxExtCodingOptionVPS::VPSId (*C++ member*), 798  
mfxExtColorConversion (*C++ struct*), 831  
mfxExtColorConversion::ChromaSiting (*C++ member*), 831  
mfxExtColorConversion::Header (*C++ member*), 831  
mfxExtContentLightLevelInfo (*C++ struct*), 801  
mfxExtContentLightLevelInfo::Header (*C++ member*), 802  
mfxExtContentLightLevelInfo::InsertPayloadToggle (*C++ member*), 802  
mfxExtContentLightLevelInfo::MaxContentLightLevel (*C++ member*), 802  
mfxExtContentLightLevelInfo::MaxPicAverageLightLevel (*C++ member*), 802  
mfxExtDecodedFrameInfo (*C++ struct*), 811  
mfxExtDecodedFrameInfo::FrameType (*C++ member*), 811

mfxExtDecodedFrameInfo::Header (*C++ member*), 811  
mfxExtDecodeErrorReport (*C++ struct*), 811  
mfxExtDecodeErrorReport::ErrorTypes (*C++ member*), 811  
mfxExtDecodeErrorReport::Header (*C++ member*), 811  
mfxExtDecVideoProcessing (*C++ struct*), 829  
mfxExtDecVideoProcessing::Header (*C++ member*), 829  
mfxExtDecVideoProcessing::In (*C++ member*), 829  
mfxExtDecVideoProcessing::mfxIn (*C++ struct*), 829  
mfxExtDecVideoProcessing::mfxIn::CropH (*C++ member*), 829  
mfxExtDecVideoProcessing::mfxIn::CropW (*C++ member*), 829  
mfxExtDecVideoProcessing::mfxIn::CropX (*C++ member*), 829  
mfxExtDecVideoProcessing::mfxIn::CropY (*C++ member*), 829  
mfxExtDecVideoProcessing::mfxOut (*C++ struct*), 829  
mfxExtDecVideoProcessing::mfxOut::ChromaFormatEncoderIPCMArea::Area (*C++ member*), 807  
mfxExtDecVideoProcessing::mfxOut::CropH mfxExtEncoderIPCMArea::Bottom (*C++ member*), 807  
mfxExtDecVideoProcessing::mfxOut::CropW mfxExtEncoderIPCMArea::Header (*C++ member*), 807  
mfxExtDecVideoProcessing::mfxOut::CropX mfxExtEncoderIPCMArea::Left (*C++ member*), 807  
mfxExtDecVideoProcessing::mfxOut::CropY mfxExtEncoderIPCMArea::Right (*C++ member*), 807  
mfxExtDecVideoProcessing::mfxOut::FourCC mfxExtEncoderIPCMArea::Top (*C++ member*), 807  
mfxExtDecVideoProcessing::mfxOut::Height mfxExtEncoderResetOption (*C++ struct*), 804  
mfxExtDecVideoProcessing::mfxOut::Width (*C++ member*), 829  
mfxExtDecVideoProcessing::Out (*C++ member*), 829  
mfxExtDirtyRect (*C++ struct*), 813  
mfxExtDirtyRect::Bottom (*C++ member*), 814  
mfxExtDirtyRect::Header (*C++ member*), 814  
mfxExtDirtyRect::Left (*C++ member*), 814  
mfxExtDirtyRect::NumRect (*C++ member*), 814  
mfxExtDirtyRect::Rect (*C++ member*), 814  
mfxExtDirtyRect::Right (*C++ member*), 814  
mfxExtDirtyRect::Top (*C++ member*), 814  
mfxExtEncodedSlicesInfo (*C++ struct*), 823  
mfxExtEncodedSlicesInfo::Header (*C++ member*), 823  
mfxExtEncodedSlicesInfo::NumSliceNonCopiant (*C++ member*), 823  
mfxExtEncodedSlicesInfo::NumSliceSizeAlloc (*C++ member*), 823  
mfxExtEncodedSlicesInfo::SliceSize (*C++ member*), 823  
mfxExtEncodedSlicesInfo::SliceSizeOverflow (*C++ member*), 823  
mfxExtEncodedUnitsInfo (*C++ struct*), 819  
mfxExtEncodedUnitsInfo::Header (*C++ member*), 820  
mfxExtEncodedUnitsInfo::NumUnitsAlloc (*C++ member*), 820  
mfxExtEncodedUnitsInfo::NumUnitsEncoded (*C++ member*), 820  
mfxExtEncodedUnitsInfo::UnitInfo (*C++ member*), 820  
mfxExtEncoderCapability (*C++ struct*), 803  
mfxExtEncoderCapability::Header (*C++ member*), 803  
mfxExtEncoderCapability::MBPerSec (*C++ member*), 803  
mfxExtEncoderIPCMArea (*C++ struct*), 807  
mfxExtEncoderIPCMArea::Area (*C++ member*), 807  
mfxExtEncoderIPCMArea::Bottom (*C++ member*), 807  
mfxExtEncoderIPCMArea::Header (*C++ member*), 807  
mfxExtEncoderIPCMArea::Left (*C++ member*), 807  
mfxExtEncoderIPCMArea::Right (*C++ member*), 807  
mfxExtEncoderIPCMArea::Top (*C++ member*), 807  
mfxExtEncoderResetOption::Header (*C++ member*), 805  
mfxExtEncoderResetOption::StartNewSequence (*C++ member*), 805  
mfxExtEncoderROI (*C++ struct*), 806  
mfxExtEncoderROI::Bottom (*C++ member*), 806  
mfxExtEncoderROI::DeltaQP (*C++ member*), 806  
mfxExtEncoderROI::Header (*C++ member*), 806  
mfxExtEncoderROI::Left (*C++ member*), 806  
mfxExtEncoderROI::NumROI (*C++ member*), 806  
mfxExtEncoderROI::Right (*C++ member*), 806  
mfxExtEncoderROI::ROI (*C++ member*), 806  
mfxExtEncoderROI::ROIMode (*C++ member*), 806  
mfxExtEncoderROI::Top (*C++ member*), 806  
mfxExtHEVCParam (*C++ struct*), 810

