

oneAPI

oneAPI Specification

Release 0.85

Intel

Jul 14, 2020

CONTENTS

1	Introduction	1
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
2	Software Architecture	4
2.1	oneAPI Platform	5
2.2	API Programming Example	6
2.3	Direct Programming Example	7
3	Library Interoperability	8
3.1	Queueing	8
3.2	API Arguments	8
3.3	Asynchronous APIs	8
3.4	Exceptions	9
4	oneAPI Elements	10
5	DPC++	11
5.1	Overview	11
5.2	Detailed API and Language Descriptions	12
5.3	Open Source Implementation	13
5.4	Testing	13
6	oneDPL	14
6.1	Namespaces	14
6.2	Supported C++ Standard Library APIs and Algorithms	14
6.2.1	Extensions to Parallel STL	14
6.2.1.1	DPC++ Execution Policy	15
6.2.1.2	Wrappers for SYCL Buffers	16
6.2.2	Specific API of oneDPL	16
7	oneDNN	19
7.1	Introduction	20
7.1.1	General API notes	21
7.1.2	Error Handling	21
7.2	Conventions	22
7.2.1	Variable (Tensor) Names	22
7.2.2	RNN-Specific Notation	23

7.3	Execution Model	23
7.3.1	Engine	24
7.3.2	Stream	25
7.4	Data model	27
7.4.1	Data types	27
7.4.1.1	Bfloat16	28
7.4.1.1.1	Workflow	28
7.4.1.1.2	Support	29
7.4.1.2	Int8	29
7.4.1.2.1	Workflow	29
7.4.1.2.2	Support	29
7.4.2	Memory	29
7.4.2.1	Memory Formats	29
7.4.2.1.1	Plain Memory Formats	30
7.4.2.1.2	Format Tags	31
7.4.2.1.3	Optimized Format ‘any’	31
7.4.2.1.4	Memory Format Propagation	32
7.4.2.1.5	API	32
7.4.2.2	Memory Descriptors and Objects	37
7.4.2.2.1	Descriptors	37
7.4.2.2.2	Objects	40
7.4.2.2.3	API	40
7.5	Primitives	43
7.5.1	Common Definitions	45
7.5.1.1	Base Class for Primitives	45
7.5.1.2	Base Class for Primitives Descriptors	47
7.5.1.3	Common Enumerations	53
7.5.1.4	Normalization Primitives Flags	56
7.5.1.5	Execution argument indices	56
7.5.2	Attributes	59
7.5.2.1	Post-ops	59
7.5.2.1.1	Supported Post-ops	60
7.5.2.1.1.1	Eltwise Post-op	60
7.5.2.1.1.2	Sum Post-op	60
7.5.2.1.1.3	Examples of Chained Post-ops	61
7.5.2.1.1.4	Sum -> ReLU	61
7.5.2.1.2	API	61
7.5.2.2	Scratchpad Mode	63
7.5.2.2.1	Examples	64
7.5.2.2.1.1	Library Manages Scratchpad	64
7.5.2.2.1.2	User Manages Scratchpad	64
7.5.2.3	Quantization	65
7.5.2.3.1	Quantization Model	65
7.5.2.3.1.1	Example: Convolution Quantization Workflow	66
7.5.2.3.1.2	Per-Channel Scaling	66
7.5.2.3.2	Output Scaling Attribute	67
7.5.2.3.2.1	Example 1: weights quantization with per-output-channel-and-group scaling	67
7.5.2.3.2.2	Example 2: convolution with groups, with per-output-channel quantization	68
7.5.2.3.2.3	Interplay of Output Scales with Post-ops	69
7.5.2.4	Attribute Related Error Handling	69
7.5.2.5	API	70
7.5.3	Batch Normalization	73

7.5.3.1	Forward	73
7.5.3.1.1	Difference Between Forward Training and Forward Inference	74
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	76
7.5.3.6	Data Representation	76
7.5.3.6.1	Source, Destination, and Their Gradients	76
7.5.3.6.2	Statistics Tensors	76
7.5.3.7	Post-ops and Attributes	76
7.5.3.8	API	77
7.5.4	Binary	80
7.5.4.1	Forward and Backward	80
7.5.4.2	Execution Arguments	81
7.5.4.3	Operation Details	81
7.5.4.4	Post-ops and Attributes	81
7.5.4.5	Data Types Support	81
7.5.4.6	Data Representation	82
7.5.4.7	API	82
7.5.5	Concat	83
7.5.5.1	Forward and Backward	83
7.5.5.2	Execution Arguments	84
7.5.5.3	Operation Details	84
7.5.5.4	Data Types Support	84
7.5.5.5	Data Representation	84
7.5.5.6	Post-ops and Attributes	84
7.5.5.7	API	84
7.5.6	Convolution and Deconvolution	85
7.5.6.1	Forward	86
7.5.6.1.1	Regular Convolution	86
7.5.6.1.2	Convolution with Groups	86
7.5.6.1.3	Convolution with Dilation	87
7.5.6.1.4	Deconvolution (Transposed Convolution)	87
7.5.6.1.5	Difference Between Forward Training and Forward Inference	87
7.5.6.2	Backward	87
7.5.6.3	Execution Arguments	88
7.5.6.4	Operation Details	88
7.5.6.5	Data Types Support	88
7.5.6.6	Data Representation	88
7.5.6.7	Post-ops and Attributes	89
7.5.6.7.1	Example 1	90
7.5.6.7.2	Example 2	90
7.5.6.8	Algorithms	91
7.5.6.9	API	91
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	111
7.5.7.4	Execution Arguments	111
7.5.7.5	Operation Details	111
7.5.7.6	Data Type Support	111
7.5.7.7	Data Representation	112
7.5.7.8	Post-ops and Attributes	112
7.5.7.9	API	112

7.5.8	Inner Product	115
7.5.8.1	Forward	115
7.5.8.1.1	Difference Between Forward Training and Forward Inference	115
7.5.8.2	Backward	115
7.5.8.2.1	Execution Arguments	115
7.5.8.3	Operation Details	116
7.5.8.4	Data Types Support	116
7.5.8.5	Data Representation	116
7.5.8.6	Post-ops and Attributes	117
7.5.8.7	API	117
7.5.9	Layer normalization	122
7.5.9.1	Forward	123
7.5.9.1.1	Difference Between Forward Training and Forward Inference	123
7.5.9.2	Backward	123
7.5.9.3	Execution Arguments	123
7.5.9.4	Operation Details	124
7.5.9.5	Data Types Support	124
7.5.9.6	Data Representation	125
7.5.9.6.1	Mean and Variance	125
7.5.9.6.2	Scale and Shift	125
7.5.9.6.3	Source, Destination, and Their Gradients	125
7.5.9.7	API	125
7.5.10	LogSoftmax	129
7.5.10.1	Forward	129
7.5.10.1.1	Difference Between Forward Training and Forward Inference	130
7.5.10.2	Backward	130
7.5.10.3	Execution Arguments	130
7.5.10.4	Operation Details	130
7.5.10.5	Post-ops and Attributes	130
7.5.10.6	Data Type Support	130
7.5.10.7	Data Representation	131
7.5.10.7.1	Source, Destination, and Their Gradients	131
7.5.10.8	API	131
7.5.11	Local Response Normalization	134
7.5.11.1	Forward	134
7.5.11.2	Backward	134
7.5.11.2.1	Execution Arguments	134
7.5.11.2.1.1	Operation Details	134
7.5.11.2.1.2	Data Type Support	135
7.5.11.2.1.3	Data Representation	135
7.5.11.2.2	Source, Destination, and Their Gradients	135
7.5.11.2.2.1	Post-ops and Attributes	135
7.5.11.2.2.2	API	135
7.5.12	Matrix Multiplication	139
7.5.12.1	Execution Arguments	139
7.5.12.2	Operation Details	139
7.5.12.3	Data Types Support	139
7.5.12.4	Data Representation	140
7.5.12.5	Attributes and Post-ops	140
7.5.12.6	API	141
7.5.13	Pooling	142
7.5.13.1	Forward	143
7.5.13.1.1	Difference Between Forward Training and Forward Inference	143
7.5.13.2	Backward	143

7.5.13.3	Execution Arguments	143
7.5.13.4	Operation Details	143
7.5.13.5	Data Type Support	144
7.5.13.6	Data Representation	144
7.5.13.6.1	Source, Destination, and Their Gradients	144
7.5.13.7	Post-ops and Attributes	144
7.5.13.8	API	144
7.5.14	Reorder	148
7.5.14.1	Execution Arguments	148
7.5.14.2	Operation Details	149
7.5.14.3	Data Types Support	149
7.5.14.4	Data Representation	149
7.5.14.5	Post-ops and Attributes	149
7.5.14.6	API	150
7.5.15	Resampling	152
7.5.15.1	Forward	152
7.5.15.1.1	Nearest Neighbor Resampling	152
7.5.15.1.2	Bilinear Resampling	152
7.5.15.1.3	Difference Between Forward Training and Forward Inference	153
7.5.15.2	Backward	153
7.5.15.3	Execution Arguments	153
7.5.15.4	Operation Details	153
7.5.15.5	Data Types Support	154
7.5.15.6	Post-ops and Attributes	154
7.5.15.7	API	154
7.5.16	RNN	158
7.5.16.1	Cell Functions	159
7.5.16.1.1	Vanilla RNN	159
7.5.16.1.2	LSTM	159
7.5.16.1.2.1	LSTM (or Vanilla LSTM)	159
7.5.16.1.2.2	LSTM with Peephole	160
7.5.16.1.2.3	LSTM with Projection	160
7.5.16.1.3	GRU	161
7.5.16.1.4	Linear-Before-Reset GRU	161
7.5.16.2	Execution Arguments	162
7.5.16.3	Operation Details	163
7.5.16.4	Data Types Support	163
7.5.16.4.1	Data Representation	163
7.5.16.4.2	Post-ops and Attributes	164
7.5.16.5	API	164
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
8	oneCCL	200
8.1	Introduction	200
8.2	Definitions	200
8.2.1	oneCCL Concepts	200
8.2.1.1	oneCCL Environment	201
8.2.1.2	oneCCL Stream	201
8.2.1.3	oneCCL Communicator	202
8.2.1.3.1	oneCCL Communicator Attributes	202
8.2.2	oneCCL Collective Communication	203
8.2.2.1	Collective Operations	203
8.2.2.1.1	Allgatherv	203
8.2.2.1.2	Allreduce	204
8.2.2.1.3	Reduce	205
8.2.2.1.4	Alltoall	206
8.2.2.1.5	Barrier	206
8.2.2.1.6	Broadcast	207
8.2.2.2	Data Types	207
8.2.2.3	Collective Call Attributes	208
8.2.2.4	Track Communication Progress	209
8.2.2.4.1	Request	209
8.2.2.4.2	Test	209
8.2.2.4.3	Wait	210
8.2.3	Error Handling	210
8.3	Programming Model	211
8.3.1	Generic Workflow	211
8.3.2	GPU support	211
8.3.2.1	Example	211
8.3.3	CPU support	213
8.3.3.1	Example	213
9	Level Zero	215
9.1	Detailed API Descriptions	215
10	oneDAL	216
10.1	Introduction	216

10.2	Glossary	218
10.2.1	Machine learning terms	218
10.2.2	oneDAL terms	219
10.2.3	Common oneAPI terms	220
10.3	Mathematical Notations	221
10.4	Programming model	221
10.4.1	Basic usage scenario	221
10.4.2	Memory objects	223
10.4.3	Algorithm Anatomy	223
10.4.3.1	Descriptors	223
10.4.3.1.1	Floating-point Types	225
10.4.3.1.2	Computational Methods	225
10.4.3.2	Operations	225
10.4.3.2.1	Input	225
10.4.3.2.2	Result	225
10.4.4	Managing execution context	225
10.4.5	Computational modes	225
10.4.5.1	Batch	225
10.4.5.2	Online	226
10.4.5.3	Distributed	226
10.5	Common Interface	226
10.5.1	Header files	226
10.5.2	Namespaces	227
10.5.3	Managing object lifetimes	227
10.5.4	Error handling	227
10.5.4.1	Exception classification	227
10.6	Data management	229
10.6.1	Key concepts	230
10.6.1.1	Dataset	230
10.6.1.2	Data source	231
10.6.1.3	Table	231
10.6.1.4	Metadata	232
10.6.1.5	Table builder	232
10.6.1.6	Accessor	232
10.6.1.7	Use-case example for table, accessor and table builder	233
10.6.2	Details	235
10.6.2.1	Data Sources	235
10.6.2.2	Tables	235
10.6.2.2.1	Requirements	235
10.6.2.2.2	Table Types	236
10.6.2.2.3	Table API	236
10.6.2.2.3.1	Homogeneous table	237
10.6.2.2.3.2	Structure-of-arrays table	239
10.6.2.2.3.3	Arrays-of-structure table	239
10.6.2.2.3.4	Compressed-sparse-row table	239
10.6.2.2.4	Metadata API	239
10.6.2.2.4.1	Data layout	240
10.6.2.2.4.2	Data format	240
10.6.2.2.4.3	Feature info	241
10.6.2.2.4.4	Data type	241
10.6.2.2.4.5	Feature type	241
10.6.2.3	Table Builders	242
10.6.2.3.1	Requirements	242
10.6.2.3.2	Table Builder Types	242

10.6.2.3.3	Simple Homogeneous Table Builder	242
10.6.2.3.4	Simple SOA Table Builder	242
10.6.2.4	Accessors	242
10.6.2.4.1	Requirements	242
10.6.2.4.2	Accessor Types	243
10.6.2.4.2.1	Row accessor	243
10.6.2.4.2.2	Column accessor	243
10.7	Algorithms	243
10.7.1	Clustering	244
10.7.1.1	K-Means	244
10.7.1.1.1	Lloyd's method	244
10.7.1.1.2	Usage example	245
10.7.1.1.3	API	245
10.7.1.1.3.1	Methods	245
10.7.1.1.3.2	Descriptor	246
10.7.1.1.3.3	Model	247
10.7.1.1.3.4	Training <code>onedal::train</code>	247
10.7.1.1.3.5	Input	247
10.7.1.1.3.6	Result	248
10.7.1.1.3.7	Operation semantics	249
10.7.1.1.3.8	Inference <code>onedal::infer</code>	249
10.7.1.1.3.9	Input	249
10.7.1.1.3.10	Result	250
10.7.1.1.3.11	Operation semantics	251
10.7.2	Nearest Neighbors (kNN)	251
10.7.2.1	k-Nearest Neighbors Classification (k-NN)	251
10.7.2.1.1	Mathematical formulation	251
10.7.2.1.1.1	Training	251
10.7.2.1.1.2	Training method: <i>brute-force</i>	251
10.7.2.1.1.3	Training method: <i>k-d tree</i>	252
10.7.2.1.1.4	Inference	252
10.7.2.1.1.5	Inference method: <i>brute-force</i>	252
10.7.2.1.1.6	Inference method: <i>k-d tree</i>	252
10.7.2.1.2	Programming Interface	252
10.7.2.1.2.1	Descriptor	252
10.7.2.1.2.2	Computational methods	253
10.7.2.1.2.3	Model	254
10.7.2.1.2.4	Training <code>train</code>	254
10.7.2.1.2.5	Input	254
10.7.2.1.2.6	Result	255
10.7.2.1.2.7	Operation	255
10.7.2.1.2.8	Inference <code>infer</code>	256
10.7.2.1.2.9	Input	256
10.7.2.1.2.10	Result	256
10.7.2.1.2.11	Operation	257
10.7.3	Decomposition	257
10.7.3.1	Principal Components Analysis (PCA)	257
10.7.3.1.1	Covariance-based method	258
10.7.3.1.2	SVD-based method	258
10.7.3.1.3	Sign-flip technique	258
10.7.3.1.4	Usage example	258
10.7.3.1.5	API	259
10.7.3.1.5.1	Methods	259
10.7.3.1.5.2	Descriptor	259

10.7.3.1.5.3	Model	260
10.7.3.1.5.4	Training <code>onedal::train</code>	261
10.7.3.1.5.5	Input	261
10.7.3.1.5.6	Result	261
10.7.3.1.5.7	Operation semantics	262
10.7.3.1.5.8	Inference <code>onedal::infer</code>	262
10.7.3.1.5.9	Input	262
10.7.3.1.5.10	Result	263
10.7.3.1.5.11	Operation semantics	264
10.8	Appendix	264
10.8.1	k-d Tree	264
10.8.1.1	Related terms	264
10.9	Bibliography	264
11	oneTBB	265
11.1	General Information	265
11.1.1	Introduction	265
11.1.2	Notational Conventions	265
11.1.3	Identifiers	267
11.1.3.1	Case	267
11.1.3.2	Reserved Identifier Prefixes	267
11.1.4	Named Requirements	267
11.1.4.1	Algorithms	268
11.1.4.1.1	Range	268
11.1.4.1.2	Splittable	269
11.1.4.1.3	ParallelForBody	270
11.1.4.1.4	ParallelReduceBody	270
11.1.4.1.5	ParallelReduceFunc	270
11.1.4.1.6	ParallelReduceReduction	271
11.1.4.1.7	ParallelForEachBody	271
11.1.4.1.7.1	ItemType	272
11.1.4.1.8	ContainerBasedSequence	272
11.1.4.1.9	ParallelScanBody	272
11.1.4.1.10	ParallelScanCombine	273
11.1.4.1.11	ParallelScanFunc	273
11.1.4.1.12	BlockedRangeValue	273
11.1.4.1.13	FilterBody	274
11.1.4.2	Mutexes	274
11.1.4.2.1	Mutex	274
11.1.4.2.2	ReaderWriterMutex	276
11.1.4.3	Containers	278
11.1.4.3.1	HashCompare	278
11.1.4.3.2	ContainerRange	279
11.1.4.4	Task scheduler	279
11.1.4.4.1	SuspendFunc	279
11.1.4.5	Flow Graph	280
11.1.4.5.1	AsyncNodeBody	280
11.1.4.5.2	ContinueNodeBody	280
11.1.4.5.3	GatewayType	281
11.1.4.5.4	FunctionNodeBody	281
11.1.4.5.5	JoinNodeFunctionObject	281
11.1.4.5.6	InputNodeBody	282
11.1.4.5.7	MultifunctionNodeBody	282
11.1.4.5.8	Sequencer	283

11.1.5 Thread Safety	283
11.2 oneTBB Interfaces	283
11.2.1 Configuration	283
11.2.1.1 Namespaces	284
11.2.1.1.1 tbb Namespace	284
11.2.1.1.2 tbb::flow Namespace	284
11.2.1.2 Version Information	284
11.2.1.2.1 TBB_runtime_interface_version Function	285
11.2.1.2.2 TBB_runtime_version Function	285
11.2.1.2.3 TBB_VERSION Environment Variable	285
11.2.1.3 Enabling Debugging Features	285
11.2.1.3.1 TBB_USE_ASSERT Macro	286
11.2.1.3.2 TBB_USE_PROFILING_TOOLS Macro	286
11.2.1.4 Feature Macros	286
11.2.1.4.1 TBB_USE_EXCEPTIONS macro	286
11.2.1.4.2 TBB_USE_GLIBCXX_VERSION macro	286
11.2.2 Algorithms	286
11.2.2.1 Parallel Functions	287
11.2.2.1.1 parallel_for	287
11.2.2.1.2 parallel_reduce	288
11.2.2.1.2.1 Example (Imperative Form)	290
11.2.2.1.2.2 Example with Lambda Expressions	291
11.2.2.1.3 parallel_deterministic_reduce	291
11.2.2.1.4 parallel_scan	293
11.2.2.1.4.1 pre_scan and final_scan Classes	294
11.2.2.1.4.2 pre_scan_tag and final_scan_tag	294
11.2.2.1.4.3 Member functions	294
11.2.2.1.4.4 Example (Imperative Form)	295
11.2.2.1.4.5 Example with Lambda Expressions	296
11.2.2.1.5 parallel_for_each	296
11.2.2.1.5.1 feeder Class	297
11.2.2.1.5.2 feeder	297
11.2.2.1.5.3 Member functions	297
11.2.2.1.5.4 Example	298
11.2.2.1.6 parallel_invoke	298
11.2.2.1.6.1 Example	298
11.2.2.1.7 parallel_pipeline	299
11.2.2.1.7.1 Example	300
11.2.2.1.7.2 filter Class Template	300
11.2.2.1.7.3 filter	300
11.2.2.1.7.4 filter_mode Enumeration	301
11.2.2.1.7.5 filter_mode	301
11.2.2.1.7.6 Member functions	302
11.2.2.1.7.7 Non-member functions	302
11.2.2.1.7.8 Deduction Guides	302
11.2.2.1.7.9 flow_control Class	302
11.2.2.1.7.10 flow_control	302
11.2.2.1.7.11 Member functions	303
11.2.2.1.8 parallel_sort	303
11.2.2.2 Blocked Ranges	304
11.2.2.2.1 blocked_range	304
11.2.2.2.1.1 Member functions	305
11.2.2.2.2 blocked_range2d	306
11.2.2.2.2.1 Member types	307

11.2.2.2.2	Member functions	307
11.2.2.2.3	blocked_range3d	309
11.2.2.3.1	Member types	309
11.2.2.3.2	Member functions	310
11.2.2.3	Partitioners	311
11.2.2.3.1	auto_partitioner	311
11.2.2.3.2	affinity_partitioner	312
11.2.2.3.3	static_partitioner	312
11.2.2.3.4	simple_partitioner	313
11.2.2.4	Split Tags	314
11.2.2.4.1	proportional split	314
11.2.2.4.1.1	Member functions	315
11.2.2.4.2	split	315
11.2.3	Flow Graph	315
11.2.3.1	Graph Class	316
11.2.3.1.1	graph	316
11.2.3.1.1.1	reset_flags enumeration	316
11.2.3.1.1.2	reset_flags Enumeration	316
11.2.3.1.1.3	Member functions	317
11.2.3.2	Nodes	317
11.2.3.2.1	Abstract Interfaces	317
11.2.3.2.1.1	graph_node	318
11.2.3.2.1.2	sender	318
11.2.3.2.1.3	receiver	318
11.2.3.2.2	Properties	319
11.2.3.2.2.1	Forwarding and Buffering	319
11.2.3.2.2.2	Forwarding	319
11.2.3.2.2.3	Buffering	319
11.2.3.2.3	Functional Nodes	320
11.2.3.2.3.1	continue_node	320
11.2.3.2.3.2	Member functions	321
11.2.3.2.3.3	Deduction Guides	323
11.2.3.2.3.4	Example	323
11.2.3.2.3.5	function_node	323
11.2.3.2.3.6	Member functions	324
11.2.3.2.3.7	Deduction Guides	325
11.2.3.2.3.8	Example	325
11.2.3.2.3.9	input_node	325
11.2.3.2.3.10	Member functions	326
11.2.3.2.3.11	Deduction Guides	327
11.2.3.2.3.12	multifunction_node	327
11.2.3.2.3.13	Member types	328
11.2.3.2.3.14	Member functions	328
11.2.3.2.3.15	async_node	329
11.2.3.2.3.16	Member functions	330
11.2.3.2.3.17	Member functions	330
11.2.3.2.3.18	Example	331
11.2.3.2.3.19	Function Nodes Policies	332
11.2.3.2.3.20	Queueing	333
11.2.3.2.3.21	Rejecting	333
11.2.3.2.3.22	Lightweight	333
11.2.3.2.3.23	Example	333
11.2.3.2.3.24	Nodes Priorities	334
11.2.3.2.3.25	Example	334

11.2.3.2.3.26 Predefined Concurrency Limits	336
11.2.3.2.3.27 copy_body	337
11.2.3.2.4 Buffering Nodes	337
11.2.3.2.4.1 overwrite_node	337
11.2.3.2.4.2 Member functions	338
11.2.3.2.4.3 Examples	338
11.2.3.2.4.4 write_once_node	339
11.2.3.2.4.5 Member functions	340
11.2.3.2.4.6 Example	341
11.2.3.2.4.7 buffer_node	342
11.2.3.2.4.8 Member functions	342
11.2.3.2.4.9 queue_node	343
11.2.3.2.4.10 Member functions	343
11.2.3.2.4.11 Example	344
11.2.3.2.4.12 priority_queue_node	344
11.2.3.2.4.13 Member functions	344
11.2.3.2.4.14 Example	345
11.2.3.2.4.15 sequencer_node	345
11.2.3.2.4.16 Member functions	346
11.2.3.2.4.17 Deduction Guides	346
11.2.3.2.4.18 Example	346
11.2.3.2.5 Service Nodes	347
11.2.3.2.5.1 limiter_node	347
11.2.3.2.5.2 Member functions	348
11.2.3.2.5.3 broadcast_node	349
11.2.3.2.5.4 Member functions	349
11.2.3.2.5.5 join_node	350
11.2.3.2.5.6 join_node Policies	351
11.2.3.2.5.7 Member types	352
11.2.3.2.5.8 Member functions	352
11.2.3.2.5.9 Non-Member Types	353
11.2.3.2.5.10 Deduction Guides	353
11.2.3.2.5.11 split_node	354
11.2.3.2.5.12 Member functions	354
11.2.3.2.5.13 indexer_node	355
11.2.3.2.5.14 Member types	356
11.2.3.2.5.15 Member functions	356
11.2.3.2.5.16 composite_node	356
11.2.3.2.5.17 Member functions	358
11.2.3.3 Ports and Edges	359
11.2.3.3.1 input_port	359
11.2.3.3.2 output_port	359
11.2.3.3.3 make_edge	360
11.2.3.3.4 remove_edge	360
11.2.3.4 Special Messages Types	361
11.2.3.4.1 continue_msg	361
11.2.3.4.2 tagged_msg	362
11.2.3.4.2.1 Member functions	362
11.2.3.4.2.2 Non-member functions	363
11.2.3.5 Examples	363
11.2.3.5.1 Dependency Flow Graph Example	363
11.2.3.5.2 Message Flow Graph Example	366
11.2.4 Task Scheduler	367
11.2.4.1 Scheduling controls	368

11.2.4.1.1	<code>task_group_context</code>	368
11.2.4.1.1.1	Member types and constants	368
11.2.4.1.1.2	Member functions	369
11.2.4.1.2	<code>global_control</code>	369
11.2.4.1.2.1	Member types and constants	370
11.2.4.1.2.2	Member functions	370
11.2.4.1.3	Resumable tasks	371
11.2.4.1.3.1	Example	371
11.2.4.2	<code>Task Group</code>	372
11.2.4.2.1	<code>task_group</code>	372
11.2.4.2.1.1	Member functions	372
11.2.4.2.1.2	Non-member functions	373
11.2.4.2.2	<code>task_group_status</code>	373
11.2.4.2.2.1	Member constants	373
11.2.4.3	<code>Task Arena</code>	373
11.2.4.3.1	<code>task_arena</code>	373
11.2.4.3.1.1	Member types and constants	374
11.2.4.3.1.2	Member functions	375
11.2.4.3.2	<code>this_task_arena</code>	376
11.2.4.3.3	<code>task_scheduler_observer</code>	377
11.2.4.3.3.1	Member functions	378
11.2.4.3.3.2	Example	379
11.2.5	<code>Containers</code>	379
11.2.5.1	<code>Sequences</code>	379
11.2.5.1.1	<code>concurrent_vector</code>	379
11.2.5.1.1.1	Class Template Synopsis	380
11.2.5.1.1.2	Requirements	382
11.2.5.1.1.3	Description	383
11.2.5.1.1.4	Exception Safety	383
11.2.5.1.1.5	Member functions	383
11.2.5.1.1.6	Construction, destruction, copying	383
11.2.5.1.1.7	Empty container constructors	383
11.2.5.1.1.8	Constructors from the sequence of elements	384
11.2.5.1.1.9	Copying constructors	384
11.2.5.1.1.10	Moving constructors	385
11.2.5.1.1.11	Destructor	385
11.2.5.1.1.12	Assignment operators	385
11.2.5.1.1.13	<code>assign</code>	386
11.2.5.1.1.14	<code>get_allocator</code>	386
11.2.5.1.1.15	Concurrent growth	386
11.2.5.1.1.16	<code>grow_by</code>	387
11.2.5.1.1.17	<code>grow_to_at_least</code>	387
11.2.5.1.1.18	<code>push_back</code>	388
11.2.5.1.1.19	<code>emplace_back</code>	388
11.2.5.1.1.20	Element access	388
11.2.5.1.1.21	Access by index	389
11.2.5.1.1.22	Access the first and the last element	389
11.2.5.1.1.23	Iterators	389
11.2.5.1.1.24	<code>begin</code> and <code>cbegin</code>	389
11.2.5.1.1.25	<code>end</code> and <code>cend</code>	390
11.2.5.1.1.26	<code>rbegin</code> and <code>crbegin</code>	390
11.2.5.1.1.27	<code>rend</code> and <code>crend</code>	390
11.2.5.1.1.28	Size and capacity	390
11.2.5.1.1.29	<code>size</code>	390

11.2.5.1.1.30	empty	390
11.2.5.1.1.31	max_size	391
11.2.5.1.1.32	capacity	391
11.2.5.1.1.33	Concurrently unsafe operations	391
11.2.5.1.1.34	Reserving	391
11.2.5.1.1.35	Resizing	391
11.2.5.1.1.36	shrink_to_fit	391
11.2.5.1.1.37	clear	392
11.2.5.1.1.38	swap	392
11.2.5.1.1.39	Parallel iteration	392
11.2.5.1.1.40	range member function	392
11.2.5.1.1.41	Non-member functions	392
11.2.5.1.1.42	Non-member binary comparisons	393
11.2.5.1.1.43	Non-member lexicographical comparisons	394
11.2.5.1.1.44	Non-member swap	394
11.2.5.1.1.45	Other	394
11.2.5.1.1.46	Deduction guides	394
11.2.5.2	Queues	395
11.2.5.2.1	concurrent_queue	395
11.2.5.2.1.1	Class Template Synopsis	395
11.2.5.2.1.2	Member functions	396
11.2.5.2.1.3	Construction, destruction, copying	396
11.2.5.2.1.4	Empty container constructors	396
11.2.5.2.1.5	Constructor from the sequence of elements	397
11.2.5.2.1.6	Copying constructors	397
11.2.5.2.1.7	Moving constructors	397
11.2.5.2.1.8	Destructor	398
11.2.5.2.1.9	Concurrently safe member functions	398
11.2.5.2.1.10	Pushing elements	398
11.2.5.2.1.11	Popping elements	398
11.2.5.2.1.12	get_allocator	399
11.2.5.2.1.13	Concurrently unsafe member functions	399
11.2.5.2.1.14	The number of elements	399
11.2.5.2.1.15	clear	399
11.2.5.2.1.16	Iterators	399
11.2.5.2.1.17	unsafe_begin and unsafe_cbegin	399
11.2.5.2.1.18	unsafe_end and unsafe_cend	400
11.2.5.2.1.19	Other	400
11.2.5.2.1.20	Deduction guides	400
11.2.5.2.2	concurrent_bounded_queue	401
11.2.5.2.2.1	Class Template Synopsis	401
11.2.5.2.2.2	Member functions	402
11.2.5.2.2.3	Construction, destruction, copying	402
11.2.5.2.2.4	Empty container constructors	402
11.2.5.2.2.5	Constructor from the sequence of elements	403
11.2.5.2.2.6	Copying constructors	403
11.2.5.2.2.7	Moving constructors	403
11.2.5.2.2.8	Destructor	403
11.2.5.2.2.9	Concurrently safe member functions	404
11.2.5.2.2.10	Pushing elements	404
11.2.5.2.2.11	Popping elements	405
11.2.5.2.2.12	abort	405
11.2.5.2.2.13	Capacity of the queue	405
11.2.5.2.2.14	get_allocator	406

11.2.5.2.2.15	Concurrently unsafe member functions	406
11.2.5.2.2.16	The number of elements	406
11.2.5.2.2.17	clear	406
11.2.5.2.2.18	Iterators	406
11.2.5.2.2.19	unsafe_begin and unsafe_cbegin	406
11.2.5.2.2.20	unsafe_end and unsafe_cend	407
11.2.5.2.2.21	Other	407
11.2.5.2.2.22	Deduction guides	407
11.2.5.2.3	concurrent_priority_queue	408
11.2.5.2.3.1	Class Template Synopsis	408
11.2.5.2.3.2	Member functions	410
11.2.5.2.3.3	Construction, destruction, copying	410
11.2.5.2.3.4	Empty container constructors	410
11.2.5.2.3.5	Constructors from the sequence of elements	410
11.2.5.2.3.6	Copying constructors	411
11.2.5.2.3.7	Moving constructors	411
11.2.5.2.3.8	Destructor	411
11.2.5.2.3.9	Assignment operators	411
11.2.5.2.3.10	assign	412
11.2.5.2.3.11	Size and capacity	412
11.2.5.2.3.12	empty	412
11.2.5.2.3.13	size	413
11.2.5.2.3.14	Concurrently safe modifiers	413
11.2.5.2.3.15	Pushing elements	413
11.2.5.2.3.16	Popping elements	413
11.2.5.2.3.17	Concurrently unsafe modifiers	414
11.2.5.2.3.18	clear	414
11.2.5.2.3.19	swap	414
11.2.5.2.3.20	Non-member functions	414
11.2.5.2.3.21	Non-member swap	415
11.2.5.2.3.22	Non-member binary comparisons	415
11.2.5.2.3.23	Other	415
11.2.5.2.3.24	Deduction guides	415
11.2.5.3	Unordered associative containers	416
11.2.5.3.1	concurrent_hash_map	416
11.2.5.3.1.1	Class Template Synopsis	417
11.2.5.3.1.2	Member classes	420
11.2.5.3.1.3	accessor and const_accessor	420
11.2.5.3.1.4	accessor member class	420
11.2.5.3.1.5	const_accessor member class	420
11.2.5.3.1.6	Member functions	421
11.2.5.3.1.7	Construction and destruction	421
11.2.5.3.1.8	Emptiness	421
11.2.5.3.1.9	Key-value pair access	421
11.2.5.3.1.10	Releasing	422
11.2.5.3.1.11	Member functions	422
11.2.5.3.1.12	Construction, destruction, copying	422
11.2.5.3.1.13	Empty container constructors	422
11.2.5.3.1.14	Constructors from the sequence of elements	422
11.2.5.3.1.15	Copying constructors	423
11.2.5.3.1.16	Moving constructors	423
11.2.5.3.1.17	Destructor	424
11.2.5.3.1.18	Assignment operators	424
11.2.5.3.1.19	get_allocator	424

11.2.5.3.1.20	Concurrently unsafe modifiers	425
11.2.5.3.1.21	clear	425
11.2.5.3.1.22	swap	425
11.2.5.3.1.23	Hash policy	425
11.2.5.3.1.24	Rehashing	425
11.2.5.3.1.25	bucket_count	425
11.2.5.3.1.26	Size and capacity	425
11.2.5.3.1.27	empty	425
11.2.5.3.1.28	size	426
11.2.5.3.1.29	max_size	426
11.2.5.3.1.30	Lookup	426
11.2.5.3.1.31	find	426
11.2.5.3.1.32	count	426
11.2.5.3.1.33	Concurrently safe modifiers	426
11.2.5.3.1.34	Inserting values	427
11.2.5.3.1.35	Inserting sequences of elements	428
11.2.5.3.1.36	Emplacing elements	428
11.2.5.3.1.37	Erasing elements	429
11.2.5.3.1.38	Iterators	429
11.2.5.3.1.39	begin and cbegin	429
11.2.5.3.1.40	end and cend	430
11.2.5.3.1.41	equal_range	430
11.2.5.3.1.42	Parallel iteration	430
11.2.5.3.1.43	range member function	430
11.2.5.3.1.44	Non member functions	430
11.2.5.3.1.45	Non-member swap	431
11.2.5.3.1.46	Non-member binary comparisons	431
11.2.5.3.1.47	Other	431
11.2.5.3.1.48	Deduction guides	431
11.2.5.3.2	concurrent_unordered_map	433
11.2.5.3.2.1	Class Template Synopsis	433
11.2.5.3.2.2	Description	438
11.2.5.3.2.3	Member functions	438
11.2.5.3.2.4	Construction, destruction, copying	438
11.2.5.3.2.5	Empty container constructors	438
11.2.5.3.2.6	Constructors from the sequence of elements	439
11.2.5.3.2.7	Copying constructors	440
11.2.5.3.2.8	Moving constructors	440
11.2.5.3.2.9	Destructor	441
11.2.5.3.2.10	Assignment operators	441
11.2.5.3.2.11	Iterators	442
11.2.5.3.2.12	begin and cbegin	442
11.2.5.3.2.13	end and cend	442
11.2.5.3.2.14	Size and capacity	442
11.2.5.3.2.15	empty	442
11.2.5.3.2.16	size	442
11.2.5.3.2.17	max_size	443
11.2.5.3.2.18	Concurrently safe modifiers	443
11.2.5.3.2.19	Emplacing elements	443
11.2.5.3.2.20	Inserting values	443
11.2.5.3.2.21	Inserting sequences of elements	445
11.2.5.3.2.22	Inserting nodes	445
11.2.5.3.2.23	Concurrently unsafe modifiers	446
11.2.5.3.2.24	Clearing	446