mfxExtHEVCParam::GeneralConstraintFlags  
     (*C++ member*), 810

mfxExtHEVCParam::Header (*C++ member*), 810

mfxExtHEVCParam::LCUSize (*C++ member*), 810

mfxExtHEVCParam::PicHeightInLumaSamples  
     (*C++ member*), 810

mfxExtHEVCParam::PicWidthInLumaSamples  
     (*C++ member*), 810

mfxExtHEVCParam::SampleAdaptiveOffset  
     (*C++ member*), 810

mfxExtHEVCRegion (*C++ struct*), 812

mfxExtHEVCRegion::Header (*C++ member*), 812

mfxExtHEVCRegion::RegionEncoding (*C++ member*), 812

mfxExtHEVCRegion::RegionId (*C++ member*), 812

mfxExtHEVCRegion::RegionType (*C++ member*), 812

mfxExtHEVCTiles (*C++ struct*), 809

mfxExtHEVCTiles::Header (*C++ member*), 810

mfxExtHEVCTiles::NumTileColumns (*C++ member*), 810

mfxExtHEVCTiles::NumTileRows (*C++ member*), 810

mfxExtInsertHeaders (*C++ struct*), 797

mfxExtInsertHeaders::Header (*C++ member*), 798

mfxExtInsertHeaders::PPS (*C++ member*), 798

mfxExtInsertHeaders::reserved (*C++ member*), 798

mfxExtInsertHeaders::SPS (*C++ member*), 798

mfxExtJPEGHuffmanTables (*C++ struct*), 837

mfxExtJPEGHuffmanTables::ACTables (*C++ member*), 838

mfxExtJPEGHuffmanTables::Bits (*C++ member*), 838

mfxExtJPEGHuffmanTables::DCTables (*C++ member*), 838

mfxExtJPEGHuffmanTables::Header (*C++ member*), 838

mfxExtJPEGHuffmanTables::NumACTable  
     (*C++ member*), 838

mfxExtJPEGHuffmanTables::NumDCTable  
     (*C++ member*), 838

mfxExtJPEGHuffmanTables::Values (*C++ member*), 838

mfxExtJPEGQuantTables (*C++ struct*), 837

mfxExtJPEGQuantTables::Header (*C++ member*), 837

mfxExtJPEGQuantTables::NumTable (*C++ member*), 837

mfxExtJPEGQuantTables::Qm (*C++ member*), 837

mfxExtMasteringDisplayColourVolume (*C++*  
     *struct*), 801

mfxExtMasteringDisplayColourVolume::DisplayPrimarie  
     (*C++ member*), 801

mfxExtMasteringDisplayColourVolume::DisplayPrimarie  
     (*C++ member*), 801

mfxExtMasteringDisplayColourVolume::Header  
     (*C++ member*), 801

mfxExtMasteringDisplayColourVolume::InsertPayloadTo  
     (*C++ member*), 801

mfxExtMasteringDisplayColourVolume::MaxDisplayMaster  
     (*C++ member*), 801

mfxExtMasteringDisplayColourVolume::MinDisplayMaster  
     (*C++ member*), 801

mfxExtMasteringDisplayColourVolume::WhitePointX  
     (*C++ member*), 801

mfxExtMasteringDisplayColourVolume::WhitePointY  
     (*C++ member*), 801

mfxExtMBDisableSkipMap (*C++ struct*), 810

mfxExtMBDisableSkipMap::Header (*C++ mem-  
     ber*), 810

mfxExtMBDisableSkipMap::Map (*C++ member*), 810

mfxExtMBDisableSkipMap::MapSize (*C++  
     member*), 810

mfxExtMBForceIntra (*C++ struct*), 808

mfxExtMBForceIntra::Header (*C++ member*), 808

mfxExtMBForceIntra::Map (*C++ member*), 808

mfxExtMBForceIntra::MapSize (*C++ member*), 808

mfxExtMBQP (*C++ struct*), 809

mfxExtMBQP::BlockSize (*C++ member*), 809

mfxExtMBQP::DeltaQP (*C++ member*), 809

mfxExtMBQP::Header (*C++ member*), 809

mfxExtMBQP::Mode (*C++ member*), 809

mfxExtMBQP::NumQPAlloc (*C++ member*), 809

mfxExtMBQP::QP (*C++ member*), 809

mfxExtMBQP::QPmode (*C++ member*), 809

mfxExtMoveRect (*C++ struct*), 814

mfxExtMoveRect::DestBottom (*C++ member*), 814

mfxExtMoveRect::DestLeft (*C++ member*), 814

mfxExtMoveRect::DestRight (*C++ member*), 814

mfxExtMoveRect::DestTop (*C++ member*), 814

mfxExtMoveRect::Header (*C++ member*), 814

mfxExtMoveRect::NumRect (*C++ member*), 814

mfxExtMoveRect::Rect (*C++ member*), 814

mfxExtMoveRect::SourceLeft (*C++ member*), 814

mfxExtMoveRect::SourceTop (*C++ member*), 814

mfxExtMVCSeqDesc (*C++ struct*), 839

mfxExtMVCSeqDesc::Header (*C++ member*), 839

mfxExtMVCSeqDesc::NumOP ( <i>C++ member</i> ), 840	( <i>C++ member</i> ), 802
mfxExtMVCSeqDesc::NumOPAlloc ( <i>C++ member</i> ), 840	mfxExtPictureTimingSEI::FullTimestampFlag ( <i>C++ member</i> ), 802
mfxExtMVCSeqDesc::NumRefsTotal ( <i>C++ member</i> ), 840	mfxExtPictureTimingSEI::Header ( <i>C++ member</i> ), 802
mfxExtMVCSeqDesc::NumView ( <i>C++ member</i> ), 839	mfxExtPictureTimingSEI::HoursFlag ( <i>C++ member</i> ), 802
mfxExtMVCSeqDesc::NumViewAlloc ( <i>C++ member</i> ), 839	mfxExtPictureTimingSEI::HoursValue ( <i>C++ member</i> ), 803
mfxExtMVCSeqDesc::NumViewId ( <i>C++ member</i> ), 839	mfxExtPictureTimingSEI::MinutesFlag ( <i>C++ member</i> ), 802
mfxExtMVCSeqDesc::NumViewIdAlloc ( <i>C++ member</i> ), 840	mfxExtPictureTimingSEI::MinutesValue ( <i>C++ member</i> ), 803
mfxExtMVCSeqDesc::OP ( <i>C++ member</i> ), 840	mfxExtPictureTimingSEI::NFrames ( <i>C++ member</i> ), 802
mfxExtMVCSeqDesc::View ( <i>C++ member</i> ), 839	mfxExtPictureTimingSEI::NuitFieldBasedFlag ( <i>C++ member</i> ), 802
mfxExtMVCSeqDesc::ViewId ( <i>C++ member</i> ), 840	mfxExtPictureTimingSEI::reserved ( <i>C++ member</i> ), 802
mfxExtMVCTargetViews ( <i>C++ struct</i> ), 840	mfxExtPictureTimingSEI::SecondsFlag ( <i>C++ member</i> ), 802
mfxExtMVCTargetViews::Header ( <i>C++ member</i> ), 840	mfxExtPictureTimingSEI::SecondsValue ( <i>C++ member</i> ), 802
mfxExtMVCTargetViews::NumView ( <i>C++ member</i> ), 840	mfxExtPictureTimingSEI::TimeOffset ( <i>C++ member</i> ), 803
mfxExtMVCTargetViews::TemporalId ( <i>C++ member</i> ), 840	mfxExtPictureTimingSEI::TimeStamp ( <i>C++ member</i> ), 803
mfxExtMVCTargetViews::ViewId ( <i>C++ member</i> ), 840	mfxExtPredWeightTable ( <i>C++ struct</i> ), 812
mfxExtMVOVerPicBoundaries ( <i>C++ struct</i> ), 815	mfxExtPredWeightTable::ChromaLog2WeightDenom ( <i>C++ member</i> ), 812
mfxExtMVOVerPicBoundaries::Header ( <i>C++ member</i> ), 815	mfxExtPredWeightTable::ChromaWeightFlag ( <i>C++ member</i> ), 812
mfxExtMVOVerPicBoundaries::StickBottom ( <i>C++ member</i> ), 815	mfxExtPredWeightTable::Header ( <i>C++ member</i> ), 812
mfxExtMVOVerPicBoundaries::StickLeft ( <i>C++ member</i> ), 815	mfxExtPredWeightTable::LumaLog2WeightDenom ( <i>C++ member</i> ), 812
mfxExtMVOVerPicBoundaries::StickRight ( <i>C++ member</i> ), 815	mfxExtPredWeightTable::LumaWeightFlag ( <i>C++ member</i> ), 812
mfxExtMVOVerPicBoundaries::StickTop ( <i>C++ member</i> ), 815	mfxExtPredWeightTable::Weights ( <i>C++ member</i> ), 813
mfxExtPartialBitstreamParam ( <i>C++ struct</i> ), 820	mfxExtThreadsParam ( <i>C++ struct</i> ), 798
mfxExtPartialBitstreamParam::BlockSize ( <i>C++ member</i> ), 820	mfxExtThreadsParam::Header ( <i>C++ member</i> ), 799
mfxExtPartialBitstreamParam::Granularity ( <i>C++ member</i> ), 820	mfxExtThreadsParam::NumThread ( <i>C++ member</i> ), 799
mfxExtPartialBitstreamParam::Header ( <i>C++ member</i> ), 820	mfxExtThreadsParam::Priority ( <i>C++ member</i> ), 799
mfxExtPictureTimingSEI ( <i>C++ struct</i> ), 802	mfxExtThreadsParam::reserved ( <i>C++ member</i> ), 799
mfxExtPictureTimingSEI::ClockTimestampFlag ( <i>C++ member</i> ), 802	mfxExtThreadsParam::SchedulingType ( <i>C++ member</i> ), 799
mfxExtPictureTimingSEI::CntDroppedFlag ( <i>C++ member</i> ), 802	mfxExtTimeCode ( <i>C++ struct</i> ), 811
mfxExtPictureTimingSEI::CountingType ( <i>C++ member</i> ), 802	mfxExtTimeCode::DropFrameFlag ( <i>C++ member</i> ), 811
mfxExtPictureTimingSEI::CtType ( <i>C++ member</i> ), 802	
mfxExtPictureTimingSEI::DiscontinuityFlag	

mfxExtTimeCode::Header (*C++ member*), 811  
mfxExtTimeCode::TimeCodeHours (*C++ member*), 811  
mfxExtTimeCode::TimeCodeMinutes (*C++ member*), 811  
mfxExtTimeCode::TimeCodePictures (*C++ member*), 812  
mfxExtTimeCode::TimeCodeSeconds (*C++ member*), 812  
mfxExtVideoSignalInfo (*C++ struct*), 799  
mfxExtVideoSignalInfo::ColourDescriptionPresent (*C++ member*), 799  
mfxExtVideoSignalInfo::ColourPrimaries (*C++ member*), 799  
mfxExtVideoSignalInfo::Header (*C++ member*), 799  
mfxExtVideoSignalInfo::MatrixCoefficients (*C++ member*), 799  
mfxExtVideoSignalInfo::TransferCharacteristics (*C++ member*), 799  
mfxExtVideoSignalInfo::VideoFormat (*C++ member*), 799  
mfxExtVideoSignalInfo::VideoFullRange (*C++ member*), 799  
mfxExtVP8CodingOption (*C++ struct*), 836  
mfxExtVP8CodingOption::CoeffTypeQPDelta (*C++ member*), 836  
mfxExtVP8CodingOption::EnableMultipleSegments (*C++ member*), 836  
mfxExtVP8CodingOption::Header (*C++ member*), 836  
mfxExtVP8CodingOption::LoopFilterLevel (*C++ member*), 836  
mfxExtVP8CodingOption::LoopFilterMbModeDRAFTVppAuxData (*C++ struct*), 836  
mfxExtVP8CodingOption::LoopFilterRefTypeDRAFTVppAuxData::PicStruct (*C++ member*), 836  
mfxExtVP8CodingOption::LoopFilterType (*C++ member*), 836  
mfxExtVP8CodingOption::NumFramesForIVFHeader (*C++ member*), 837  
mfxExtVP8CodingOption::NumTokenPartitions (*C++ member*), 836  
mfxExtVP8CodingOption::SegmentQPDelta (*C++ member*), 836  
mfxExtVP8CodingOption::SharpnessLevel (*C++ member*), 836  
mfxExtVP8CodingOption::Version (*C++ member*), 836  
mfxExtVP8CodingOption::WriteIVFHeaders (*C++ member*), 837  
mfxExtVP9Param (*C++ struct*), 818  
mfxExtVP9Param::FrameHeight (*C++ member*), 818  
mfxExtVP9Param::FrameWidth (*C++ member*), 818  
mfxExtVP9Param::Header (*C++ member*), 818  
mfxExtVP9Param::NumTileColumns (*C++ member*), 818  
mfxExtVP9Param::NumTileRows (*C++ member*), 818  
mfxExtVP9Param::QIndexDeltaChromaAC (*C++ member*), 818  
mfxExtVP9Param::QIndexDeltaChromaDC  
mfxExtVP9Param::QIndexDeltaLumaDC (*C++ member*), 818  
mfxExtVP9Param::WriteIVFHeaders (*C++ member*), 818  
mfxExtVP9Segmentation (*C++ struct*), 816  
mfxExtVP9Segmentation::Header (*C++ member*), 816  
mfxExtVP9Segmentation::NumSegmentIdAlloc (*C++ member*), 816  
mfxExtVP9Segmentation::NumSegments (*C++ member*), 816  
mfxExtVP9Segmentation::Segment (*C++ member*), 816  
mfxExtVP9Segmentation::SegmentId (*C++ member*), 816  
mfxExtVP9Segmentation::SegmentIdBlockSize (*C++ member*), 816  
mfxExtVP9TemporalLayers (*C++ struct*), 817  
mfxExtVP9TemporalLayers::Header (*C++ member*), 817  
mfxExtVP9TemporalLayers::Layer (*C++ member*), 817  
mfxExtVppAuxData (*C++ struct*), 824  
mfxExtVppAuxData::Header (*C++ member*), 824  
mfxExtVppAuxData::PicStruct (*C++ member*), 824  
mfxExtVPPColorFill (*C++ struct*), 831  
mfxExtVPPColorFill::Enable (*C++ member*), 831  
mfxExtVPPColorFill::Header (*C++ member*), 831  
mfxExtVPPComposite (*C++ struct*), 826  
mfxExtVPPComposite::B (*C++ member*), 827  
mfxExtVPPComposite::G (*C++ member*), 827  
mfxExtVPPComposite::Header (*C++ member*), 827  
mfxExtVPPComposite::InputStream (*C++ member*), 828  
mfxExtVPPComposite::NumInputStream (*C++ member*), 827  
mfxExtVPPComposite::NumTiles (*C++ member*), 827  
mfxExtVPPComposite::R (*C++ member*), 827