11.2.5.3.2.25	Erasing elements	446
11.2.5.3.2.26	Erasing sequences	447
11.2.5.3.2.27	Extracting nodes	447
11.2.5.3.2.28	swap	448
11.2.5.3.2.29	Element access	448
11.2.5.3.2.30	at	448
11.2.5.3.2.31	operator[]	449
11.2.5.3.2.32	Lookup	449
11.2.5.3.2.33	count	449
11.2.5.3.2.34	find	450
11.2.5.3.2.35	contains	450
11.2.5.3.2.36	equal_range	450
11.2.5.3.2.37	Bucket interface	451
11.2.5.3.2.38	Bucket begin and bucket end	451
11.2.5.3.2.39	The number of buckets	451
11.2.5.3.2.40	Size of the bucket	452
11.2.5.3.2.41	Bucket number	452
11.2.5.3.2.42	Hash policy	452
11.2.5.3.2.43	Load factor	452
11.2.5.3.2.44	Manual rehashing	452
11.2.5.3.2.45	Observers	453
11.2.5.3.2.46	get_allocator	453
11.2.5.3.2.47	hash_function	453
11.2.5.3.2.48	key_eq	453
11.2.5.3.2.49	Parallel iteration	453
11.2.5.3.2.50	range member function	453
11.2.5.3.2.51	Non-member functions	453
11.2.5.3.2.52	Non-member swap	454
11.2.5.3.2.53	Non-member binary comparisons	454
11.2.5.3.2.54	Other	455
11.2.5.3.2.55	Deduction guides	455
11.2.5.3.3	concurrent_unordered_multimap	457
11.2.5.3.3.1	Class Template Synopsis	457
11.2.5.3.3.2	Description	462
11.2.5.3.3.3	Member functions	462
11.2.5.3.3.4	Construction, destruction, copying	462
11.2.5.3.3.5	Empty container constructors	462
11.2.5.3.3.6	Constructors from the sequence of elements	463
11.2.5.3.3.7	Copying constructors	464
11.2.5.3.3.8	Moving constructors	464
11.2.5.3.3.9	Destructor	464
11.2.5.3.3.10	Assignment operators	465
11.2.5.3.3.11	Iterators	465
11.2.5.3.3.12	begin and cbegin	466
11.2.5.3.3.13	end and cend	466
11.2.5.3.3.14	Size and capacity	466
11.2.5.3.3.15	empty	466
11.2.5.3.3.16	size	466
11.2.5.3.3.17	max_size	466
11.2.5.3.3.18	Concurrently safe modifiers	467
11.2.5.3.3.19	Emplacing elements	467
11.2.5.3.3.20	Inserting values	467
11.2.5.3.3.21	Inserting sequences of elements	468
11.2.5.3.3.22	Inserting nodes	468

11.2.5.3.3.23	Merging containers	469
11.2.5.3.3.24	Concurrently unsafe modifiers	469
11.2.5.3.3.25	Clearing	470
11.2.5.3.3.26	Erasing elements	470
11.2.5.3.3.27	Erasing sequences	470
11.2.5.3.3.28	Extracting nodes	471
11.2.5.3.3.29	<i>swap</i>	472
11.2.5.3.3.30	Lookup	472
11.2.5.3.3.31	count	472
11.2.5.3.3.32	find	472
11.2.5.3.3.33	contains	473
11.2.5.3.3.34	<i>equal_range</i>	473
11.2.5.3.3.35	Bucket interface	474
11.2.5.3.3.36	Bucket begin and bucket end	474
11.2.5.3.3.37	The number of buckets	474
11.2.5.3.3.38	Size of the bucket	475
11.2.5.3.3.39	Bucket number	475
11.2.5.3.3.40	Hash policy	475
11.2.5.3.3.41	Load factor	475
11.2.5.3.3.42	Manual rehashing	475
11.2.5.3.3.43	Observers	476
11.2.5.3.3.44	<i>get_allocator</i>	476
11.2.5.3.3.45	<i>hash_function</i>	476
11.2.5.3.3.46	<i>key_eq</i>	476
11.2.5.3.3.47	Parallel iteration	476
11.2.5.3.3.48	range member function	476
11.2.5.3.3.49	Non-member functions	476
11.2.5.3.3.50	Non-member <i>swap</i>	477
11.2.5.3.3.51	Non-member binary comparisons	477
11.2.5.3.3.52	Other	478
11.2.5.3.3.53	Deduction guides	478
11.2.5.3.4	<i>concurrent_unordered_set</i>	480
11.2.5.3.4.1	Class Template Synopsis	480
11.2.5.3.4.2	Description	485
11.2.5.3.4.3	Member functions	485
11.2.5.3.4.4	Construction, destruction, copying	485
11.2.5.3.4.5	Empty container constructors	485
11.2.5.3.4.6	Constructors from the sequence of elements	486
11.2.5.3.4.7	Copying constructors	487
11.2.5.3.4.8	Moving constructors	487
11.2.5.3.4.9	Destructor	487
11.2.5.3.4.10	Assignment operators	487
11.2.5.3.4.11	Iterators	488
11.2.5.3.4.12	<i>begin</i> and <i>cbegin</i>	488
11.2.5.3.4.13	<i>end</i> and <i>cend</i>	489
11.2.5.3.4.14	Size and capacity	489
11.2.5.3.4.15	<i>empty</i>	489
11.2.5.3.4.16	<i>size</i>	489
11.2.5.3.4.17	<i>max_size</i>	489
11.2.5.3.4.18	Concurrently safe modifiers	489
11.2.5.3.4.19	Inserting values	489
11.2.5.3.4.20	Inserting sequences of elements	490
11.2.5.3.4.21	Inserting nodes	491
11.2.5.3.4.22	Emplacing elements	491

11.2.5.3.4.23	Merging containers	492
11.2.5.3.4.24	Concurrently unsafe modifiers	492
11.2.5.3.4.25	Clearing	492
11.2.5.3.4.26	Erasing elements	492
11.2.5.3.4.27	Erasing sequences	493
11.2.5.3.4.28	Extracting nodes	493
11.2.5.3.4.29	<i>swap</i>	494
11.2.5.3.4.30	Lookup	495
11.2.5.3.4.31	<i>count</i>	495
11.2.5.3.4.32	<i>find</i>	495
11.2.5.3.4.33	<i>contains</i>	495
11.2.5.3.4.34	<i>equal_range</i>	496
11.2.5.3.4.35	Bucket interface	496
11.2.5.3.4.36	Bucket begin and bucket end	496
11.2.5.3.4.37	The number of buckets	497
11.2.5.3.4.38	Size of the bucket	497
11.2.5.3.4.39	Bucket number	497
11.2.5.3.4.40	Hash policy	497
11.2.5.3.4.41	Load factor	497
11.2.5.3.4.42	Manual rehashing	498
11.2.5.3.4.43	Observers	498
11.2.5.3.4.44	<i>get_allocator</i>	498
11.2.5.3.4.45	<i>hash_function</i>	498
11.2.5.3.4.46	<i>key_eq</i>	498
11.2.5.3.4.47	Parallel iteration	498
11.2.5.3.4.48	range member function	499
11.2.5.3.4.49	Non-member functions	499
11.2.5.3.4.50	Non-member swap	499
11.2.5.3.4.51	Non-member binary comparisons	500
11.2.5.3.4.52	Other	500
11.2.5.3.4.53	Deduction guides	500
11.2.5.3.5	<i>concurrent_unordered_multiset</i>	502
11.2.5.3.5.1	Class Template Synopsis	502
11.2.5.3.5.2	Description	507
11.2.5.3.5.3	Member functions	507
11.2.5.3.5.4	Construction, destruction, copying	507
11.2.5.3.5.5	Empty container constructors	507
11.2.5.3.5.6	Constructors from the sequence of elements	508
11.2.5.3.5.7	Copying constructors	509
11.2.5.3.5.8	Moving constructors	509
11.2.5.3.5.9	Destructor	509
11.2.5.3.5.10	Assignment operators	509
11.2.5.3.5.11	Iterators	510
11.2.5.3.5.12	<i>begin</i> and <i>cbegin</i>	510
11.2.5.3.5.13	<i>end</i> and <i>cend</i>	511
11.2.5.3.5.14	Size and capacity	511
11.2.5.3.5.15	<i>empty</i>	511
11.2.5.3.5.16	<i>size</i>	511
11.2.5.3.5.17	<i>max_size</i>	511
11.2.5.3.5.18	Concurrently safe modifiers	511
11.2.5.3.5.19	Inserting values	511
11.2.5.3.5.20	Inserting sequences of elements	512
11.2.5.3.5.21	Inserting nodes	512
11.2.5.3.5.22	Emplacing elements	513

11.2.5.3.5.23	Merging containers	513
11.2.5.3.5.24	Concurrently unsafe modifiers	514
11.2.5.3.5.25	Clearing	514
11.2.5.3.5.26	Erasing elements	514
11.2.5.3.5.27	Erasing sequences	515
11.2.5.3.5.28	Extracting nodes	515
11.2.5.3.5.29	<i>swap</i>	516
11.2.5.3.5.30	Lookup	516
11.2.5.3.5.31	<i>count</i>	516
11.2.5.3.5.32	<i>find</i>	517
11.2.5.3.5.33	<i>contains</i>	517
11.2.5.3.5.34	<i>equal_range</i>	518
11.2.5.3.5.35	Bucket interface	518
11.2.5.3.5.36	Bucket begin and bucket end	518
11.2.5.3.5.37	The number of buckets	519
11.2.5.3.5.38	Size of the bucket	519
11.2.5.3.5.39	Bucket number	519
11.2.5.3.5.40	Hash policy	519
11.2.5.3.5.41	Load factor	519
11.2.5.3.5.42	Manual rehashing	520
11.2.5.3.5.43	Observers	520
11.2.5.3.5.44	<i>get_allocator</i>	520
11.2.5.3.5.45	<i>hash_function</i>	520
11.2.5.3.5.46	<i>key_eq</i>	520
11.2.5.3.5.47	Parallel iteration	520
11.2.5.3.5.48	range member function	521
11.2.5.3.5.49	Non-member functions	521
11.2.5.3.5.50	Non-member swap	521
11.2.5.3.5.51	Non-member binary comparisons	522
11.2.5.3.5.52	Other	522
11.2.5.3.5.53	Deduction guides	522
11.2.5.4	Ordered associative containers	524
11.2.5.4.1	<i>concurrent_map</i>	524
11.2.5.4.1.1	Class Template Synopsis	524
11.2.5.4.1.2	Member classes	528
11.2.5.4.1.3	<i>value_compare</i>	528
11.2.5.4.1.4	Class Synopsis	528
11.2.5.4.1.5	Member objects	529
11.2.5.4.1.6	Member functions	529
11.2.5.4.1.7	Member functions	529
11.2.5.4.1.8	Construction, destruction, copying	529
11.2.5.4.1.9	Empty container constructors	529
11.2.5.4.1.10	Constructors from the sequence of elements	530
11.2.5.4.1.11	Copying constructors	530
11.2.5.4.1.12	Moving constructors	531
11.2.5.4.1.13	Destructor	531
11.2.5.4.1.14	Assignment operators	531
11.2.5.4.1.15	Element access	532
11.2.5.4.1.16	<i>at</i>	532
11.2.5.4.1.17	<i>operator[]</i>	532
11.2.5.4.1.18	Iterators	533
11.2.5.4.1.19	<i>begin</i> and <i>cbegin</i>	533
11.2.5.4.1.20	<i>end</i> and <i>cend</i>	533
11.2.5.4.1.21	Size and capacity	533

11.2.5.4.1.22	empty	533
11.2.5.4.1.23	size	533
11.2.5.4.1.24	max_size	534
11.2.5.4.1.25	Concurrently safe modifiers	534
11.2.5.4.1.26	Inserting values	534
11.2.5.4.1.27	Inserting sequences of elements	535
11.2.5.4.1.28	Inserting nodes	535
11.2.5.4.1.29	Emplacing elements	536
11.2.5.4.1.30	Concurrently unsafe modifiers	537
11.2.5.4.1.31	Clearing	537
11.2.5.4.1.32	Erasing elements	537
11.2.5.4.1.33	Erasing sequences	538
11.2.5.4.1.34	Extracting nodes	538
11.2.5.4.1.35	swap	539
11.2.5.4.1.36	Lookup	539
11.2.5.4.1.37	count	539
11.2.5.4.1.38	find	540
11.2.5.4.1.39	contains	540
11.2.5.4.1.40	lower_bound	540
11.2.5.4.1.41	upper_bound	541
11.2.5.4.1.42	equal_range	541
11.2.5.4.1.43	Observers	542
11.2.5.4.1.44	get_allocator	542
11.2.5.4.1.45	key_comp	542
11.2.5.4.1.46	value_comp	542
11.2.5.4.1.47	Parallel iteration	542
11.2.5.4.1.48	range member function	542
11.2.5.4.1.49	Non-member functions	542
11.2.5.4.1.50	Non-member swap	543
11.2.5.4.1.51	Non-member binary comparisons	543
11.2.5.4.1.52	Non-member lexicographical comparisons	544
11.2.5.4.1.53	Other	544
11.2.5.4.1.54	Deduction guides	544
11.2.5.4.2	concurrent_multimap	546
11.2.5.4.2.1	Class Template Synopsis	546
11.2.5.4.2.2	Member classes	550
11.2.5.4.2.3	value_compare	550
11.2.5.4.2.4	Class Synopsis	550
11.2.5.4.2.5	Member objects	550
11.2.5.4.2.6	Member functions	550
11.2.5.4.2.7	Member functions	551
11.2.5.4.2.8	Construction, destruction, copying	551
11.2.5.4.2.9	Empty container constructors	551
11.2.5.4.2.10	Constructors from the sequence of elements	551
11.2.5.4.2.11	Copying constructors	552
11.2.5.4.2.12	Moving constructors	552
11.2.5.4.2.13	Destructor	552
11.2.5.4.2.14	Assignment operators	552
11.2.5.4.2.15	Iterators	553
11.2.5.4.2.16	begin and cbegin	553
11.2.5.4.2.17	end and cend	553
11.2.5.4.2.18	Size and capacity	554
11.2.5.4.2.19	empty	554
11.2.5.4.2.20	size	554

11.2.5.4.2.21	max_size	554
11.2.5.4.2.22	Concurrently safe modifiers	554
11.2.5.4.2.23	Emplacing elements	554
11.2.5.4.2.24	Inserting values	555
11.2.5.4.2.25	Inserting sequences of elements	556
11.2.5.4.2.26	Inserting nodes	556
11.2.5.4.2.27	Merging containers	557
11.2.5.4.2.28	Concurrently unsafe modifiers	557
11.2.5.4.2.29	Clearing	557
11.2.5.4.2.30	Erasing elements	557
11.2.5.4.2.31	Erasing sequences	558
11.2.5.4.2.32	Extracting nodes	558
11.2.5.4.2.33	swap	559
11.2.5.4.2.34	Lookup	560
11.2.5.4.2.35	count	560
11.2.5.4.2.36	find	560
11.2.5.4.2.37	contains	561
11.2.5.4.2.38	lower_bound	561
11.2.5.4.2.39	upper_bound	561
11.2.5.4.2.40	equal_range	562
11.2.5.4.2.41	Observers	562
11.2.5.4.2.42	get_allocator	562
11.2.5.4.2.43	key_comp	562
11.2.5.4.2.44	value_comp	563
11.2.5.4.2.45	Parallel iteration	563
11.2.5.4.2.46	range member function	563
11.2.5.4.2.47	Non-member functions	563
11.2.5.4.2.48	Non-member swap	564
11.2.5.4.2.49	Non-member binary comparisons	564
11.2.5.4.2.50	Non-member lexicographical comparisons	564
11.2.5.4.2.51	Other	565
11.2.5.4.2.52	Deduction guides	565
11.2.5.4.3	concurrent_set	566
11.2.5.4.3.1	Class Template Synopsis	566
11.2.5.4.3.2	Member functions	570
11.2.5.4.3.3	Construction, destruction, copying	570
11.2.5.4.3.4	Empty container constructors	570
11.2.5.4.3.5	Constructors from the sequence of elements	570
11.2.5.4.3.6	Copying constructors	571
11.2.5.4.3.7	Moving constructors	571
11.2.5.4.3.8	Destructor	572
11.2.5.4.3.9	Assignment operators	572
11.2.5.4.3.10	Iterators	572
11.2.5.4.3.11	begin and cbegin	573
11.2.5.4.3.12	end and cend	573
11.2.5.4.3.13	Size and capacity	573
11.2.5.4.3.14	empty	573
11.2.5.4.3.15	size	573
11.2.5.4.3.16	max_size	573
11.2.5.4.3.17	Concurrently safe modifiers	574
11.2.5.4.3.18	Inserting values	574
11.2.5.4.3.19	Inserting sequences of elements	575
11.2.5.4.3.20	Inserting nodes	575
11.2.5.4.3.21	Emplacing elements	576

11.2.5.4.3.22	Merging containers	576
11.2.5.4.3.23	Concurrently unsafe modifiers	577
11.2.5.4.3.24	Clearing	577
11.2.5.4.3.25	Erasing elements	577
11.2.5.4.3.26	Erasing sequences	578
11.2.5.4.3.27	Extracting nodes	578
11.2.5.4.3.28	<i>swap</i>	579
11.2.5.4.3.29	Lookup	579
11.2.5.4.3.30	<i>count</i>	579
11.2.5.4.3.31	<i>find</i>	579
11.2.5.4.3.32	<i>contains</i>	580
11.2.5.4.3.33	<i>lower_bound</i>	580
11.2.5.4.3.34	<i>upper_bound</i>	580
11.2.5.4.3.35	<i>equal_range</i>	581
11.2.5.4.3.36	Observers	581
11.2.5.4.3.37	<i>get_allocator</i>	581
11.2.5.4.3.38	<i>key_comp</i>	581
11.2.5.4.3.39	<i>value_comp</i>	582
11.2.5.4.3.40	Parallel iteration	582
11.2.5.4.3.41	range member function	582
11.2.5.4.3.42	Non-member functions	582
11.2.5.4.3.43	Non-member <i>swap</i>	583
11.2.5.4.3.44	Non-member binary comparisons	583
11.2.5.4.3.45	Non-member lexicographical comparisons	583
11.2.5.4.3.46	Other	584
11.2.5.4.3.47	Deduction guides	584
11.2.5.4.4	<i>concurrent_multiset</i>	585
11.2.5.4.4.1	Class Template Synopsis	585
11.2.5.4.4.2	Member functions	589
11.2.5.4.4.3	Construction, destruction, copying	589
11.2.5.4.4.4	Empty container constructors	589
11.2.5.4.4.5	Constructors from the sequence of elements	589
11.2.5.4.4.6	Copying constructors	590
11.2.5.4.4.7	Moving constructors	590
11.2.5.4.4.8	Destructor	590
11.2.5.4.4.9	Assignment operators	591
11.2.5.4.4.10	Iterators	591
11.2.5.4.4.11	<i>begin</i> and <i>cbegin</i>	591
11.2.5.4.4.12	<i>end</i> and <i>cend</i>	592
11.2.5.4.4.13	Size and capacity	592
11.2.5.4.4.14	<i>empty</i>	592
11.2.5.4.4.15	<i>size</i>	592
11.2.5.4.4.16	<i>max_size</i>	592
11.2.5.4.4.17	Concurrently safe modifiers	592
11.2.5.4.4.18	Inserting values	592
11.2.5.4.4.19	Inserting sequences of elements	593
11.2.5.4.4.20	Inserting nodes	594
11.2.5.4.4.21	Emplacing elements	594
11.2.5.4.4.22	Merging containers	595
11.2.5.4.4.23	Concurrently unsafe modifiers	595
11.2.5.4.4.24	Clearing	595
11.2.5.4.4.25	Erasing elements	595
11.2.5.4.4.26	Erasing sequences	596
11.2.5.4.4.27	Extracting nodes	596

11.2.5.4.4.28	swap	597
11.2.5.4.4.29	Lookup	597
11.2.5.4.4.30	count	597
11.2.5.4.4.31	find	598
11.2.5.4.4.32	contains	598
11.2.5.4.4.33	lower_bound	599
11.2.5.4.4.34	upper_bound	599
11.2.5.4.4.35	equal_range	599
11.2.5.4.4.36	Observers	600
11.2.5.4.4.37	get_allocator	600
11.2.5.4.4.38	key_comp	600
11.2.5.4.4.39	value_comp	600
11.2.5.4.4.40	Parallel iteration	600
11.2.5.4.4.41	range member function	601
11.2.5.4.4.42	Non-member functions	601
11.2.5.4.4.43	Non-member swap	602
11.2.5.4.4.44	Non-member binary comparisons	602
11.2.5.4.4.45	Non-member lexicographical comparisons	602
11.2.5.4.4.46	Other	603
11.2.5.4.4.47	Deduction guides	603
11.2.5.5	Auxiliary classes	604
11.2.5.5.1	tbb_hash_compare	604
11.2.5.5.1.1	Class Template Synopsis	604
11.2.5.5.1.2	Member functions	604
11.2.5.5.2	Node handles	604
11.2.5.5.2.1	Class synopsis	605
11.2.5.5.2.2	Member functions	606
11.2.5.5.2.3	Constructors	606
11.2.5.5.2.4	Assignment	606
11.2.5.5.2.5	Destructor	606
11.2.5.5.2.6	Swap	606
11.2.5.5.2.7	State	607
11.2.5.5.2.8	Access to the stored element	607
11.2.5.5.2.9	get_allocator	607
11.2.6	Thread Local Storage	608
11.2.6.1	combinable	608
11.2.6.1.1	Member functions	608
11.2.6.2	enumerable_thread_specific	610
11.2.6.2.1	Member functions	612
11.2.6.2.1.1	Construction, destruction, copying	612
11.2.6.2.1.2	Empty container constructors	612
11.2.6.2.1.3	Copying constructors	613
11.2.6.2.1.4	Moving constructors	613
11.2.6.2.1.5	Destructor	613
11.2.6.2.1.6	Assignment operators	613
11.2.6.2.1.7	Concurrently safe modifiers	614
11.2.6.2.1.8	Concurrently unsafe modifiers	614
11.2.6.2.1.9	clear	614
11.2.6.2.1.10	Size and capacity	614
11.2.6.2.1.11	Iteration	615
11.2.6.2.1.12	Combining	615
11.2.6.2.2	Non-member types and constants	616
11.2.6.3	flattened2d	616
11.2.6.3.1	Member functions	617

11.2.6.3.2	Non-member functions	618
11.3	oneTBB Auxiliary Interfaces	618
11.3.1	Memory Allocation	618
11.3.1.1	Allocators	618
11.3.1.1.1	tbb_allocator	618
11.3.1.1.1.1	Member Functions	619
11.3.1.1.1.2	Non-member Functions	619
11.3.1.1.2	scalable_allocator	620
11.3.1.1.2.1	Member Functions	620
11.3.1.1.2.2	Non-member Functions	620
11.3.1.1.3	cache_aligned_allocator	621
11.3.1.1.3.1	Member Functions	622
11.3.1.1.3.2	Non-member Functions	622
11.3.1.2	Memory Resources	622
11.3.1.2.1	cache_aligned_resource	623
11.3.1.2.1.1	Member Functions	623
11.3.1.2.2	scalable_memory_resource	624
11.3.1.3	Library Functions	624
11.3.1.3.1	C Interface to Scalable Allocator	624
11.3.2	Mutual Exclusion	627
11.3.2.1	Mutex Classes	627
11.3.2.1.1	spin_mutex	627
11.3.2.1.1.1	Member classes	628
11.3.2.1.1.2	Member functions	628
11.3.2.1.2	spin_rw_mutex	628
11.3.2.1.2.1	Member classes	629
11.3.2.1.2.2	Member functions	629
11.3.2.1.3	speculative_spin_mutex	629
11.3.2.1.3.1	Member classes	630
11.3.2.1.3.2	Member functions	630
11.3.2.1.4	speculative_spin_rw_mutex	630
11.3.2.1.4.1	Member classes	631
11.3.2.1.4.2	Member functions	631
11.3.2.1.5	queuing_mutex	631
11.3.2.1.5.1	Member classes	632
11.3.2.1.5.2	Member functions	632
11.3.2.1.6	queuing_rw_mutex	632
11.3.2.1.6.1	Member classes	633
11.3.2.1.6.2	Member functions	633
11.3.2.1.7	null_mutex	633
11.3.2.1.7.1	Member classes	633
11.3.2.1.7.2	Member functions	634
11.3.2.1.8	null_rw_mutex	634
11.3.2.1.8.1	Member classes	635
11.3.2.1.8.2	Member functions	635
11.3.3	Timing	635
11.3.3.1	Syntax	635
11.3.3.2	Classes	636
11.3.3.2.1	tick_count class	636
11.3.3.2.2	tick_count::interval_t class	636
11.3.3.2.3	Non-member functions	637
12	oneVPL	638
12.1	oneVPL for Intel® Media Software Development Kit Users	638

12.1.1	oneVPL Usability Enhancements	638
12.1.2	Obsolete MSDK Features omitted from oneVPL	638
12.1.3	MSDK API's not present in oneVPL	639
12.1.4	oneVPL API's not present in MSDK	640
12.2	oneVPL API versioning	641
12.3	Acronyms and Abbreviations	641
12.4	Architecture	642
12.4.1	Video Decoding	643
12.4.2	Video Encoding	643
12.4.3	Video Processing	644
12.5	Programming Guide	646
12.5.1	Status Codes	647
12.5.2	SDK Session	647
12.5.2.1	Media SDK dispatcher (legacy)	647
12.5.2.2	oneVPL diapatcher	648
12.5.2.3	Multiple Sessions	649
12.5.3	Frame and Fields	650
12.5.3.1	Frame Surface Locking	650
12.5.4	Decoding Procedures	651
12.5.4.1	Bitstream Repositioning	653
12.5.4.2	Broken Streams Handling	653
12.5.4.3	VP8 Specific Details	654
12.5.4.4	JPEG	654
12.5.4.5	Multi-view video decoding	656
12.5.5	Encoding Procedures	657
12.5.5.1	Encoding procedure	657
12.5.5.2	Configuration Change	658
12.5.5.3	External Bit Rate Control	659
12.5.5.4	JPEG	663
12.5.5.5	Multi-view video encoding	664
12.5.6	Video Processing Procedures	665
12.5.6.1	Configuration	666
12.5.6.2	Region of Interest	668
12.5.6.3	Multi-view video processing	668
12.5.7	Transcoding Procedures	669
12.5.7.1	Asynchronous Pipeline	669
12.5.7.2	Surface Pool Allocation	670
12.5.7.3	Pipeline Error Reporting	671
12.5.8	Working with hardware acceleration	672
12.5.8.1	Working with multiple Intel media devices	672
12.5.8.2	Working with video memory	674
12.5.8.3	Working with Microsoft* DirectX* Applications	675
12.5.8.4	Working with VA API Applications	677
12.5.8.5	Memory Allocation and External Allocators	678
12.5.8.6	External memory managment	678
12.5.8.7	Internal memory managment	679
12.5.8.8	mfxFrameSurfaceInterface	680
12.5.8.9	Hardware Device Error Handling	680
12.6	Summary Tables	681
12.6.1	Mandatory API reference	681
12.6.1.1	Functions per API Version	681
12.7	Appendices	682
12.7.1	Configuration Parameter Constraints	682
12.7.2	Multiple-Segment Encoding	688

12.7.3	Streaming and Video Conferencing Features	689
12.7.3.1	Dynamic Bitrate Change	689
12.7.3.2	Dynamic Resolution Change	690
12.7.3.3	Dynamic reference frame scaling	690
12.7.3.4	Forced Key Frame Generation	690
12.7.3.5	Reference List Selection	691
12.7.3.6	Low Latency Encoding and Decoding	691
12.7.3.7	Reference Picture Marking Repetition SEI message	691
12.7.3.8	Long-term Reference frame	692
12.7.3.9	Temporal scalability	692
12.7.4	Switchable Graphics and Multiple Monitors	693
12.7.4.1	Switchable Graphics	693
12.7.4.2	Multiple Monitors	693
12.7.4.3	Working directly with VA API for Linux*	694
12.7.4.4	CQP HRD mode encoding	696
12.8	oneVPL API Reference	697
12.8.1	Basic Types	697
12.8.2	Typedefs	698
12.8.3	oneVPL Dispatcher API	698
12.8.3.1	Defines	698
12.8.3.2	Structures	698
12.8.3.3	mfxVariant	698
12.8.3.4	mfxDecoderDescription	700
12.8.3.5	mfxEncoderDescription	701
12.8.3.6	mfxVPPDescription	702
12.8.3.7	mfxImplDescription	704
12.8.3.8	Functions	705
12.8.4	Enums	708
12.8.4.1	mfxStatus	708
12.8.4.2	mfxIMPL	710
12.8.4.3	mfxImplCapsDeliveryFormat	711
12.8.4.4	mfxPriority	711
12.8.4.5	GPUCopy	711
12.8.4.6	PlatformCodeName	711
12.8.4.7	mfxMediaAdapterType	712
12.8.4.8	mfxMemoryFlags	712
12.8.4.9	mfxResourceType	713
12.8.4.10	ColorFourCC	713
12.8.4.11	ChromaFormatIdc	715
12.8.4.12	PicStruct	715
12.8.4.13	Frame Data Flags	716
12.8.4.14	Corruption	716
12.8.4.15	TimeStampCalc	717
12.8.4.16	IOPattern	717
12.8.4.17	CodecFormatFourCC	717
12.8.4.18	CodecProfile	718
12.8.4.19	CodecLevel	719
12.8.4.20	HEVC Tiers	721
12.8.4.21	GopOptFlag	721
12.8.4.22	TargetUsage	721
12.8.4.23	RateControlMethod	722
12.8.4.24	TrellisControl	723
12.8.4.25	BRefControl	723
12.8.4.26	LookAheadDownSampling	723

12.8.4.27	BPSEIControl	724
12.8.4.28	SkipFrame	724
12.8.4.29	IntraRefreshTypes	724
12.8.4.30	WeightedPred	724
12.8.4.31	PRefType	725
12.8.4.32	ScenarioInfo	725
12.8.4.33	ContentInfo	725
12.8.4.34	IntraPredBlockSize/InterPredBlockSize	726
12.8.4.35	MVPrecision	726
12.8.4.36	CodingOptionValue	726
12.8.4.37	BitstreamDataFlag	726
12.8.4.38	ExtendedBufferID	727
12.8.4.39	PayloadCtrlFlags	731
12.8.4.40	ExtMemFrameType	731
12.8.4.41	FrameType	732
12.8.4.42	MfxNalUnitType	733
12.8.4.43	mfxHandleType	734
12.8.4.44	mfxSkipMode	734
12.8.4.45	FrcAlgm	735
12.8.4.46	ImageStabMode	735
12.8.4.47	InsertHDRPayload	735
12.8.4.48	LongTermIdx	735
12.8.4.49	TransferMatrix	735
12.8.4.50	NominalRange	736
12.8.4.51	ROImode	736
12.8.4.52	DeinterlacingMode	736
12.8.4.53	TelecinePattern	737
12.8.4.54	VPPFieldProcessingMode	737
12.8.4.55	PicType	737
12.8.4.56	MBQPMode	738
12.8.4.57	GeneralConstraintFlags	738
12.8.4.58	SampleAdaptiveOffset	738
12.8.4.59	ErrorTypes	739
12.8.4.60	HEVCRegionType	739
12.8.4.61	HEVCRegionEncoding	739
12.8.4.62	Angle	739
12.8.4.63	ScalingMode	740
12.8.4.64	InterpolationMode	740
12.8.4.65	MirroringType	740
12.8.4.66	ChromaSiting	740
12.8.4.67	VP9ReferenceFrame	741
12.8.4.68	SegmentIdBlockSize	741
12.8.4.69	SegmentFeature	741
12.8.4.70	MCTFTemporalMode	742
12.8.4.71	mfxComponentType	742
12.8.4.72	PartialBitstreamOutput	742
12.8.4.73	BRCStatus	742
12.8.4.74	Rotation	743
12.8.4.75	JPEGColorFormat	743
12.8.4.76	JPEGScanType	743
12.8.4.77	Protected	744
12.8.5	Structs	744
12.8.5.1	mfxRange32U	744
12.8.5.2	mfxI16Pair	744

12.8.5.3 mfxHDLPair	744
12.8.5.4 mfx Version	745
12.8.5.5 mfxStructVersion	745
12.8.5.6 mfxPlatform	746
12.8.5.7 mfxInitParam	746
12.8.5.8 mfxInfoMFX	747
12.8.5.9 mfxFrameInfo	751
12.8.5.10 mfxVideoParam	753
12.8.5.11 mfxFrameData	754
12.8.5.12 mfxFrameSurfaceInterface	757
12.8.5.13 mfxFrameSurface1	760
12.8.5.14 mfxBitstream	760
12.8.5.15 mfxEncodeStat	761
12.8.5.16 mfxDecodeStat	761
12.8.5.17 mfxPayload	762
12.8.5.18 mfxEncodeCtrl	763
12.8.5.19 mfxFrameAllocRequest	764
12.8.5.20 mfxFrameAllocResponse	764
12.8.5.21 mfxFrameAllocator	764
12.8.5.22 mfxComponentInfo	766
12.8.5.23 mfxAdapterInfo	766
12.8.5.24 mfxAdaptersInfo	767
12.8.5.25 mfxQPandMode	767
12.8.5.26 VPP Structures	767
12.8.5.26.1 mfxInfoVPP	767
12.8.5.26.2 mfxVPPStat	768
12.8.5.27 Extension buffers structures	768
12.8.5.27.1 mfxExtBuffer	768
12.8.5.27.2 mfxExtCodingOption	768
12.8.5.27.3 mfxExtCodingOption2	770
12.8.5.27.4 mfxExtCodingOption3	774
12.8.5.27.5 mfxExtCodingOptionSPSPPS	778
12.8.5.27.6 mfxExtInsertHeaders	779
12.8.5.27.7 mfxExtCodingOptionVPS	779
12.8.5.27.8 mfxExtThreadsParam	780
12.8.5.27.9 mfxExtVideoSignalInfo	780
12.8.5.27.10mfxExtAVCRefListCtrl	781
12.8.5.27.11mfxExtMasteringDisplayColourVolume	782
12.8.5.27.12mfxExtContentLightLevelInfo	782
12.8.5.27.13mfxExtPictureTimingSEI	783
12.8.5.27.14mfxExtAvcTemporalLayers	784
12.8.5.27.15mfxExtEncoderCapability	784
12.8.5.27.16mfxExtEncoderResetOption	785
12.8.5.27.17mfxExtAVCEncodedFrameInfo	786
12.8.5.27.18mfxExtEncoderROI	787
12.8.5.27.19mfxExtEncoderIPCMArea	788
12.8.5.27.20mfxExtAVCRefLists	788
12.8.5.27.21mfxExtChromaLocInfo	789
12.8.5.27.22mfxExtMBForceIntra	789
12.8.5.27.23mfxExtMBQP	790
12.8.5.27.24mfxExtHEVCTiles	790
12.8.5.27.25mfxExtMBDisableSkipMap	791
12.8.5.27.26mfxExtHEVCParam	791
12.8.5.27.27mfxExtDecodeErrorReport	792