mfxExtVPPComposite::U (*C++ member*), 827  
 mfxExtVPPComposite::V (*C++ member*), 827  
 mfxExtVPPComposite::Y (*C++ member*), 827  
 mfxExtVPPDeinterlacing (*C++ struct*), 823  
 mfxExtVPPDeinterlacing::Header (*C++ member*), 823  
 mfxExtVPPDeinterlacing::Mode (*C++ member*), 823  
 mfxExtVPPDeinterlacing::reserved (*C++ member*), 823  
 mfxExtVPPDeinterlacing::TelecineLocation (*C++ member*), 823  
 mfxExtVPPDeinterlacing::TelecinePattern (*C++ member*), 823  
 mfxExtVPPDenoise (*C++ struct*), 821  
 mfxExtVPPDenoise::DenoiseFactor (*C++ member*), 821  
 mfxExtVPPDenoise::Header (*C++ member*), 821  
 mfxExtVPPDetail (*C++ struct*), 822  
 mfxExtVPPDetail::DetailFactor (*C++ member*), 822  
 mfxExtVPPDetail::Header (*C++ member*), 822  
 mfxExtVPPDoNotUse (*C++ struct*), 820  
 mfxExtVPPDoNotUse::AlgList (*C++ member*), 821  
 mfxExtVPPDoNotUse::Header (*C++ member*), 821  
 mfxExtVPPDoNotUse::NumAlg (*C++ member*), 821  
 mfxExtVPPDoUse (*C++ struct*), 821  
 mfxExtVPPDoUse::AlgList (*C++ member*), 821  
 mfxExtVPPDoUse::Header (*C++ member*), 821  
 mfxExtVPPDoUse::NumAlg (*C++ member*), 821  
 mfxExtVPPFieldProcessing (*C++ struct*), 828  
 mfxExtVPPFieldProcessing::Header (*C++ member*), 828  
 mfxExtVPPFieldProcessing::InField (*C++ member*), 828  
 mfxExtVPPFieldProcessing::Mode (*C++ member*), 828  
 mfxExtVPPFieldProcessing::OutField (*C++ member*), 828  
 mfxExtVPPFrameRateConversion (*C++ struct*), 824  
 mfxExtVPPFrameRateConversion::Algorithm (*C++ member*), 824  
 mfxExtVPPFrameRateConversion::Header (*C++ member*), 824  
 mfxExtVPPImageStab (*C++ struct*), 825  
 mfxExtVPPImageStab::Header (*C++ member*), 825  
 mfxExtVPPImageStab::Mode (*C++ member*), 825  
 mfxExtVppMctf (*C++ struct*), 832  
 mfxExtVppMctf::FilterStrength (*C++ member*), 832  
 ber), 832  
 mfxExtVppMctf::Header (*C++ member*), 832  
 mfxExtVPPMirroring (*C++ struct*), 831  
 mfxExtVPPMirroring::Header (*C++ member*), 831  
 mfxExtVPPMirroring::Type (*C++ member*), 831  
 mfxExtVPPProcAmp (*C++ struct*), 822  
 mfxExtVPPProcAmp::Brightness (*C++ member*), 822  
 mfxExtVPPProcAmp::Contrast (*C++ member*), 822  
 mfxExtVPPProcAmp::Header (*C++ member*), 822  
 mfxExtVPPProcAmp::Hue (*C++ member*), 822  
 mfxExtVPPProcAmp::Saturation (*C++ member*), 822  
 mfxExtVPPRotation (*C++ struct*), 830  
 mfxExtVPPRotation::Angle (*C++ member*), 830  
 mfxExtVPPRotation::Header (*C++ member*), 830  
 mfxExtVPPScaling (*C++ struct*), 830  
 mfxExtVPPScaling::Header (*C++ member*), 830  
 mfxExtVPPScaling::InterpolationMethod (*C++ member*), 830  
 mfxExtVPPScaling::ScalingMode (*C++ member*), 830  
 mfxExtVPPVideoSignalInfo (*C++ struct*), 828  
 mfxExtVPPVideoSignalInfo::Header (*C++ member*), 828  
 mfxExtVPPVideoSignalInfo::NominalRange (*C++ member*), 828  
 mfxExtVPPVideoSignalInfo::TransferMatrix (*C++ member*), 828  
 mfxF32 (*C++ type*), 712  
 mfxF64 (*C++ type*), 712  
 mfxFrameAllocator (*C++ struct*), 783  
 mfxFrameAllocator::Alloc (*C++ member*), 784  
 mfxFrameAllocator::Free (*C++ member*), 785  
 mfxFrameAllocator::GetHDL (*C++ member*), 784  
 mfxFrameAllocator::Lock (*C++ member*), 784  
 mfxFrameAllocator::pthis (*C++ member*), 784  
 mfxFrameAllocator::Unlock (*C++ member*), 784  
 mfxFrameAllocRequest (*C++ struct*), 783  
 mfxFrameAllocRequest::AllocId (*C++ member*), 783  
 mfxFrameAllocRequest::Info (*C++ member*), 783  
 mfxFrameAllocRequest::NumFrameMin (*C++ member*), 783  
 mfxFrameAllocRequest::NumFrameSuggested (*C++ member*), 783  
 mfxFrameAllocRequest::Type (*C++ member*), 783

mfxFrameAllocResponse (*C++ struct*), 783  
 mfxFrameAllocResponse::AllocId (*C++ member*), 783  
 mfxFrameAllocResponse::mids (*C++ member*), 783  
 mfxFrameAllocResponse::NumFrameActual (*C++ member*), 783  
 mfxFrameData (*C++ struct*), 773  
 mfxFrameData::A (*C++ member*), 774  
 mfxFrameData::A2RGB10 (*C++ member*), 775  
 mfxFrameData::B (*C++ member*), 775  
 mfxFrameData::Cb (*C++ member*), 775  
 mfxFrameData::CbCr (*C++ member*), 775  
 mfxFrameData::Corrupted (*C++ member*), 774  
 mfxFrameData::Cr (*C++ member*), 775  
 mfxFrameData::CrCb (*C++ member*), 775  
 mfxFrameData::DataFlag (*C++ member*), 774  
 mfxFrameData::ExtParam (*C++ member*), 775  
 mfxFrameData::FrameOrder (*C++ member*), 774  
 mfxFrameData::G (*C++ member*), 775  
 mfxFrameData::Locked (*C++ member*), 774  
 mfxFrameData::MemId (*C++ member*), 774  
 mfxFrameData::MemType (*C++ member*), 774  
 mfxFrameData::NumExtParam (*C++ member*), 774  
 mfxFrameData::PitchHigh (*C++ member*), 774  
 mfxFrameData::PitchLow (*C++ member*), 775  
 mfxFrameData::R (*C++ member*), 775  
 mfxFrameData::reserved (*C++ member*), 774  
 mfxFrameData::TimeStamp (*C++ member*), 774  
 mfxFrameData::U (*C++ member*), 775  
 mfxFrameData::U16 (*C++ member*), 775  
 mfxFrameData::UV (*C++ member*), 775  
 mfxFrameData::V (*C++ member*), 775  
 mfxFrameData::V16 (*C++ member*), 775  
 mfxFrameData::VU (*C++ member*), 775  
 mfxFrameData::Y (*C++ member*), 775  
 mfxFrameData::Y16 (*C++ member*), 775  
 mfxFrameData::Y410 (*C++ member*), 775  
 mfxFrameId (*C++ struct*), 769  
 mfxFrameId::DependencyId (*C++ member*), 769  
 mfxFrameId::PriorityId (*C++ member*), 769  
 mfxFrameId::QualityId (*C++ member*), 769  
 mfxFrameId::TemporalId (*C++ member*), 769  
 mfxFrameId::ViewId (*C++ member*), 769  
 mfxFrameInfo (*C++ struct*), 769  
 mfxFrameInfo::AspectRatioH (*C++ member*), 770  
 mfxFrameInfo::AspectRatioW (*C++ member*), 770  
 mfxFrameInfo::BitDepthChroma (*C++ member*), 771  
 mfxFrameInfo::BitDepthLuma (*C++ member*), 771  
 mfxFrameInfo::BufferSize (*C++ member*), 771  
 mfxFrameInfo::ChromaFormat (*C++ member*), 771  
 mfxFrameInfo::CropH (*C++ member*), 770  
 mfxFrameInfo::CropW (*C++ member*), 770  
 mfxFrameInfo::CropX (*C++ member*), 770  
 mfxFrameInfo::CropY (*C++ member*), 770  
 mfxFrameInfo::FourCC (*C++ member*), 771  
 mfxFrameInfo::FrameId (*C++ member*), 771  
 mfxFrameInfo::FrameRateExtD (*C++ member*), 770  
 mfxFrameInfo::FrameRateExtN (*C++ member*), 770  
 mfxFrameInfo::Height (*C++ member*), 771  
 mfxFrameInfo::PicStruct (*C++ member*), 771  
 mfxFrameInfo::reserved (*C++ member*), 771  
 mfxFrameInfo::reserved4 (*C++ member*), 771  
 mfxFrameInfo::Shift (*C++ member*), 771  
 mfxFrameInfo::Width (*C++ member*), 771  
 mfxFrameSurface1 (*C++ struct*), 779  
 mfxFrameSurface1::Data (*C++ member*), 779  
 mfxFrameSurface1::FrameInterface (*C++ member*), 779  
 mfxFrameSurface1::Info (*C++ member*), 779  
 mfxFrameSurfaceInterface (*C++ struct*), 776  
 mfxFrameSurfaceInterface::AddRef (*C++ member*), 776  
 mfxFrameSurfaceInterface::Context (*C++ member*), 776  
 mfxFrameSurfaceInterface::GetDeviceHandle (*C++ member*), 778  
 mfxFrameSurfaceInterface::GetNativeHandle (*C++ member*), 777  
 mfxFrameSurfaceInterface::GetRefCounter (*C++ member*), 776  
 mfxFrameSurfaceInterface::Map (*C++ member*), 777  
 mfxFrameSurfaceInterface::Release (*C++ member*), 776  
 mfxFrameSurfaceInterface::Synchronize (*C++ member*), 778  
 mfxFrameSurfaceInterface::Unmap (*C++ member*), 777  
 mfxFrameSurfaceInterface::Version (*C++ member*), 776  
 MFXGetPriority (*C++ function*), 844  
 mfxHandleType (*C++ enum*), 752  
 mfxHandleType::MFX\_HANDLE\_D3D11\_DEVICE (*C++ enumerator*), 752  
 mfxHandleType::MFX\_HANDLE\_D3D9\_DEVICE\_MANAGER (*C++ enumerator*), 752  
 mfxHandleType::MFX\_HANDLE\_DIRECT3D\_DEVICE\_MANAGER9 (*C++ enumerator*), 752  
 mfxHandleType::MFX\_HANDLE\_RESERVED1