12.8.5.27.28mfxExtDecodedFrameInfo	792
12.8.5.27.29mfxExtTimeCode	792
12.8.5.27.30mfxExtHEVCRegion	793
12.8.5.27.31mfxExtPredWeightTable	793
12.8.5.27.32mfxExtAVCRoundingOffset	794
12.8.5.27.33mfxExtDirtyRect	794
12.8.5.27.34mfxExtMoveRect	795
12.8.5.27.35mfxExtMVOVerPicBoundaries	796
12.8.5.27.36mfxVP9SegmentParam	796
12.8.5.27.37mfxExtVP9Segmentation	797
12.8.5.27.38mfxVP9TemporalLayer	798
12.8.5.27.39mfxExtVP9TemporalLayers	798
12.8.5.27.40mfxExtVP9Param	799
12.8.5.27.41mfxEncodedUnitInfo	800
12.8.5.27.42mfxExtEncodedUnitsInfo	800
12.8.5.27.43mfxExtPartialBitstreamParam	801
12.8.5.28 VPP Extention buffers	801
12.8.5.28.1 mfxExtVPPDoNotUse	801
12.8.5.28.2 mfxExtVPPDoUse	802
12.8.5.29 mfxExtVPPDenoise	802
12.8.5.29.1 mfxExtVPPDetail	803
12.8.5.29.2 mfxExtVPPProcAmp	803
12.8.5.29.3 mfxExtVPPDeinterlacing	804
12.8.5.29.4 mfxExtEncodedSlicesInfo	804
12.8.5.29.5 mfxExtVppAuxData	805
12.8.5.29.6 mfxExtVPPFrameRateConversion	805
12.8.5.29.7 mfxExtVPPImageStab	806
12.8.5.29.8 mfxVPPCompInputStream	806
12.8.5.29.9 mfxExtVPPComposite	807
12.8.5.29.10mfxExtVPPVideoSignalInfo	809
12.8.5.29.11mfxExtVPPFieldProcessing	809
12.8.5.29.12mfxExtDecVideoProcessing	810
12.8.5.29.13mfxExtVPPRotation	811
12.8.5.29.14mfxExtVPPScaling	811
12.8.5.29.15mfxExtVPPMirroring	812
12.8.5.29.16mfxExtVPPColorFill	812
12.8.5.29.17mfxExtColorConversion	812
12.8.5.29.18mfxExtVppMctf	813
12.8.5.30 Bit Rate Control Extension Buffers	813
12.8.5.30.1 mfxBRCFrameParam	813
12.8.5.30.2 mfxBRCFrameCtrl	814
12.8.5.30.3 mfxBRCFrameStatus	815
12.8.5.30.4 mfxExtBRC	815
12.8.5.31 VP8 Extenrion Buffers	817
12.8.5.31.1 mfxExtVP8CodingOption	817
12.8.5.32 JPEG Extension Buffers	818
12.8.5.32.1 mfxExtJPEGQuantTables	818
12.8.5.32.2 mfxExtJPEGHuffmanTables	818
12.8.5.33 MVC Extension Buffers	819
12.8.5.33.1 mfxMVCViewDependency	819
12.8.5.33.2 mfxMVCOperationPoint	820
12.8.5.33.3 mfxExtMVCSeqDesc	820
12.8.5.33.4 mfxExtMVCTargetViews	821
12.8.5.33.5 mfxExtEncToolsConfig	821

12.8.5.34 PCP Extension Buffers	822
12.8.6 Functions	823
12.8.6.1 Implementation Capabilities	823
12.8.6.2 Session Management	823
12.8.6.3 VideoCORE	826
12.8.6.4 Memory	827
12.8.6.5 VideoENCODE	828
12.8.6.6 VideoDECODE	832
12.8.6.7 VideoVPP	837
13 oneMKL	840
13.1 oneMKL Architecture	840
13.1.1 Execution Model	840
13.1.1.1 Use of Queues	840
13.1.1.1.1 Non-Member Functions	841
13.1.1.1.2 Member Functions	841
13.1.1.2 Device Usage	841
13.1.1.3 Asynchronous Execution	841
13.1.1.3.1 Synchronization When Using Buffers	842
13.1.1.3.2 Synchronization When Using USM APIs	842
13.1.1.4 Host Thread Safety	842
13.1.2 Memory Model	842
13.1.2.1 The Buffer Memory Model	842
13.1.2.2 Unified Shared Memory Model	843
13.1.3 API Design	843
13.1.3.1 oneMKL namespaces	843
13.1.3.2 Standard C++ datatype usage	843
13.1.3.3 DPC++ datatype usage	844
13.1.3.4 oneMKL defined datatypes	844
13.1.4 Exceptions and Error Handling	846
13.1.5 Other Features	846
13.1.5.1 oneMKL Specification Versioning Strategy	846
13.1.5.2 Current Version of this oneMKL Specification	846
13.1.5.3 Pre/Post Condition Checking	846
13.2 oneMKL Domains	846
13.2.1 Dense Linear Algebra	846
13.2.1.1 Matrix Storage	847
13.2.1.2 BLAS Routines	853
13.2.1.2.1 BLAS Level 1 Routines	853
13.2.1.2.1.1 asum	854
13.2.1.2.1.2 asum (Buffer Version)	854
13.2.1.2.1.3 asum (USM Version)	855
13.2.1.2.1.4 axpy	855
13.2.1.2.1.5 axpy (Buffer Version)	856
13.2.1.2.1.6 axpy (USM Version)	856
13.2.1.2.1.7 copy	857
13.2.1.2.1.8 copy (Buffer Version)	858
13.2.1.2.1.9 copy (USM Version)	858
13.2.1.2.1.10 dot	859
13.2.1.2.1.11 dot (Buffer Version)	859
13.2.1.2.1.12 dot (USM Version)	860
13.2.1.2.1.13 sdsdot	861
13.2.1.2.1.14 sdsdot (Buffer Version)	861
13.2.1.2.1.15 sdsdot (USM Version)	862

13.2.1.2.1.16	dotc	862
13.2.1.2.1.17	dotc (Buffer Version)	863
13.2.1.2.1.18	dotc (USM Version)	863
13.2.1.2.1.19	dotu	864
13.2.1.2.1.20	dotu (Buffer Version)	865
13.2.1.2.1.21	dotu (USM Version)	865
13.2.1.2.1.22	nrm2	866
13.2.1.2.1.23	nrm2 (Buffer Version)	866
13.2.1.2.1.24	nrm2 (USM Version)	867
13.2.1.2.1.25	rot	867
13.2.1.2.1.26	rot (Buffer Version)	868
13.2.1.2.1.27	rot (USM Version)	868
13.2.1.2.1.28	rotg	869
13.2.1.2.1.29	rotg (Buffer Version)	870
13.2.1.2.1.30	rotg (USM Version)	870
13.2.1.2.1.31	rotm	871
13.2.1.2.1.32	rotm (Buffer Version)	871
13.2.1.2.1.33	rotm (USM Version)	873
13.2.1.2.1.34	rotmg	874
13.2.1.2.1.35	rotmg (Buffer Version)	875
13.2.1.2.1.36	rotmg (USM Version)	876
13.2.1.2.1.37	scal	877
13.2.1.2.1.38	scal (Buffer Version)	878
13.2.1.2.1.39	scal (USM Version)	878
13.2.1.2.1.40	swap	879
13.2.1.2.1.41	swap (Buffer Version)	879
13.2.1.2.1.42	swap (USM Version)	880
13.2.1.2.1.43	iamax	880
13.2.1.2.1.44	iamax (Buffer Version)	881
13.2.1.2.1.45	iamax (USM Version)	881
13.2.1.2.1.46	iamin	882
13.2.1.2.1.47	iamin (Buffer Version)	883
13.2.1.2.1.48	iamin (USM Version)	883
13.2.1.2.2	BLAS Level 2 Routines	884
13.2.1.2.2.1	gbmv	884
13.2.1.2.2.2	gbmv (Buffer Version)	885
13.2.1.2.2.3	gbmv (USM Version)	886
13.2.1.2.2.4	gemv	887
13.2.1.2.2.5	gemv (Buffer Version)	887
13.2.1.2.2.6	gemv (USM Version)	888
13.2.1.2.2.7	ger	889
13.2.1.2.2.8	ger (Buffer Version)	890
13.2.1.2.2.9	ger (USM Version)	890
13.2.1.2.2.10	gerc	891
13.2.1.2.2.11	gerc (Buffer Version)	892
13.2.1.2.2.12	gerc (USM Version)	892
13.2.1.2.2.13	geru	893
13.2.1.2.2.14	geru (Buffer Version)	894
13.2.1.2.2.15	geru (USM Version)	894
13.2.1.2.2.16	hbmv	895
13.2.1.2.2.17	hbmv (Buffer Version)	896
13.2.1.2.2.18	hbmv (USM Version)	896
13.2.1.2.2.19	hemv	897
13.2.1.2.2.20	hemv (Buffer Version)	898

13.2.1.2.2.21	hemv (USM Version)	899
13.2.1.2.2.22	her	900
13.2.1.2.2.23	her (Buffer Version)	900
13.2.1.2.2.24	her (USM Version)	901
13.2.1.2.2.25	her2	902
13.2.1.2.2.26	her2 (Buffer Version)	902
13.2.1.2.2.27	her2 (USM Version)	903
13.2.1.2.2.28	hpmv	904
13.2.1.2.2.29	hpmv (Buffer Version)	904
13.2.1.2.2.30	hpmv (USM Version)	905
13.2.1.2.2.31	hpr	906
13.2.1.2.2.32	hpr (Buffer Version)	906
13.2.1.2.2.33	hpr (USM Version)	907
13.2.1.2.2.34	hpr2	908
13.2.1.2.2.35	hpr2 (Buffer Version)	908
13.2.1.2.2.36	hpr2 (USM Version)	909
13.2.1.2.2.37	sbmv	910
13.2.1.2.2.38	sbmv (Buffer Version)	910
13.2.1.2.2.39	sbmv (USM Version)	911
13.2.1.2.2.40	spmv	912
13.2.1.2.2.41	spmv (Buffer Version)	913
13.2.1.2.2.42	spmv (USM Version)	913
13.2.1.2.2.43	spr	914
13.2.1.2.2.44	spr (Buffer Version)	915
13.2.1.2.2.45	spr (USM Version)	915
13.2.1.2.2.46	spr2	916
13.2.1.2.2.47	spr2 (Buffer Version)	916
13.2.1.2.2.48	spr2 (USM Version)	917
13.2.1.2.2.49	symv	918
13.2.1.2.2.50	symv (Buffer Version)	918
13.2.1.2.2.51	symv (USM Version)	919
13.2.1.2.2.52	syr	920
13.2.1.2.2.53	syr (Buffer Version)	920
13.2.1.2.2.54	syr (USM Version)	921
13.2.1.2.2.55	syr2	922
13.2.1.2.2.56	syr2 (Buffer Version)	922
13.2.1.2.2.57	syr2 (USM Version)	923
13.2.1.2.2.58	tbmv	924
13.2.1.2.2.59	tbmv (Buffer Version)	924
13.2.1.2.2.60	tbmv (USM Version)	925
13.2.1.2.2.61	tbsv	926
13.2.1.2.2.62	tbsv (Buffer Version)	926
13.2.1.2.2.63	tbsv (USM Version)	927
13.2.1.2.2.64	tpmv	928
13.2.1.2.2.65	tpmv (Buffer Version)	928
13.2.1.2.2.66	tpmv (USM Version)	929
13.2.1.2.2.67	tpsv	930
13.2.1.2.2.68	tpsv (Buffer Version)	930
13.2.1.2.2.69	tpsv (USM Version)	931
13.2.1.2.2.70	trmv	932
13.2.1.2.2.71	trmv (Buffer Version)	932
13.2.1.2.2.72	trmv (USM Version)	933
13.2.1.2.2.73	trsv	934
13.2.1.2.2.74	trsv (Buffer Version)	934

13.2.1.2.2.75	trsv (USM Version)	935
13.2.1.2.3	BLAS Level 3 Routines	936
13.2.1.2.3.1	gemm	936
13.2.1.2.3.2	gemm (Buffer Version)	937
13.2.1.2.3.3	gemm (USM Version)	938
13.2.1.2.3.4	hemm	939
13.2.1.2.3.5	hemm (Buffer Version)	940
13.2.1.2.3.6	hemm (USM Version)	941
13.2.1.2.3.7	herk	942
13.2.1.2.3.8	herk (Buffer Version)	942
13.2.1.2.3.9	herk (USM Version)	943
13.2.1.2.3.10	her2k	944
13.2.1.2.3.11	her2k (Buffer Version)	945
13.2.1.2.3.12	her2k (USM Version)	946
13.2.1.2.3.13	symm	947
13.2.1.2.3.14	symm (Buffer Version)	947
13.2.1.2.3.15	symm (USM Version)	948
13.2.1.2.3.16	syrk	950
13.2.1.2.3.17	syrk (Buffer Version)	950
13.2.1.2.3.18	syrk (USM Version)	951
13.2.1.2.3.19	syr2k	952
13.2.1.2.3.20	syr2k (Buffer Version)	952
13.2.1.2.3.21	syr2k (USM Version)	953
13.2.1.2.3.22	trmm	954
13.2.1.2.3.23	trmm (Buffer Version)	955
13.2.1.2.3.24	trmm (USM Version)	956
13.2.1.2.3.25	trsm	957
13.2.1.2.3.26	trsm (Buffer Version)	958
13.2.1.2.3.27	trsm (USM Version)	959
13.2.1.2.4	BLAS-like Extensions	960
13.2.1.2.4.1	axpy_batch	960
13.2.1.2.4.2	axpy_batch (Buffer Version)	960
13.2.1.2.4.3	axpy_batch (USM Version)	961
13.2.1.2.4.4	gemm_batch	964
13.2.1.2.4.5	gemm_batch (Buffer Version)	964
13.2.1.2.4.6	gemm_batch (USM Version)	966
13.2.1.2.4.7	trsm_batch	969
13.2.1.2.4.8	trsm_batch (Buffer Version)	969
13.2.1.2.4.9	trsm_batch (USM Version)	971
13.2.1.2.4.10	gemmt	974
13.2.1.2.4.11	gemmt (Buffer Version)	975
13.2.1.2.4.12	gemmt (USM Version)	976
13.2.1.2.4.13	gemm_bias	977
13.2.1.2.4.14	gemm_bias (Buffer Version)	978
13.2.1.2.4.15	gemm_bias (USM Version)	979
13.2.1.3	LAPACK Routines	980
13.2.1.3.1	LAPACK Linear Equation Routines	981
13.2.1.3.1.1	geqrf	981
13.2.1.3.1.2	geqrf (BUFFER Version)	982
13.2.1.3.1.3	geqrf (USM Version)	983
13.2.1.3.1.4	geqrf_scratchpad_size	984
13.2.1.3.1.5	geqrf_scratchpad_size	984
13.2.1.3.1.6	getrf	985
13.2.1.3.1.7	getrf (BUFFER Version)	985

13.2.1.3.1.8	getrf (USM Version)	986
13.2.1.3.1.9	getrf_scratchpad_size	987
13.2.1.3.1.10	getrf_scratchpad_size	988
13.2.1.3.1.11	getri	988
13.2.1.3.1.12	getri (BUFFER Version)	989
13.2.1.3.1.13	getri (USM Version)	989
13.2.1.3.1.14	getri_scratchpad_size	990
13.2.1.3.1.15	getri_scratchpad_size	991
13.2.1.3.1.16	getrs	991
13.2.1.3.1.17	getrs (BUFFER Version)	992
13.2.1.3.1.18	getrs (USM Version)	993
13.2.1.3.1.19	getrs_scratchpad_size	994
13.2.1.3.1.20	getrs_scratchpad_size	994
13.2.1.3.1.21	orgqr	995
13.2.1.3.1.22	orgqr (BUFFER Version)	996
13.2.1.3.1.23	orgqr (USM Version)	997
13.2.1.3.1.24	orgqr_scratchpad_size	998
13.2.1.3.1.25	orgqr_scratchpad_size	998
13.2.1.3.1.26	ormqr	999
13.2.1.3.1.27	ormqr (BUFFER Version)	999
13.2.1.3.1.28	ormqr (USM Version)	1000
13.2.1.3.1.29	ormqr_scratchpad_size	1001
13.2.1.3.1.30	ormqr_scratchpad_size	1002
13.2.1.3.1.31	potrf	1002
13.2.1.3.1.32	potrf (BUFFER Version)	1003
13.2.1.3.1.33	potrf (USM Version)	1004
13.2.1.3.1.34	potrf_scratchpad_size	1005
13.2.1.3.1.35	potrf_scratchpad_size	1006
13.2.1.3.1.36	potri	1006
13.2.1.3.1.37	potri (BUFFER Version)	1007
13.2.1.3.1.38	potri (USM Version)	1008
13.2.1.3.1.39	potri_scratchpad_size	1009
13.2.1.3.1.40	potri_scratchpad_size	1009
13.2.1.3.1.41	potrs	1010
13.2.1.3.1.42	potrs (BUFFER Version)	1010
13.2.1.3.1.43	potrs (USM Version)	1011
13.2.1.3.1.44	potrs_scratchpad_size	1013
13.2.1.3.1.45	potrs_scratchpad_size	1013
13.2.1.3.1.46	sytrf	1014
13.2.1.3.1.47	sytrf (BUFFER Version)	1014
13.2.1.3.1.48	sytrf (USM Version)	1016
13.2.1.3.1.49	sytrf_scratchpad_size	1017
13.2.1.3.1.50	sytrf_scratchpad_size	1017
13.2.1.3.1.51	trtrs	1018
13.2.1.3.1.52	trtrs (BUFFER Version)	1019
13.2.1.3.1.53	trtrs (USM Version)	1020
13.2.1.3.1.54	trtrs_scratchpad_size	1021
13.2.1.3.1.55	trtrs_scratchpad_size	1021
13.2.1.3.1.56	ungqr	1022
13.2.1.3.1.57	ungqr (BUFFER Version)	1023
13.2.1.3.1.58	ungqr (USM Version)	1024
13.2.1.3.1.59	ungqr_scratchpad_size	1025
13.2.1.3.1.60	ungqr_scratchpad_size	1025
13.2.1.3.1.61	unmqr	1026

13.2.1.3.1.62	unmqr (BUFFER Version)	1026
13.2.1.3.1.63	unmqr (USM Version)	1027
13.2.1.3.1.64	unmqr_scratchpad_size	1029
13.2.1.3.1.65	unmqr_scratchpad_size	1029
13.2.1.3.2	LAPACK Singular Value and Eigenvalue Problem Routines	1030
13.2.1.3.2.1	gebrd	1030
13.2.1.3.2.2	gebrd (BUFFER Version)	1031
13.2.1.3.2.3	gebrd (USM Version)	1032
13.2.1.3.2.4	gebrd_scratchpad_size	1033
13.2.1.3.2.5	gebrd_scratchpad_size	1034
13.2.1.3.2.6	gesvd	1034
13.2.1.3.2.7	gesvd (BUFFER Version)	1035
13.2.1.3.2.8	gesvd (USM Version)	1037
13.2.1.3.2.9	gesvd_scratchpad_size	1039
13.2.1.3.2.10	gesvd_scratchpad_size	1039
13.2.1.3.2.11	heevd	1040
13.2.1.3.2.12	heevd (BUFFER Version)	1041
13.2.1.3.2.13	heevd (USM Version)	1042
13.2.1.3.2.14	heevd_scratchpad_size	1043
13.2.1.3.2.15	heevd_scratchpad_size	1043
13.2.1.3.2.16	hegvd	1044
13.2.1.3.2.17	hegvd (BUFFER Version)	1044
13.2.1.3.2.18	hegvd (USM Version)	1046
13.2.1.3.2.19	hegvd_scratchpad_size	1048
13.2.1.3.2.20	hegvd_scratchpad_size	1048
13.2.1.3.2.21	hetrd	1049
13.2.1.3.2.22	hetrd (BUFFER Version)	1049
13.2.1.3.2.23	hetrd (USM Version)	1050
13.2.1.3.2.24	hetrd_scratchpad_size	1052
13.2.1.3.2.25	hetrd_scratchpad_size	1052
13.2.1.3.2.26	orgbr	1053
13.2.1.3.2.27	orgbr (BUFFER Version)	1054
13.2.1.3.2.28	orgbr (USM Version)	1055
13.2.1.3.2.29	orgbr_scratchpad_size	1056
13.2.1.3.2.30	orgbr_scratchpad_size	1056
13.2.1.3.2.31	orgtr	1057
13.2.1.3.2.32	orgtr (BUFFER Version)	1058
13.2.1.3.2.33	orgtr (USM Version)	1058
13.2.1.3.2.34	orgtr_scratchpad_size	1059
13.2.1.3.2.35	orgtr_scratchpad_size	1060
13.2.1.3.2.36	ormtr	1060
13.2.1.3.2.37	ormtr (BUFFER Version)	1061
13.2.1.3.2.38	ormtr (USM Version)	1062
13.2.1.3.2.39	ormtr_scratchpad_size	1063
13.2.1.3.2.40	ormtr_scratchpad_size	1064
13.2.1.3.2.41	syevd	1065
13.2.1.3.2.42	syevd (BUFFER Version)	1065
13.2.1.3.2.43	syevd (USM Version)	1066
13.2.1.3.2.44	syevd_scratchpad_size	1067
13.2.1.3.2.45	syevd_scratchpad_size	1068
13.2.1.3.2.46	sygvd	1069
13.2.1.3.2.47	sygvd (BUFFER Version)	1069
13.2.1.3.2.48	sygvd (USM Version)	1071
13.2.1.3.2.49	sygvd_scratchpad_size	1073

13.2.1.3.2.50	<code>sygvd_scratchpad_size</code>	1073
13.2.1.3.2.51	<code>sytrd</code>	1074
13.2.1.3.2.52	<code>sytrd (BUFFER Version)</code>	1074
13.2.1.3.2.53	<code>sytrd (USM Version)</code>	1075
13.2.1.3.2.54	<code>sytrd_scratchpad_size</code>	1077
13.2.1.3.2.55	<code>sytrd_scratchpad_size</code>	1077
13.2.1.3.2.56	<code>ungbr</code>	1078
13.2.1.3.2.57	<code>ungbr (BUFFER Version)</code>	1078
13.2.1.3.2.58	<code>ungbr (USM Version)</code>	1080
13.2.1.3.2.59	<code>ungbr_scratchpad_size</code>	1081
13.2.1.3.2.60	<code>ungbr_scratchpad_size</code>	1081
13.2.1.3.2.61	<code>ungtr</code>	1082
13.2.1.3.2.62	<code>ungtr (BUFFER Version)</code>	1083
13.2.1.3.2.63	<code>ungtr (USM Version)</code>	1083
13.2.1.3.2.64	<code>ungtr_scratchpad_size</code>	1084
13.2.1.3.2.65	<code>ungtr_scratchpad_size</code>	1085
13.2.1.3.2.66	<code>unmtr</code>	1085
13.2.1.3.2.67	<code>unmtr (BUFFER Version)</code>	1086
13.2.1.3.2.68	<code>unmtr (USM Version)</code>	1087
13.2.1.3.2.69	<code>unmtr_scratchpad_size</code>	1088
13.2.1.3.2.70	<code>unmtr_scratchpad_size</code>	1089
13.2.1.3.3	LAPACK-like Extensions Routines	1090
13.2.1.3.3.1	<code>geqrf_batch</code>	1090
13.2.1.3.3.2	<code>geqrf_batch (BUFFER Version)</code>	1091
13.2.1.3.3.3	<code>getrf_batch</code>	1091
13.2.1.3.3.4	<code>getrf_batch (BUFFER Version)</code>	1092
13.2.1.3.3.5	<code>getri_batch</code>	1093
13.2.1.3.3.6	<code>getri_batch (BUFFER Version)</code>	1093
13.2.1.3.3.7	<code>getrs_batch</code>	1094
13.2.1.3.3.8	<code>getrs_batch (BUFFER Version)</code>	1094
13.2.1.3.3.9	<code>orgqr_batch</code>	1095
13.2.1.3.3.10	<code>orgqr_batch (BUFFER Version)</code>	1096
13.2.1.3.3.11	<code>potrf_batch</code>	1097
13.2.1.3.3.12	<code>potrf_batch (BUFFER Version)</code>	1097
13.2.1.3.3.13	<code>potrs_batch</code>	1098
13.2.1.3.3.14	<code>potrs_batch (BUFFER Version)</code>	1098
13.2.2	Sparse Linear Algebra	1100
13.2.2.1	Sparse BLAS Routines	1100
13.2.2.1.1	<code>onemkl::sparse::matrixInit</code>	1100
13.2.2.1.2	<code>onemkl::sparse::setCSRstructure</code>	1101
13.2.2.1.3	<code>onemkl::sparse::gemm</code>	1102
13.2.2.1.4	<code>onemkl::sparse::gemv</code>	1104
13.2.2.1.5	<code>onemkl::sparse::gemvdot</code>	1106
13.2.2.1.6	<code>onemkl::sparse::gemvOptimize</code>	1107
13.2.2.1.7	<code>onemkl::sparse::symv</code>	1108
13.2.2.1.8	<code>onemkl::sparse::trmv</code>	1110
13.2.2.1.9	<code>onemkl::sparse::trmvOptimize</code>	1112
13.2.2.1.10	<code>onemkl::sparse::trsv</code>	1113
13.2.2.1.11	<code>onemkl::sparse::trsvOptimize</code>	1115
13.2.2.1.12	Supported Types	1116
13.2.2.1.13	Exceptions	1116
13.2.3	Discrete Fourier Transforms	1116
13.2.3.1	Fourier Transform Functions	1117
13.2.3.1.1	<code>onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain></code>	1117

13.2.3.1.2	onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::Init	1118
13.2.3.1.3	onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::setValue	1119
13.2.3.1.4	onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::getValue	1120
13.2.3.1.5	onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::commit	1122
13.2.3.1.6	onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::computeForward<typename IOType>	1123
13.2.3.1.7	onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::computeBackward<typename IOType>	1125
13.2.4	Random Number Generators	1126
13.2.4.1	oneMKL RNG Usage Model	1127
13.2.4.2	Generate Routine	1129
13.2.4.2.1	onemkl::rng::generate	1130
13.2.4.3	Engines (Basic Random Number Generators)	1131
13.2.4.3.1	onemkl::rng::mrg32k3a	1132
13.2.4.3.2	onemkl::rng::philox4x32x10	1133
13.2.4.3.3	onemkl::rng::mcg31m1	1134
13.2.4.3.4	onemkl::rng::mcg59	1134
13.2.4.3.5	onemkl::rng::r250	1135
13.2.4.3.6	onemkl::rng::wichmann_hill	1136
13.2.4.3.7	onemkl::rng::mt19937	1137
13.2.4.3.8	onemkl::rng::sfmt19937	1138
13.2.4.3.9	onemkl::rng::mt2203	1138
13.2.4.3.10	onemkl::rng::ars5	1139
13.2.4.3.11	onemkl::rng::sobol	1140
13.2.4.3.12	onemkl::rng::niederreiter	1141
13.2.4.3.13	onemkl::rng::nondeterministic	1142
13.2.4.4	Service Routines	1143
13.2.4.4.1	onemkl::rng::leapfrog	1143
13.2.4.4.2	onemkl::rng::skip_ahead	1144
13.2.4.5	Distributions	1147
13.2.4.5.1	Distributions Template Parameter onemkl::rng::method Values	1150
13.2.4.5.2	onemkl::rng::uniform (Continuous)	1151
13.2.4.5.3	onemkl::rng::gaussian	1152
13.2.4.5.4	onemkl::rng::exponential	1154
13.2.4.5.5	onemkl::rng::laplace	1155
13.2.4.5.6	onemkl::rng::weibull	1157
13.2.4.5.7	onemkl::rng::cauchy	1158
13.2.4.5.8	onemkl::rng::rayleigh	1160
13.2.4.5.9	onemkl::rng::lognormal	1161
13.2.4.5.10	onemkl::rng::gumbel	1163
13.2.4.5.11	onemkl::rng::gamma	1164
13.2.4.5.12	onemkl::rng::beta	1165
13.2.4.5.13	onemkl::rng::chi_square	1167
13.2.4.5.14	onemkl::rng::uniform (Discrete)	1168
13.2.4.5.15	onemkl::rng::uniform_bits	1169
13.2.4.5.16	onemkl::rng::bits	1170
13.2.4.5.17	onemkl::rng::bernoulli	1170
13.2.4.5.18	onemkl::rng::geometric	1172
13.2.4.5.19	onemkl::rng::binomial	1173
13.2.4.5.20	onemkl::rng::hypergeometric	1174
13.2.4.5.21	onemkl::rng::poisson	1176

13.2.4.5.22	onemkl::rng::poisson_v	1177
13.2.4.5.23	onemkl::rng::negbinomial	1178
13.2.4.5.24	onemkl::rng::multinomial	1179
13.2.4.6	Bibliography	1181
13.2.5	Vector Math	1182
13.2.5.1	Special Value Notations	1182
13.2.5.2	VM Mathematical Functions	1183
13.2.5.2.1	Arithmetic Functions	1185
13.2.5.2.1.1	add	1185
13.2.5.2.1.2	sub	1187
13.2.5.2.1.3	sqr	1189
13.2.5.2.1.4	mul	1190
13.2.5.2.1.5	mulbyconj	1193
13.2.5.2.1.6	conj	1194
13.2.5.2.1.7	abs	1195
13.2.5.2.1.8	arg	1197
13.2.5.2.1.9	linearfrac	1199
13.2.5.2.1.10	fmod	1201
13.2.5.2.1.11	remainder	1203
13.2.5.2.2	Power and Root Functions	1205
13.2.5.2.2.1	inv	1205
13.2.5.2.2.2	div	1207
13.2.5.2.2.3	sqrt	1209
13.2.5.2.2.4	invsqrt	1211
13.2.5.2.2.5	cbrt	1213
13.2.5.2.2.6	invcbrt	1214
13.2.5.2.2.7	pow2o3	1216
13.2.5.2.2.8	pow3o2	1217
13.2.5.2.2.9	pow	1219
13.2.5.2.2.10	powx	1222
13.2.5.2.2.11	powr	1224
13.2.5.2.2.12	hypot	1226
13.2.5.2.3	Exponential and Logarithmic Functions	1228
13.2.5.2.3.1	exp	1228
13.2.5.2.3.2	exp2	1230
13.2.5.2.3.3	exp10	1232
13.2.5.2.3.4	expm1	1234
13.2.5.2.3.5	ln	1235
13.2.5.2.3.6	log2	1237
13.2.5.2.3.7	log10	1239
13.2.5.2.3.8	log1p	1241
13.2.5.2.3.9	logb	1243
13.2.5.2.4	Trigonometric Functions	1244
13.2.5.2.4.1	cos	1245
13.2.5.2.4.2	sin	1247
13.2.5.2.4.3	sincos	1249
13.2.5.2.4.4	cis	1250
13.2.5.2.4.5	tan	1252
13.2.5.2.4.6	acos	1254
13.2.5.2.4.7	asin	1256
13.2.5.2.4.8	atan	1258
13.2.5.2.4.9	atan2	1259
13.2.5.2.4.10	cospi	1261
13.2.5.2.4.11	sinpi	1263

13.2.5.2.4.12	tanpi	1265
13.2.5.2.4.13	acospri	1267
13.2.5.2.4.14	asinpri	1269
13.2.5.2.4.15	atanpri	1271
13.2.5.2.4.16	atan2pri	1272
13.2.5.2.4.17	cosd	1274
13.2.5.2.4.18	sind	1276
13.2.5.2.4.19	tand	1278
13.2.5.2.5	Hyperbolic Functions	1280
13.2.5.2.5.1	cosh	1280
13.2.5.2.5.2	sinh	1282
13.2.5.2.5.3	tanh	1285
13.2.5.2.5.4	acosh	1286
13.2.5.2.5.5	asinh	1289
13.2.5.2.5.6	atanh	1290
13.2.5.2.6	Special Functions	1293
13.2.5.2.6.1	erf	1293
13.2.5.2.6.2	erfc	1296
13.2.5.2.6.3	cdfnorm	1299
13.2.5.2.6.4	erfinv	1302
13.2.5.2.6.5	erfcinv	1305
13.2.5.2.6.6	cdfnorminv	1308
13.2.5.2.6.7	lgamma	1311
13.2.5.2.6.8	tgamma	1312
13.2.5.2.6.9	expint1	1314
13.2.5.2.7	Rounding Functions	1316
13.2.5.2.7.1	floor	1316
13.2.5.2.7.2	ceil	1318
13.2.5.2.7.3	trunc	1320
13.2.5.2.7.4	round	1321
13.2.5.2.7.5	nearbyint	1323
13.2.5.2.7.6	rint	1324
13.2.5.2.7.7	modf	1326
13.2.5.2.7.8	frac	1328
13.2.5.3	VM Service Functions	1330
13.2.5.3.1	setmode	1330
13.2.5.3.2	get_mode	1331
13.2.5.3.3	set_status	1332
13.2.5.3.4	get_status	1334
13.2.5.3.5	clear_status	1335
13.2.5.3.6	create_error_handler	1336
13.2.5.4	Miscellaneous VM Functions	1340
13.2.5.4.1	copysign	1340
13.2.5.4.2	nextafter	1341
13.2.5.4.3	fdim	1343
13.2.5.4.4	fmax	1345
13.2.5.4.5	fmin	1346
13.2.5.4.6	maxmag	1348
13.2.5.4.7	minmag	1349
13.2.5.5	Bibliography	1351

14 Contributors 1352

15 HTML and PDF Versions 1353

15.1	Release Notes	1353
15.1.1	0.85	1353
15.1.2	0.8	1354
15.1.3	0.7	1354
15.1.4	0.5	1354
16	Legal Notices and Disclaimers	1355
17	Page Not Found	1356
	Bibliography	1357
	Index	1358

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

Todo: create link to “Video Processing procedure / Configuration”

(The [original entry](#) is located in source/elements/oneVPL/source/MSDK_processing.inc.rst, line 26.)

Todo: create link to mfxFrameInfo::Shift

(The [original entry](#) is located in source/elements/oneVPL/source/MSDK_processing.inc.rst, line 130.)

Todo: Keep or remove HW?

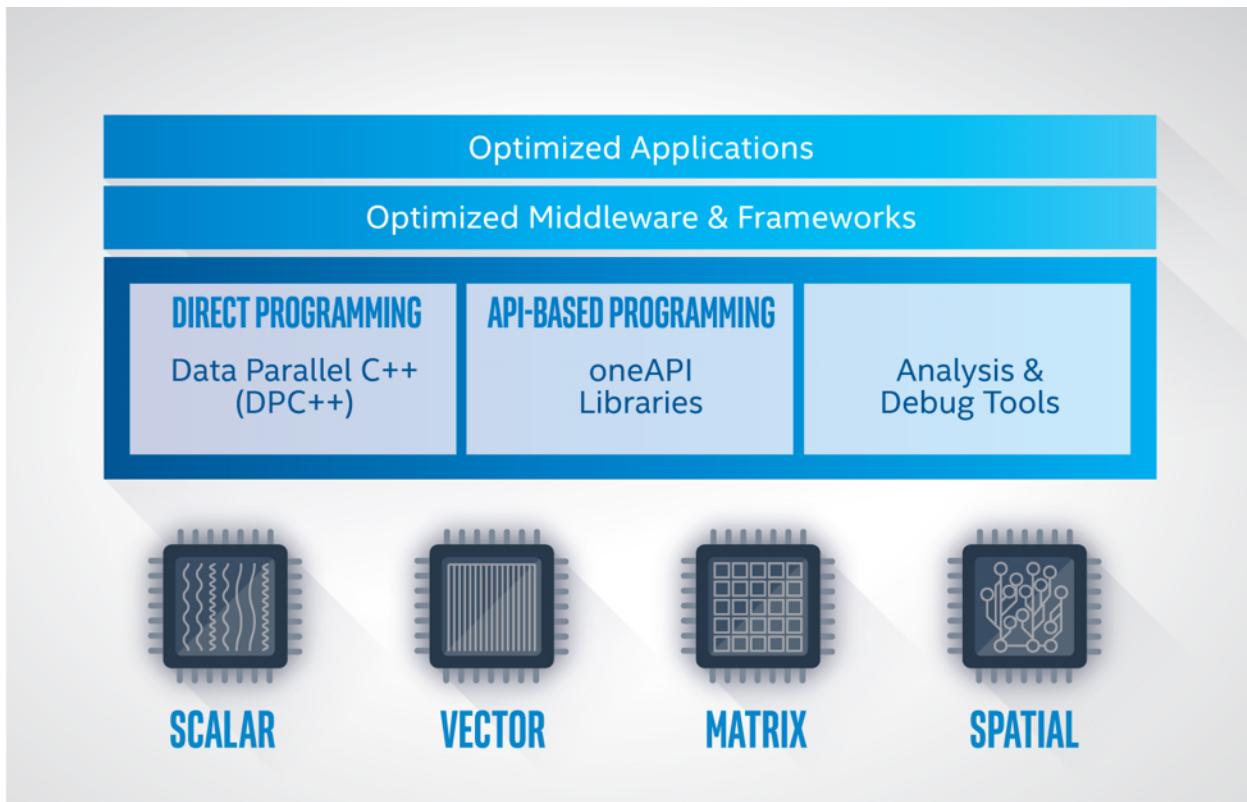
(The [original entry](#) is located in source/elements/oneVPL/source/MSDK_processing.inc.rst, line 134.)