mfxHandleType::MFX_HANDLE_RESERVED3 (C++ enumerator), 752	720
mfxHandleType::MFX_HANDLE_VA_CONFIG_ID (C++ enumerator), 752	mfxImplDescription::VPP (C++ member), 721
mfxHandleType::MFX_HANDLE_VA_CONTEXT_ID (C++ enumerator), 753	mfxImplType (C++ enum), 713
mfxHandleType::MFX_HANDLE_VA_DISPLAY (C++ enumerator), 752	mfxImplType::MFX_IMPL_TYPE_HARDWARE (C++ enumerator), 713
mfxHDL (C++ type), 712	mfxImplType::MFX_IMPL_TYPE_SOFTWARE (C++ enumerator), 713
mfxHDPair (C++ struct), 763	mfxInfoMFX (C++ struct), 765
mfxHDPair::first (C++ member), 763	mfxInfoMFX::Accuracy (C++ member), 767
mfxHDPair::second (C++ member), 763	mfxInfoMFX::BRCPParamMultiplier (C++ member), 765
mfxI16 (C++ type), 712	mfxInfoMFX::BufferSizeInKB (C++ member), 767
mfxI16Pair (C++ struct), 763	mfxInfoMFX::CodecId (C++ member), 765
mfxI16Pair::x (C++ member), 763	mfxInfoMFX::CodecLevel (C++ member), 765
mfxI16Pair::y (C++ member), 763	mfxInfoMFX::CodecProfile (C++ member), 765
mfxI32 (C++ type), 712	mfxInfoMFX::Convergence (C++ member), 767
mfxI64 (C++ type), 712	mfxInfoMFX::DecodedOrder (C++ member), 767
mfxI8 (C++ type), 712	mfxInfoMFX::EnableReallocRequest (C++ member), 768
mfxIMPL (C++ type), 727	mfxInfoMFX::EncodedOrder (C++ member), 767
mfxImplCapsDeliveryFormat (C++ enum), 728	mfxInfoMFX::ExtendedPicStruct (C++ member), 768
mfxImplCapsDeliveryFormat::MFX_IMPLCAPS_IMPLDESCSTRUCTURE (C++ enumerator), 728	mfxInfoMFX::FrameInfo (C++ member), 765
mfxImplDescription (C++ struct), 720	mfxInfoMFX::GopOptFlag (C++ member), 766
mfxImplDescription::AccelerationMode (C++ member), 720	mfxInfoMFX::GopPicSize (C++ member), 766
mfxImplDescription::ApiVersion (C++ member), 720	mfxInfoMFX::GopRefDist (C++ member), 766
mfxImplDescription::Dec (C++ member), 721	mfxInfoMFX::ICQQuality (C++ member), 767
mfxImplDescription::Dev (C++ member), 721	mfxInfoMFX::IdrInterval (C++ member), 766
mfxImplDescription::Enc (C++ member), 721	mfxInfoMFX::InitialDelayInKB (C++ member), 766
mfxImplDescription::ExtParam (C++ member), 721	mfxInfoMFX::Interleaved (C++ member), 768
mfxImplDescription::ExtParams (C++ member), 721	mfxInfoMFX::InterleavedDec (C++ member), 768
mfxImplDescription::Impl (C++ member), 720	mfxInfoMFX::JPEGChromaFormat (C++ member), 768
mfxImplDescription::ImplName (C++ member), 720	mfxInfoMFX::JPEGColorFormat (C++ member), 768
mfxImplDescription::Keywords (C++ member), 720	mfxInfoMFX::LowPower (C++ member), 765
mfxImplDescription::License (C++ member), 720	mfxInfoMFX::MaxDecFrameBuffering (C++ member), 768
mfxImplDescription::NumExtParam (C++ member), 721	mfxInfoMFX::MaxKbps (C++ member), 767
mfxImplDescription::reserved (C++ member), 721	mfxInfoMFX::NumRefFrame (C++ member), 767
mfxImplDescription::Reserved2 (C++ member), 721	mfxInfoMFX::NumSlice (C++ member), 767
mfxImplDescription::VendorID (C++ member), 721	mfxInfoMFX::QPB (C++ member), 767
mfxImplDescription::VendorImplID (C++ member), 721	mfxInfoMFX::QPI (C++ member), 767
mfxImplDescription::Version (C++ member), 721	mfxInfoMFX::QPP (C++ member), 767
	mfxInfoMFX::Quality (C++ member), 768
	mfxInfoMFX::reserved (C++ member), 765
	mfxInfoMFX::RestartInterval (C++ member), 768
	mfxInfoMFX::Rotation (C++ member), 768
	mfxInfoMFX::SamplingFactorH (C++ member), 768

mfxInfoMFX::SamplingFactorV (*C++ member*), 768  
mfxInfoMFX::SliceGroupsPresent (*C++ member*), 768  
mfxInfoMFX::TargetKbps (*C++ member*), 767  
mfxInfoMFX::TargetUsage (*C++ member*), 766  
mfxInfoMFX::TimeStampCalc (*C++ member*), 768  
mfxInfoVPP (*C++ struct*), 786  
mfxInfoVPP::In (*C++ member*), 787  
mfxInfoVPP::Out (*C++ member*), 787  
MFXInit (*C++ function*), 842  
MFXInitEx (*C++ function*), 842  
mfxInitParam (*C++ struct*), 764  
mfxInitParam::ExternalThreads (*C++ member*), 765  
mfxInitParam::ExtParam (*C++ member*), 765  
mfxInitParam::GPUCopy (*C++ member*), 765  
mfxInitParam::Implementation (*C++ member*), 765  
mfxInitParam::NumExtParam (*C++ member*), 765  
mfxInitParam::Version (*C++ member*), 765  
mfxInitParam::[anonymous] (*C++ member*), 765  
MFXJoinSession (*C++ function*), 843  
mfxL32 (*C++ type*), 712  
MFXLoad (*C++ function*), 721  
mfxLoader (*C++ type*), 713  
mfxMediaAdapterType (*C++ enum*), 729  
mfxMediaAdapterType::MFX\_MEDIA\_DISCRETE (*C++ enumerator*), 729  
mfxMediaAdapterType::MFX\_MEDIA\_INTEGRATED (*C++ enumerator*), 729  
mfxMediaAdapterType::MFX\_MEDIA\_UNKNOWN (*C++ enumerator*), 729  
mfxMemId (*C++ type*), 712  
MFXMemory\_GetSurfaceForDecode (*C++ function*), 847  
MFXMemory\_GetSurfaceForEncode (*C++ function*), 846  
MFXMemory\_GetSurfaceForVPP (*C++ function*), 846  
mfxMemoryFlags (*C++ enum*), 729  
mfxMemoryFlags::MFX\_MAP\_NOWAIT (*C++ enumerator*), 729  
mfxMemoryFlags::MFX\_MAP\_READ (*C++ enumerator*), 729  
mfxMemoryFlags::MFX\_MAP\_READ\_WRITE (*C++ enumerator*), 729  
mfxMemoryFlags::MFX\_MAP\_WRITE (*C++ enumerator*), 729  
mfxMVCOperationPoint (*C++ struct*), 839  
mfxMVCOperationPoint::LevelIdc (*C++ member*), 839  
mfxMVCOperationPoint::NumTargetViews (*C++ member*), 839  
mfxMVCOperationPoint::NumViews (*C++ member*), 839  
mfxMVCOperationPoint::TargetViewId (*C++ member*), 839  
mfxMVCOperationPoint::TemporalId (*C++ member*), 839  
mfxMVCViewDependency (*C++ struct*), 838  
mfxMVCViewDependency::AnchorRefL0 (*C++ member*), 838  
mfxMVCViewDependency::AnchorRefL1 (*C++ member*), 838  
mfxMVCViewDependency::NonAnchorRefL0 (*C++ member*), 839  
mfxMVCViewDependency::NumAnchorRefsL0 (*C++ member*), 838  
mfxMVCViewDependency::NumAnchorRefsL1 (*C++ member*), 838  
mfxMVCViewDependency::NumNonAnchorRefsL0 (*C++ member*), 838  
mfxMVCViewDependency::NumNonAnchorRefsL1 (*C++ member*), 839  
mfxMVCViewDependency::ViewId (*C++ member*), 838  
mfxPayload (*C++ struct*), 781  
mfxPayload::BufSize (*C++ member*), 781  
mfxPayload::CtrlFlags (*C++ member*), 781  
mfxPayload::Data (*C++ member*), 781  
mfxPayload::NumBit (*C++ member*), 781  
mfxPayload::Type (*C++ member*), 781  
mfxPlatform (*C++ struct*), 764  
mfxPlatform::CodeName (*C++ member*), 764  
mfxPlatform::DeviceId (*C++ member*), 764  
mfxPlatform::MediaAdapterType (*C++ member*), 764  
mfxPlatform::reserved (*C++ member*), 764  
mfxPriority (*C++ enum*), 728  
mfxPriority::MFX\_PRIORITY\_HIGH (*C++ enumerator*), 728  
mfxPriority::MFX\_PRIORITY\_LOW (*C++ enumerator*), 728  
mfxPriority::MFX\_PRIORITY\_NORMAL (*C++ enumerator*), 728  
mfxQPandMode (*C++ struct*), 786  
mfxQPandMode::DeltaQP (*C++ member*), 786  
mfxQPandMode::Mode (*C++ member*), 786  
mfxQPandMode::QP (*C++ member*), 786  
MFXQueryAdapters (*C++ function*), 859  
MFXQueryAdaptersDecode (*C++ function*), 859  
MFXQueryAdaptersNumber (*C++ function*), 860  
MFXQueryIMPL (*C++ function*), 843

MFXQueryImplsDescription (*C++ function*), 841  
MFXQueryVersion (*C++ function*), 843  
mfxRange32U (*C++ struct*), 762  
mfxRange32U::Max (*C++ member*), 762  
mfxRange32U::Min (*C++ member*), 762  
mfxRange32U::Step (*C++ member*), 762  
MFXReleaseImplDescription (*C++ function*), 841  
mfxResourceType (*C++ enum*), 730  
mfxResourceType::MFX\_RESOURCE\_DMA\_RESOURCE (C++ Status), 725  
mfxResourceType::MFX\_RESOURCE\_DX11\_TEXTURE (C++ Status), 725  
mfxResourceType::MFX\_RESOURCE\_DX12\_RESOURCE (C++ Status), 725  
mfxResourceType::MFX\_RESOURCE\_DX9\_SURFACE (C++ Status), 725  
mfxResourceType::MFX\_RESOURCE\_SYSTEM\_SURFACE (C++ Status), 725  
mfxResourceType::MFX\_RESOURCE\_VA\_BUFFER (C++ Status), 725  
mfxResourceType::MFX\_RESOURCE\_VA\_SURFACE (C++ Status), 725  
mfxSession (*C++ type*), 713  
MFXSetConfigFilterProperty (*C++ function*), 722  
MFXSetPriority (*C++ function*), 844  
mfxSkipMode (*C++ enum*), 753  
mfxSkipMode::MFX\_SKIPMODE\_LESS (*C++ enumerator*), 753  
mfxSkipMode::MFX\_SKIPMODE\_MORE (*C++ enumerator*), 753  
mfxSkipMode::MFX\_SKIPMODE\_NOSKIP (*C++ enumerator*), 753  
mfxStatus (*C++ enum*), 725  
mfxStatus::MFX\_ERR\_ABORTED (*C++ enumerator*), 725  
mfxStatus::MFX\_ERR\_DEVICE\_FAILED (*C++ enumerator*), 725  
mfxStatus::MFX\_ERR\_DEVICE\_LOST (*C++ enumerator*), 725  
mfxStatus::MFX\_ERR\_GPU\_HANG (*C++ enumerator*), 726  
mfxStatus::MFX\_ERR\_INCOMPATIBLE\_VIDEO\_PARAM (*C++ enumerator*), 725  
mfxStatus::MFX\_ERR\_INVALID\_HANDLE (*C++ enumerator*), 725  
mfxStatus::MFX\_ERR\_INVALID\_VIDEO\_PARAM (*C++ enumerator*), 725  
mfxStatus::MFX\_ERR\_LOCK\_MEMORY (*C++ enumerator*), 725  
mfxStatus::MFX\_ERR\_MEMORY\_ALLOC (*C++ enumerator*), 725  
mfxStatus::MFX\_ERR\_MORE\_BITSTREAM (*C++ enumerator*), 725  
mfxStatus::MFX\_ERR\_MORE\_DATA (*C++ enumerator*), 725  
mfxStatus::MFX\_ERR\_MORE\_DATA\_SUBMIT\_TASK (*C++ enumerator*), 726  
mfxStatus::MFX\_ERR\_MORE\_SURFACE (*C++ enumerator*), 725  
mfxStatus::MFX\_ERR\_NONE (*C++ enumerator*), 725  
mfxStatus::MFX\_ERR\_NONE\_PARTIAL\_OUTPUT (*C++ enumerator*), 726  
mfxStatus::MFX\_ERR\_NOT\_ENOUGH\_BUFFER (*C++ enumerator*), 725  
mfxStatus::MFX\_ERR\_NOT\_FOUND (*C++ enumerator*), 725  
mfxStatus::MFX\_ERR\_NOT\_IMPLEMENTED (*C++ enumerator*), 726  
mfxStatus::MFX\_ERR\_NOT\_INITIALIZED (*C++ enumerator*), 725  
mfxStatus::MFX\_ERR\_NULL\_PTR (*C++ enumerator*), 725  
mfxStatus::MFX\_ERR\_REALLOC\_SURFACE (*C++ enumerator*), 726  
mfxStatus::MFX\_ERR\_RESOURCE\_MAPPED (*C++ enumerator*), 726  
mfxStatus::MFX\_ERR\_UNDEFINED\_BEHAVIOR (*C++ enumerator*), 725  
mfxStatus::MFX\_ERR\_UNKNOWN (*C++ enumerator*), 725  
mfxStatus::MFX\_ERR\_UNSUPPORTED (*C++ enumerator*), 725  
mfxStatus::MFX\_TASK\_BUSY (*C++ enumerator*), 726  
mfxStatus::MFX\_TASK\_DONE (*C++ enumerator*), 726  
mfxStatus::MFX\_TASK\_WORKING (*C++ enumerator*), 726  
mfxStatus::MFX\_WRN\_DEVICE\_BUSY (*C++ enumerator*), 726  
mfxStatus::MFX\_WRN\_FILTER\_SKIPPED (*C++ enumerator*), 726  
mfxStatus::MFX\_WRN\_IN\_EXECUTION (*C++ enumerator*), 726  
mfxStatus::MFX\_WRN\_INCOMPATIBLE\_VIDEO\_PARAM (*C++ enumerator*), 726  
mfxStatus::MFX\_WRN\_OUT\_OF\_RANGE (*C++ enumerator*), 726  
mfxStatus::MFX\_WRN\_PARTIAL\_ACCELERATION (*C++ enumerator*), 726  
mfxStatus::MFX\_WRN\_VALUE\_NOT\_CHANGED (*C++ enumerator*), 726  
mfxStatus::MFX\_WRN\_VIDEO\_PARAM\_CHANGED (*C++ enumerator*), 726  
mfxStructVersion (*C++ union*), 764

mfxStructVersion::Major (*C++ member*), 764  
mfxStructVersion::Minor (*C++ member*), 764  
mfxStructVersion::Version (*C++ member*),  
    764  
mfxStructVersion::[anonymous] (*C++ mem-  
ber*), 764  
mfxSyncPoint (*C++ type*), 713  
mfxThreadTask (*C++ type*), 712  
mfxU16 (*C++ type*), 712  
mfxU32 (*C++ type*), 712  
mfxU64 (*C++ type*), 712  
mfxU8 (*C++ type*), 712  
mfxUL32 (*C++ type*), 712  
MFXUnload (*C++ function*), 721  
mfxVariant (*C++ struct*), 714  
mfxVariant::Data (*C++ member*), 714  
mfxVariant::data (*C++ union*), 714  
mfxVariant::data::F32 (*C++ member*), 715  
mfxVariant::data::F64 (*C++ member*), 715  
mfxVariant::data::I16 (*C++ member*), 715  
mfxVariant::data::I32 (*C++ member*), 715  
mfxVariant::data::I64 (*C++ member*), 715  
mfxVariant::data::I8 (*C++ member*), 715  
mfxVariant::data::Ptr (*C++ member*), 715  
mfxVariant::data::U16 (*C++ member*), 715  
mfxVariant::data::U32 (*C++ member*), 715  
mfxVariant::data::U64 (*C++ member*), 715  
mfxVariant::data::U8 (*C++ member*), 715  
mfxVariant::Type (*C++ member*), 714  
mfxVariant::Version (*C++ member*), 714  
mfxVariantType (*C++ enum*), 714  
mfxVariantType::MFX\_VARIANT\_TYPE\_F32  
    (*C++ enumerator*), 714  
mfxVariantType::MFX\_VARIANT\_TYPE\_F64  
    (*C++ enumerator*), 714  
mfxVariantType::MFX\_VARIANT\_TYPE\_I16  
    (*C++ enumerator*), 714  
mfxVariantType::MFX\_VARIANT\_TYPE\_I32  
    (*C++ enumerator*), 714  
mfxVariantType::MFX\_VARIANT\_TYPE\_I64  
    (*C++ enumerator*), 714  
mfxVariantType::MFX\_VARIANT\_TYPE\_I8  
    (*C++ enumerator*), 714  
mfxVariantType::MFX\_VARIANT\_TYPE\_PTR  
    (*C++ enumerator*), 714  
mfxVariantType::MFX\_VARIANT\_TYPE\_U16  
    (*C++ enumerator*), 714  
mfxVariantType::MFX\_VARIANT\_TYPE\_U32  
    (*C++ enumerator*), 714  
mfxVariantType::MFX\_VARIANT\_TYPE\_U64  
    (*C++ enumerator*), 714  
mfxVariantType::MFX\_VARIANT\_TYPE\_U8  
    (*C++ enumerator*), 714  
mfxVariantType::MFX\_VARIANT\_TYPE\_UNSET  
    (*C++ enumerator*), 714  
mfxVersion (*C++ union*), 763  
mfxVersion::Major (*C++ member*), 763  
mfxVersion::Minor (*C++ member*), 763  
mfxVersion::Version (*C++ member*), 763  
mfxVersion::[anonymous] (*C++ member*), 763  
MFXVideoCORE\_GetHandle (*C++ function*), 845  
MFXVideoCORE\_QueryPlatform (*C++ function*),  
    845  
MFXVideoCORE\_SetFrameAllocator (*C++ func-  
tion*), 845  
MFXVideoCORE\_SetHandle (*C++ function*), 845  
MFXVideoCORE\_SyncOperation (*C++ function*),  
    846  
MFXVideoDECODE\_Close (*C++ function*), 853  
MFXVideoDECODE\_DecodeFrameAsync  
    (*C++ function*), 854  
MFXVideoDECODE\_DecodeHeader (*C++ function*),  
    851  
MFXVideoDECODE\_GetDecodeStat  
    (*C++ func-  
tion*), 854  
MFXVideoDECODE\_GetPayload  
    (*C++ function*),  
    854  
MFXVideoDECODE\_GetVideoParam  
    (*C++ func-  
tion*), 853  
MFXVideoDECODE\_Init (*C++ function*), 852  
MFXVideoDECODE\_Query (*C++ function*), 851  
MFXVideoDECODE\_QueryIOSurf  
    (*C++ function*),  
    852  
MFXVideoDECODE\_Reset  
    (*C++ function*), 853  
MFXVideoDECODE\_SetSkipMode  
    (*C++ function*),  
    854  
MFXVideoENCODE\_Close (*C++ function*), 849  
MFXVideoENCODE\_EncodeFrameAsync  
    (*C++ function*), 850  
MFXVideoENCODE\_GetEncodeStat  
    (*C++ func-  
tion*), 850  
MFXVideoENCODE\_GetVideoParam  
    (*C++ func-  
tion*), 849  
MFXVideoENCODE\_Init (*C++ function*), 848  
MFXVideoENCODE\_Query  
    (*C++ function*), 847  
MFXVideoENCODE\_QueryIOSurf  
    (*C++ function*),  
    848  
MFXVideoENCODE\_Reset  
    (*C++ function*), 849  
mfxVideoParam (*C++ struct*), 772  
mfxVideoParam::AllocId (*C++ member*), 772  
mfxVideoParam::AsyncDepth  
    (*C++ member*),  
    772  
mfxVideoParam::ExtParam  
    (*C++ member*), 773  
mfxVideoParam::IOPattern  
    (*C++ member*), 772  
mfxVideoParam::mfx  
    (*C++ member*), 772  
mfxVideoParam::NumExtParam  
    (*C++ member*),  
    773