Todo: Add link to “Allocation and External Allocators”

(The [original entry](#) is located in source/elements/oneVPL/source/MSDK_prg_hw.inc.rst, line 295.)

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.

**CHAPTER
FOUR**

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++ includes the SYCL language. SYCL 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. A compliant DPC++ implementation must also be a conformant SYCL 1.2.1 (or later) implementation, a process which includes Khronos* conformance testing and an adopter process. (See the [SYCL Specification](#) and [SYCL Adopters](#).)
- 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, SYCL 1.2.1, and DPC++ extensions. These version and feature coverage requirements will evolve over time, with newer 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 ¹
Unified Shared Memory	Required ²	Required ²	Required ²	usm
In-order queues	Required	Required	Required	NA ³
Optional lambda name	Required	Required	Required	NA ³
Deduction guides	Required	Required	Required	NA ³
Reductions	Required	Required	Required	NA ³
Sub-groups	Required	Required	Not required ⁴	sub_group
Sub-group algorithms	Required	Required	Not required ⁴	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.

¹ Test directory within [extension tests](#)

² Minimum of explicit USM support

³ Not yet available.

⁴ 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 Khronos SYCL 1.2.1 conformance test suite (SYCL-1.2.1/master branch).
2. 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)

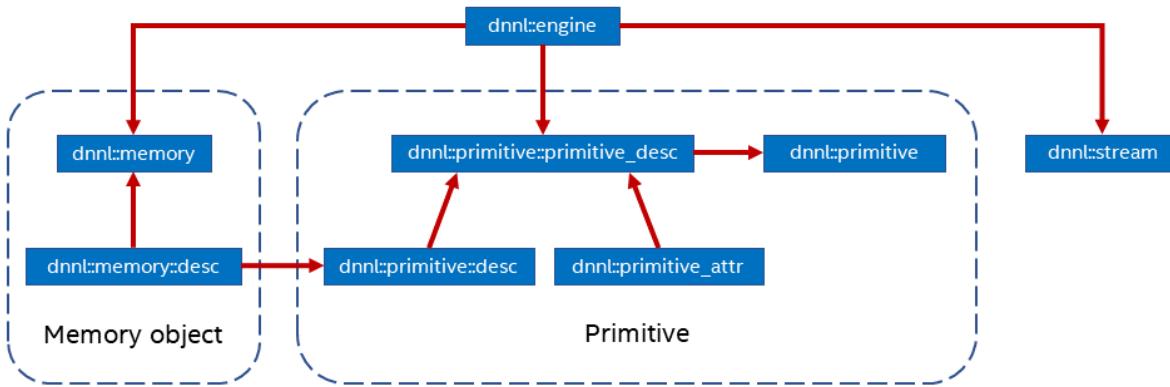
(continued from previous page)

```
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

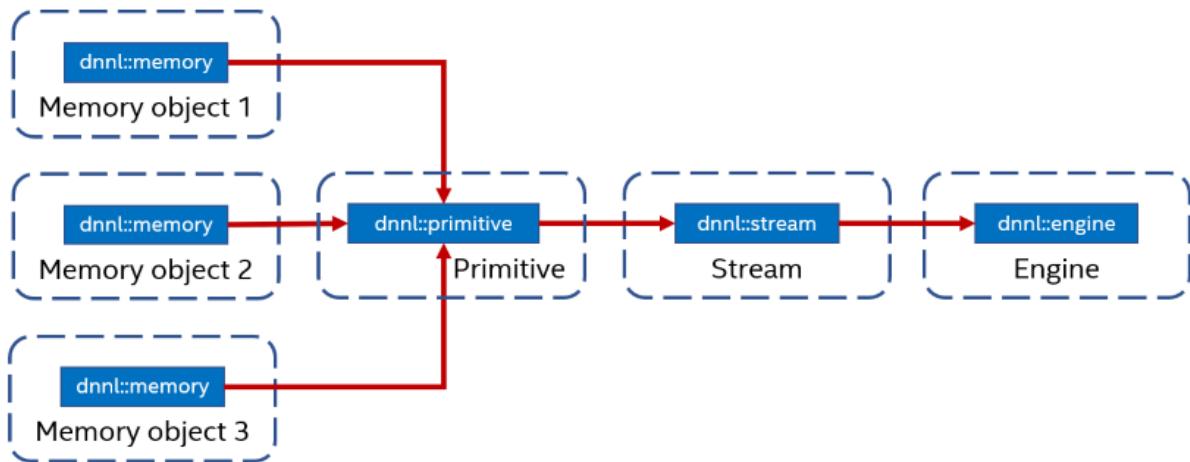
7.2.2 RNN-Specific Notation

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.

f32	s	8-bit exp	23 bit mantissa
bf16	s	8-bit exp	7 bit mantissa
f16	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 σ 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 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
Compute intensive: (De-)convolution, Inner product, RNN	Use <code>any</code>	Use <code>any</code>	N/A
Memory-bandwidth limited: 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
Memory-bandwidth limited: 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 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 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 memory::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)`
`const`
 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:

- i. Add a dimension of size 1. This is always possible.
- ii. Remove a dimension of size 1.
- iii. Split a dimension into multiple ones. This is possible only if the product of all tensor dimensions stays constant.
- iv. 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 $\text{dim}[i] < \text{stride for } \text{dim}[j]$.

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.

7.4.2.2.2 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 data handle 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 data handle 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

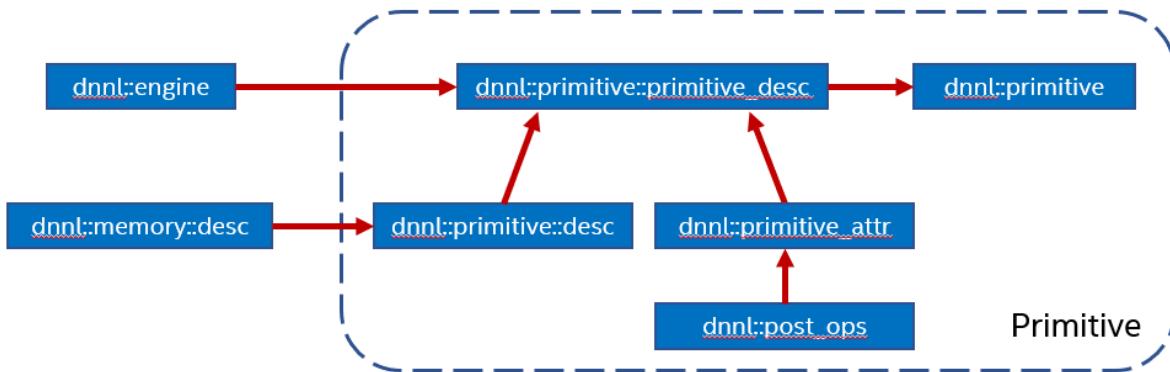
Primitives are functor objects that encapsulates a particular computation such as forward convolution, backward LSTM computations, or a data transformation operation. A single primitive can sometimes represent more complex fused computations such as a forward convolution followed by a ReLU.

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 2.

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 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_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_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 = 0x1u

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 = 0x2u

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 = 0x4u

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
```

(continues on next page)

(continued from previous page)

```
// (!) 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

$$\text{dst}[:] = \text{scale} \cdot \text{as}_d\text{ata}_t\text{ype}(\text{dst}[:]) + \text{Op}(...)$$

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();
```

(continues on next page)

(continued from previous page)

```

// 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 src_{f32} , weights_{f32} , 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 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

- $\text{output_scale} := \frac{\text{scale}_{\text{src}} \cdot \text{scale}_{\text{weights}}}{\text{scale}_{\text{dst}}};$
- 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 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 α 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 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:

$$\text{weights}_{int8}(oc, ic, kh, kw) = \text{f32_to_int8}(\text{output_scale}(oc) \cdot \text{weights}_{f32}(oc, ic, kh, kw)),$$

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 `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.
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
```

(continues on next page)

(continued from previous page)

```

// (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
    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,

```

(continues on next page)

(continued from previous page)

```

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);

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: $scales.size() = \prod_{d \in mask} weights.dims[d]$. 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

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

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
- ε 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:

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

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

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.

	<i>forward_infer</i>	<i>backward_training</i>	<i>backward</i>	<i>backward_data</i>
<i>none</i>	<i>In:</i> src; <i>Out:</i> dst	<i>In:</i> src; <i>Out:</i> dst, μ , σ^2	<i>In:</i> diff_dst, src, μ , σ^2 ; <i>Out:</i> diff_src	Same as for <i>backward</i>
<i>use_global_stats</i>	<i>In:</i> src, μ , σ^2 ; <i>Out:</i> dst	<i>In:</i> src, μ , σ^2 ; <i>Out:</i> dst	<i>In:</i> diff_dst, src, μ , σ^2 ; <i>Out:</i> diff_src	Same as for <i>backward</i>
<i>use_scaleshift</i>	<i>In:</i> src, γ , β ; <i>Out:</i> dst	<i>In:</i> src, γ , β ; <i>Out:</i> dst, μ , σ^2	<i>In:</i> diff_dst, src, μ , σ^2 , γ , β ; <i>Out:</i> diff_src, diff_γ, diff_β	Not supported
<i>use_global_stats</i> <i>use_scaleshift</i>	<i>In:</i> src, μ , σ^2 , γ , β ; <i>Out:</i> dst	<i>In:</i> src, μ , σ^2 , γ , β ; <i>Out:</i> dst	<i>In:</i> diff_dst, src, μ , σ^2 , γ , β ; <i>Out:</i> diff_src, diff_γ, diff_β	Not supported
<i>flags</i> <i>fuse_norm_relu</i>	<i>In:</i> same as with flags; <i>Out:</i> same as with flags	<i>In:</i> same as with flags; <i>Out:</i> same as with flags, workspace	<i>In:</i> same as with flags, workspace; <i>Out:</i> same as with flags	Same as for <i>backward</i> if flags do not contain <i>use_scaleshift</i> ; 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	<i>DNNL_ARG_SRC</i>
γ, β	<i>DNNL_ARG_SCALE_SHIFT</i>
mean (μ)	<i>DNNL_ARG_MEAN</i>
variance (σ)	<i>DNNL_ARG_VARIANCE</i>
dst	<i>DNNL_ARG_DST</i>
workspace	<i>DNNL_ARG_WORKSPACE</i>
diff_dst	<i>DNNL_ARG_DIFF_DST</i>
diff_src	<i>DNNL_ARG_DIFF_SRC</i>
diff_γ, diff_β	<i>DNNL_ARG_DIFF_SCALE_SHIFT</i>

7.5.3.4 Operation Details

- 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.
- 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*.
- 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.
- 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</code> , <code>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)</code> , <code>nwc(acb)</code> , <i>optimized</i>
2D	NCHW	<code>nchw(abcd)</code> , <code>nhwc(acdb)</code> , <i>optimized</i>
3D	NCDHW	<code>ncdhw(abce)</code> , <code>ndhwc(acdeb)</code> , <i>optimized</i>

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

7.5.3.6.2 Statistics Tensors

The mean (μ) and variance (σ^2) are separate 1D tensors of size C .

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

If used, the scale (γ) and shift (β) 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 allow_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 *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
src ₀	<i>DNNL_ARG_SRC_0</i>
src ₁	<i>DNNL_ARG_SRC_1</i>
dst	<i>DNNL_ARG_DST</i>

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	<i>Scales</i>	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	<i>Sum</i>	Adds the operation result to the destination tensor instead of overwriting it.	Must precede eltwise post-op.
Post-op	<i>Eltwi</i> 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 left)
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 right)
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		<i>any</i>	<i>optimized</i>
1D	f32, bf16	NCW / OIW, GOIW	<i>ncw (abc) / oiw (abc), goiw (abcd)</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	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	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)
```

This 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)
```

This 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 mem-
ory::dims &padding_r)
```

Constructs a descriptor for a convolution 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`.
- 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.

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.

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 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.
- `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_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 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 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 of 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 deconvolu-  
tion_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 al-  
low_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>	$d = 0.5s(1 + \operatorname{erf}[\frac{s}{\sqrt{2}}])$
<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_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 `diff_src(\bar{s})`, based on `diff_dst(\bar{s})` and `src(\bar{s})`. However, some operations support a computation using `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 `src`, `dst`, `diff_src`, and a `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()` and `dnnl::eltwise_backward::desc()` constructors take both parameters α , and β . 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()` 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:

$$dst(n, oc) = \text{bias}(oc) + \sum_{ic=0}^{IC-1} \text{src}(n, ic) \cdot \text{weights}(oc, ic)$$

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
- ε 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, μ, σ^2	<i>In:</i> diff_dst, src, μ, σ^2 <i>Out:</i> diff_src	Same as for <code>backward</code>
<code>use_global_stats</code>	<i>In:</i> src, μ, σ^2 <i>Out:</i> dst	<i>In:</i> src, μ, σ^2 <i>Out:</i> dst	<i>In:</i> diff_dst, src, μ, σ^2 <i>Out:</i> diff_src	Same as for <code>backward</code>
<code>use_scaleshift</code>	<i>In:</i> src, γ, β <i>Out:</i> dst	<i>In:</i> src, γ, β <i>Out:</i> dst, μ, σ^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>
γ, β	<i>DNNL_ARG_SCALE_SHIFT</i>
mean (μ)	<i>DNNL_ARG_MEAN</i>
variance (σ)	<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_ γ , diff_ β	<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 (μ) and variance (σ^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 (γ) and shift (β) 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 aprop_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

- *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*: 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
- ν 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())
```

Constructs a primitive descriptor for reorder primitive.

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).

```
primitive_desc(const memory &src, const memory &dst, const primitive_attr &attr = primitive_attr())
```

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).

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

- $ih = \left\lceil \frac{oh+0.5}{F_h} - 0.5 \right\rceil$,
- $iw = \left\lceil \frac{ow+0.5}{F_w} - 0.5 \right\rceil$.

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
<code>src</code>	<code>DNNL_ARG_SRC</code>
<code>dst</code>	<code>DNNL_ARG_DST</code>
<code>diff_src</code>	<code>DNNL_ARG_DIFF_SRC</code>
<code>diff_dst</code>	<code>DNNL_ARG_DIFF_DST</code>

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	<code>f32, bf16</code>
forward	<code>f16, s8, u8</code>

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`

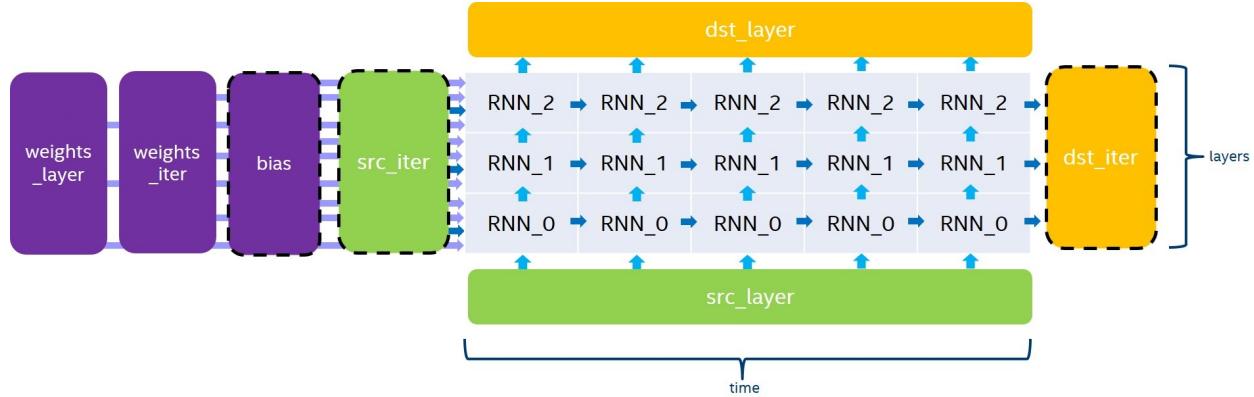
Returns a diff destination memory descriptor.

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:math: i , f , o . The following equation gives the mathematical description of these gates and output for the forward pass:

$$\begin{aligned} i_t &= \sigma(W_i \cdot h_{t,l-1} + U_i \cdot h_{t-1,l} + P_i \cdot c_{t-1} + B_i) \\ f_t &= \sigma(W_f \cdot h_{t,l-1} + U_f \cdot h_{t-1,l} + P_f \cdot c_{t-1} + B_f) \\ \tilde{c}_t &= \tanh(W_{\tilde{c}} \cdot h_{t,l-1} + U_{\tilde{c}} \cdot h_{t-1,l} + B_{\tilde{c}}) \\ c_t &= f_t * c_{t-1} + i_t * \tilde{c}_t \\ o_t &= \sigma(W_o \cdot h_{t,l-1} + U_o \cdot h_{t-1,l} + P_o \cdot c_t + B_o) \\ h_t &= \tanh(c_t) * o_t \end{aligned}$$

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.

$$\begin{aligned} u_t &= \sigma(W_u \cdot h_{t,l-1} + U_u \cdot h_{t-1,l} + B_u) \\ r_t &= \sigma(W_r \cdot h_{t,l-1} + U_r \cdot h_{t-1,l} + B_r) \\ o_t &= \tanh(W_o \cdot h_{t,l-1} + r_t * (U_o \cdot h_{t-1,l} + B_{u'}) + B_o) \\ h_t &= u_t * h_{t-1,l} + (1 - u_t) * o_t \end{aligned}$$

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	<code>f32</code>	<code>f32</code>	<code>f32</code>	<code>f32</code>	<code>f32</code>
Forward / Backward (2)	All (3)	<code>bf16</code>	<code>bf16</code>	<code>bf16</code>	<code>f32</code>	<code>bf16</code>
Forward	All (3)	<code>f16</code>	<code>f16</code>	<code>f16</code>	<code>f16</code>	<code>f16</code>
Forward inference	Vanilla LSTM	<code>u8</code>	<code>u8</code>	<code>s8</code>	<code>f32</code>	<code>u8, f32</code>

- (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) 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.

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 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 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_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 memory::desc
&dst_iter_desc, const memory::desc &dst_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 memory::desc &diff_bias_desc, const
memory::desc &diff_dst_layer_desc, const memory::desc &diff_dst_iter_desc, const
memory::desc &diff_dst_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 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_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 src_iter_desc, bias_desc, and dst_iter, may point to a zero memory descriptor. 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 dnnl::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
- ν 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:

- Allreduce
- Allgatherv
- Broadcast
- Reduce
- Alltoall

oneCCL exposes controls over additional optimizations and capabilities such as:

- User-defined pre-/post-processing of incoming buffers and reduction operation
- Prioritization for communication operations
- Persistent communication operations (enables decoupling one-time initialization and repetitive execution)
- Fusion of multiple communication operations into the single one
- Unordered communication operations
- Allreduce on sparse data

Intel has published an [open source implementation](#) with the Apache license. The [open source implementation](#) includes a comprehensive test suite. Consult the [README](#) for directions.

8.2 Definitions

8.2.1 oneCCL Concepts

oneAPI Collective Communications Library introduces the following list of concepts:

- *oneCCL Environment*
- *oneCCL Stream*
- *oneCCL Communicator*

8.2.1.1 oneCCL Environment

oneCCL Environment is a singleton object which is used as an entry point into the oneCCL library and which is defined only for C++ version of API. oneCCL Environment exposes a number of helper methods to manage other oneCCL objects, such as streams, communicators, etc.

8.2.1.2 oneCCL Stream

C version of oneCCL API:

```
typedef void* ccl_stream_t;
```

C++ version of oneCCL API:

```
class stream;
using stream_t = std::unique_ptr<ccl::stream>;
```

CCL Stream encapsulates execution context for communication primitives declared by oneCCL specification. It is an opaque handle that is managed by oneCCL API:

C version of oneCCL API:

```
ccl_status_t ccl_stream_create(ccl_stream_type_t type,
                               void* native_stream,
                               ccl_stream_t* ccl_stream);
ccl_status_t ccl_stream_free(ccl_stream_t stream);
```

C++ version of oneCCL API:

```
class environment
{
public:
    ...
    /**
     * Creates a new ccl stream of @c type with @c native stream
     * @param type the @c ccl::stream_type and may be @c cpu or @c sycl (if configured)
     * @param native_stream the existing handle of stream
     */
    stream_t create_stream(ccl::stream_type type = ccl::stream_type::cpu, void* native_stream = nullptr) const;
}
```

When you create oneCCL stream object using the API described above, you need to specify the stream type and pass the pointer to the underlying command queue object. For example, for oneAPI device you should pass `ccl::stream_type::sycl` and `cl::sycl::queue` objects.

8.2.1.3 oneCCL Communicator

C version of oneCCL API:

```
typedef void* ccl_comm_t;
```

C++ version of oneCCL API:

```
class communicator;
using communicator_t = std::unique_ptr<ccl::communicator>;
```

oneCCL Communicator defines participants of collective communication operations. It is an opaque handle that is managed by oneCCL API:

C version of oneCCL API:

```
ccl_status_t ccl_comm_create(ccl_comm_t* comm,
                             const ccl_comm_attr_t* attr);
ccl_status_t ccl_comm_free(ccl_comm_t comm);
```

C++ version of oneCCL API:

```
class environment
{
public:
...
 /**
 * Creates a new communicator according to @c attr parameters
 * or creates a copy of global communicator, if @c attr is @c nullptr(default)
 * @param attr
 */
communicator_t create_communicator(const ccl::comm_attr* attr = nullptr) const;
```

When you create oneCCL Communicator, you can optionally specify attributes that control the runtime behavior of oneCCL implementation.

8.2.1.3.1 oneCCL Communicator Attributes

```
typedef struct
{
    /**
     * Used to split global communicator into parts. Ranks with identical color
     * will form a new communicator.
     */
    int color;
} ccl_comm_attr_t;
```

`ccl_comm_attr_t` (`ccl::comm_attr` in C++ version of API) is extendable structure that serves as a modifier of communicator behavior.

8.2.2 oneCCL Collective Communication

This section covers collective communication operations implemented in oneAPI Collective Communications Library.

8.2.2.1 Collective Operations

oneAPI Collective Communications Library introduces the following list of communication primitives:

- *Allgatherv*
- *Allreduce*
- *Reduce*
- *Alltoall*
- *Barrier*
- *Broadcast*

These operations are collective, meaning that all participants of oneCCL communicator should make a call.

8.2.2.1.1 Allgatherv

Allgatherv is a collective communication operation that collects data from all processes within oneCCL communicator. Each participant gets the same result data. Different participants can contribute segments of different sizes.

C version of oneCCL API:

```
ccl_status_t ccl_allgatherv(
    const void* send_buf,
    size_t send_count,
    void* recv_buf,
    const size_t* recv_counts,
    ccl_datatype_t dtype,
    const ccl_coll_attr_t* attr,
    ccl_comm_t comm,
    ccl_stream_t stream,
    ccl_request_t* req);
```

C++ version of oneCCL API:

```
template<class buffer_type>
coll_request_t communicator::allgatherv(const buffer_type* send_buf,
                                         size_t send_count,
                                         buffer_type* recv_buf,
                                         const size_t* recv_counts,
                                         const ccl::coll_attr* attr,
                                         const ccl::stream_t& stream);
```

send_buf the buffer with **count** elements of type **buffer_type** that stores local data to be sent

recv_buf [out] the buffer to store received data, must have the same dimension as **send_buf**

count the number of elements of type **buffer_type** to be sent by a participant of oneCCL communicator

recv_counts the number of elements of type **buffer_type** to be received from each participant of oneCCL communicator

dtype datatype of the elements (for C++ API gets inferred from the buffer type)

attr optional attributes that customize operation
comm oneCCL communicator for the operation
stream oneCCL stream associated with the operation
req object that can be used to track the progress of the operation (returned value for C++ API)

8.2.2.1.2 Allreduce

Global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to all members of oneCCL communicator.

C version of oneCCL API:

```
ccl_status_t ccl_allreduce(
    const void* send_buf,
    void* recv_buf,
    size_t count,
    ccl_datatype_t dtype,
    ccl_reduction_t reduction,
    const ccl_coll_attr_t* attr,
    ccl_comm_t comm,
    ccl_stream_t stream,
    ccl_request_t* req);
```

C++ version of oneCCL API:

```
template<class buffer_type>
coll_request_t communicator::allreduce(const buffer_type* send_buf,
                                       buffer_type* recv_buf,
                                       size_t count,
                                       ccl::reduction reduction,
                                       const ccl::coll_attr* attr,
                                       const ccl::stream_t& stream);
```

send_buf the buffer with **count** elements of **buffer_type** that stores local data to be reduced

recv_buf [out] the buffer to store reduced result, must have the same dimension as **send_buf**

count the number of elements of **buffer_type** in **send_buf**

dtype datatype of the elements (for C++ API gets inferred from the buffer type)

reduction type of reduction operation to be applied

attr optional attributes that customize operation

comm oneCCL communicator for the operation

stream oneCCL stream associated with the operation

req object that can be used to track the progress of the operation (returned value for C++ API)

8.2.2.1.3 Reduce

Global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to a single member of oneCCL communicator (root).

C version of oneCCL API:

```
ccl_status_t ccl_reduce(
    const void* send_buf,
    void* recv_buf,
    size_t count,
    ccl_datatype_t dtype,
    ccl_reduction_t reduction,
    size_t root,
    const ccl_coll_attr_t* attr,
    ccl_comm_t comm,
    ccl_stream_t stream,
    ccl_request_t* req);
```

C++ version of oneCCL API:

```
template<class buffer_type>
coll_request_t communicator::reduce(const buffer_type* send_buf,
                                    buffer_type* recv_buf,
                                    size_t count,
                                    ccl::reduction reduction,
                                    size_t root,
                                    const ccl::coll_attr* attr,
                                    const ccl::stream_t& stream);
```

send_buf the buffer with **count** elements of **buffer_type** that stores local data to be reduced

recv_buf [out] the buffer to store reduced result, must have the same dimension as **send_buf**

count the number of elements of **buffer_type** in **send_buf**

dtype datatype of the elements (for C++ API gets inferred from the buffer type)

reduction type of reduction operation to be applied

root the rank of the process that gets the result of reduction

attr optional attributes that customize operation

comm oneCCL communicator for the operation

stream oneCCL stream associated with the operation

req object that can be used to track the progress of the operation (returned value for C++ API)

The following reduction operations are supported for *Allreduce* and *Reduce* primitives:

ccl_reduction_sum elementwise summation

ccl_reduction_prod elementwise multiplication

ccl_reduction_min elementwise min

ccl_reduction_max elementwise max

ccl_reduction_custom: class of user-defined operations

8.2.2.1.4 Alltoall

oneCCL Alltoall is an extension of oneCCL Allgather for cases when each process sends different data to each receiver. The j -th block sent from the process i is received by the process j and is placed in the i -th block of `recvbuf`. For each pair of processes the amount of sent data must be equal to the amount of received data.

C version of oneCCL API:

```
ccl_status_t ccl_alltoall(
    const void* send_buf,
    void* recv_buf,
    size_t count,
    ccl_datatype_t dtype,
    const ccl_coll_attr_t* attr,
    ccl_comm_t comm,
    ccl_stream_t stream,
    ccl_request_t* req);
```

C++ version of oneCCL API:

```
template<class buffer_type>
coll_request_t communicator::alltoall(const buffer_type* send_buf,
                                      buffer_type* recv_buf,
                                      size_t count,
                                      const ccl::coll_attr* attr,
                                      const ccl::stream_t& stream);
```

send_buf the buffer with `count` elements of `buffer_type` that stores local data to be sent

recv_buf [out] the buffer to store received data, must have the same dimension as `send_buf`

count the number of elements of type `buffer_type` to be sent to or received from each participant of oneCCL communicator

dtype datatype of the elements (for C++ API gets inferred from the buffer type)

attr optional attributes that customize operation

comm oneCCL communicator for the operation

stream oneCCL stream associated with the operation

req object that can be used to track the progress of the operation (returned value for C++ API)

8.2.2.1.5 Barrier

Blocking barrier synchronization across all members of oneCCL communicator.

C version of oneCCL API:

```
ccl_status_t ccl_barrier(ccl_comm_t comm,
                         ccl_stream_t stream);
```

C++ version of oneCCL API:

```
void communicator::barrier(const ccl::stream_t& stream);
```

comm oneCCL communicator for the operation

stream oneCCL stream associated with the operation

8.2.2.1.6 Broadcast

Collective communication operation that broadcasts data from one participant of oneCCL communicator (denoted as root) to all other participants.

C version of oneCCL API:

```
ccl_status_t ccl_bcast(
    void* buf,
    size_t count,
    ccl_datatype_t dtype,
    size_t root,
    const ccl_coll_attr_t* attr,
    ccl_comm_t comm,
    ccl_stream_t stream,
    ccl_request_t* req);
```

C++ version of oneCCL API:

```
template<class buffer_type>
col_request_t communicator::bcast(buffer_type* buf, size_t count,
                                    size_t root,
                                    const ccl::coll_attr* attr,
                                    const ccl::stream_t& stream);
```

buf serves as send buffer for root and as receive buffer for other participants

count the number of elements of type `buffer_type` in `send_buf`

dtype datatype of the elements (for C++ API gets inferred from the buffer type)

root the rank of the process that broadcasts the data

attr optional attributes that customize the operation

comm oneCCL communicator for the operation

stream oneCCL stream associated with the operation

req object that can be used to track the progress of the operation (returned value for C++ API)

8.2.2.2 Data Types

oneCCL specification defines the following data types that can be used for collective communication operations:

C version of API:

```
typedef enum
{
    ccl_dtype_char    = 0,
    ccl_dtype_int     = 1,
    ccl_dtype_bfp16   = 2,
    ccl_dtype_float   = 3,
    ccl_dtype_double  = 4,
    ccl_dtype_int64   = 5,
    ccl_dtype_uint64  = 6,
} ccl_datatype_t;
```

C++ version of API:

```
enum class data_type
{
    dt_char = ccl_dtype_char,
    dt_int = ccl_dtype_int,
    dt_bfp16 = ccl_dtype_bfp16,
    dt_float = ccl_dtype_float,
    dt_double = ccl_dtype_double,
    dt_int64 = ccl_dtype_int64,
    dt_uint64 = ccl_dtype_uint64,
};
```

ccl_dtype_char Corresponds to *char* in C language

ccl_dtype_int Corresponds to *signed int* in C language

ccl_dtype_bfp16 BFloat16 datatype

ccl_dtype_float Corresponds to *float* in C language

ccl_dtype_double Corresponds to *double* in C language

ccl_dtype_int64 Corresponds to *int64_t* in C language

ccl_dtype_uint64 Corresponds to *uint64_t* in C language

8.2.2.3 Collective Call Attributes

```
/* Extendable list of collective attributes */
typedef struct
{
    /**
     * Callbacks into application code
     * for pre-/post-processing data
     * and custom reduction operation
     */
    ccl_prologue_fn_t prologue_fn;
    ccl_epilogue_fn_t epilogue_fn;
    ccl_reduction_fn_t reduction_fn;
    /* Priority for collective operation */
    size_t priority;
    /* Blocking/non-blocking */
    int synchronous;
    /* Persistent/non-persistent */
    int to_cache;
    /* Treat buffer as vector/regular - applicable for allgatherv only */
    int vector_buf;
    /**
     * Id of the operation. If specified, new communicator will be created and
     collective
     * operations with the same @b match_id will be executed in the same order.
     */
    const char* match_id;
} ccl_coll_attr_t;
```

ccl_coll_attr_t (`ccl::coll_attr` in C++ version of API) is extendable structure which serves as a modifier of communication primitive behavior. It can be optionally passed into any collective operation exposed by oneCCL.

8.2.2.4 Track Communication Progress

The progress for any of the collective operations provided by oneCCL can be tracked using *Test* or *Wait* function for *Request* object.

8.2.2.4.1 Request

Each collective communication operation of oneCCL returns a request that can be used to query completion of this operation or to block the execution while the operation is in progress. oneCCL request is an opaque handle that is managed by corresponding APIs.

C version of oneCCL API:

```
typedef void* ccl_request_t;
```

C++ version of oneCCL API:

```
/***
 * A request interface that allows the user to track collective operation progress
 */
class request
{
public:
    /**
     * Blocking wait for collective operation completion
     */
    virtual void wait() = 0;

    /**
     * Non-blocking check for collective operation completion
     * @retval true if the operations has been completed
     * @retval false if the operations has not been completed
     */
    virtual bool test() = 0;

    virtual ~request() = default;
};
```

8.2.2.4.2 Test

Non-blocking operation that returns the completion status.

C version of oneCCL API:

```
ccl_status_t ccl_test(ccl_request_t req, int* is_completed);
```

req requests the handle for communication operation being tracked

is_completed indicates the status:

- 0 - operation is in progress
- otherwise, the operation is completed

C++ version of oneCCL API:

```
bool request::test();
```

Returns the value that indicates the status:

- 0 - operation is in progress
- otherwise, the operation is completed.

8.2.2.4.3 Wait

Operation that blocks the execution until communication operation is completed.

C version of oneCCL API:

```
ccl_status_t ccl_wait(ccl_request_t req);
```

req requests the handle for communication operation being tracked

C++ version of oneCCL API:

```
void request::wait();
```

8.2.3 Error Handling

Error handling in oneCCL is implemented differently in C and C++ versions of API. C version of API uses error codes that are returned by every exposed function, while C++ API uses exceptions.

C version of oneCCL API:

```
typedef enum
{
    ccl_status_success          = 0,
    ccl_status_out_of_resource  = 1,
    ccl_status_invalid_arguments = 2,
    ccl_status_unimplemented    = 3,
    ccl_status_runtime_error    = 4,
    ccl_status_blocked_due_to_resize = 5,
    ccl_status_last_value
} ccl_status_t;
```

C++ version of oneCCL API:

```
class ccl_error : public std::runtime_error
```

8.3 Programming Model

8.3.1 Generic Workflow

Below is a generic flow for using C++ API of oneCCL:

1. Initialize the library:

```
ccl::environment::instance();
```

Alternatively, you can create communicator objects:

```
ccl::communicator_t comm = ccl::environment::instance().create_
˓→communicator();
```

2. Execute collective operation of choice on this communicator:

```
auto request = comm.allreduce(...);
request->wait();
```

8.3.2 GPU support

The choice between CPU and GPU backends is performed by specifying `ccl_stream_type` value at the moment when `ccl stream` object is created:

- For GPU backend you should specify `ccl_stream_sycl` as the first argument.
- For collective operations, which operate on SYCL* stream, C version of oneCCL API expects communication buffers to be `sycl::buffer*` objects cast to `void*`.

The example below demonstrates these concepts.

8.3.2.1 Example

Consider simple `allreduce` example for GPU.

1. Create GPU `ccl stream` object:

C version of oneCCL API:

```
ccl_stream_create(ccl_stream_sycl, &q, &stream);
```

C++ version of oneCCL API:

```
ccl::stream_t stream = ccl::environment::instance().create_
˓→stream(cc::stream_type::sycl, &q);
```

`q` is an object of type `sycl::queue`.

2. To illustrate the `ccl_allreduce` execution, initialize `sendbuf` (in real scenario it is provided by application):

```
auto host_acc_sbuf = sendbuf.get_access<mode::write>();
for (i = 0; i < COUNT; i++) {
    host_acc_sbuf[i] = rank;
}
```

3. For demonstration purposes only, modify the `sendbuf` on the GPU side:

```
q.submit ([&] (cl::sycl::handler& cgh) {
    auto dev_acc_sbuf = sendbuf.get_access<mode::write>(cgh);
    cgh.parallel_for<class allreduce_test_sbuf_modify>(range<1>{COUNT}, [
        [=] (item<1> id) {
            dev_acc_sbuf[id] += 1;
        });
});
```

`ccl_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 the size equal to the number of processes (p), where all elements are equal to the sum of arithmetical progression:

$$p \cdot (p - 1)/2$$

C version of oneCCL API:

```
ccl_allreduce(&sendbuf,
              &recvbuf,
              COUNT,
              ccl_dtype_int,
              ccl_reduction_sum,
              NULL, /* attr */
              NULL, /* comm */
              stream,
              &request);
ccl_wait(request);
```

C++ version of oneCCL API:

```
comm.allreduce(sendbuf,
               recvbuf,
               COUNT,
               ccl::reduction::sum,
               nullptr, /* attr */
               stream)->wait();
```

4. Check the correctness of `ccl_allreduce` on the GPU:

```
q.submit ([&] (handler& cgh) {
    auto dev_acc_rbuf = recvbuf.get_access<mode::write>(cgh);
    cgh.parallel_for<class allreduce_test_rbuf_check>(range<1>{COUNT}, [
        [=] (item<1> id) {
            if (dev_acc_rbuf[id] != size*(size+1)/2) {
                dev_acc_rbuf[id] = -1;
            }
        });
});
```