mfxVideoParam::Protected (*C++ member*), 772  
mfxVideoParam::vpp (*C++ member*), 772  
MFXVideoVPP\_Close (*C++ function*), 858  
MFXVideoVPP\_GetVideoParam (*C++ function*),  
  858  
MFXVideoVPP\_GetVPPStat (*C++ function*), 858  
MFXVideoVPP\_Init (*C++ function*), 857  
MFXVideoVPP\_Query (*C++ function*), 856  
MFXVideoVPP\_QueryIOSurf (*C++ function*), 856  
MFXVideoVPP\_Reset (*C++ function*), 857  
MFXVideoVPP\_RunFrameVPPAsync (*C++ func-  
tion*), 858  
mfxVP9SegmentParam (*C++ struct*), 815  
mfxVP9SegmentParam::FeatureEnabled (*C++  
member*), 815  
mfxVP9SegmentParam::LoopFilterLevelDelta  
  (*C++ member*), 815  
mfxVP9SegmentParam::QIndexDelta  
  (*C++ member*), 815  
mfxVP9SegmentParam::ReferenceFrame (*C++  
member*), 815  
mfxVP9TemporalLayer (*C++ struct*), 817  
mfxVP9TemporalLayer::FrameRateScale  
  (*C++ member*), 817  
mfxVP9TemporalLayer::TargetKbps  
  (*C++ member*), 817  
mfxVPPCompInputStream (*C++ struct*), 825  
mfxVPPCompInputStream::DstH (*C++ member*),  
  825  
mfxVPPCompInputStream::DstW (*C++ member*),  
  825  
mfxVPPCompInputStream::DstX (*C++ member*),  
  825  
mfxVPPCompInputStream::DstY (*C++ member*),  
  825  
mfxVPPCompInputStream::GlobalAlpha (*C++  
member*), 826  
mfxVPPCompInputStream::GlobalAlphaEnable  
  mfxVPPDescription::Version (*C++ member*),  
  718  
mfxVPPCompInputStream::LumaKeyEnable  
  (*C++ member*), 825  
mfxVPPCompInputStream::LumaKeyMax  
  (*C++ member*), 825  
mfxVPPCompInputStream::LumaKeyMin  
  (*C++ member*), 825  
mfxVPPCompInputStream::PixelAlphaEnable  
  (*C++ member*), 826  
mfxVPPCompInputStream::TileId  
  (*C++ member*), 826  
mfxVPPDescription (*C++ struct*), 718  
mfxVPPDescription::filter (*C++ struct*), 718  
mfxVPPDescription::filter::FilterFourCC  
  (*C++ member*), 718  
mfxVPPDescription::filter::MaxDelayInFrame  
  (*C++ member*), 718  
mfxVPPDescription::filter::MemDesc (*C++  
member*), 719  
mfxVPPDescription::filter::memdesc (*C++  
struct*), 719  
mfxVPPDescription::filter::memdesc::format  
  (*C++ struct*), 719  
mfxVPPDescription::filter::memdesc::format::InFor-  
  mat (*C++ member*), 719  
mfxVPPDescription::filter::memdesc::format::NumOut-  
  Put (*C++ member*), 719  
mfxVPPDescription::filter::memdesc::format::OutFor-  
  mat (*C++ member*), 719  
mfxVPPDescription::filter::memdesc::format::reserved  
  (*C++ member*), 719  
mfxVPPDescription::filter::memdesc::Formats  
  (*C++ member*), 719  
mfxVPPDescription::filter::memdesc::Height  
  (*C++ member*), 719  
mfxVPPDescription::filter::memdesc::MemHandleType  
  (*C++ member*), 719  
mfxVPPDescription::filter::memdesc::NumInFormats  
  (*C++ member*), 719  
mfxVPPDescription::filter::memdesc::reserved  
  (*C++ member*), 719  
mfxVPPDescription::filter::memdesc::Width  
  (*C++ member*), 719  
mfxVPPDescription::filter::NumMemTypes  
  (*C++ member*), 718  
mfxVPPDescription::filter::reserved  
  (*C++ member*), 718  
mfxVPPDescription::Filters (*C++ member*),  
  718  
mfxVPPDescription::NumFilters (*C++ mem-  
ber*), 718  
mfxVPPDescription::reserved (*C++ member*),  
  718  
mfxVPPStat (*C++ struct*), 787  
mfxVPPStat::NumCachedFrame (*C++ member*),  
  787  
mfxVPPStat::NumFrame (*C++ member*), 787  
mfxY410 (*C++ struct*), 773  
mfxY410::A (*C++ member*), 773  
mfxY410::U (*C++ member*), 773  
mfxY410::V (*C++ member*), 773  
mfxY410::Y (*C++ member*), 773  
minmag (*C++ function*), 1371  
Misc, 648  
Model, 228  
modf (*C++ function*), 1347  
MPEG, 711  
MPEG-2, 711

`mul (C++ function)`, 1212  
`mulbyconj (C++ function)`, 1214

## N

`NAL`, 711  
`nearbyint (C++ function)`, 1344  
`nextafter (C++ function)`, 1363  
Nominal feature, 228  
`not_complete (C macro)`, 383  
`not_initialized (C++ member)`, 384  
`null_mutex (C++ function)`, 644  
`null_rw_mutex (C++ function)`, 645  
NV12, 711  
NV16, 711

## O

Observation, 228  
`observe (C++ function)`, 388  
`on_scheduler_entry (C++ function)`, 388  
`on_scheduler_exit (C++ function)`, 389  
`onedal::data_format (C++ class)`, 250  
`onedal::data_layout (C++ class)`, 250  
`onedal::data_type (C++ class)`, 251  
`onedal::feature_info (C++ class)`, 251  
`onedal::feature_type (C++ class)`, 251  
`onedal::homogen_table (C++ class)`, 248  
`onedal::homogen_table::data_pointer (C++ member)`, 248  
`onedal::homogen_table::data_type (C++ member)`, 248  
`onedal::homogen_table::feature_types_equal (C++ member)`, 248  
`onedal::homogen_table::homogen_table (C++ function)`, 248  
`onedal::homogen_table::operator= (C++ function)`, 248  
`onedal::infer (C++ function)`, 261, 274  
`onedal::kmeans::desc (C++ class)`, 256  
`onedal::kmeans::desc::accuracy_threshold (C++ member)`, 256  
`onedal::kmeans::desc::cluster_count (C++ member)`, 256  
`onedal::kmeans::desc::desc (C++ function)`, 256  
`onedal::kmeans::desc::max_iteration_count (C++ member)`, 256  
`onedal::kmeans::infer_input (C++ class)`, 259  
`onedal::kmeans::infer_input::data (C++ member)`, 260  
`onedal::kmeans::infer_input::infer_input (C++ function)`, 259, 260  
`onedal::kmeans::infer_input::model (C++ member)`, 260  
`onedal::kmeans::infer_result (C++ class)`, 260  
`onedal::kmeans::infer_result::infer_result (C++ function)`, 260  
`onedal::kmeans::infer_result::labels (C++ member)`, 260  
`onedal::kmeans::infer_result::objective_function_value (C++ member)`, 260  
`onedal::kmeans::method::by_default (C++ type)`, 255  
`onedal::kmeans::method::lloyd (C++ struct)`, 255  
`onedal::kmeans::model (C++ class)`, 257  
`onedal::kmeans::model::centroids (C++ member)`, 257  
`onedal::kmeans::model::cluster_count (C++ member)`, 257  
`onedal::kmeans::model::model (C++ function)`, 257  
`onedal::kmeans::train_input (C++ class)`, 257  
`onedal::kmeans::train_input::data (C++ member)`, 258  
`onedal::kmeans::train_input::initial_centroids (C++ member)`, 258  
`onedal::kmeans::train_input::train_input (C++ function)`, 257, 258  
`onedal::kmeans::train_result (C++ class)`, 258  
`onedal::kmeans::train_result::iteration_count (C++ member)`, 258  
`onedal::kmeans::train_result::labels (C++ member)`, 258  
`onedal::kmeans::train_result::model (C++ member)`, 258  
`onedal::kmeans::train_result::objective_function_value (C++ member)`, 259  
`onedal::kmeans::train_result::train_result (C++ function)`, 258  
`onedal::knn::descriptor (C++ class)`, 263  
`onedal::knn::descriptor::class_count (C++ member)`, 263  
`onedal::knn::descriptor::descriptor (C++ function)`, 263  
`onedal::knn::descriptor::neighbor_count (C++ member)`, 263  
`onedal::knn::infer (C++ function)`, 267  
`onedal::knn::infer_input (C++ class)`, 266  
`onedal::knn::infer_input::data (C++ member)`, 266  
`onedal::knn::infer_input::infer_input (C++ function)`, 266  
`onedal::knn::infer_input::model (C++ member)`, 266