```
if (rank == COLL_ROOT) {
    auto host_acc_rbuf_new = recvbuf.get_access<mode::read>();
    for (i = 0; i < COUNT; i++) {
        if (host_acc_rbuf_new[i] == -1) {
            cout << "FAILED" << endl;
            break;
    }
}
```

(continues on next page)

(continued from previous page)

```
    }  
    if (i == COUNT) {  
        cout<<"PASSED"<<endl;  
    }  
}
```

Note: When using C version of oneCCL API, it is required to explicitly free the created GPU ccl stream object:

```
ccl_stream_free(stream);
```

For C++ version of oneCCL API this will be performed implicitly.

8.3.3 CPU support

The choice between CPU and GPU backends is performed by specifying `ccl_stream_type` value at the moment of creating ccl stream object.

- For CPU backend one should specify `ccl_stream_cpu` there.
 - For collective operations performed using CPU stream, oneCCL expects communication buffers to reside in the host memory.

The example below demonstrates these concepts.

8.3.3.1 Example

Consider simple allreduce example for CPU.

- #### 1. Create CPU ccl stream object:

C version of oncCCL API:

```
/* For CPU stream, NULL is passed instead of native stream pointer */
ccl_stream_create(ccl_stream_cpu, NULL, &stream);
```

C++ version of oneCCL API:

```
/* For CPU, NULL is passed instead of native stream pointer */
ccl::stream_t stream = ccl::environment::instance().create_
    ↵stream(cc::stream_type::cpu, NULL);
```

or just

```
ccl::stream stream;
```

2. To illustrate the `ccl_allreduce` execution, initialize `sendbuf` (in real scenario it is supplied by application):

```
/* initialize sendbuf */
for (i = 0; i < COUNT; i++) {
    sendbuf[i] = rank;
}
```

`ccl_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 the size equal to the number of processes (p), where all elements are equal to the sum of arithmetical progression:

$$p \cdot (p - 1)/2$$

C version of oneCCL API:

```
ccl_allreduce(&sendbuf,
              &recvbuf,
              COUNT,
              ccl_dtype_int,
              ccl_reduction_sum,
              NULL, /* attr */
              NULL, /* comm */
              stream,
              &request);
ccl_wait(request);
```

C++ version of oneCCL API:

```
comm.allreduce(&sendbuf,
               &recvbuf,
               COUNT,
               ccl::reduction::sum,
               nullptr, /* attr */
               stream)->wait();
```

Note: When using C version of oneCCL API, it is required to explicitly free `ccl` stream object:

```
ccl_stream_free(stream);
```

For C++ version of oneCCL API this is performed implicitly.

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

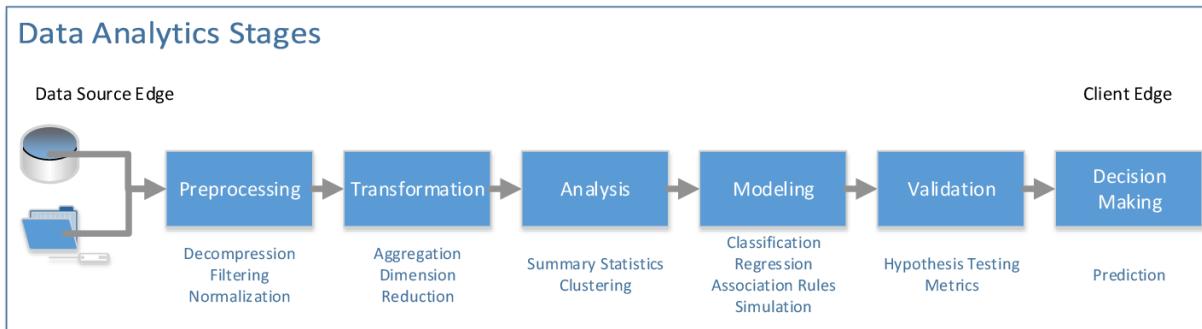
This document specifies requirements for implementations of oneAPI Data Analytics Library (oneDAL).

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.

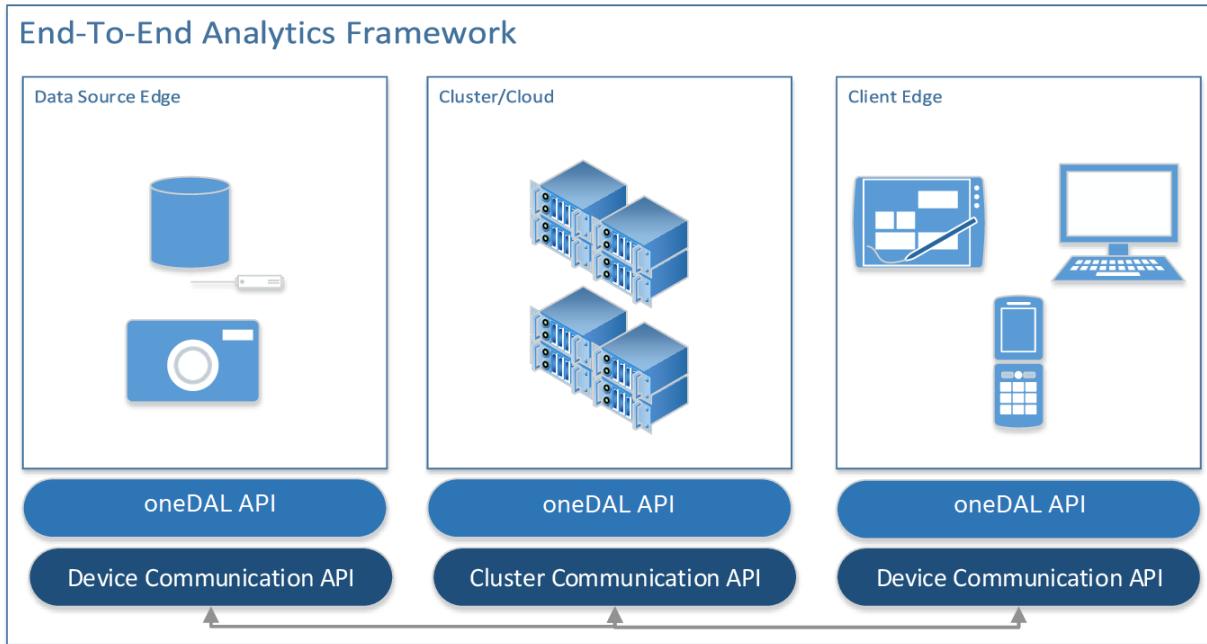
For general information, visit [oneDAL GitHub*](#) page.

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"> If A is a set, this denotes its cardinality, i.e., the number of elements in the set A. If A is a real number, this denotes an absolute value of A.
$\ 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

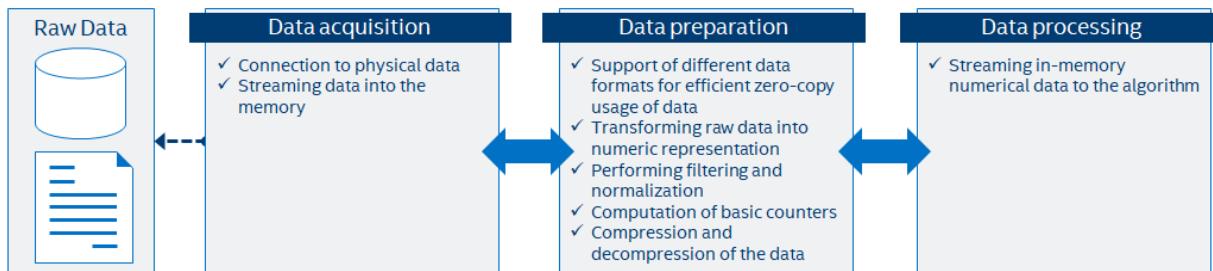
10.6 Data management

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

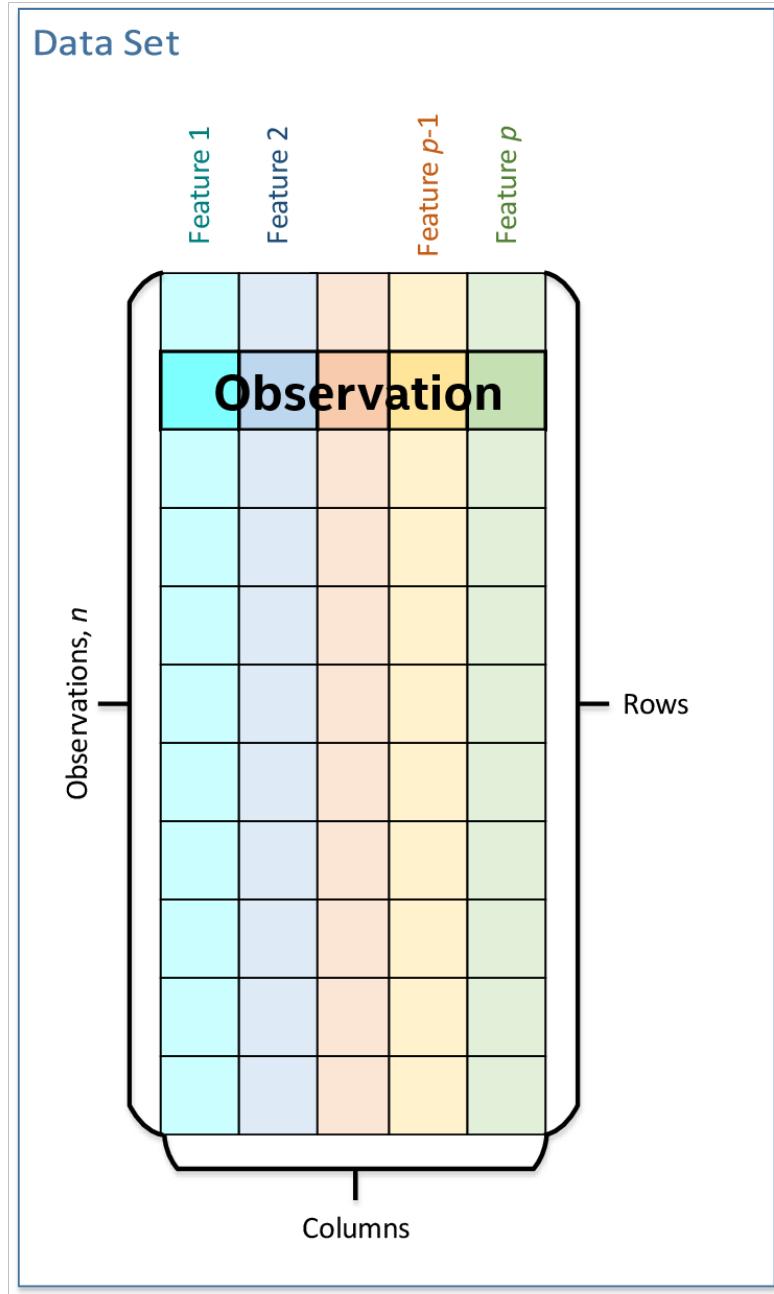


10.6.1 Key concepts

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:

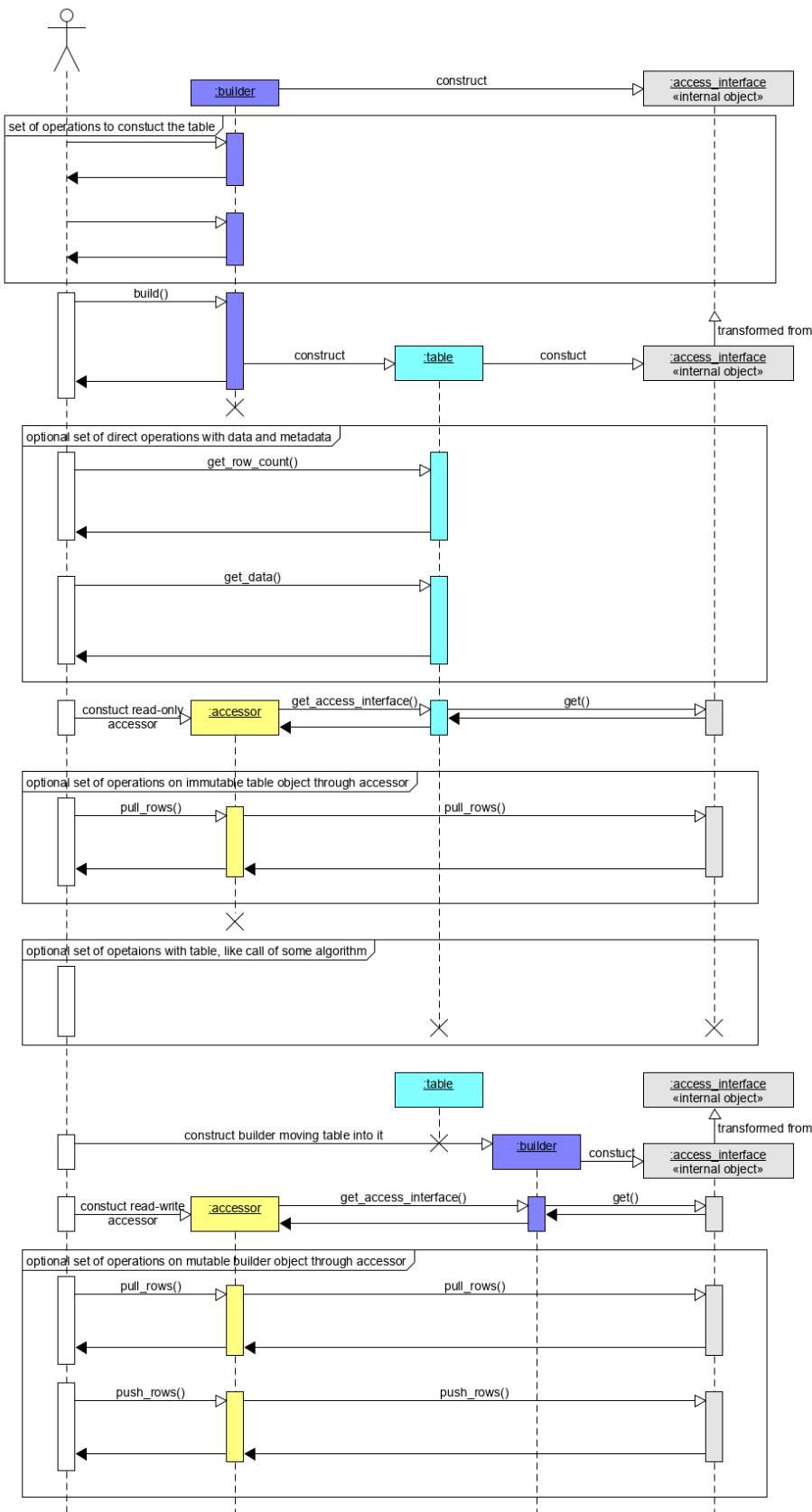
- To have unified access to the data from various sets of different objects, such as *tables* or *table builders*, without exposing their implementation details.
- To convert a variety of numeric *data formats* into a smaller set of formats.
- To provide a *flat* view on the data blocks of a *dataset* 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).
<i>aos_table</i>	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_equalFlag 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 ε 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, $1 \leq i \leq n$.

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 ≥ 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.	filename ... 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.	word ,... 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 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"> Windows* OS: 1 if <code>_DEBUG</code> is defined, 0 otherwise. All other systems: 0. 	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`.

`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 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 \times 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 \times 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 \times . Template function `parallel_scan` decides if and when to generate parallel work. It is thus crucial that \times 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_control` 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
Functional Nodes		
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
Buffering Nodes		
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
Split/Join Nodes		
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
Other Nodes		
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`, `multipfunction_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)

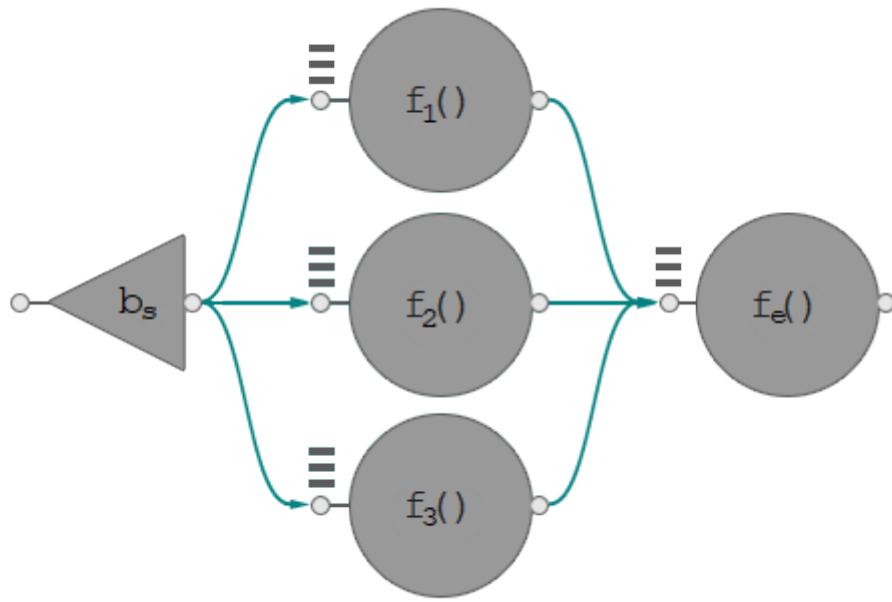


Fig. 1: Data flow graph with critical path.

(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 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>
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 );
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 );
```

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)

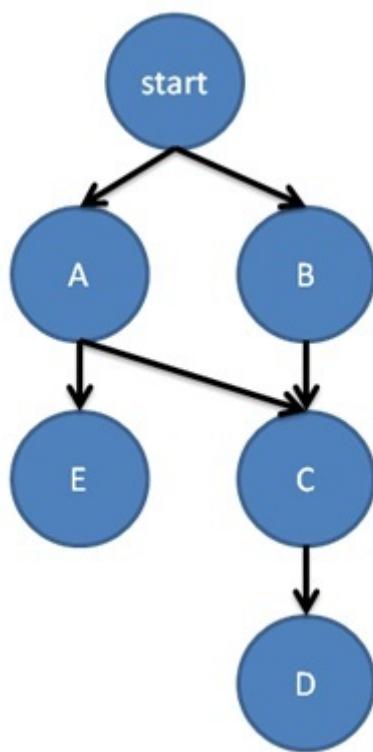


Fig. 2: A simple dependency graph.

(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.

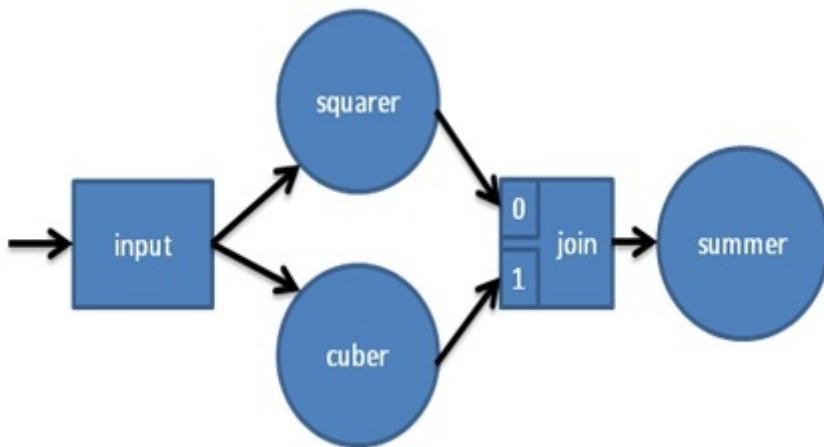


Fig. 3: A simple message flow graph.

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 {
        int current_thread_index();
```

(continues on next page)

(continued from previous page)

```

    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 {
public:
    task_scheduler_observer();
}

```

(continues on next page)

(continued from previous page)

```

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, boolconst value_type& value );
iterator insert( const_iterator hint, const value_type& value );

template <typename P>
std::pair<iterator, booltemplate <typename P>
iterator insert( const_iterator hint, P&& value );

std::pair<iterator, booltemplate <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(),
                                     const key_equal& equal = key_equal(),
                                     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,
                         const allocator_type& alloc );

explicit concurrent_unordered_set( const allocator_type& alloc );

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_
 $\leftrightarrow$ );

template <typename InputIterator>
concurrent_unordered_set( InputIterator first, InputIterator last,
                        size_type bucket_count, const hasher& hash,
                        const allocator_type& alloc );

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() );

concurrent_unordered_set( std::initializer_list<value_type> init,
                        size_type bucket_count, const allocator_type& alloc_
 $\leftrightarrow$ );

concurrent_unordered_set( std::initializer_list<value_type> init,
                        size_type bucket_count, const hasher& hash,
                        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(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_bount() 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 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.

11.3.1.1.2 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 $n * \text{sizeof}(T)$ 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_rw_mutex

[mutex.queueing_rw_mutex]

A `queueing_rw_mutex` is a class that models the *ReaderWriterMutex requirement* concept. The `queueing_rw_mutex` is not recursive. The `queueing_rw_mutex` is fair, threads acquire a lock on a mutex in the order that they request it.

```

// Defined in header <tbb/queueing_rw_mutex.h>

namespace tbb {
    class queueing_rw_mutex {
public:
    queueing_rw_mutex() noexcept;
    ~queueing_rw_mutex();

    queueing_rw_mutex(const queueing_rw_mutex&) = delete;
    queueing_rw_mutex& operator=(const queueing_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 = 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 is a programming interface for video decoding, encoding, and processing to build portable media pipelines on CPU's, GPU's, and other accelerators. It provides API primitives for zero-copy buffer sharing, device discovery and selection in media centric and video analytics workloads. oneVPL's backwards and cross-architecture compatibility ensures optimal execution on current and next generation hardware without source code changes.

See [oneVPL API Reference](#) for the detailed API description.

12.1 oneVPL for Intel® Media Software Development Kit Users

oneVPL is source compatible with Intel® Media Software Development Kit (MSDK), allowing applications to use MSDK to target older hardware and oneVPL to target everything else. oneVPL offers improved usability over MSDK. Some obsolete features of MSDK have been omitted from oneVPL.

12.1.1 oneVPL Usability Enhancements

1. Smart dispatcher with implementations capabilities discovery. Explore [SDK Session](#) for more details.
2. Simplified decoder initialization. Explore [Decoding Procedures](#) for more details.
3. New memory management and components (session) interoperability. Explore [Internal memory management](#) and [Decoding Procedures](#) for more details.
4. Improved internal threading and internal task scheduling.

12.1.2 Obsolete MSDK Features omitted from oneVPL

The following MSDKt features are not included in oneVPL:

Audio Support oneVPL is for video processing, and removes audio APIs that duplicate functionality from other audio libraries like [Sound Open Firmware](#)

ENC and PAK interfaces Available as part of Flexible Encode Infrastructure (FEI) and plugin interfaces. FEI is the Intel Graphic specific feature designed for AVC and HEVC encoders, not widely used by customers.

User plugins architecture oneVPL enables robust video acceleration through API implementations of many different video processing frameworks, making support of its own user plugin framework obsolete.

External Buffer memory managment A set of callback functions to replace internal memory allocation is obsolete.

Video Processing extended runtime functionality Video processing function MFXVideoVPP_RunFrameVPPAsynchEx is used for plugins only and is obsolete.

External threading New threading model makes MFXDoWork function obsolete

The following behaviors occur when attempting to use a MSDK API that is not supported by oneVPL:

Code compilation 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 MSDK and used with a oneVPL runtime Code previously compiled with MSDK and executing using a oneVPL runtime will generate an MFX_ERR_UNSUPPORTED error when calling a removed function.

12.1.3 MSDK API's not present in oneVPL

Audio related functions:

```
MFXAudioCORE_SyncOperation(mfxSession session, mfxSyncPoint syncp, mfxU32 wait)
MFXAudioDECODE_Close(mfxSession session)
MFXAudioDECODE_DecodeFrameAsync(mfxSession session, mfxBitstream *bs,
                                mfxAudioFrame *frame_out, mfxSyncPoint *syncp)
MFXAudioDECODE_DecodeHeader(mfxSession session, mfxBitstream *bs, mfxAudioParam *par)
MFXAudioDECODE_GetAudioParam(mfxSession session, mfxAudioParam *par)
MFXAudioDECODE_Init(mfxSession session, mfxAudioParam *par)
MFXAudioDECODE_Query(mfxSession session, mfxAudioParam *in, mfxAudioParam *out)
MFXAudioDECODE_QueryIOSize(mfxSession session, mfxAudioParam *par, ▾
    ↵mfxAudioAllocRequest *request)
MFXAudioDECODE_Reset(mfxSession session, mfxAudioParam *par)
MFXAudioENCODE_Close(mfxSession session)
MFXAudioENCODE_EncodeFrameAsync(mfxSession session, mfxAudioFrame *frame,
                                mfxBitstream *buffer_out, mfxSyncPoint *syncp)
MFXAudioENCODE_GetAudioParam(mfxSession session, mfxAudioParam *par)
MFXAudioENCODE_Init(mfxSession session, mfxAudioParam *par)
MFXAudioENCODE_Query(mfxSession session, mfxAudioParam *in, mfxAudioParam *out)
MFXAudioENCODE_QueryIOSize(mfxSession session, mfxAudioParam *par, ▾
    ↵mfxAudioAllocRequest *request)
MFXAudioENCODE_Reset(mfxSession session, mfxAudioParam *par)
```

Flexible encode infrastructure functions:

```
MFXVideoENC_Close(mfxSession session)
MFXVideoENC_GetVideoParam(mfxSession session, mfxVideoParam *par)
MFXVideoENC_Init(mfxSession session, mfxVideoParam *par)
MFXVideoENC_ProcessFrameAsync (mfxSession session, mfxENCIInput *in,
                               mfxENCOOutput *out, mfxSyncPoint *syncp)
MFXVideoENC_Query(mfxSession session, mfxVideoParam *in, mfxVideoParam *out)
MFXVideoENC_QueryIOSurf(mfxSession session, mfxVideoParam *par,
                        mfxFrameAllocRequest *request)
MFXVideoENC_Reset(mfxSession session, mfxVideoParam *par)
MFXVideoPAK_Close(mfxSession session)
MFXVideoPAK_GetVideoParam(mfxSession session, mfxVideoParam *par)
MFXVideoPAK_Init(mfxSession session, mfxVideoParam *par)
MFXVideoPAK_ProcessFrameAsync(mfxSession session, mfxPAKInput *in,
                             mfxPAKOutput *out, mfxSyncPoint *syncp)
MFXVideoPAK_Query(mfxSession session, mfxVideoParam *in, mfxVideoParam *out)
MFXVideoPAK_QueryIOSurf(mfxSession session, mfxVideoParam *par,
                        mfxFrameAllocRequest *request)
MFXVideoPAK_Reset(mfxSession session, mfxVideoParam *par)
```

User Plugin functions:

```

MFXAudioUSER_ProcessFrameAsync(mfxSession session, const mfxHDL *in,
                                mfxU32 in_num, const mfxHDL *out,
                                mfxU32 out_num, mfxSyncPointx *syncp)
MFXAudioUSER_Register(mfxSession session, mfxU32 type, const mfxPlugin *par)
MFXAudioUSER_Unregister(mfxSession session, mfxU32 type)
MFXVideoUSER_GetPlugin(mfxSession session, mfxU32 type, mfxPlugin *par)
MFXVideoUSER_ProcessFrameAsync(mfxSession session, const mfxHDL *in, mfxU32 in_num,
                                const mfxHDL *out, mfxU32 out_num, mfxSyncPoint *syncp)
MFXVideoUSER_Register(mfxSession session, mfxU32 type, const mfxPlugin *par)
MFXVideoUSER_Unregister(mfxSession session, mfxU32 type)
MFXVideoUSER_Load(mfxSession session, const mfxPluginUID *uid, mfxU32 version)
MFXVideoUSER_LoadByPath(mfxSession session, const mfxPluginUID *uid, mfxU32 version,
                        const mfxChar *path, mfxU32 len)
MFXVideoUSER_UnLoad(mfxSession session, const mfxPluginUID *uid)
MFXDoWork(mfxSession session)

```

Memory functions:

```
MFXVideoCORE_SetBufferAllocator(mfxSession session, mfxBufferAllocator *allocator)
```

Video processing functions:

```

MFXVideoVPP_RunFrameVPPAsyncEx(mfxSession session, mfxFrameSurface1 *in,
                                mfxFrameSurface1 *surface_work, mfxFrameSurface1 *
                                surface_out,
                                mfxSyncPoint *syncp)

```

Important: Corresponding extension buffers are also removed.

12.1.4 oneVPL API's not present in MSDK

oneVPL dispatcher functions:

```

MFXLoad()
MFXUnload()
MFXCreateConfig()
MFXSetConfigFilterProperty()
MFXEnumImplementations()
MFXCreateSession()
MFXDispReleaseImplDescription()

```

Memory management functions:

```

MFXMemory_GetSurfaceForVPP()
MFXMemory_GetSurfaceForEncode()
MFXMemory_GetSurfaceForDecode()

```

Implementation capabilities retrieval functions:

```
MFXQueryImplDescription()
MFXReleaseImplDescription()
```

12.2 oneVPL API versioning

As a successor of MSDKt oneVPL API version starts from 2.0.

Experimental API's 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.

12.3 Acronyms and Abbreviations

Acronyms / Abbreviations	Meaning
API	Application Programming Interface
AVC	Advanced Video Codec (same as H.264 and MPEG-4, part 10)
Direct3D	Microsoft* Direct3D* version 9 or 11.1
Direct3D9	Microsoft* Direct3D* version 9
Direct3D11	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
HRD	Hypothetical Reference Decoder
IDR	Instantaneous decoding fresh picture, a term used in the H.264 specification
LA	Look Ahead. Special encoding mode where encoder performs pre analysis of several frames before actual encoding
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 Associated Audio
NAL	Network Abstraction Layer
NV12	A color format for raw video frames
PPS	Picture Parameter Set
QP	Quantization Parameter
RGB3	Twenty-four-bit RGB color format. Also known as RGB24
RGB4	Thirty-two-bit RGB color format. Also known as RGB32
SDK	Intel® Media Software Development Kit – SDK
SEI	Supplemental Enhancement Information
SPS	Sequence Parameter Set
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
video memory	memory used by hardware acceleration device, also known as GPU, to hold frame and other types of video data
VPP	Video Processing
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
IYUV	A color format for raw video frames, also known as I420

Table 1 – continued from previous page

Acronyms / Abbreviations	Meaning
P010	A color format for raw video frames, extends NV12 for 10 bit
I010	A color format for raw video frames, extends IYUV/I420 for 10 bit
GPB	Generalized P/B picture. B-picture, containing only forward references in both L0 and L1
HDR	High Dynamic Range
BRC	Bit Rate Control
MCTF	Motion Compensated Temporal Filter. Special type of a noise reduction filter which utilizes motion to
iGPU/iGfx	Integrated Intel® HD Graphics
dGPU/dGfx	Discrete Intel® Graphics

12.4 Architecture

SDK functions fall into the following categories:

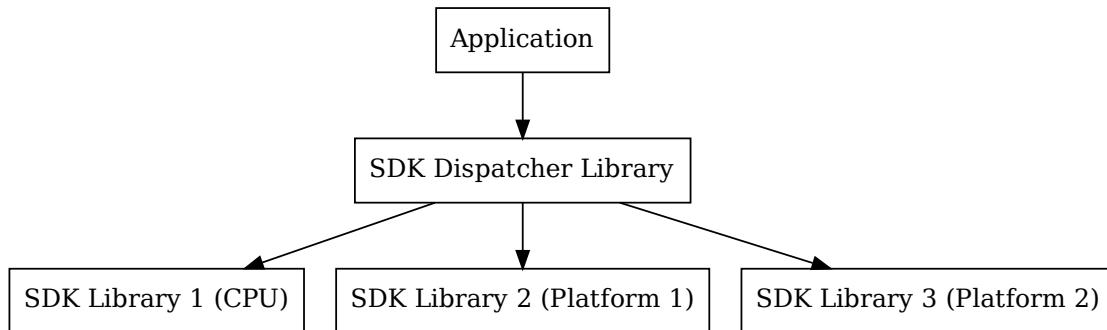
Category	Description
DECODE	Decode compressed video streams into raw video frames
ENCODE	Encode raw video frames into compressed bitstreams
VPP	Perform video processing on raw video frames
CORE	Auxiliary functions for synchronization
Misc	Global auxiliary functions

With the exception of the global auxiliary functions, SDK functions are named after their functioning domain and category, as illustrated below. Here, SDK only exposes video domain functions.

MFVideoDECODE_DecodeFrameAsync

Prefix Domain Class Name

Applications use SDK functions by linking with the SDK dispatcher library, as illustrated below. The dispatcher library identifies the hardware acceleration device on the running platform, determines the most suitable platform library, and then redirects function calls. If the dispatcher is unable to detect any suitable platform-specific hardware, the dispatcher redirects SDK function calls to the default software library.



12.4.1 Video Decoding

The **DECODE** class of functions takes a compressed bitstream as input and converts it to raw frames as output.

DECODE processes only pure or elementary video streams. The library cannot process bitstreams that reside in a container format, such as MP4 or MPEG. The application must first de-multiplex the bitstreams. De-multiplexing extracts pure video streams out of the container format. The application can provide the input bitstream as one complete frame of data, less than one frame (a partial frame), or 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. That is, the first byte of a video coding layer NAL unit for H.264, or picture header for MPEG-2 and VC-1. **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 that 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.

12.4.2 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 key frames: 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 in two ways:

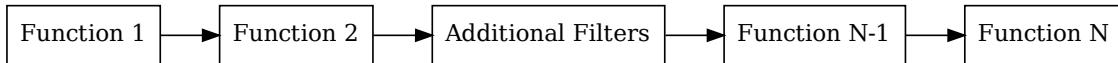
- Display order: **ENCODE** receives input frames in the display order. A few GOP structure parameters specify the GOP sequence during **ENCODE** initialization. Scene change results from the video processing stage of a pipeline can alter the GOP sequence.
- Encoded order: **ENCODE** receives input frames 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.

An **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.)

12.4.3 Video Processing

Video processing (**VPP**) takes raw frames as input and provides raw frames as output.



The actual conversion process is a chain operation with many single-function filters, as Figure 3 illustrates. 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.

Table 1 shows the 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 procedure / Configuration” for more details on how to configure optional filters.

Todo: create link to “Video Processing procedure / Configuration”

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)

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: ‘X’ indicates a supported function.

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 pipeline. All optional filters are skipped. See description of `MFX_WRN_FILTER_SKIPPED` warning in `mfxStatus` enum for more details on how to retrieve list of active filters.

Deinterlacing/Inverse Telecine Support in **VPP**:

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

Note: ‘X’ indicates a supported function.

This table describes pure deinterlacing algorithm. The application can combine it with frame rate conversion to achieve any desirable input/output frame rate ratio. Note, that in this table input rate is field rate, i.e. number of video fields in one second of video. The SDK uses frame rate in all configuration parameters, so this input field rate should be divided by two during the SDK configuration. For example, 60i to 60p conversion in this table is represented by right bottom cell. It should be described in `mfxVideoParam` as input frame rate equal to 30 and output 60.

SDK support two HW-accelerated deinterlacing algorithms: BOB DI (in Linux’s 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. So user can select as usual between speed and quality.

User can exactly configure DI modes via `mfxExtVPPDeinterlacing`.

There is one special mode of deinterlacing available in combination with frame rate conversion. If VPP input frame is interlaced (TFF or BFF) and output is progressive and 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), special mode of VPP turned on: for 30 interlaced input frames application will get 60 different progressive output frames

Color formats supported by **VPP** filters:

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: ‘X’ indicates a supported function.

Note: The SDK video processing pipeline supports limited HW acceleration for P010 format - zeroed mfxFrameInfo::Shift leads to partial acceleration.

Todo: create link to mfxFrameInfo::Shift

Note: The SDK video processing pipeline does not support HW acceleration for P210 format.

Todo: Keep or remove HW?

12.5 Programming Guide

This chapter describes the concepts used in programming the 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) */
```

12.5.1 Status Codes

The SDK functions organize into classes for easy reference. The classes 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 de-initialize specific operations defined for the class. Call all other member functions within a given class (except **Query** and **QueryIOSurf**) within the **Init** ... **Reset** (optional) ... **Close** sequence.

The **Init** and **Reset** member functions both set up necessary internal structures for media processing. The difference between the two is that the **Init** functions allocate memory while the **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 whether an operation succeeded or failed. See the mfxStatus enumerator for all defined status codes. The status code `MFX_ERR_NONE` indicates that the function successfully completed its operation. Status codes are less than `MFX_ERR_NONE` for all errors and greater than `MFX_ERR_NONE` for all warnings.