onedal::knn::infer\_result (*C++ class*), 266  
 onedal::knn::infer\_result::infer\_result  
     (*C++ function*), 266  
 onedal::knn::infer\_result::labels (*C++ member*), 266  
 onedal::knn::method::bruteforce     (*C++ struct*), 263  
 onedal::knn::method::by\_default     (*C++ type*), 264  
 onedal::knn::method::kd\_tree (*C++ struct*), 263  
 onedal::knn::model (*C++ class*), 264  
 onedal::knn::model::model (*C++ function*), 264  
 onedal::knn::train (*C++ function*), 265  
 onedal::knn::train\_input (*C++ class*), 264  
 onedal::knn::train\_input::data (*C++ member*), 264  
 onedal::knn::train\_input::labels (*C++ member*), 264  
 onedal::knn::train\_input::train\_input  
     (*C++ function*), 264  
 onedal::knn::train\_result (*C++ class*), 265  
 onedal::knn::train\_result::model (*C++ member*), 265  
 onedal::knn::train\_result::train\_result  
     (*C++ function*), 265  
 onedal::pca::desc (*C++ class*), 269  
 onedal::pca::desc::component\_count (*C++ member*), 269  
 onedal::pca::desc::desc (*C++ function*), 269  
 onedal::pca::desc::set\_deterministic  
     (*C++ member*), 270  
 onedal::pca::infer\_input (*C++ class*), 273  
 onedal::pca::infer\_input::data (*C++ member*), 273  
 onedal::pca::infer\_input::infer\_input  
     (*C++ function*), 273  
 onedal::pca::infer\_input::model     (*C++ member*), 273  
 onedal::pca::infer\_result (*C++ class*), 273  
 onedal::pca::infer\_result::infer\_result  
     (*C++ function*), 273  
 onedal::pca::infer\_result::transformed\_data  
     (*C++ member*), 273  
 onedal::pca::method::by\_default     (*C++ type*), 269  
 onedal::pca::method::cov (*C++ struct*), 269  
 onedal::pca::method::svd (*C++ struct*), 269  
 onedal::pca::model (*C++ class*), 270  
 onedal::pca::model::component\_count  
     (*C++ member*), 270  
 onedal::pca::model::eigenvectors    (*C++ member*), 270  
 onedal::pca::model::model (*C++ function*),  
     270  
 onedal::pca::train\_input (*C++ class*), 271  
 onedal::pca::train\_input::data (*C++ member*), 271  
 onedal::pca::train\_input::train\_input  
     (*C++ function*), 271  
 onedal::pca::train\_result (*C++ class*), 271  
 onedal::pca::train\_result::eigenvalues  
     (*C++ member*), 272  
 onedal::pca::train\_result::means    (*C++ member*), 271  
 onedal::pca::train\_result::model    (*C++ member*), 271  
 onedal::pca::train\_result::train\_result  
     (*C++ function*), 271  
 onedal::pca::train\_result::variances  
     (*C++ member*), 272  
 onedal::table (*C++ class*), 246  
 onedal::table::feature\_count (*C++ member*), 247  
 onedal::table::is\_empty (*C++ member*), 247  
 onedal::table::metadata (*C++ member*), 247  
 onedal::table::observation\_count (*C++ member*), 247  
 onedal::table::operator= (*C++ function*), 246  
 onedal::table::table (*C++ function*), 246  
 onedal::table\_meta (*C++ class*), 249  
 onedal::table\_meta::feature (*C++ member*),  
     249  
 onedal::table\_meta::feature\_count (*C++ member*), 249  
 onedal::table\_meta::format (*C++ member*),  
     250  
 onedal::table\_meta::is\_contiguous (*C++ member*), 250  
 onedal::table\_meta::is\_homogeneous (*C++ function*), 250  
 onedal::table\_meta::layout (*C++ member*),  
     250  
 onedal::train (*C++ function*), 259, 272  
 onemkl::blas::asum (*C++ function*), 875, 876  
 onemkl::blas::axpy (*C++ function*), 877  
 onemkl::blas::axpy\_batch (*C++ function*),  
     982–984  
 onemkl::blas::copy (*C++ function*), 879  
 onemkl::blas::dot (*C++ function*), 880, 881  
 onemkl::blas::dotc (*C++ function*), 884  
 onemkl::blas::dotu (*C++ function*), 886  
 onemkl::blas::gbmv (*C++ function*), 906, 907  
 onemkl::blas::gemm (*C++ function*), 958, 959  
 onemkl::blas::gemm\_batch (*C++ function*), 985,  
     987, 989  
 onemkl::blas::gemm\_bias (*C++ function*), 999,

1000  
onemkl::blas::gemmt (*C++ function*), 996, 997  
onemkl::blas::gemv (*C++ function*), 908, 909  
onemkl::blas::ger (*C++ function*), 911  
onemkl::blas::gerc (*C++ function*), 913  
onemkl::blas::geru (*C++ function*), 915  
onemkl::blas::hbmv (*C++ function*), 917  
onemkl::blas::hemm (*C++ function*), 961, 962  
onemkl::blas::hemv (*C++ function*), 919, 920  
onemkl::blas::her (*C++ function*), 921, 922  
onemkl::blas::her2 (*C++ function*), 923, 924  
onemkl::blas::her2k (*C++ function*), 966, 967  
onemkl::blas::herk (*C++ function*), 963, 964  
onemkl::blas::hpmv (*C++ function*), 925, 926  
onemkl::blas::hpr (*C++ function*), 927, 928  
onemkl::blas::hpr2 (*C++ function*), 929, 930  
onemkl::blas::iamax (*C++ function*), 902  
onemkl::blas::iamin (*C++ function*), 904  
onemkl::blas::nrm2 (*C++ function*), 887, 888  
onemkl::blas::rot (*C++ function*), 889  
onemkl::blas::rotg (*C++ function*), 891  
onemkl::blas::rotm (*C++ function*), 892, 894  
onemkl::blas::rotmg (*C++ function*), 896, 897  
onemkl::blas::sbmv (*C++ function*), 931, 932  
onemkl::blas::scal (*C++ function*), 899  
onemkl::blas::sdssdot (*C++ function*), 882, 883  
onemkl::blas::spmv (*C++ function*), 934  
onemkl::blas::spr (*C++ function*), 936  
onemkl::blas::spr2 (*C++ function*), 937, 938  
onemkl::blas::swap (*C++ function*), 900, 901  
onemkl::blas::symm (*C++ function*), 968, 969  
onemkl::blas::symv (*C++ function*), 939, 940  
onemkl::blas::syr (*C++ function*), 941, 942  
onemkl::blas::syr2 (*C++ function*), 943, 944  
onemkl::blas::syr2k (*C++ function*), 973, 974  
onemkl::blas::syrk (*C++ function*), 971, 972  
onemkl::blas::tbmv (*C++ function*), 945, 946  
onemkl::blas::tbsv (*C++ function*), 947, 948  
onemkl::blas::tpmv (*C++ function*), 949, 950  
onemkl::blas::tpsv (*C++ function*), 951, 952  
onemkl::blas::trmm (*C++ function*), 976, 977  
onemkl::blas::trmv (*C++ function*), 953, 954  
onemkl::blas::trsm (*C++ function*), 979, 980  
onemkl::blas::trsm\_batch (*C++ function*), 991,  
993, 994  
onemkl::blas::trsv (*C++ function*), 955, 956  
onemkl::lapack::gebrd (*C++ function*), 1052,  
1053  
onemkl::lapack::gebrd\_scratchpad\_size  
(*C++ function*), 1055  
onemkl::lapack::geqrf (*C++ function*), 1003,  
1004  
onemkl::lapack::geqrf\_batch (*C++ function*),  
1112  
onemkl::lapack::geqrf\_scratchpad\_size  
(*C++ function*), 1005  
onemkl::lapack::gesvd (*C++ function*), 1056,  
1058  
onemkl::lapack::gesvd\_scratchpad\_size  
(*C++ function*), 1060  
onemkl::lapack::getrf (*C++ function*), 1006,  
1007  
onemkl::lapack::getrf\_batch (*C++ function*),  
1113  
onemkl::lapack::getrf\_scratchpad\_size  
(*C++ function*), 1009  
onemkl::lapack::getri (*C++ function*), 1010  
onemkl::lapack::getri\_batch (*C++ function*),  
1114  
onemkl::lapack::getri\_scratchpad\_size  
(*C++ function*), 1012  
onemkl::lapack::getrs (*C++ function*), 1013,  
1014  
onemkl::lapack::getrs\_batch (*C++ function*),  
1115  
onemkl::lapack::getrs\_scratchpad\_size  
(*C++ function*), 1015  
onemkl::lapack::heevd (*C++ function*), 1062,  
1063  
onemkl::lapack::heevd\_scratchpad\_size  
(*C++ function*), 1064  
onemkl::lapack::hegvd (*C++ function*), 1065,  
1067  
onemkl::lapack::hegvd\_scratchpad\_size  
(*C++ function*), 1069  
onemkl::lapack::hetrd (*C++ function*), 1070,  
1071  
onemkl::lapack::hetrd\_scratchpad\_size  
(*C++ function*), 1073  
onemkl::lapack::orgbr (*C++ function*), 1075,  
1076  
onemkl::lapack::orgbr\_scratchpad\_size  
(*C++ function*), 1077  
onemkl::lapack::orgqr (*C++ function*), 1017,  
1018  
onemkl::lapack::orgqr\_batch (*C++ function*),  
1117  
onemkl::lapack::orgqr\_scratchpad\_size  
(*C++ function*), 1019  
onemkl::lapack::orgtr (*C++ function*), 1079  
onemkl::lapack::orgtr\_scratchpad\_size  
(*C++ function*), 1081  
onemkl::lapack::ormqr (*C++ function*), 1020,  
1021  
onemkl::lapack::ormqr\_scratchpad\_size  
(*C++ function*), 1023  
onemkl::lapack::ormtr (*C++ function*), 1082,  
1083