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 `MFX_ERR_MORE_DATA` or `MFX_ERR_MORE_SURFACE` or `MFX_ERR_MORE_BITSTREAM`), the function aborts the operation. The application must call either the **Reset** function to put 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**.

12.5.2 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.

12.5.2.1 Media SDK 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 the former case, the SDK functions execute on a CPU, and in the latter case, 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 (`MFX_IMPL_HARDWARE1`, ..., `MFX_IMPL_HARDWARE4`).

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, dispatcher works in that way:

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	libmfxhw64.so.1	64-bit hardware-based implementation
Linux	libmfxhw64.so.1	32-bit hardware-based implementation
Windows	libmfxsw32.dll	64-bit software-based implementation
Windows	libmfxsw32.dll	32-bit software-based implementation
Windows	libmfxhw64.dll	64-bit hardware-based implementation
Windows	libmfxhw64.dll	32-bit hardware-based implementation

- Once library is loaded, dispatcher obtains addresses of each SDK function. See table with the list of functions to export.

12.5.2.2 oneVPL dispatcher

oneVPL dispatcher extends the legacy dispatcher by providing additional ability to select 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, image (frame) size in pixels and so on.

This is recommended way for the user to configure the dispatcher's capabilities search filters and create session based on suitable implementation:

- Create loader ([MFXLoad\(\)](#) dispatcher's function).
- Create loader's config ([MFXCreateConfig\(\)](#) dispatcher's function).
- Add config properties ([MFXSetConfigFilterProperty\(\)](#) dispatcher's function).
- Explore available implementations according ([MFXEnumImplementations\(\)](#) dispatcher's function).
- Create suitable session ([MFXCreateSession\(\)](#) dispatcher's function).

This is application termination procedure:

- Destroys session ([MFXClose\(\)](#) function).
- Destroys loader ([MFXUnload\(\)](#) dispatcher's function).

Note: Multiple loader instances can be created.

Note: Each loader may have multiple config objects associated with it.

Important: One config object can handle only one filter property.

Note: Multiple sessions can be created by using one loader object.

When dispatcher searches for the implementation it uses following priority rules:

- HW implementation has priority over SW implementation.
- Gen HW implementation has priority over VSI HW implementation.

-
3. Highest API version has higher priority over lower API version.

Note: Implementation has priority over the API version. In other words, dispatcher must return implementation with highest API priority (greater or equal to the requested).

Dispatcher searches implementation in the following folders at runtime (in priority order):

1. User-defined search folders.
2. oneVPL package.
3. Standalone MSDK package (or driver).

User has ability to develop its own implementation and guide oneVPL dispatcher to load his implementation by providing list of search folders. The way how it can be done depends on OS.

- linux: User can provide 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, user can use Windows registry.

Different SW implementations is supported by the dispatcher. User can use field *mfxImplDescription::VendorID* or *mfxImplDescription::VendorImplID* or *mfxImplDescription::ImplName* to search for the particular implementation.

Internally, dispatcher works in that way:

1. Dispatcher loads any shared library with in given search folders.
2. For each loaded library, dispatcher tries to resolve address of the `MFXQueryImplCapabilities()` function to collect the implementation's capabilities.
3. Once user requested to create the session based on this implementation, dispatcher obtains addresses of all each SDK function. See table with the list of functions to export.

This table summarizes list of environmental variables to control the dispatcher behaviour:

Variable	Purpose
ONEVPL_SEARCH_PATH	List of user-defined search folders.

Note: Each implementation must support both dispatchers for backward compatibility with existing applications.

12.5.2.3 Multiple Sessions

Each SDK session can run exactly one instance of **DECODE**, **ENCODE** and **VPP** functions. This is good for a simple transcoding operation. If the application needs more than one instance of **DECODE**, **ENCODE** and **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 SDK session can independently 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 (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 function `MFXCloneSession()` 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, one of the sessions (the first join) serves as a parent session, scheduling execution resources, with all others child sessions relying 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 less CPU cycles. Session priority does not affect hardware accelerated processing.

After the completion of all session operations, the application can use the function [MFXDisjoinSession\(\)](#) to remove the joined state of a session. Do not close the parent session until all child sessions are disjoined or closed.

12.5.3 Frame and Fields

In SDK terminology, a frame (or frame surface, interchangeably) 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.

12.5.3.1 Frame Surface Locking

During encoding, decoding or video processing, cases arise that require reserving input or output frames for future use. In the case of decoding, for example, a frame that is ready for output must remain as a reference frame until the current sequence pattern ends. The usual approach 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. This mechanism is as follows:

- 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, part of the mfxFrameData structure. Initially, the Locked counter is set to zero.
- The application calls an SDK function with frame surfaces from the pool, whose Locked counter is set as appropriate: for decoding or video processing operations where the SDK uses the surfaces to write it 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.” That is, the application can read, but cannot alter, move, delete or free the frame surface.
- 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 race condition.

Attention: Modifying the Locked counter is not recommended.

Starting from API version 2.0 mfxFrameSurfaceInterface structure as a set of callback functions was introduced for mfxFrameSurface1 to work with frames. This interface defines mfxFrameSurface1 as a reference counted object which can be allocated by the SDK or application. Application has to follow the general rules of operations with reference counted objects. As example, when surfaces are allocated by the SDK during MFXVideoDECODE_CODE_DecodeFrameAsync or with help of MFXMemory_GetSurfaceForVPP, MFXMemory_GetSurfaceForEncode, application has to call correspondent mfxFrameSurfaceInterface->(*Release) for the surfaces whose are no longer in use.

Attention: Need to distinguish Locked counter which defines read/write access polices and reference counter responsible for managing frames' lifetime.

Note: all mfxFrameSurface1 structures starting from mfxFrameSurface1::mfxStructVersion = {1,1} supports mfxFrameSurfaceInterface.

12.5.4 Decoding Procedures

Example 1 shows the pseudo code of the decoding procedure. The following describes a few key points:

- The application can use the `MFXVideoDECODE_DecodeHeader()` function to retrieve decoding initialization parameters from the bitstream. This step is optional if such parameters are retrievable from other sources such as an audio/video splitter.
- The application uses the `MFXVideoDECODE_QueryIOSurf()` function to obtain the number of working frame surfaces required to reorder output frames. This call is optional and required when application uses external 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 from API version 2.0 application can provide NULL as 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:

- MFX_ERR_MORE_DATA: The function needs additional bitstream input. The existing buffer contains less than a frame worth of bitstream data.
- MFX_ERR_MORE_SURFACE: The function needs one more frame surface to produce any output.
- MFX_ERR_REALLOC_SURFACE: Dynamic resolution change case - the function needs bigger working frame surface (work).
- Upon successful decoding, the `MFXVideoDECODE_DecodeFrameAsync()` function returns MFX_ERR_NONE. However, the decoded frame data (identified by the disp pointer) is not yet available because the `MFXVideoDECODE_DecodeFrameAsync()` function is asynchronous. The application has to 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 MFX_ERR_MORE_DATA.

Example 2 below demonstrates simplified decoding procedure.

Starting for API version 2.0 new decoding approach has been introduced. For simple use cases, when user just wants to decode some elementary stream and don't want to set additional parameters, the simplified procedure of Decoder's initialization has been proposed. For such situations it is possible to skip explicit stages of stream's header decoding and Decoder's initialization and perform it implicitly during decoding of first frame. This change also requires additional field in mfxBitstream object to indicate codec type. In that mode decoder allocates mfxFrameSurface1 internally, so users should set input surface to zero.

Example 1: Decoding Pseudo Code

```

MFXVideoDECODE_DecodeHeader(session, bitstream, &init_param);
MFXVideoDECODE_QueryIOSurf(session, &init_param, &request);
allocate_pool_of_frame_surfaces(request.NumFrameSuggested);
MFXVideoDECODE_Init(session, &init_param);
sts=MFX_ERR_MORE_DATA;
for (;;) {
    if (sts==MFX_ERR_MORE_DATA && !end_of_stream())
        append_more_bitstream(bitstream);
    find_unlocked_surface_from_the_pool(&work);
    bits=(end_of_stream())?NULL:bitstream;
    sts=MFXVideoDECODE_DecodeFrameAsync(session,bits,work,&disp,&syncp);
    if (sts==MFX_ERR_MORE_SURFACE) continue;
    if (end_of_bitstream() && sts==MFX_ERR_MORE_DATA) break;
    if (sts==MFX_ERR_REALLOC_SURFACE) {
        MFXVideoDECODE_GetVideoParam(session, &param);
        realloc_surface(work, param.mfx.FrameInfo);
        continue;
    }
    // skipped other error handling
    if (sts==MFX_ERR_NONE) {
        MFXVideoCORE_SyncOperation(session, syncp, INFINITE);
        do_something_with_decoded_frame(disp);
    }
}
MFXVideoDECODE_Close();
free_pool_of_frame_surfaces();

```

Example 2: Simplified decoding procedure

```

sts=MFX_ERR_MORE_DATA;
for (;;) {
    if (sts==MFX_ERR_MORE_DATA && !end_of_stream())
        append_more_bitstream(bitstream);
    bits=(end_of_stream())?NULL:bitstream;
    sts=MFXVideoDECODE_DecodeFrameAsync(session,bits,NULL,&disp,&syncp);
    if (sts==MFX_ERR_MORE_SURFACE) continue;
    if (end_of_bitstream() && sts==MFX_ERR_MORE_DATA) break;
    // skipped other error handling
    if (sts==MFX_ERR_NONE) {
        MFXVideoCORE_SyncOperation(session, syncp, INFINITE);
        do_something_with_decoded_frame(disp);
        release_surface(disp);
    }
}

```

12.5.4.1 Bitstream Repositioning

The application can use the following procedure for bitstream repositioning during decoding:

- Use the `MFXVideoDECODE_Reset()` function to reset the SDK decoder.
- Optionally, if the application maintains a sequence header that decodes correctly the bitstream at the new position, the application may insert the sequence header to the bitstream buffer.
- Append the bitstream from the new location to the bitstream buffer.
- Resume the decoding procedure. If the sequence header is not inserted in the above steps, the SDK decoder searches for a new sequence header before starting decoding.

12.5.4.2 Broken Streams Handling

Robustness and capability to handle broken input stream is important part of the decoder.

First of all, 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 bitstream are parsed and verified. If any of elements violate the specification then input bitstream is considered as invalid and decoder tries to re-sync (find next start code). The further decoder's behavior is depend on which syntax element is broken:

- SPS header – return `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 a lot of streams which violate the letter of standard but can be decoded without errors.

- Many streams have IDR frames with `frame_num != 0` while 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 doesn't invalidate the whole SPS, decoder either doesn't use corrupted VUI (AVC) or resets incorrect values to default (HEVC).

The corruption at reference frame is spread over all inter-coded pictures which use this reference for prediction. To cope with this problem you either have to periodically insert I-frames (intra-coded) or use ‘intra refresh’ technique. The latter allows to recover corruptions within a pre-defined time interval. The main point of ‘intra refresh’ is to insert cyclic intra-coded pattern (usually row) of macroblocks into the inter-coded pictures, restricting motion vectors accordingly. Intra-refresh is often used in combination with Recovery point SEI, where `recovery_frame_cnt` is derived from intra-refresh interval. Recovery point SEI message is well described at ITU-T H.264 D.2.7 and ITU-T H.265 D.2.8. This message can be used by the decoder to understand from which picture all subsequent (in display order) pictures contain no errors, if we start decoding from AU associated with this SEI message. In opposite to IDR, recovery point message doesn't mark reference pictures as “unused for reference”.

Besides validation of syntax elements and theirs constrains, decoder also uses various hints to handle broken streams.

- If there are no valid slices for current frame – 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 will be explicitly cleared by IDR or MMCO, the incorrect LTR could cause long standing visual artifacts. AVC decoder uses the following approaches to care about this:
 - When we have DPB overflow in case incorrect MMCO command which marks reference picture as LT, we rollback this operation
 - An IDR frame with frame_num != 0 can't be LTR
- If 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 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 really used for inter-prediction.

12.5.4.3 VP8 Specific Details

Unlike other supported by SDK decoders, VP8 can accept only complete frame as input and application should provide it accompanied by [MFX_BITSTREAM_COMPLETE_FRAME](#) flag. This is the single specific difference.

12.5.4.4 JPEG

The application can use the same decoding procedures for JPEG/motion JPEG decoding, as illustrated in 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 matrixes
- Interleaved and non-interleaved scans
- Single and multiple scans
 - chroma subsampling 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 channels images

The `MFXVideoDECODE_Query()` function will return `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 one 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 input bitstream is described by `JPEGChromaFormat` and `JPEGColorFormat` fields in `mfxInfoMFX` structure. The `MFXVideoDECODE_DecodeHeader()` function usually fills them in. But if JPEG bitstream does not contain color format information, application should provide it. Output color format is described by general SDK parameters - `FourCC` and `ChromaFormat` fields in `mfxFrameInfo` structure.

Motion JPEG supports interlaced content by compressing each field (a half-height frame) individually. This behavior is incompatible with the rest SDK transcoding pipeline, where SDK requires that fields 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:
 - Set the `PicStruct` field of the `mfxVideoParam` structure with proper interlaced type, `MFX_PICSTRUCT_FIELD_TFF` or `MFX_PICSTRUCT_FIELD_BFF`, from motion JPEG header.
 - Double the `Height` field of 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, application sends both fields for decoding together in the same `mfxBitstream`. Application also should set `DataFlag` in `mfxBitstream` structure to `MFX_BITSTREAM_COMPLETE_FRAME`. The SDK decodes both fields and combines them into odd and even lines as in the SDK convention.

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 `mfxVideoParam` structure. In this case, decoder ignores tables from bitstream and uses specified by application. The application can also retrieve these tables by attaching the same buffers to `mfxVideoParam` and calling `MFXVideoDECODE_GetVideoParam()` or `MFXVideoDECODE_DecodeHeader()` functions.

12.5.4.5 Multi-view video decoding

The SDK MVC decoder operates on complete MVC streams that contain all view/temporal configurations. The application can configure the SDK decoder to generate a subset at the decoding output. To do this, the application needs to understand the stream structure and based on such information configure the SDK decoder for target views.

The decoder initialization procedure is as follows:

- The application calls the `MFXVideoDECODE_DecodeHeader` function to obtain the stream structural information. This is actually done in two sub-steps:

- **The application calls the `MFXVideoDECODE_DecodeHeader` function with the `mfxExtMVCSeqDesc` structure attached.**

Do not allocate memory for the arrays in the `mfxExtMVCSeqDesc` structure just yet. Set the `View`, `ViewId` and `OP` pointers to `NULL` and set `NumViewAlloc`, `NumViewIdAlloc` and `NumOPAlloc` to zero. The function parses the bitstream and returns `MFX_ERR_NOT_ENOUGH_BUFFER` with the correct values `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.

- **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.

- The application fills the `mfxExtMvcTargetViews` structure to choose the target views, based on information described in the `mfxExtMVCSeqDesc` structure.
- **The application initializes the SDK decoder using the `MFXVideoDECODE_Init` function. The application must attach both 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 decoding procedure remains unchanged. As illustrated 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.

```

mfxExtBuffer *eb[2];
mfxExtMVCSeqDesc seq_desc;
mfxVideoParam init_param;

init_param.ExtParam=&eb;
init_param.NumExtParam=1;
eb[0]=&seq_desc;
MFXVideoDECODE_DecodeHeader(session, bitstream, &init_param);

/* select views to decode */
mfxExtMvcTargetViews tv;
init_param.NumExtParam=2;
eb[1]=&tv;

/* initialize decoder */
MFXVideoDECODE_Init(session, &init_param);

/* perform decoding */
for (;;) {
    MFXVideoDECODE_DecodeFrameAsync(session, bits, work, &disp,
                                    &syncp);
    MFXVideoCORE_SyncOperation(session, &syncp, INFINITE);
}

```

(continues on next page)

(continued from previous page)

```
/* close decoder */
MFXVideoDECODE_Close();
```

12.5.5 Encoding Procedures

12.5.5.1 Encoding procedure

There are two ways of allocation and handling in SDK for shared memory: external and internal.

Example below shows the pseudo code of the encoding procedure with external memory (legacy mode).

```
MFXVideoENCODE_QueryIOSurf(session, &init_param, &request);
allocate_pool_of_frame_surfaces(request.NumFrameSuggested);
MFXVideoENCODE_Init(session, &init_param);
sts=MFX_ERR_MORE_DATA;
for (;;) {
    if (sts==MFX_ERR_MORE_DATA && !end_of_stream()) {
        find_unlocked_surface_from_the_pool(&surface);
        fill_content_for_encoding(surface);
    }
    surface2=end_of_stream() ?NULL:surface;
    sts=MFXVideoENCODE_EncodeFrameAsync(session,NULL,surface2,bits,&syncp);
    if (end_of_stream() && sts==MFX_ERR_MORE_DATA) break;
    // Skipped other error handling
    if (sts==MFX_ERR_NONE) {
        MFXVideoCORE_SyncOperation(session, syncp, INFINITE);
        do_something_with_encoded_bits(bits);
    }
}
MFXVideoENCODE_Close();
free_pool_of_frame_surfaces();
```

The following describes a few key points:

- The application uses the MFXVideoENCODE_QueryIOSurf function to obtain the number of working frame surfaces required for reordering input frames.
- The application calls the MFXVideoENCODE_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 MFX_ERR_MORE_DATA to request additional input frames.
- Upon successful encoding, the MFXVideoENCODE_EncodeFrameAsync function returns MFX_ERR_NONE. However, the encoded bitstream is not yet available because the MFXVideoENCODE_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 MFXVideoENCODE_EncodeFrameAsync function with NULL surface pointer to drain any remaining bitstreams cached within the SDK encoder, until the function returns MFX_ERR_MORE_DATA.

Note: It is the application's responsibility to fill pixels outside of crop window when it is smaller than frame to be encoded. Especially in cases when crops are not aligned to minimum coding block size (16 for AVC, 8 for HEVC and VP9).

Another approach is when SDK allocates memory for shared objects internally.

```

MFXVideoENCODE_Init(session, &init_param);
sts=MFX_ERR_MORE_DATA;
for (;;) {
    if (sts==MFX_ERR_MORE_DATA && !end_of_stream()) {
        MFXMemory_GetSurfaceForEncode(&surface);
        fill_content_for_encoding(surface);
    }
    surface2=end_of_stream() ?NULL:surface;
    sts=MFXVideoENCODE_EncodeFrameAsync(session,NULL,surface2,bits,&syncp);
    if (surface2) surface->FrameInterface->(*Release)(surface2);
    if (end_of_stream() && sts==MFX_ERR_MORE_DATA) break;
    // Skipped other error handling
    if (sts==MFX_ERR_NONE) {
        MFXVideoCORE_SyncOperation(session, syncp, INFINITE);
        do_something_with_encoded_bits(bits);
    }
}
MFXVideoENCODE_Close();

```

There are several key points which are different from legacy mode:

- The application doesn't need to call MFXVideoENCODE_QueryIOSurf function to obtain the number of working frame surfaces since allocation is done by SDK
- The application calls the MFXMemory_GetSurfaceForEncode function to get free surface for the following encode operation.
- The application needs to call the FrameInterface->(*Release) function to decrement reference counter of the obtained surface after MFXVideoENCODE_EncodeFrameAsync call.

12.5.5.2 Configuration Change

The application changes configuration during encoding by calling MFXVideoENCODE_Reset function. Depending on difference in configuration parameters before and after change, the SDK encoder either continues current sequence or starts a new one. If the SDK encoder starts a new sequence it completely resets internal state and begins a new sequence with IDR frame.

The application controls encoder behavior during parameter change by attaching mfxExtEncoderResetOption to mfxVideoParam structure during reset. By using this structure, the application instructs encoder to start or not to start a new sequence after reset. In some cases request to continue current sequence cannot be satisfied and encoder fails during reset. To avoid such cases the application may query reset outcome before actual reset by calling MFXVideoENCODE_Query function with mfxExtEncoderResetOption attached to mfxVideoParam structure.

The application uses the following procedure to change encoding configurations:

- 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 MFX_ERR_MORE_DATA.

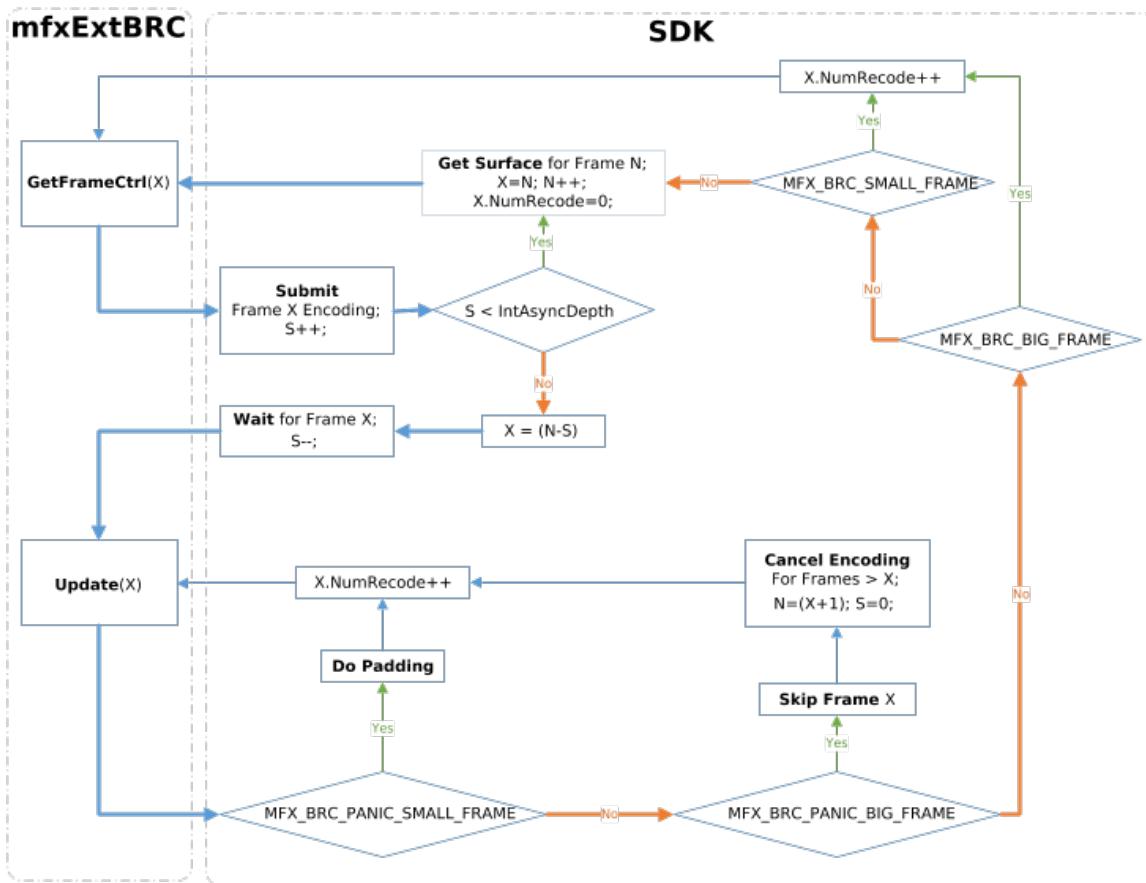
Note: The application must set the initial encoding configuration flag EndOfStream of the mfxExtCodingOption structure to OFF to avoid inserting an End of Stream (EOS) marker into the bitstream. An EOS marker causes the bitstream to terminate before encoding is complete.

- The application calls the MFXVideoENCODE_Reset function with the new configuration:
 - If the function successfully set the configuration, the application can continue encoding as usual.

- If the new configuration requires a new memory allocation, the function returns MFX_ERR_INCOMPATIBLE_VIDEO_PARAM. The application must close the SDK encoder and reinitialize the encoding procedure with the new configuration.

12.5.5.3 External Bit Rate Control

The application can make encoder use external BRC instead of native one. In order to do that it should attach to mfxVideoParam structure mfxExtCodingOption2 with ExtBRC = MFX_CODINGOPTION_ON and callback structure mfxExtBRC during encoder initialization. Callbacks Init, Reset and Close will be invoked inside MFXVideoENCODE_Init, MFXVideoENCODE_Reset and MFXVideoENCODE_Close correspondingly. Figure below illustrates asynchronous encoding flow with external BRC (usage of GetFrameCtrl and Update):



Note: **IntAsyncDepth** is the SDK max internal asynchronous encoding queue size; it is always less than or equal to mfxVideoParam::AsyncDepth.

External BRC Pseudo Code:

```

#include "mfxvideo.h"
#include "mfxbrc.h"

typedef struct {

```

(continues on next page)

(continued from previous page)

```

mfxU32 EncodedOrder;
mfxI32 QP;
mfxU32 MaxSize;
mfxU32 MinSize;
mfxU16 Status;
mfxU64 StartTime;
// ... skipped
} MyBrcFrame;

typedef struct {
    MyBrcFrame* frame_queue;
    mfxU32 frame_queue_size;
    mfxU32 frame_queue_max_size;
    mfxI32 max_qp[3]; //I,P,B
    mfxI32 min_qp[3]; //I,P,B
    // ... skipped
} MyBrcContext;

mfxStatus MyBrcInit(mfxHDL pthis, mfxVideoParam* par) {
    MyBrcContext* ctx = (MyBrcContext*)pthis;
    mfxI32 QpBdOffset;
    mfxExtCodingOption2* co2;

    if (!pthis || !par)
        return MFX_ERR_NULL_PTR;

    if (!IsParametersSupported(par))
        return MFX_ERR_UNSUPPORTED;

    frame_queue_max_size = par->AsyncDepth;
    frame_queue = (MyBrcFrame*)malloc(sizeof(MyBrcFrame) * frame_queue_max_size);

    if (!frame_queue)
        return MFX_ERR_MEMORY_ALLOC;

    co2 = (mfxExtCodingOption2*)GetExtBuffer(par->ExtParam, par->NumExtParam, MFX_
EXTBUFF_CODING_OPTION2);
    QpBdOffset = (par->BitDepthLuma > 8) : (6 * (par->BitDepthLuma - 8)) : 0;

    for (<X = I,P,B>) {
        ctx->max_qp[X] = (co2 && co2->MaxQPX) ? (co2->MaxQPX - QpBdOffset) : <Default>;
        ctx->min_qp[X] = (co2 && co2->MinQPX) ? (co2->MinQPX - QpBdOffset) : <Default>;
    }

    // skipped initialization of other other BRC parameters

    frame_queue_size = 0;

    return MFX_ERR_NONE;
}

mfxStatus MyBrcReset(mfxHDL pthis, mfxVideoParam* par) {
    MyBrcContext* ctx = (MyBrcContext*)pthis;

    if (!pthis || !par)
        return MFX_ERR_NULL_PTR;
}

```

(continues on next page)

(continued from previous page)

```

if (!IsParametersSupported(par))
    return MFX_ERR_UNSUPPORTED;

if (!IsResetPossible(ctx, par))
    return MFX_ERR_INCOMPATIBLE_VIDEO_PARAM;

// reset here BRC parameters if required

return MFX_ERR_NONE;
}

mfxStatus MyBrcClose(mfxHDL pthis) {
    MyBrcContext* ctx = (MyBrcContext*)pthis;

    if (!pthis)
        return MFX_ERR_NULL_PTR;

    if (ctx->frame_queue) {
        free(ctx->frame_queue);
        ctx->frame_queue = NULL;
        ctx->frame_queue_max_size = 0;
        ctx->frame_queue_size = 0;
    }

    return MFX_ERR_NONE;
}

mfxStatus MyBrcGetFrameCtrl(mfxHDL pthis, mfxBRCFrameParam* par, mfxBRCFrameCtrl* ctrl) {
    MyBrcContext* ctx = (MyBrcContext*)pthis;
    MyBrcFrame* frame = NULL;
    mfxU32 cost;

    if (!pthis || !par || !ctrl)
        return MFX_ERR_NULL_PTR;

    if (par->NumRecode > 0)
        frame = GetFrame(ctx->frame_queue, ctx->frame_queue_size, par->EncodedOrder);
    else if (ctx->frame_queue_size < ctx->frame_queue_max_size)
        frame = ctx->frame_queue[ctx->frame_queue_size++];

    if (!frame)
        return MFX_ERR_UNDEFINED_BEHAVIOR;

    if (par->NumRecode == 0) {
        frame->EncodedOrder = par->EncodedOrder;
        cost = GetFrameCost(par->FrameType, par->PyramidLayer);
        frame->MinSize = GetMinSize(ctx, cost);
        frame->MaxSize = GetMaxSize(ctx, cost);
        frame->QP = GetInitQP(ctx, frame->MinSize, frame->MaxSize, cost); // from QP/
size stat
        frame->StartTime = GetTime();
    }

    ctrl->QpY = frame->QP;

    return MFX_ERR_NONE;
}

```

(continues on next page)

(continued from previous page)

```

}

mfxStatus MyBrcUpdate(mfxHDL pthis, mfxBRCFParam* par, mfxBRCFctrl* ctrl, ↵
    ↵mfxBRCFstatus* status) {
    MyBrcContext* ctx = (MyBrcContext*)pthis;
    MyBrcFrame* frame = NULL;
    bool panic = false;

    if (!pthis || !par || !ctrl || !status)
        return MFX_ERR_NULL_PTR;

    frame = GetFrame(ctx->frame_queue, ctx->frame_queue_size, par->EncodedOrder);
    if (!frame)
        return MFX_ERR_UNDEFINED_BEHAVIOR;

    // update QP/size stat here

    if (frame->Status == MFX_BRC_PANIC_BIG_FRAME
        || frame->Status == MFX_BRC_PANIC_SMALL_FRAME_FRAME)
        panic = true;

    if (panic || (par->CodedFrameSize >= frame->MinSize && par->CodedFrameSize <= ↵
        ↵frame->MaxSize)) {
        UpdateBRCState(par->CodedFrameSize, ctx);
        RemoveFromQueue(ctx->frame_queue, ctx->frame_queue_size, frame);
        ctx->frame_queue_size--;
        status->BRCStatus = MFX_BRC_OK;

        // Here update Min/MaxSize for all queued frames

        return MFX_ERR_NONE;
    }

    panic = ((GetTime() - frame->StartTime) >= GetMaxFrameEncodingTime(ctx));

    if (par->CodedFrameSize > frame->MaxSize) {
        if (panic || (frame->QP >= ctx->max_qp[X])) {
            frame->Status = MFX_BRC_PANIC_BIG_FRAME;
        } else {
            frame->Status = MFX_BRC_BIG_FRAME;
            frame->QP = <increase QP>;
        }
    }

    if (par->CodedFrameSize < frame->MinSize) {
        if (panic || (frame->QP <= ctx->min_qp[X])) {
            frame->Status = MFX_BRC_PANIC_SMALL_FRAME;
            status->MinFrameSize = frame->MinSize;
        } else {
            frame->Status = MFX_BRC_SMALL_FRAME;
            frame->QP = <decrease QP>;
        }
    }

    status->BRCStatus = frame->Status;

    return MFX_ERR_NONE;
}

```

(continues on next page)

(continued from previous page)

```

}

// initialize encoder
MyBrcContext brc_ctx;
mfxExtBRC ext_brc;
mfxExtCodingOption2 co2;
mfxExtBuffer* ext_buf[2] = {&co2.Header, &ext_brc.Header};

memset(&brc_ctx, 0, sizeof(MyBrcContext));
memset(&ext_brc, 0, sizeof(mfxExtBRC));
memset(&co2, 0, sizeof(mfxExtCodingOption2));

vpar.ExtParam = ext_buf;
vpar.NumExtParam = sizeof(ext_buf) / sizeof(ext_buf[0]);

co2.Header.BufferId = MFX_EXTBUFF_CODING_OPTION2;
co2.Header.BufferSz = sizeof(mfxExtCodingOption2);
co2.ExtBRC = MFX_CODINGOPTION_ON;

ext_brc.Header.BufferId = MFX_EXTBUFF_BRC;
ext_brc.Header.BufferSz = sizeof(mfxExtBRC);
ext_brc.pthis = &brc_ctx;
ext_brc.Init = MyBrcInit;
ext_brc.Reset = MyBrcReset;
ext_brc.Close = MyBrcClose;
ext_brc.GetFrameCtrl = MyBrcGetFrameCtrl;
ext_brc.Update = MyBrcUpdate;

status = MFXVideoENCODE_Query(session, &vpar, &vpar);
if (status == MFX_ERR_UNSUPPORTED || co2.ExtBRC != MFX_CODINGOPTION_ON)
    // unsupported case
else
    status = MFXVideoENCODE_Init(session, &vpar);

```

12.5.5.4 JPEG

The application can use the same encoding procedures for JPEG/motion JPEG encoding, as illustrated by the 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 matrixes

- Interleaved and non-interleaved scans
- Single and multiple scans
 - chroma subsampling 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 channels images

The application may specify Huffman and quantization tables during encoder initialization by attaching `mfxExtJPEGQuantTables` and `mfxExtJPEGHuffmanTables` buffers to `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 quantization table it has to specify Quality parameter in `mfxInfoMF`X structure. In this case, the SDK encoder scales default quantization table according to specified Quality parameter.

The application should properly configured chroma sampling format and color format. FourCC and ChromaFormat fields in `mfxFrameInfo` structure are used for this. For example, to encode 4:2:2 vertically sampled YCbCr picture, the application should set FourCC to `MFX_FOURCC_YUY2` and ChromaFormat to `MFX_CHROMAFORMAT_YUV422V`. To encode 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 has to call `MFXVideoEncode_Query()` function to check if required color format is supported on given platform and then initialize encoder with proper values of FourCC and ChromaFormat in `mfxFrameInfo` structure.

The application should not define number of scans and number of components. They are derived by the SDK encoder from Interleaved flag in `mfxInfoMF`X structure and from chroma type. If interleaved coding is specified then one scan is encoded that contains all image components. Otherwise, number of scans is equal to number of components. The SDK encoder uses next component IDs - “1” for luma (Y), “2” for chroma Cb (U) and “3” for chroma Cr (V).

The application should allocate big enough buffer to hold encoded picture. Roughly, its upper limit may be calculated using next equation:

```
BufferSizeInKB = 4 + (Width * Height * BytesPerPx + 1023) / 1024;
```

where Width and Height are weight and height of the picture in pixel, BytesPerPx is number of byte for one pixel. It equals to 1 for 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).

12.5.5.5 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 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. The view dependencies follow the View dependency 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 mfxExtCodingOptionSP-SPPS structure, or configuring reference frame list via the mfxExtRefListCtrl structure.

During encoding, the SDK encoding function MFXVideoENCODE_EncodeFrameAsync accumulates input frames until encoding of a picture is possible. The function returns MFX_ERR_MORE_DATA for more data at input or MFX_ERR_NONE if having successfully accumulated enough data for encoding of a picture. The generated bitstream contains the complete picture (multiple views). The application can change this behavior and instruct encoder to output each view in a separate bitstream buffer. To do so the application has to turn on the ViewOutput flag in the mfxExtCodingOption structure. In this case, encoder returns MFX_ERR_MORE_BITSTREAM if it needs more bitstream buffers at output and MFX_ERR_NONE when processing of picture (multiple views) has been finished. It is recommended that the application provides a new input frame each time the SDK encoder requests new bitstream buffer. The application must submit views 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 pseudo code.

```
mfxExtBuffer *eb;
mfxExtMVCSeqDesc seq_desc;
mfxVideoParam init_param;

init_param.ExtParam=&eb;
init_param.NumExtParam=1;
eb=&seq_desc;

/* init encoder */
MFXVideoENCODE_Init(session, &init_param);

/* perform encoding */
for (;;) {
    MFXVideoENCODE_EncodeFrameAsync(session, NULL, surface2, bits,
                                    &syncp);
    MFXVideoCORE_SyncOperation(session, syncp, INFINITE);
}

/* close encoder */
MFXVideoENCODE_Close();
```

12.5.6 Video Processing Procedures

Example below shows the pseudo code of the video processing procedure.

```
MFXVideoVPP_QueryIOSurf(session, &init_param, response);
allocate_pool_of_surfaces(in_pool, response[0].NumFrameSuggested);
allocate_pool_of_surfaces(out_pool, response[1].NumFrameSuggested);
MFXVideoVPP_Init(session, &init_param);
in=find_unlocked_surface_and_fill_content(in_pool);
out=find_unlocked_surface_from_the_pool(out_pool);
```

(continues on next page)

(continued from previous page)

```

for (;;) {
    sts=MFXVideoVPP_RunFrameVPPAsync(session,in,out,aux,&syncp);
    if (sts==MFX_ERR_MORE_SURFACE || sts==MFX_ERR_NONE) {
        MFXVideoCore_SyncOperation(session, syncp, INFINITE);
        process_output_frame(out);
        out=find_unlocked_surface_from_the_pool(out_pool);
    }
    if (sts==MFX_ERR_MORE_DATA && in==NULL)
        break;
    if (sts==MFX_ERR_NONE || sts==MFX_ERR_MORE_DATA) {
        in=find_unlocked_surface(in_pool);
        fill_content_for_video_processing(in);
        if (end_of_input_sequence())
            in=NULL;
    }
}
MFXVideoVPP_Close(session);
free_pool_of_surfaces(in_pool);
free_pool_of_surfaces(out_pool);

```

The following describes a few key points:

- 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 the other for the output.
- The video processing function `MFXVideoVPP_RunFrameVPPAsync` is asynchronous. The application must synchronize to make the output result ready, through the `MFXVideoCORE_SyncOperation` function.
- The body of the video processing procedures covers three scenarios as follows:
 - If the number of frames consumed at input is equal to the number of frames generated at output, **VPP** returns `MFX_ERR_NONE` when an output is ready. The application must process the output frame after synchronization, as the `MFXVideoVPP_RunFrameVPPAsync` function is asynchronous. At the end of a sequence, the application must provide a NULL input 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 `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 and provide a NULL input at the end of 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 `MFX_ERR_MORE_SURFACE` (when more than one output is ready), or `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 sequence to drain any remaining frames.

12.5.6.1 Configuration

The SDK configures the video processing pipeline operation based on the difference between the input and output formats, specified in the `mfxVideoParam` structure. A few examples follow:

- 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 de-interlacing.
- 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 pipeline by using `mfxExtVPPDoNotUse` structure; enable them by using `mfxExtVPPDoUse` structure and configure them by using dedicated configuration structures. See Table 4 for 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 input format to output one are included in pipeline. However, the SDK can skip some optional filters even if they are explicitly requested by the application, for example, due to limitation of underlying hardware. To notify application about this skip, the SDK returns warning `MFX_WRN_FILTER_SKIPPED`. 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.

Configurable VPP filters:

Filter ID	Configuration structure
<code>MFX_EXTBUFF_VPP_DENOISE</code>	<code>mfxExtVPPDenoise</code>
<code>MFX_EXTBUFF_VPP_MCTF</code>	<code>mfxExtVppMctf</code>
<code>MFX_EXTBUFF_VPP_DETAIL</code>	<code>mfxExtVPPDetail</code>
<code>MFX_EXTBUFF_VPP_FRAME_RATE_CONVERSION</code>	<code>mfxExtVPPFrameRateConversion</code>
<code>MFX_EXTBUFF_VPP_IMAGE_STABILIZATION</code>	<code>mfxExtVPPIImageStab</code>
<code>MFX_EXTBUFF_VPP_PICSTRUCT_DETECTION</code>	<code>none</code>
<code>MFX_EXTBUFF_VPP_PROCAMP</code>	<code>mfxExtVPPProcAmp</code>
<code>MFX_EXTBUFF_VPP_FIELD_PROCESSING</code>	<code>mfxExtVPPFieldProcessing</code>

Example of Video Processing configuration:

```
/* enable image stabilization filter with default settings */
mfxExtVPPDoUse du;
mfxU32 al=MFX_EXTBUFF_VPP_IMAGE_STABILIZATION;

du.Header.BufferId=MFX_EXTBUFF_VPP_DOUSE;
du.Header.BufferSz=sizeof(mfxExtVPPDoUse);
du.NumAlg=1;
du.AlgList=&al;

/* configure the mfxVideoParam structure */
mfxVideoParam conf;
mfxExtBuffer *eb=&du;

memset(&conf,0,sizeof(conf));
conf.IOPattern=MFX_IOPATTERN_IN_SYSTEM_MEMORY | MFX_IOPATTERN_OUT_SYSTEM_MEMORY;
conf.NumExtParam=1;
conf.ExtParam=&eb;

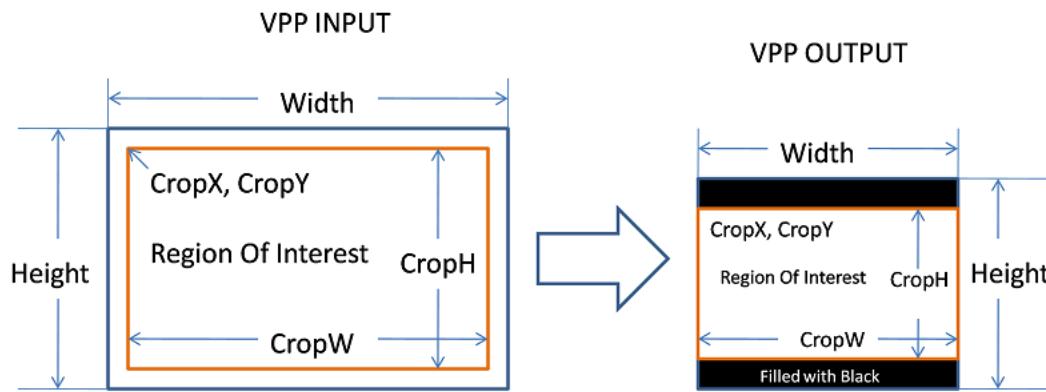
conf.vpp.In.FourCC=MFX_FOURCC_YV12;
conf.vpp.Out.FourCC=MFX_FOURCC_NV12;
conf.vpp.In.Width=conf.vpp.Out.Width=1920;
conf.vpp.In.Height=conf.vpp.Out.Height=1088;

/* video processing initialization */
MFXVideoVPP_Init(session, &conf);
```

12.5.6.2 Region of Interest

During video processing operations, the application can specify a region of interest for each frame, as illustrated below:

VPP Region of Interest Operation:



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.

Examples of VPP Operations on Region of Interest:

Operation	VPP Input Width/Height	VPP Input CropX, CropY, CropW, CropH	VPP Output Width/Height	VPP Output CropX, CropY, CropW, CropH
Cropping	720x480	16,16,688,448	720x480	16,16,688,448
Resizing	720x480	0,0,720,480	1440x960	0,0,1440,960
Horizontal stretching	720x480	0,0,720,480	640x480	0,0,640,480
16:9 4:3 with letter boxing at the top and bottom	1920x1088	0,0,1920,1088	720x480	0,36,720,408
4:3 16:9 with pillar boxing at the left and right	720x480	0,0,720,480	1920x1088	144,0,1632,1088

12.5.6.3 Multi-view video processing

The 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 independently, one view at a time. The SDK refers to the FrameID field of the mfxFrameInfo structure to configure each view according to its processing pipeline. The application needs to fill the the FrameID field before calling the MFXVideoVPP_RunFrameVPPAsync function, if the video processing source frame is not the output from the SDK MVC decoder. The following pseudo code illustrates it:

```
mfxExtBuffer *eb;
mfxExtMVCSeqDesc seq_desc;
mfxVideoParam init_param;

init_param.ExtParam = &eb;
```

(continues on next page)

(continued from previous page)

```

init_param.NumExtParam=1;
eb=&seq_desc;

/* init VPP */
MFXVideoVPP_Init(session, &init_param);

/* perform processing */
for (;;) {
    MFXVideoVPP_RunFrameVPPAsync(session,in,out,aux,&syncp);
    MFXVideoCORE_SyncOperation(session, syncp, INFINITE);
}

/* close VPP */
MFXVideoVPP_Close(session);

```

12.5.7 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.

12.5.7.1 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. Such pipeline construction is done at runtime and can be dynamically changed, as illustrated below:

```

mfxSyncPoint sp_d, sp_e;
MFXVideoDECODE_DecodeFrameAsync(session,bs,work,&vin, &sp_d);
if (going_through_vpp) {
    MFXVideoVPP_RunFrameVPPAsync(session,vin,vout, &sp_d);
    MFXVideoENCODE_EncodeFrameAsync(session,NULL,vout,bits2,&sp_e);
} else {
    MFXVideoENCODE_EncodeFrameAsync(session,NULL,vin,bits2,&sp_e);
}
MFXVideoCORE_SyncOperation(session,sp_e,INFINITE);

```

The SDK simplifies the requirement for asynchronous pipeline synchronization. The application only needs to synchronize after the last SDK function. Explicit synchronization of intermediate results is not required and in fact can slow performance.

The SDK tracks the dynamic pipeline construction and verifies dependency on input and output parameters to ensure the execution order of the pipeline function. In Example 6, 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 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 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: Note that the dependency check works on the pointers to the structures only.

There are two exceptions with respect to intermediate synchronization:

- The application must synchronize any input before calling the SDK function `MFXVideoDECODE_DecodeFrameAsync`, if the input is from any asynchronous operation.
- 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.

Pseudo Code of Asynchronous **ENC->**ENCODE**** Pipeline Construction:

```
mfxENCInput enc_in = ...;
mfxENCOOutput enc_out = ...;
mfxSyncPoint sp_e, sp_n;
mfxFrameSurface1* surface = get_frame_to_encode();
mfxExtBuffer dependency;
dependency.BufferId = MFX_EXTBUFF_TASK_DEPENDENCY;
dependency.BufferSz = sizeof(mfxExtBuffer);

enc_in.InSurface = surface;
enc_out.ExtParam[enc_out.NumExtParam++] = &dependency;
MFXVideoENC_ProcessFrameAsync(session, &enc_in, &enc_out, &sp_e);

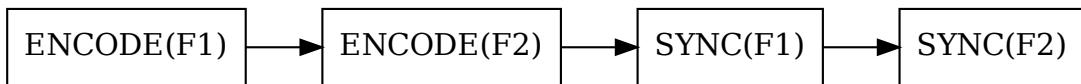
surface->Data.ExtParam[surface->Data.NumExtParam++] = &dependency;
MFXVideoENCODE_EncodeFrameAsync(session, NULL, surface, &bs, &sp_n);

MFXVideoCORE_SyncOperation(session, sp_n, INFINITE);
surface->Data.NumExtParam--;
```

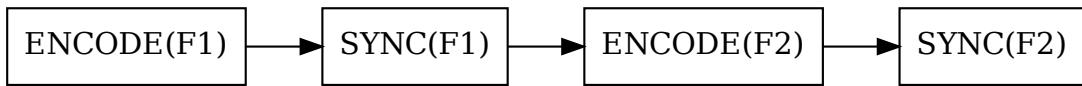
12.5.7.2 Surface Pool Allocation

When connecting SDK function **A** to SDK function **B**, the application must take into account the needs 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 needs from SDK function **A** output, and **Nb** is the frame surface needs from SDK function **B** input.

For performance considerations, the application must submit multiple operations and delays 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 case, 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 that how many asynchronous operations the application plans to perform before synchronization. The corresponding SDK **QueryIOSurf** function will reflect such consideration in the NumFrameSuggested value. Example below shows a way of calculating the surface needs based on NumFrameSuggested values:

```

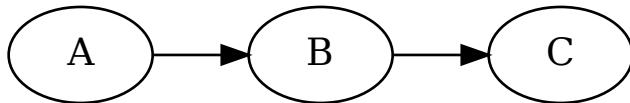
async_depth=4;
init_param_v.AsyncDepth=async_depth;
MFXVideoVPP_QueryIOSurf(session, &init_param_v, response_v);
init_param_e.AsyncDepth=async_depth;
MFXVideoENCODE_QueryIOSurf(session, &init_param_e, &response_e);
num_surfaces=    response_v[1].NumFrameSuggested
                +response_e.NumFrameSuggested
                -async_depth; /* double counted in ENCODE & VPP */

```

12.5.7.3 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.

Assume the pipeline is:



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 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 MFX_ERR_ABORTED. Same logic applies if the error occurs in SDK function **A**.

12.5.8 Working with hardware acceleration

12.5.8.1 Working with multiple Intel media devices

If your system has multiple Intel Gen Graphics adapters you may need hints on which adapter suits better to process some particular workload. The SDK provides helper API to select best suitable adapter for your workload based on passed workload description. Example below showcases workload initialization on discrete adapter:

```
mfxU32 num_adapters_available;

// Query number of Intel Gen Graphics adapters available on system
mfxStatus sts = MFXQueryAdaptersNumber(&num_adapters_available);
MSDK_CHECK_STATUS(sts, "MFXQueryAdaptersNumber failed");

// Allocate memory for response
std::vector<mfxAdapterInfo> displays_data(num_adapters_available);
mfxAdaptersInfo adapters = { displays_data.data(), mfxU32(displays_data.size()), 0u };

// Query information about all adapters (mind that first parameter is NULL)
sts = MFXQueryAdapters(nullptr, &adapters);
MSDK_CHECK_STATUS(sts, "MFXQueryAdapters failed");

// Find dGfx adapter in list of adapters
auto idx_d = std::find_if(adapters.Adapters, adapters.Adapters + adapters.NumActual,
    [] (const mfxAdapterInfo info)
{
    return info.Platform.MediaAdapterType == mfxMediaAdapterType::MFX_MEDIA_DISCRETE;
});

// No dGfx in list
if (idx_d == adapters.Adapters + adapters.NumActual)
{
    printf("Warning: No dGfx detected on machine\n");
    return -1;
}

mfxU32 idx = static_cast<mfxU32>(std::distance(adapters.Adapters, idx));

// Choose correct implementation for discrete adapter
switch (adapters.Adapters[idx].Number)
{
case 0:
    impl = MFX_IMPL_HARDWARE;
    break;
case 1:
    impl = MFX_IMPL_HARDWARE2;
    break;
case 2:
    impl = MFX_IMPL_HARDWARE3;
    break;
case 3:
    impl = MFX_IMPL_HARDWARE4;
    break;

default:
    // Try searching on all display adapters
    impl = MFX_IMPL_HARDWARE_ANY;
```

(continues on next page)

(continued from previous page)

```

break;
}

// Initialize mfxSession in regular way with obtained implementation

```

As you see in this example, after obtaining adapter list with MFXQueryAdapters further initialization of mfxSession is performed in regular way. Particular adapter choice is performed with `MFX_IMPL_HARDWARE`,..., `MFX_IMPL_HARDWARE4` values of `mfxIMPL`.

Example below showcases usage of MFXQueryAdapters for querying best suitable adapter for particular encode workload (see MFXQueryAdapters description for adapter priority rules):

```

mfxU32 num_adapters_available;

// Query number of Intel Gen Graphics adapters available on system
mfxStatus sts = MFXQueryAdaptersNumber(&num_adapters_available);
MSDK_CHECK_STATUS(sts, "MFXQueryAdaptersNumber failed");

// Allocate memory for response
std::vector<mfxAdapterInfo> displays_data(num_adapters_available);
mfxAdaptersInfo adapters = { displays_data.data(), mfxU32(displays_data.size()), 0 };

// Fill description of Encode workload
mfxComponentInfo interface_request = { MFX_COMPONENT_ENCODE, Encode_mfxVideoParam };

// Query information about suitable adapters for Encode workload described by Encode_
→mfxVideoParam
sts = MFXQueryAdapters(&interface_request, &adapters);

if (sts == MFX_ERR_NOT_FOUND)
{
    printf("Error: No adapters on machine capable to process desired workload\n");
    return -1;
}

MSDK_CHECK_STATUS(sts, "MFXQueryAdapters failed");

// Choose correct implementation for discrete adapter. Mind usage of index 0, this is
→best suitable adapter from MSDK perspective
switch (adapters.Adapters[0].Number)
{
case 0:
    impl = MFX_IMPL_HARDWARE;
    break;
case 1:
    impl = MFX_IMPL_HARDWARE2;
    break;
case 2:
    impl = MFX_IMPL_HARDWARE3;
    break;
case 3:
    impl = MFX_IMPL_HARDWARE4;
    break;

default:
    // Try searching on all display adapters
    impl = MFX_IMPL_HARDWARE_ANY;
}

```

(continues on next page)

(continued from previous page)

```

break;
}

// Initialize mfxSession in regular way with obtained implementation

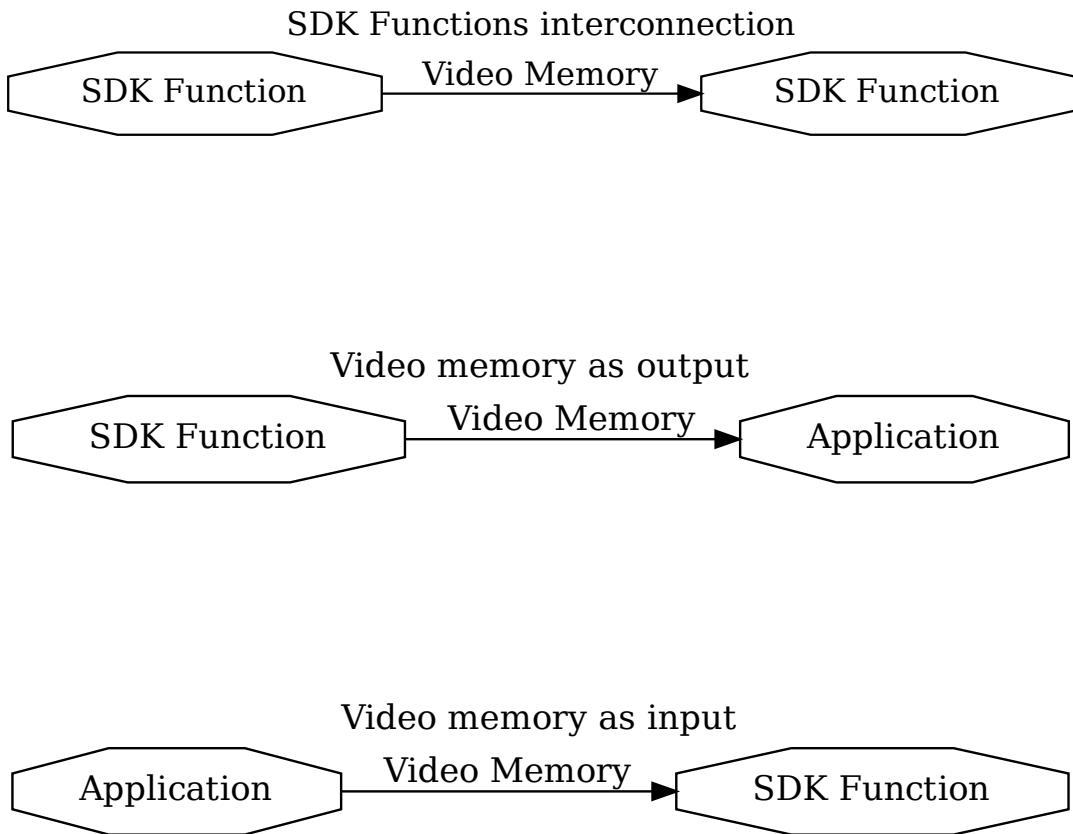
```

12.5.8.2 Working with video memory

To fully utilize the SDK acceleration capability, the application should support OS specific infrastructures, Microsoft* DirectX* for Microsoft* Windows* and VA API for Linux*.

The hardware acceleration support in application consists of video memory support and acceleration device support.

Depending on usage model, the application can use video memory on different stages of pipeline. Three major scenarios are illustrated below:



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 SDK component to query HW capabilities. The application can share its device with the SDK by passing device handle through the MFXVideoCORE_SetHandle function. It is recommended to share the handle before any actual usage of the SDK.

12.5.8.3 Working with Microsoft* DirectX* Applications

The SDK supports two different infrastructures for hardware acceleration on Microsoft* Windows* OS, “Direct3D 9 DXVA2” and “Direct3D 11 Video API”. In the first one the application should use the IDirect3DDeviceManager9 interface as the acceleration device handle, in the second one - ID3D11Device interface. The application should share one of these interfaces with the SDK through the MFXVideoCORE_SetHandle function. If the application does not provide it, then the SDK creates its own internal acceleration device. This internal device could not be accessed by the application and as a result, the SDK input and output will be limited to system memory only. That in turn will reduce SDK performance. If the SDK fails to create a valid acceleration device, then SDK cannot proceed with hardware acceleration and returns an error status to the application.

The application must create the Direct3D9* device with the flag **D3DCREATE_MULTITHREADED**. Additionally the flag **D3DCREATE_FPU_PRESERVE** is recommended. This influences floating-point calculations, including PTS values.

The application must also set multithreading mode for Direct3D11* device. Example below illustrates how to do it:

```
ID3D11Device          *pD11Device;
ID3D11DeviceContext  *pD11Context;
ID3D10Multithread    *pD10Multithread;

pD11Device->GetImmediateContext (&pD11Context);
pD11Context->QueryInterface(IID_ID3D10Multithread, &pD10Multithread);
pD10Multithread->SetMultithreadProtected(true);
```

During hardware acceleration, if a Direct3D* “device lost” event occurs, the SDK operation terminates with the return status **MFX_ERR_DEVICE_LOST**. 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 decoder Init function requests 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, SDK requires different surface types. It is strongly recommended to call one of the MFXVideoENCODE_QueryIOSurf, MFXVideoDECODE_QueryIOSurf or MFXVideoVPP_QueryIOSurf functions to determine the appropriate type.

Supported SDK Surface Types and Color Formats for Direct3D9:

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: “Decoder Render Target” corresponds to **DXVA2_VideoDecoderRenderTarget** type.

Note: “Processor Render Target” corresponds to **DXVA2_VideoProcessorRenderTarget**.

Supported SDK Surface Types and Color Formats for Direct3D11:

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: “Decoder Render Target” corresponds to **D3D11_BIND_DECODER** flag.

Note: “Processor Render Target” corresponds to **D3D11_BIND_RENDER_TARGET**.

Note: that NV12 is the major encoding and decoding color format.

Note: Additionally, JPEG/MJPEG decoder supports RGB32 and YUY2 output.

Note: JPEG/MJPEG encoder supports RGB32 and YUY2 input for Direct3D9/Direct3D11 and YV12 input for

Direct3D9 only.

Note: VPP supports RGB32 output.

12.5.8.4 Working with VA API Applications

The SDK supports single infrastructure for hardware acceleration on Linux* - “VA API”. 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 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.

Obtaining VA display from X Window System:

```
Display *x11_display;
VADisplay va_display;

x11_display = XOpenDisplay(current_display);
va_display = vaGetDisplay(x11_display);

MFXVideoCORE_SetHandle(session, MFX_HANDLE_VA_DISPLAY, (mfxHDL) va_display);
```

Obtaining VA display from Direct Rendering Manager:

```
int card;
VADisplay va_display;

card = open("/dev/dri/card0", O_RDWR); /* primary card */
va_display = vaGetDisplayDRM(card);
vaInitialize(va_display, &major_version, &minor_version);

MFXVideoCORE_SetHandle(session, MFX_HANDLE_VA_DISPLAY, (mfxHDL) va_display);
```

When the SDK decoder creates 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.

Todo: Add link to “Allocation and External Allocators”

Only 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.

Supported SDK Surface Types and Color Formats for VA API:

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	

12.5.8.5 Memory Allocation and External Allocators

There are two models of memory management in SDK implementations: internal and external.

12.5.8.6 External memory management

In external memory model the application must allocate sufficient memory for input and output parameters and buffers, and de-allocate 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 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 the section [Working with hardware acceleration](#) for additional details.

The external frame allocator responds only to frame allocation requests for the requested memory type and returns `MFX_ERR_UNSUPPORTED` for all others. The allocation request uses flags, part of memory type field, to indicate which SDK class initiates the request, so the external frame allocator can respond accordingly.

Simple external frame allocator:

```

typedef struct {
    mfxU16 width, height;
    mfxU8 *base;
} mid_struct;

mfxStatus fa_alloc(mfxHDL pthis, mfxFrameAllocRequest *request, mfxFrameAllocResponse*
                    response) {
    if (! (request->type&MFX_MEMTYPE_SYSTEM_MEMORY))
        return MFX_ERR_UNSUPPORTED;
    if (request->Info->FourCC!=MFX_FOURCC_NV12)
        return MFX_ERR_UNSUPPORTED;
    response->NumFrameActual=request->NumFrameMin;
    for (int i=0;i<request->NumFrameMin;i++) {
        mid_struct *mmid=(mid_struct *)malloc(sizeof(mid_struct));
        mmid->width=ALIGN32(request->Info->Width);
        mmid->height=ALIGN32(request->Info->Height);
        mmid->base=(mfxU8*)malloc(mmid->width*mmid->height*3/2);
        response->mids[i]=mmid;
    }
}

```

(continues on next page)

(continued from previous page)

```

    return MFX_ERR_NONE;
}

mfxStatus fa_lock(mfxHDL pthis, mfxMemId mid, mfxFrameData *ptr) {
    mid_struct *mmid=(mid_struct *)mid;
    ptr->pitch=mmid->width;
    ptr->Y=mmid->base;
    ptr->U=ptr->Y+mmid->width*mmid->height;
    ptr->V=ptr->U+1;
    return MFX_ERR_NONE;
}

mfxStatus fa_unlock(mfxHDL pthis, mfxMemId mid, mfxFrameData *ptr) {
    if (ptr) ptr->Y=ptr->U=ptr->V=ptr->A=0;
    return MFX_ERR_NONE;
}

mfxStatus fa_gethdl(mfxHDL pthis, mfxMemId mid, mfxHDL *handle) {
    return MFX_ERR_UNSUPPORTED;
}

mfxStatus fa_free(mfxHDL pthis, mfxFrameAllocResponse *response) {
    for (int i=0;i<response->NumFrameActual;i++) {
        mid_struct *mmid=(mid_struct *)response->mids[i];
        free(mmid->base); free(mid);
    }
    return MFX_ERR_NONE;
}

```

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).

12.5.8.7 Internal memory management

In the internal memory management model SDK provides interface functions for frames allocation:

MFXMemory_GetSurfaceForVPP ()

MFXMemory_GetSurfaceForEncode ()

MFXMemory_GetSurfaceForDecode ()

which are used together with *mfxFrameSurfaceInterface* for surface management. The surface returned by these function is reference counted object and the application has to call *mfxFrameSurfaceInterface::Release* after finishing all operations with the surface. In this model the application doesn't need to create and set external allocator to SDK. Another possibility to obtain internally allocated surface is to call *MFXVideoDECODE_DecodeFrameAsync ()* with working surface equal to NULL (see *Simplified decoding procedure*). In such situation Decoder will allocate new refcountable *mfxFrameSurface1* and return to the user. All assumed contracts with user are similar with such in functions *MFXMemory_GetSurfaceForXXX*.

12.5.8.8 mfxFrameSurfaceInterface

Starting from API version 2.0 SDK support `mfxFrameSurfaceInterface`. This interface is a set of callback functions to manage lifetime of allocated surfaces, get access to pixel data, and obtain native handles and device abstractions (if suitable). It's recommended to use `mfxFrameSurface1::mfxFrameSurfaceInterface` if presents instead of directly accessing `mfxFrameSurface1` structure members or call external allocator callback functions if set.

The following example demonstrates the usage of `mfxFrameSurfaceInterface` for memory sharing:

```
// let decode frame and try to access output optimal way.
sts = MFXVideoDECODE_DecodeFrameAsync(session, NULL, NULL, &outsurface, &syncp);
if (MFX_ERR_NONE == sts)
{
    outsurface->FrameInterface->(*GetDeviceHandle)(outsurface, &device_handle, &
    ~device_type);
    // if application or component is familiar with mfxHandleType and it's possible to
    // share memory created by device_handle.
    if (isDeviceTypeCompatible(device_type) && isPossibleForMemorySharing(device_
    ~handle)) {
        // get native handle and type
        outsurface->FrameInterface->(*GetNativeHandle)(outsurface, &resource, &
        ~resource_type);
        if (isResourceTypeCompatible(resource_type)) {
            //use memory directly
            ProcessNativeMemory(resource);
            outsurface->FrameInterface->(*Release)(outsurface);
        }
    }
    // Application or component is not aware about such DeviceHandle or Resource type,
    // need to map to system memory.
    outsurface->FrameInterface->(*Map)(outsurface, MFX_MAP_READ);
    ProcessSystemMemory(outsurface);
    outsurface->FrameInterface->(*Unmap)(outsurface);
    outsurface->FrameInterface->(*Release)(outsurface);
}
```

12.5.8.9 Hardware Device Error Handling

The SDK accelerates decoding, encoding and video processing through a hardware device. The SDK functions may return the following errors or warnings if the hardware device encounters errors:

Status	Description
MFX_ERR_DEVICE_FAILURE	Hardware device returned unexpected errors. SDK was unable to restore operation.
MFX_ERR_DEVICE_LOST	Hardware device was lost due to system lock or shutdown.
MFX_WRN_PARTIAL_ACCELERATION	The hardware does not fully support the specified configuration. The encoding, decoding, or video processing operation may be partially accelerated.
MFX_WRN_DEVICE_BUSY	Hardware device is currently busy.

SDK functions **Query**, **QueryIOSurf**, and **Init** return `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 `MFX_WRN_PARTIAL_ACCELERATION`, as it is a lower priority warning.)

SDK functions return `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. Example below shows the decoding pseudo-code. The same procedure applies to encoding and video processing.

SDK functions return MFX_ERR_DEVICE_LOST or 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.

Pseudo-Code to Handle MFX_WRN_DEVICE_BUSY:

```
mfxStatus sts=MFX_ERR_NONE;
for (;;) {
    // do something
    sts=MFXVideoDECODE_DecodeFrameAsync(session, bitstream, surface_work, &surface_
→disp, &syncp);
    if (sts == MFX_WRN_DEVICE_BUSY) Sleep(5);
}
```

12.6 Summary Tables

12.6.1 Mandatory API reference

12.6.1.1 Functions per API Version

This is list of functions exported by any implementation with corresponding 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
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

continues on next page

Table 2 – continued from previous page

Function	API Version
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
MFXQueryImplDescription	2.0
MFXReleaseImplDescription	2.0

12.7 Appendices

12.7.1 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 following constraints apply:

Constraints common for **DECODE**, **ENCODE** and **VPP**:

Pa-ram-eters	During SDK ini-tialization	During SDK operation
FourCC	Any valid value	The value must be the same as the initialization value. The only exception is VPP 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.

Constraints for **DECODE**:

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	DECODE 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.	DECODE output.
FrameRate-ExtN FrameRate-ExtD	If unspecified, values from the input bitstream will be used; see note below the table.	DECODE output.
PicStruct	Ignored	DECODE output.

Note: Note about priority of initialization parameters.

Note: If application explicitly sets FrameRateExtN/FrameRateExtD or AspectRatioW/AspectRatioH during initialization then decoder uses these values during decoding regardless of values from bitstream and does not update them on new SPS. If application sets them to 0, then decoder uses values from stream and update them on each SPS.

Constraints for **VPP**:

Pa- ram-e- ters	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.
As- pec- tRa- tioW As- pec- tRa- tioH	Ignored	Aspect ratio values will be passed through from input to output.
Fram- eRate- ExtN Fram- eRate- ExtD	Any valid values	Frame rate values will be updated with the initialization value at output.
Pic- Struct	<code>MFX_PICSTRUCT_UNKNOWN</code> <code>MFX_PICSTRUCT_PROGRESSIVE</code> <code>MFX_PICSTRUCT_FIELD_TFF</code> <code>MFX_PICSTRUCT_FIELD_BFF</code> <code>MFX_PICSTRUCT_FIELD_SINGLE</code> <code>MFX_PICSTRUCT_FIELD_TOP</code> <code>MFX_PICSTRUCT_FIELD_BOTTOM</code>	<p>The base value must be the same as the initialization value unless <code>MFX_PICSTRUCT_UNKNOWN</code> is specified during initialization.</p> <p>Other decorative picture structure flags are passed through or added as needed. See the <i>PicStruct</i> enumerator for details.</p>

Constraints for **ENCODE**:

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	<i>MFX_PICSTRUCT_UNKNOWN</i> <i>MFX_PICSTRUCT_PROGRESSIVE</i> <i>MFX_PICSTRUCT_FIELD_TF</i> <i>MFX_PICSTRUCT_FIELD_BF</i>	The base value must be the same as the initialization value unless <i>MFX_PICSTRUCT_UNKNOWN</i> is specified during initialization. Add other decorative picture structure flags to indicate additional display attributes. Use <i>MFX_PICSTRUCT_UNKNOWN</i> during initialization for field attributes and <i>MFX_PICSTRUCT_PROGRESSIVE</i> for frame attributes. See the <i>PicStruct</i> enumerator for details.

The following table summarizes how to specify the configuration parameters during initialization and during encoding, decoding and video processing:

Structure (param)	ENCODE Init	ENCODE Encoding	DECODE Init	DECODE Decoding	VPP Init	VPP Processing
<i>mfxVideoParam</i>						
Protected	R	•	R	•	R	•
IOPattern	M	•	M	•	M	•

continues on next page

Table 3 – continued from previous page

Structure (param)	ENCODE Init	ENCODE Encoding	DECODE Init	DECODE Decoding	VPP Init	VPP Processing
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	O	•	•	•	•	•
InitialDelayInKB	O	•	•	•	•	•
BufferSizeInKB	O	•	•	•	•	•
TargetKbps	M	•	•	•	•	•
MaxKbps	O	•	•	•	•	•

continues on next page

Table 3 – continued from previous page

Structure (param)	ENCODE Init	ENCODE Encoding	DECODE Init	DECODE Decoding	VPP Init	VPP Processing
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

Table Legend:

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 application doesn't explicitly set *CodecProfile* during initialization, HEVC decoder will use profile up to Main10.

12.7.2 Multiple-Segment Encoding

Multiple-segment encoding is useful in video editing applications when during production; 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:

Segment already Encoded	Segment in encoding	Segment to be encoded
0s	200s	500s

Note: Note that 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 the 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). Nevertheless, 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. Table below 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.

Function Name	Error Code if Import Fails
MFXVideoENCODE_Init()	MFX_ERR_INCOMPATIBLE_VIDEO_PARAM
MFXVideoENCODE_QueryIOSurf()	MFX_ERR_INCOMPATIBLE_VIDEO_PARAM
MFXVideoENCODE_Reset()	MFX_ERR_INCOMPATIBLE_VIDEO_PARAM
MFXVideoENCODE_Query()	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 it anywhere in the final bitstream. After encoding, each encoded segment is HRD compliant. However, the concatenated segments may not be HRD compliant.

Example below shows an example of the encoder initialization procedure that imports H.264 sequence and picture parameter sets:

```
mfxStatus init_encoder() {
    mfxExtCodingOptionSPSPPS option, *option_array;

    /* configure mfxExtCodingOptionSPSPPS */
    memset(&option, 0, sizeof(option));
    option.Header.BufferId=MFX_EXTBUFF_CODING_OPTION_SPSPPS;
    option.Header.BufferSz=sizeof(option);
    option.SPSBuffer=sps_buffer;
    option.SPSBufSize=sps_buffer_length;
    option.PPSBuffer=pps_buffer;
```

(continues on next page)

(continued from previous page)

```

option.PPSBufSize=pps_buffer_length;

/* configure mfxVideoParam */
mfxVideoParam param;
//...
param.NumExtParam=1;
option_array=&option;
param.ExtParam=&option_array;

/* encoder initialization */
mfxStatus status;
status=MFXVideoENCODE_Init(session, &param);
if (status==MFX_ERR_INCOMPATIBLE_VIDEO_PARAM) {
    printf("Initialization failed\n");
} else {
    printf("Initialized\n");
}
return status;
}

```

12.7.3 Streaming and Video Conferencing Features

The following sections address a few aspects of additional requirements that streaming or video conferencing applications may use in the encoding or transcoding process. See also [Configuration Change](#) chapter.

12.7.3.1 Dynamic Bitrate Change

The SDK encoder supports dynamic bitrate change differently depending on bitrate control mode and HRD conformance requirement. If HRD conformance is required, i.e. if application sets **NalHrdConformance** option in [*mfxExtCodingOption*](#) structure to **ON**, the only allowed bitrate control mode is VBR. In this mode, the application can change **TargetKbps** and **MaxKbps** values. The application can change these values by calling the [*MFXVideoENCODE_Reset\(\)*](#) function. Such change in bitrate usually results in generation of a new key-frame and sequence header. There are some exceptions though. For example, if HRD Information is absent in the stream then change of **TargetKbps** does not require change of sequence header and as a result the SDK encoder does not insert a key frame.

If HRD conformance is not required, i.e. if application turns off **NalHrdConformance** option in [*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. Such change in bitrate will not result in generation of a new key-frame 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. That is why it is strongly recommended to retrieve the actual working parameters by [*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 soon the actual bitrate can catch up with the specified bitrate is implementation dependent.

Alternatively, the application may use the CQP (constant quantization parameter) 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.

12.7.3.2 Dynamic Resolution Change

The SDK encoder supports dynamic resolution change in all bitrate control modes. The application may change resolution by calling `MFXVideoENCODE_Reset()` function. The application may decrease or increase resolution up to the size specified during encoder initialization.