onemkl::lapack::ormtr\_scratchpad\_size  
     (*C++ function*), 1085  
 onemkl::lapack::potrf (*C++ function*), 1024,  
     1025  
 onemkl::lapack::potrf\_batch (*C++ function*),  
     1118  
 onemkl::lapack::potrf\_scratchpad\_size  
     (*C++ function*), 1027  
 onemkl::lapack::potri (*C++ function*), 1028,  
     1029  
 onemkl::lapack::potri\_scratchpad\_size  
     (*C++ function*), 1030  
 onemkl::lapack::potrs (*C++ function*), 1031,  
     1032  
 onemkl::lapack::potrs\_batch (*C++ function*),  
     1119  
 onemkl::lapack::potrs\_scratchpad\_size  
     (*C++ function*), 1034  
 onemkl::lapack::syevd (*C++ function*), 1086,  
     1087  
 onemkl::lapack::syevd\_scratchpad\_size  
     (*C++ function*), 1089  
 onemkl::lapack::sygvd (*C++ function*), 1090,  
     1092  
 onemkl::lapack::sygvd\_scratchpad\_size  
     (*C++ function*), 1094  
 onemkl::lapack::sytrd (*C++ function*), 1095,  
     1096  
 onemkl::lapack::sytrd\_scratchpad\_size  
     (*C++ function*), 1098  
 onemkl::lapack::sytrf (*C++ function*), 1035,  
     1037  
 onemkl::lapack::sytrf\_scratchpad\_size  
     (*C++ function*), 1038  
 onemkl::lapack::trtrs (*C++ function*), 1040,  
     1041  
 onemkl::lapack::trtrs\_scratchpad\_size  
     (*C++ function*), 1042  
 onemkl::lapack::ungbr (*C++ function*), 1099,  
     1101  
 onemkl::lapack::ungbr\_scratchpad\_size  
     (*C++ function*), 1102  
 onemkl::lapack::ungqr (*C++ function*), 1044,  
     1045  
 onemkl::lapack::ungqr\_scratchpad\_size  
     (*C++ function*), 1046  
 onemkl::lapack::ungtr (*C++ function*), 1104  
 onemkl::lapack::ungtr\_scratchpad\_size  
     (*C++ function*), 1106  
 onemkl::lapack::unmqr (*C++ function*), 1047,  
     1048  
 onemkl::lapack::unmqr\_scratchpad\_size  
     (*C++ function*), 1050  
 onemkl::lapack::unmtr (*C++ function*), 1107,  
     1108  
 onemkl::lapack::unmtr\_scratchpad\_size  
     (*C++ function*), 1110  
 onemkl::sparse::gemm (*C++ function*), 1123  
 onemkl::sparse::gemv (*C++ function*), 1125  
 onemkl::sparse::gemvdot (*C++ function*), 1127  
 onemkl::sparse::gemvOptimize (*C++ func-  
     tion*), 1128  
 onemkl::sparse::matrixInit (*C++ function*),  
     1121  
 onemkl::sparse::setCSRstructure (*C++  
     function*), 1122  
 onemkl::sparse::symv (*C++ function*), 1129  
 onemkl::sparse::trmv (*C++ function*), 1131  
 onemkl::sparse::trmvOptimize (*C++ func-  
     tion*), 1133  
 onemkl::sparse::trsv (*C++ function*), 1134  
 onemkl::sparse::trsvOptimize (*C++ func-  
     tion*), 1136  
 Online mode, 230  
 operator bool (*C++ function*), 304  
 operator split (*C++ function*), 325  
 operator!= (*C++ function*), 629, 631, 632  
 operator+ (*C++ function*), 283  
 operator= (*C++ function*), 283, 290–293, 619  
 operator== (*C++ function*), 629, 631, 632  
 operator& (*C++ function*), 312  
 operator- (*C++ function*), 283  
 operator< (*C++ function*), 278, 283  
 Ordinal feature, 229  
 Outlier, 229  
 output\_ports (*C++ function*), 365, 368  
 overwrite\_node (*C++ function*), 348

## P

P010, 711  
 P210, 711  
 parameter::max\_allowed\_parallelism (*C++  
     enum*), 380  
 parameter::thread\_stack\_size (*C++ enum*),  
     380  
 pow (*C++ function*), 1240  
 pow2o3 (*C++ function*), 1237  
 pow3o2 (*C++ function*), 1238  
 powr (*C++ function*), 1245  
 powx (*C++ function*), 1243  
 PPS, 711  
 priority\_queue\_node (*C++ function*), 354  
 proportional\_split (*C++ function*), 325

## Q

QP, 711  
 queue\_node (*C++ function*), 353  
 queuing\_mutex (*C++ function*), 642

`queuing_rw_mutex (C++ function)`, 643

## R

`R::~R (C++ function)`, 278  
`R::empty (C++ function)`, 278  
`R::is_divisible (C++ function)`, 279  
`R::R (C++ function)`, 278, 279  
`range (C++ function)`, 625  
`Ratio feature`, 229  
`Reduction::operator () (C++ function)`, 281  
`Reference-counted object`, 230  
`Regression`, 229  
`remainder (C++ function)`, 1224  
`reset (C++ function)`, 327, 379  
`Response`, 229  
`RGB32`, 711  
`RGB4`, 711  
`right (C++ function)`, 325  
`rint (C++ function)`, 1346  
`round (C++ function)`, 1343  
`run (C++ function)`, 382  
`run_and_wait (C++ function)`, 382  
`RWM::~scoped_lock (C++ function)`, 287  
`RWM::scoped_lock (C++ function)`, 287  
`RWM::scoped_lock (C++ type)`, 287  
`RWM::scoped_lock::acquire (C++ function)`, 287  
`RWM::scoped_lock:: downgrade_to_reader (C++ function)`, 287  
`RWM::scoped_lock::release (C++ function)`, 287  
`RWM::scoped_lock::try_acquire (C++ function)`, 287  
`RWM::scoped_lock::upgrade_to_writer (C++ function)`, 287

## S

`S::~S (C++ function)`, 293  
`S::operator () (C++ function)`, 293  
`S::S (C++ function)`, 293  
`scalable_allocation_command (C function)`, 636  
`scalable_allocation_mode (C++ function)`, 635  
`scalable_msize (C++ function)`, 635  
`Scan::operator () (C++ function)`, 283  
`scoped_lock (C++ class)`, 638–643, 645  
`SDK`, 711  
`SEI`, 711  
`sequencer_node (C++ function)`, 356  
`set_external_ports (C++ function)`, 368  
`set_mode (C++ function)`, 1351  
`set_status (C++ function)`, 1353  
`Setter`, 230  
`setValue (C++ function)`, 1140

`sin (C++ function)`, 1268  
`sincos (C++ function)`, 1270  
`sind (C++ function)`, 1297  
`sinh (C++ function)`, 1303  
`sinpi (C++ function)`, 1284  
`size (C++ function)`, 316, 624, 627  
`size_type (C++ type)`, 315  
`skip_ahead (C++ function)`, 1166  
`speculative_spin_mutex (C++ function)`, 640  
`speculative_spin_rw_mutex (C++ function)`, 641  
`spin_mutex (C++ function)`, 638  
`spin_rw_mutex (C++ function)`, 639  
`SPIR-V`, 230  
`split_node (C++ function)`, 364  
`SPS`, 711  
`sqr (C++ function)`, 1210  
`sqrt (C++ function)`, 1230  
`stop (C++ function)`, 313  
`sub (C++ function)`, 1208  
`Supervised learning`, 229  
`swap (C++ function)`, 278  
`SYCL`, 230

## T

`T::begin (C++ function)`, 282  
`T::end (C++ function)`, 282  
`T::release_wait (C++ function)`, 291  
`T::reserve_wait (C++ function)`, 291  
`T::try_put (C++ function)`, 291  
`Table`, 230  
`tag (C++ function)`, 372  
`tagged_msg (C++ function)`, 372  
`tan (C++ function)`, 1273  
`tand (C++ function)`, 1299  
`tanh (C++ function)`, 1306  
`tanpi (C++ function)`, 1286  
`task_arena (C++ function)`, 385  
`task_group (C++ function)`, 382  
`task_group_context (C++ function)`, 379  
`task_scheduler_observer (C++ function)`, 388  
`TBBMALLOC_CLEAN_ALL_BUFFERS (C macro)`, 636  
`TBBMALLOC_CLEAN_THREAD_BUFFERS (C macro)`, 636  
`TBBMALLOC_SET_HUGE_SIZE_THRESHOLD (C macro)`, 636  
`TBBMALLOC_SET_SOFT_HEAP_LIMIT (C macro)`, 636  
`TBBMALLOC_USE_HUGE_PAGES (C macro)`, 635  
`terminate (C++ function)`, 385  
`tgamma (C++ function)`, 1333  
`Training`, 229  
`Training set`, 229  
`traits (C++ function)`, 379

traits\_type::fp\_settings (*C++ enum*), 378  
trunc (*C++ function*), 1341  
try\_get (*C++ function*), 337, 348, 350, 352, 353, 356,  
    358, 359, 366  
try\_lock (*C++ function*), 638, 639, 644, 645  
try\_lock\_shared (*C++ function*), 639, 645  
try\_put (*C++ function*), 348, 350, 352–356, 358, 359,  
    364

## U

unlock (*C++ function*), 638, 639, 644, 645  
unlock\_shared (*C++ function*), 639, 645  
Unsupervised learning, 229  
upstream\_resource (*C++ function*), 633  
UYVY, 711

## V

VA API, 711  
Value::~Value (*C++ function*), 283  
Value::Value (*C++ function*), 283  
VBR, 711  
VBV, 711  
VC-1, 711  
video memory, 711  
VPP, 648  
VUI, 711

## W

wait (*C++ function*), 382  
wait\_for\_all (*C++ function*), 327  
Workload, 230  
write\_once\_node (*C++ function*), 350

## X

X::X (*C++ function*), 279

## Y

YUY2, 711  
YV12, 711