Resolution change always results in insertion of key IDR frame and new sequence parameter set header. The only exception is SDK VP9 encoder (see section for *Dynamic reference frame scaling*). The SDK encoder does not guarantee HRD conformance across 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. That is why it is strongly recommended to retrieve the actual working parameters set by `MFXVideoENCODE_GetVideoParam()` function before making any resolution change.

12.7.3.3 Dynamic reference frame scaling

VP9 standard allows to change resolution without insertion of key-frame. It's possible because of native built-in capability of VP9 decoder to upscale and downscale reference frames to match resolution of frame which is being encoded. By default SDK VP9 encoder inserts key-frame when application does *Dynamic Resolution Change*. In this case first frame with new resolution is encoded using Inter prediction from scaled reference frame of previous resolution. Dynamic scaling has following limitation coming from VP9 specification: resolution of any active reference frame cannot exceed 2x resolution of current frame, and can't be smaller than 1/16 of current frame resolution. In case of dynamic scaling SDK VP9 encoder always uses single active reference frame for first frame after resolution change. So SDK VP9 encoder has following limitation for dynamic resolution change: new resolution shouldn't exceed 16x and be below than 1/2 of current resolution.

Application may force insertion of key-frame at the place of resolution change by invoking encoder reset with `mfxExtEncoderResetOption::StartNewSequence` set to `MFX_CODINGOPTION_ON`. In case of inserted key-frame above limitations for new resolution are not in force.

It should be noted that resolution change with dynamic reference scaling is compatible with multiref (`mfxVideoParam::NumRefFrame > 1`). For multiref configuration SDK VP9 encoder uses multiple references within stream pieces of same resolution, and uses single reference at the place of resolution change.

12.7.3.4 Forced Key Frame Generation

The SDK supports forced key frame 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 key frame. 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 exactly specify frame type for every frame thus the application can enforce the current frame to have any frame type that particular coding standard allows.

12.7.3.5 Reference List Selection

During streaming or video conferencing, if the application can obtain feedbacks about how good the client receives certain frames, the application may need to adjust the encoding process to use or not use certain frames as reference. The following paragraphs describe how to fine-tune the encoding process based on such feedbacks.

The application can specify the reference window size by specifying the parameter `mfxInfoMFX::NumRefFrame` during encoding initialization. Certain platform may have limitation on how big the size of the reference window is. Use the function `MFXVideoENCODE_GetVideoParam()` 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 `MFXVideoENCODE_EncodeFrameAsync()` function. The `mfxExtAVCRefListCtrl::NumRefIdxL0Active` member specifies the length of the reference list L0 and the `mfxExtAVCRefListCtrl::NumRefIdxL1Active` member specifies the length of the reference list L1. These two numbers must be less or equal to the parameter `mfxInfoMFX::NumRefFrame` 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 must uniquely identify each input frame, by setting the `mfxFrameData::FrameOrder` parameter. The application then specifies the preferred reference frame list `mfxExtAVCRefListCtrl::PreferredRefList` and/or the rejected frame list `mfxExtAVCRefListCtrl::RejectedRefList`, and attach the `mfxExtAVCRefListCtrl` structure to the `MFXVideoENCODE_EncodeFrameAsync()` function. The two lists fine-tune how the SDK encoder chooses the reference frames of the current frame. The SDK encoder does not keep `PreferredRefList` and application has to send it for each frame if necessary. There are a few limitations:

- 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.

12.7.3.6 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 number of reference frames.

12.7.3.7 Reference Picture Marking Repetition SEI message

The application can request writing the reference picture marking repetition SEI message during encoding initialization, by setting the `mfxExtCodingOption::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 such message exists in the bitstream, and check with 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.

12.7.3.8 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 frame as long-term.

There are two control lists in the `mfxExtAVCRefListCtrl` extended buffer. The `mfxExtAVCRefListCtrl::LongTermRefList` list contains the frame orders (the `mfxFrameData::FrameOrder` value in the `mfxFrameData` structure) of the frames that should be marked as long-term frames. The `mfxExtAVCRefListCtrl::RejectedRefList` list contains the frame order of the frames that should be unmarked as long-term frames. The application can only mark/unmark those frames that are buffered inside encoder. Because of this, it is recommended that the application marks a frame when it is submitted for encoding. Application can either explicitly unmark long-term reference frame or wait for IDR frame, there 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 `mfxExtAVCRefListCtrl::NumRefIdxL0Active` and `mfxExtAVCRefListCtrl::NumRefIdxL1Active` values in the `mfxExtAVCRefListCtrl` extended buffer) is smaller than the total reference frame number (the `mfxInfoMFx::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 list of preferred reference frames in the `mfxExtAVCRefListCtrl::PreferredRefList` list in the `mfxExtAVCRefListCtrl` extended buffer. In this case, the SDK encoder reorders the reference list based on the specified list.

12.7.3.9 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 `mfxExtAvcTemporalLayers::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 `mfxExtCodingOptionSPSPPS::SPSID`/`mfxExtCodingOptionSPSPPS::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 `mfxExtAvcTemporalLayers::Scale` value. Scale for layer N is equal to ratio between the frame rate of subsequence consisted of temporal layers with temporal ID lower or equal to N and frame rate of base temporal layer. The application may skip some of the temporal layers by specifying the `mfxExtAvcTemporalLayers::Scale` value as zero. The application should use an integer ratio of the frame rates for two consecutive temporal layers.

For example, 30 frame per second video sequence typically is 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). `mfxExtAvcTemporalLayers::Scale` for this case should have next values {1,2,4,0,0,0,0,0}.

12.7.4 Switchable Graphics and Multiple Monitors

The following sections address a few aspects of supporting switchable graphics and multiple monitors configurations.

12.7.4.1 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 document discusses the implication of switchable graphics to the SDK and the SDK applications.

As the SDK performs hardware acceleration through Intel graphic device, it is critical that the SDK can access to the Intel graphic device in the switchable graphics setting. If possible, it is recommended to add the application to the Intel graphic device affinity list. Otherwise, the application must handle the following cases:

- By the SDK design, during the SDK library initialization, the function `MFXInit()` searches for Intel graphic devices. If a SDK implementation is successfully loaded, the function `MFXInit()` returns `MFX_ERR_NONE` and the `MFXQueryIMPL()` function returns the actual implementation type. If no SDK implementation is loaded, the function `MFXInit()` returns `MFX_ERR_UNSUPPORTED`. In the switchable graphics environment, if the application is not in the Intel graphic device affinity list, it is possible that the Intel graphic device is not accessible during the SDK library initialization. The fact that the `MFXInit()` function returns `MFX_ERR_UNSUPPORTED` does not mean that hardware acceleration is not possible permanently. The user may switch the graphics later and by then the Intel 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 Intel graphic device affinity list, the previously accessible Intel graphic device may become inaccessible due to a switch event. The SDK functions will return `MFX_ERR_DEVICE_LOST` or `MFX_ERR_DEVICE_FAILED`, depending on when the switch occurs and what stage the SDK functions operate. The application needs to handle these errors and exits gracefully.

12.7.4.2 Multiple Monitors

Multiple monitors refer to the machine configuration that multiple graphic devices are available. Some of the graphic devices connect to a display, they become active and accessible under the Microsoft* DirectX* infrastructure. For those graphic devices not connected to a display, they are inactive. Specifically, under the Microsoft Direct3D9* infrastructure, those devices are not accessible.

The SDK uses the adapter number to access to a specific graphic device. Usually, the graphic device that drives the main desktop becomes the primary adapter. Other graphic devices take subsequent adapter numbers after the primary adapter. Under the Microsoft Direct3D9 infrastructure, only active adapters are accessible and thus have an adapter number.

The SDK extends the implementation type `mfxIMPL` as follows:

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

The application can use the above definitions to instruct the SDK library to initialize on a specific graphic device. The application can also use the following definitions for automatic detection:

Implementation Type	Definition
<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.

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:

- 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 `MFX_ERR_NONE` if successful.
- The application uses the `MFXQueryIMPL()` function to check the actual implementation type. The implementation type `MFX_IMPL_HARDWARE1`, ..., `MFX_IMPL_HARDWARE4` indicates the graphic adapter the SDK works on.
- The application creates the Direct3D* device on the respective graphic adapter, and passes it to the SDK through the `MFXVideoCORE_SetHandle()` function.

Finally, similar to the switchable graphics cases, it is possible that the user disconnects monitors from the graphic devices or remaps the primary adapter thus causes interruption. If the interruption occurs during the SDK library initialization, the `MFXInit()` function may return `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 `MFX_ERR_DEVICE_LOST` or `MFX_ERR_DEVICE_FAILED`. The application needs to handle these errors and exit gracefully.

12.7.4.3 Working directly with VA API for Linux*

The SDK takes care of all memory and synchronization related operations in VA API. However, in some cases the application may need to extend the SDK functionality by working directly with VA API for Linux*. For example, to implement customized external allocator. This chapter describes some basic memory management and synchronization techniques.

To create VA surface pool the application should call `vaCreateSurfaces`:

```
VASurfaceAttrib attrib;
attrib.type = VASurfaceAttribPixelFormat;
attrib.value.type = VAGenericValueTypeInteger;
attrib.value.value.i = VA_FOURCC_NV12;
attrib.flags = VA_SURFACE_ATTRIB_SETTABLE;

#define NUM_SURFACES 5;
VASurfaceID surfaces[NUMSURFACES];

vaCreateSurfaces(va_display, VA_RT_FORMAT_YUV420, width, height, surfaces, NUM_
->SURFACES, &attrib, 1);
```

To destroy surface pool the application should call `vaDestroySurfaces`:

```
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 `MFXVideoCORE_SyncOperation()` call. If the application works with hardware acceleration directly then it has to check surface status before accessing data in video memory. This check can be done asynchronously by calling `vaQuerySurfaceStatus` function or synchronously by `vaSyncSurface` function.

After successful synchronization the application can access surface data. It is performed in two steps. At the first step `VAImage` is created from surface and at the second step image buffer is mapped to system memory. After mapping `VAImage.offsets[3]` array holds offsets to each color plain in mapped buffer and `VAImage.pitches[3]` array holds color plain pitches, in bytes. For packed data formats, only first entries in these arrays are valid. How to access data in NV12 surface:

```
VAImage image;
unsigned char *buffer, Y, U, V;

vaDeriveImage(va_display, surface_id, &image);
vaMapBuffer(va_display, image.buf, &buffer);

/* NV12 */
Y = buffer + image.offsets[0];
U = buffer + image.offsets[1];
V = U + 1;
```

After processing data in VA surface the application should release resources allocated for mapped buffer and `VAImage` object:

```
vaUnmapBuffer(va_display, image.buf);
vaDestroyImage(va_display, image.image_id);
```

In some cases, for example, to retrieve encoded bitstream from video memory, the application has to use `VABuffer` to store data. Example below shows how to create, use and then destroy VA buffer. Note, that `vaMapBuffer` function returns pointers to different objects depending on mapped buffer type. It is plain data buffer for `VAImage` and `VACodedBufferSegment` structure for encoded bitstream. The application cannot use `VABuffer` for synchronization and in case of encoding it is recommended to synchronize by input VA surface as described above.

```
/* create buffer */
VABufferID buf_id;
vaCreateBuffer(va_display, va_context, VAEncCodedBufferType, buf_size, 1, NULL, &buf_id);

/* encode frame */
// ...

/* map buffer */
VACodedBufferSegment *coded_buffer_segment;

vaMapBuffer(va_display, buf_id, (void **)(& coded_buffer_segment));

size = coded_buffer_segment->size;
offset = coded_buffer_segment->bit_offset;
buf = coded_buffer_segment->buf;

/* retrieve encoded data*/
// ...

/* unmap and destroy buffer */
vaUnmapBuffer(va_display, buf_id);
vaDestroyBuffer(va_display, buf_id);
```

12.7.4.4 CQP HRD mode encoding

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 appropriate profile/level. It's responsibility of application to provide per-frame QP, track HRD conformance and insert required SEI messages to the bitstream.

Example below shows how to enable CQP HRD mode. 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). SDK will choose CBR or VBR HRD mode based on **MaxKbps** parameter. If **MaxKbps** is set to zero, SDK will use CBR HRD model (write *cbr_flag* = 1 to VUI), otherwise VBR model will be used (and *cbr_flag* = 0 is written to VUI).

```
mfxExtCodingOption option, *option_array;

/* configure mfxExtCodingOption */
memset(&option, 0, sizeof(option));
option.Header.BufferId      = MFX_EXTBUFF_CODING_OPTION;
option.Header.BufferSz       = sizeof(option);
option.VuiNalHrdParameters  = MFX_CODINGOPTION_ON;
option.NalHrdConformance    = MFX_CODINGOPTION_OFF;

/* configure mfxVideoParam */
mfxVideoParam param;

// ...

param.mfx.RateControlMethod      = MFX_RATECONTROL_CQP;
param.mfx.FrameInfo.FrameRateExtN = <valid_non_zero_value>;
param.mfx.FrameInfo.FrameRateExtD = <valid_non_zero_value>;
param.mfx.BufferSizeInKB        = <valid_non_zero_value>;
param.mfx.InitialDelayInKB     = <valid_non_zero_value>;
param.mfx.TargetKbps           = <valid_non_zero_value>;

if (<write cbr_flag = 1>)
    param.mfx.MaxKbps = 0;
else /* <write cbr_flag = 0> */
    param.mfx.MaxKbps = <valid_non_zero_value>;

param.NumExtParam = 1;
option_array      = &option;
param.ExtParam    = &option_array;

/* encoder initialization */
mfxStatus sts;
sts = MFXVideoENCODE_Init(session, &param);

// ...

/* encoding */
mfxEncodeCtrl ctrl;
memset(&ctrl, 0, sizeof(ctrl));
ctrl.QP = <frame_qp>

sts=MFXVideoENCODE_EncodeFrameAsync(session,&ctrl,surface2,bits,&syncp);
```

12.8 oneVPL API Reference

12.8.1 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**

Unigned integer, 64 bit type

typedef __INT64 **mfxI64**

Signed integer, 64 bit type

typedef float **mfxF32**

Single-presesion floating point, 32 bit type

typedef double **mfxF64**

Double-presesion 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**

ASCII character, 8 bit type

12.8.2 Typedefs

```
typedef struct _mfxSession *mfxSession
SDK session handle

typedef struct _mfxSyncPoint *mfxSyncPoint
Syncronization point object handle

typedef struct _mfxLoader *mfxLoader
SDK loader handle

typedef struct _mfxConfig *mfxConfig
SDK config handle
```

12.8.3 oneVPL Dispatcher API

12.8.3.1 Defines

MFX_IMPL_NAME
Maximum allowed lenght of the implementation name.

MFX_STRFIELD_LEN
Maximum allowed lenght of the implementation name.

12.8.3.2 Structures

12.8.3.3 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 mfxVariant 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.

12.8.3.4 mfxDecoderDescription

struct mfxDecoderDescription

This structure represents decoders description.

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 decoder description.

Public Members

mfxU32 CodecID

decoder ID in fourCC format.

mfxU16 reserved[8]

reserved for future use.

mfxU16 MaxcodecLevel

Maximum supported codec's level.

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 codec's profile description.

Public Members

mfxU32 Profile

Profile ID in fourCC format.

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 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 output color formats.

mfxU32 ***ColorFormats**

Pointer to the array of supported output color formats (in fourCC).

12.8.3.5 mfxEncoderDescription

struct mfxEncoderDescription

This structure represents 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's level.

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 codec's profile description.

Public Members

mfxU32 Profile

Profile ID in fourCC format.

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 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 input color formats.

mfxU32 *ColorFormats

Pointer to the array of supported input color formats (in fourCC).

12.8.3.6 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 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 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 input color format description.

Public Members

mfxU16 reserved[7]
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).

12.8.3.7 mfxImplDescription

struct mfxImplDescription

This structure represents implementation description

Public Members

mfxStructVersion Version

Version of the structure.

mfxIMPL Impl

Impl type: SW/GEN/Bay/Custom.

mfxU16 accelerationMode

Hardware acceleration stack to use. OS depended parameter. On linux - VA, on Windows - DX*.

mfxVersion ApiVersion

Supported API version.

mfxU8 ImplName[MFX_IMPL_NAME]

Null-terminated string with implementation name given by vendor.

mfxU8 License[MFX_STRFIELD_LEN]

Null-terminated string with licence name of the implementation.

mfxU8 Keywords[MFX_STRFIELD_LEN]

Null-terminated string with comma-separated list of keywords specific to this implementation that dispatcher can search for.

mfxU32 VendorID

Standart vendor ID 0x8086 - Intel.

mfxU32 VendorImplID

Vendor specific number with gived implementation ID.

mfxDecoderDescription Dec

Decoders config.

mfxEncoderDescription Enc

Encoders config.

mfxVPPDescription VPP

VPP config.

mfxU32 reserved[16]

reserved for future use.

mfxU32 NumExtParam

Number of extension buffers. Reserved for future. Must be 0.

mfxExtBuffer ****ExtParam**
 Array of extension buffers.

mfxU64 **Reserved2**
 reserved for future use.

union *mfxImplDescription*::[anonymous] **ExtParams**
 Extension buffers. Reserved for future.

12.8.3.8 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 dispatcher internal coniguration, which is used to filter out avialable implementations. Then this config is used to walk through selected implementations to gather more details and select appropriate implementation to load. Loader object remembers all created mfxConfig objects and desrtoyes them during mfxUnload function call.

Multilple configurations per single mfxLoader object is 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 used to add additional filter property (any fields of *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_SOFTWARE;
MFXSetConfigFilterProperty(cfg, "mfxImplDescription.Impl", ImplValue);
MFXCreateSession(loader, 0, &session);
```

Note Each new call with the same parameter “name” will overwrite previously set “value”. This may invalidate other properties.

Note Each new call with another parameter “name” will delete previous property and create new property based on new “name”’s value.

Two sessions usage example (Multiple loaders example):

```
// 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_SOFTWARE;
MFXSetConfigFilterProperty(cfg1, "mfxImplDescriptionImpl", 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_HARDWARE;
MFXSetConfigFilterProperty(cfg2, "mfxImplDescriptionImpl", ImplValueHW);
MFXCreateSession(loader2, 0, &sessionHW);

// use both sessionSW and sessionHW
// ...
// Close everything
MFXCLOSE(sessionSW);
MFXCLOSE(sessionHW);
MFXUnload(loader1); // cfg1 will be destroyed here.
MFXUnload(loader2); // cfg2 will be destroyed here.
```

Two decoders example (Multiple Config objects example):

```
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 paramer 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 used to iterate over filtered out implementations to gather their details. This function allocates memory to store *mfxImplDescription* structure instance. Use 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 *mfxImplCapsDeliveryFormat* enumerator for more details.
- [out] idesc: Poiner to the *mfxImplDescription* structure.

mfxStatus **MFXCreateSession** (*mfxLoader* loader, *mfxU32* i, *mfxSession* **session*)

This function 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 MFXQueryImplCapabilities 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.

12.8.4 Enums

12.8.4.1 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 | undeveloped 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 HW 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_WRN_IN_EXECUTION = 1
    the previous asynchronous operation is in execution

enumerator MFX_WRN_DEVICE_BUSY = 2
    the HW acceleration device is busy

enumerator MFX_WRN_VIDEO_PARAM_CHANGED = 3
    the video parameters are changed during decoding

enumerator MFX_WRN_PARTIAL_ACCELERATION = 4
    SW 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

```

12.8.4.2 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.

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_SINGLE_THREAD = 0x0009

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* Direct3D9* infrastructure.

enumerator MFX_IMPL_VIA_D3D11 = 0x0300
Hardware acceleration goes through the Microsoft* Direct3D11* infrastructure.

enumerator MFX_IMPL_VIA_VAAPI = 0x0400
Hardware acceleration goes through the Linux* VA API infrastructure.

enumerator MFX_IMPL_EXTERNAL_THREADING = 0x10000

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.

12.8.4.3 `mfxImplCapsDeliveryFormat`

`enum mfxImplCapsDeliveryFormat`

Values:

`enumerator MFX_IMPLCAPS_IMPLDESCSTRUCTURE = 1`

Deliver capabilities as `mfxImplDescription` structure.

12.8.4.4 `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.

12.8.4.5 `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.

12.8.4.6 `PlatformCodeName`

`enumerator MFX_PLATFORM_UNKNOWN = 0`

Unknown platform

`enumerator MFX_PLATFORM_SANDYBRIDGE = 1`

Sandy Bridge

`enumerator MFX_PLATFORM_IVYBRIDGE = 2`

Ivy Bridge

`enumerator MFX_PLATFORM_HASWELL = 3`

Haswell

`enumerator MFX_PLATFORM_BAYTRAIL = 4`

Bay Trail

`enumerator MFX_PLATFORM_BROADWELL = 5`

Broadwell

`enumerator MFX_PLATFORM_CHERRYTRAIL = 6`

Cherry Trail

```

enumerator MFX_PLATFORM_SKYLAKE = 7
    Skylake

enumerator MFX_PLATFORM_APOLLOLAKE = 8
    Apollo Lake

enumerator MFX_PLATFORM_KABYLAKE = 9
    Kaby Lake

enumerator MFX_PLATFORM_GEMINILAKE = 10
    Gemini Lake

enumerator MFX_PLATFORM_COFFEE LAKE = 11
    Coffe Lake

enumerator MFX_PLATFORM_CANNONLAKE = 20
    Cannon Lake

enumerator MFX_PLATFORM_ICELAKE = 30
    Ice Lake

enumerator MFX_PLATFORM_JASPERLAKE = 32
    Jasper Lake

enumerator MFX_PLATFORM_ELKHARTLAKE = 33
    Elkhart Lake

enumerator MFX_PLATFORM_TIGERLAKE = 40
    Tiger Lake

```

12.8.4.7 mfxMediaAdapterType

enum mfxMediaAdapterType

The mfxMediaAdapterType enumerator itemizes types of Intel Gen Graphics adapters.

Values:

```

enumerator MFX_MEDIA_UNKNOWN = 0xffff
    Unknown type.

enumerator MFX_MEDIA_INTEGRATED = 0
    Integrated Intel Gen Graphics adapter.

enumerator MFX_MEDIA_DISCRETE = 1
    Discrete Intel Gen Graphics adapter.

```

12.8.4.8 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

12.8.4.9 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.

12.8.4.10 ColorFourCC

The ColorFourCC enumerator itemizes color formats.

```
enumerator MFX_FOURCC_NV12 = MFX_MAKEFOURCC('N', 'V', '1', '2')
```

NV12 color planes. Native Format

```
enumerator MFX_FOURCC_NV21 = MFX_MAKEFOURCC('N', 'V', '2', '1')
```

```
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 NV12 but with weaved V and U values.

```
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. ARGB is the order, A channel is 8 MSBs

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.

Direct3D9: IDirectXVideoDecoderService::CreateSurface()

Direct3D11: 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.

Direct3D9: IDirectXVideoDecoderService::CreateSurface()

Direct3D11: 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')

< same as YV12 except that the U and V plane order is reversed.

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')

0 bit per sample 4:2:2 color format with similar to NV12 layout

enumerator MFX_FOURCC_BGR4 = MFX_MAKEFOURCC('B', 'G', 'R', '4')

ABGR color format. It is similar to MFX_FOURCC_RGB4 but with interchanged R and B channels. 'A' is 8 MSBs, then 8 bits for 'B' channel, then 'G' and 'R' 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.
```

12.8.4.11 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 subsampling. It is equal to 4:2:2 color.

enumerator MFX_CHROMAFORMAT_YUV422V = 5
    4:2:2 color, vertical subsampling

enumerator MFX_CHROMAFORMAT_RESERVED1 = 6
    Reserved

enumerator MFX_CHROMAFORMAT_JPEG_SAMPLING = 6
    Color sampling specified via mfxInfoMFx::SamplingFactorH and SamplingFactorV.
```

12.8.4.12 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

```

12.8.4.13 Frame Data Flags

```

enumerator MFX_TIMESTAMP_UNKNOWN = -1
    Indicates that timestamp is unknown for this frame/bitsream 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.

```

12.8.4.14 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 refers to frame was decoded with minor/major corruption flag – this frame is also marked with 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 bitstream. Which field is absent depends on value of bottom_field_flag (ITU-T H.264 7.4.3).

12.8.4.15 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 telecining attributes are available in the bitstream and time stamp is not explicitly specified. The input frame rate must be specified.

12.8.4.16 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.

12.8.4.17 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
```

12.8.4.18 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 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)
```

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.
```

MPEG-2 profiles

```
enumerator MFX_PROFILE_MPEG2_SIMPLE = 0x50
enumerator MFX_PROFILE_MPEG2_MAIN = 0x40
enumerator MFX_PROFILE_MPEG2_HIGH = 0x10
```

VC-1 Profiles

```
enumerator MFX_PROFILE_VC1_SIMPLE = (0 + 1)
enumerator MFX_PROFILE_VC1_MAIN = (4 + 1)
```

```

enumerator MFX_PROFILE_VC1_ADVANCED = (12 + 1)

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

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

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

JPEG Profiles

enumerator MFX_PROFILE_JPEG_BASELINE = 1
    Baseline JPEG Profile.

```

12.8.4.19 CodecLevel

The CodecLevel enumerator itemizes codec levels for all codecs.

```

enumerator MFX_LEVEL_UNKNOWN = 0
    Unspecified level

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

H.264 level 2-2.2

enumerator MFX_LEVEL_AVC_2 = 20

enumerator MFX_LEVEL_AVC_21 = 21

enumerator MFX_LEVEL_AVC_22 = 22

H.264 level 3-3.2

enumerator MFX_LEVEL_AVC_3 = 30

```

```
enumerator MFX_LEVEL_AVC_31 = 31
enumerator MFX_LEVEL_AVC_32 = 32
H.264 level 4-4.2
enumerator MFX_LEVEL_AVC_4 = 40
enumerator MFX_LEVEL_AVC_41 = 41
enumerator MFX_LEVEL_AVC_42 = 42
H.264 level 5-5.2
enumerator MFX_LEVEL_AVC_5 = 50
enumerator MFX_LEVEL_AVC_51 = 51
enumerator MFX_LEVEL_AVC_52 = 52
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
VC-1 Level Low (simple & main profiles)
enumerator MFX_LEVEL_VC1_LOW = (0 + 1)
enumerator MFX_LEVEL_VC1_MEDIAN = (2 + 1)
enumerator MFX_LEVEL_VC1_HIGH = (4 + 1)
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)
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
```

12.8.4.20 HEVC Tiers

```
enumerator MFX_TIER_HEVC_MAIN = 0
enumerator MFX_TIER_HEVC_HIGH = 0x100
```

12.8.4.21 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.

12.8.4.22 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.
```

12.8.4.23 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 compliancy are satisfied. It uses the same set of parameters as VBR and quality factor specified by *mfxExtCodingOption3::QVBRQuality*.

12.8.4.24 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.

12.8.4.25 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.

12.8.4.26 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.

12.8.4.27 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

12.8.4.28 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.

12.8.4.29 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.

12.8.4.30 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).
```

12.8.4.31 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.
```

12.8.4.32 ScenarioInfo

The ScenarioInfo enumerator itemizes scenarios for the encoding session.

```
enumerator MFX_SCENARIO_UNKNOWN = 0
enumerator MFX_SCENARIO_DISPLAY_REMOTEING = 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
```

12.8.4.33 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
```

12.8.4.34 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

12.8.4.35 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)

12.8.4.36 CodingOptionValue

The CodingOptionValue enumerator defines a three-state coding option setting.

enumerator MFX_CODINGOPTION_UNKNOWN = 0

Unspecified.

enumerator MFX_CODINGOPTION_ON = 0x10

Coding option set.

enumerator MFX_CODINGOPTION_OFF = 0x20

Coding option not set.

enumerator MFX_CODINGOPTION_ADAPTIVE = 0x30

Reserved

12.8.4.37 BitstreamDataFlag

The BitstreamDataFlag enumerator uses bit-ORed values to itemize additional information about the bitstream buffer.

enumerator MFX_BITSTREAM_COMPLETE_FRAME = 0x0001

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.

enumerator MFX_BITSTREAM_EOS = 0x0002

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.

12.8.4.38 ExtendedBufferID

The ExtendedBufferID enumerator itemizes and defines identifiers (BufferId) for extended buffers or video processing algorithm identifiers.

enumerator MFX_EXTBUFF_THREADS_PARAM = MFX_MAKEFOURCC('T', 'H', 'D', 'P')
mfxExtThreadsParam buffer ID

enumerator MFX_EXTBUFF_CODING_OPTION = MFX_MAKEFOURCC('C', 'D', 'O', 'P')

This extended buffer defines additional encoding controls. See the *mfxExtCodingOption* structure for details.
The application can attach this buffer to the structure for encoding initialization.

enumerator MFX_EXTBUFF_CODING_OPTION_SPSPPS = MFX_MAKEFOURCC('C', 'O', 'S', 'P')

This extended buffer defines sequence header and picture header for encoders and decoders. See the *mfxExtCodingOptionSPSPPS* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for encoding initialization, and for obtaining raw headers from the decoders and encoders.

enumerator MFX_EXTBUFF_VPP_DONOTUSE = MFX_MAKEFOURCC('N', 'U', 'S', 'E')

This extended buffer defines a list of VPP algorithms that applications should not use. See the *mfxExtVPP-DoNotUse* structure for details. The application can attach this buffer to the *mfxVideoParam* structure for video processing initialization.

enumerator MFX_EXTBUFF_VPP_AUXDATA = MFX_MAKEFOURCC('A', 'U', 'X', 'D')

This extended buffer defines auxiliary information at the VPP output. See the *mfxExtVppAuxData* structure for details. The application can attach this buffer to the *mfxEncodeCtrl* structure for per-frame encoding control.

enumerator MFX_EXTBUFF_VPP_DENOISE = MFX_MAKEFOURCC('D', 'N', 'T', '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', '')

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', '')

This extended buffer defines control parameters for the VPP frame rate conversion algorithm. See the *mfx-*

ExtVPPFrameRateConversion 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 *mfxExtAvctTemporalLayers* 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', 'T')

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 *mfxExtHEVCP param* 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 [mfxExtAvTemporalLayers](#) 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 [mfxExtVPPDoUse](#) 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_ENCTOOLS_CONFIG = MFX_MAKEFOURCC('E', 'E', 'T', 'C')

See the *mfxExtEncToolsConfig* structure for details.

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.

12.8.4.39 PayloadCtrlFlags

The PayloadCtrlFlags enumerator itemizes additional payload properties.

enumerator MFX_PAYLOAD_CTRL_SUFFIX = 0x00000001

Insert this payload into HEVC Suffix SEI NAL-unit.

12.8.4.40 ExtMemFrameType

The ExtMemFrameType enumerator specifies the memory type of frame. It is a bit-ORed value of the following. For information on working with video memory surfaces, see the section Working with hardware acceleration.

enumerator MFX_MEMTYPE_PERSISTENT_MEMORY = 0x0002

Memory page for persistent use.

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.

```

12.8.4.41 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.

12.8.4.42 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.

12.8.4.43 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.

enumerator MFX_HANDLE_CM_DEVICE = 8

12.8.4.44 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.

12.8.4.45 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.

12.8.4.46 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.

12.8.4.47 InsertHDPayload

The InsertHDPayload 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.

12.8.4.48 LongTermIdx

The LongTermIdx specifies long term index of picture control

enumerator MFX_LONGTERM_IDX_NO_IDX = 0xFFFF

Long term index of picture is undefined.

12.8.4.49 TransferMatrix

The TransferMatrix enumerator itemizes color transfer matrixes.

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.

12.8.4.50 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].
```

12.8.4.51 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
```

12.8.4.52 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 framerate output.
```

```
enumerator MFX_DEINTERLACING_AUTO_SINGLE = 4
    Auto mode with deinterlacing single framerate output.
```

```
enumerator MFX_DEINTERLACING_FULL_FR_OUT = 5
    Deinterlace only mode with full framerate output.
```

```
enumerator MFX_DEINTERLACING_HALF_FR_OUT = 6
    Deinterlace only Mode with half framerate 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.
```

12.8.4.53 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.
```

12.8.4.54 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.
```

12.8.4.55 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.
```

12.8.4.56 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.

12.8.4.57 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)
```

12.8.4.58 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).

12.8.4.59 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.

12.8.4.60 HEVCRegionType

The HEVCRegionType enumerator itemizes type of HEVC region.

enumerator MFX_HEVC_REGION_SLICE = 0

Slice type.

12.8.4.61 HEVCRegionEncoding

The HEVCRegionEncoding enumerator itemizes HEVC region's encoding.

enumerator MFX_HEVC_REGION_ENCODING_ON = 0

enumerator MFX_HEVC_REGION_ENCODING_OFF = 1

12.8.4.62 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.

12.8.4.63 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

12.8.4.64 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

12.8.4.65 MirroringType

The MirroringType enumerator itemizes mirroring types.

enumerator MFX_MIRRORING_DISABLED = 0

enumerator MFX_MIRRORING_HORIZONTAL = 1

enumerator MFX_MIRRORING_VERTICAL = 2

12.8.4.66 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.

12.8.4.67 VP9ReferenceFrame

The VP9ReferenceFrame enumerator itemizes reference frame type by 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.

12.8.4.68 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.

12.8.4.69 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.

12.8.4.70 MCTFTemporalMode

The MCTFTemporalMode enumerator itemizes temporal filtering modes.

```
enumerator MFX_MCTF_TEMPORAL_MODE_UNKNOWN = 0
enumerator MFX_MCTF_TEMPORAL_MODE_SPATIAL = 1
enumerator MFX_MCTF_TEMPORAL_MODE_1REF = 2
enumerator MFX_MCTF_TEMPORAL_MODE_2REF = 3
enumerator MFX_MCTF_TEMPORAL_MODE_4REF = 4
```

12.8.4.71 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.
```

12.8.4.72 PartialBitstreamOutput

The PartialBitstreamOutput enumerator indicates flags of partial bitstream output type.

```
enumerator MFX_PARTIAL_BITSTREAM_NONE = 0
    Don't use partial output

enumerator MFX_PARTIAL_BITSTREAM_SLICE = 1
    Partial bitstream output will be aligned to slice granularity

enumerator MFX_PARTIAL_BITSTREAM_BLOCK = 2
    Partial bitstream output will be aligned to user-defined block size granularity

enumerator MFX_PARTIAL_BITSTREAM_ANY = 3
    Partial bitstream output will be return any coded data available at the end of SyncOperation timeout
```

12.8.4.73 BRCStatus

The BRCStatus enumerator itemizes instructions to the SDK encoder by `mfxExtBrc::Update`.

```
enumerator MFX_BRC_OK = 0
    CodedFrameSize is acceptable, no further recoding/padding/skip required, proceed to next frame.

enumerator MFX_BRC_BIG_FRAME = 1
    Coded frame is too big, recoding required.

enumerator MFX_BRC_SMALL_FRAME = 2
    Coded frame is too small, recoding required.
```

```
enumerator MFX_BRC_PANIC_BIG_FRAME = 3
```

Coded frame is too big, no further recoding possible - skip frame.

```
enumerator MFX_BRC_PANIC_SMALL_FRAME = 4
```

Coded frame is too small, no further recoding possible - required padding to *mfxBRCFrameStatus::MinFrameSize*.

12.8.4.74 Rotation

The Rotation enumerator itemizes the JPEG rotation options.

```
enumerator MFX_ROTATION_0 = 0
```

No rotation.

```
enumerator MFX_ROTATION_90 = 1
```

90 degree rotation

```
enumerator MFX_ROTATION_180 = 2
```

180 degree rotation

```
enumerator MFX_ROTATION_270 = 3
```

270 degree rotation

12.8.4.75 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.

12.8.4.76 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.

12.8.4.77 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*.

12.8.5 Structs

12.8.5.1 mfxRange32U

struct mfxRange32U

This structure represents 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

12.8.5.2 mfxI16Pair

struct mfxI16Pair

Thus structure represents pair of numbers of mfxI16 type

Public Members

mfxI16 x

First number

mfxI16 y

Second number

12.8.5.3 mfxHDLPair

struct mfxHDLPair

Thus structure represents pair of handles of mfxHDL type

Public Members

mfxHDL **first**

First handle

mfxHDL **second**

Second number

12.8.5.4 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

12.8.5.5 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

12.8.5.6 mfxPlatform

struct mfxPlatform

The *mfxPlatform* structure contains information about hardware platform.

Public Members

mfxU16 CodeName

Intel® microarchitecture code name. See the PlatformCodeName enumerator for a list of possible values.

mfxU16 DeviceId

Unique identifier of graphics device.

mfxU16 MediaAdapterType

Description of Intel Gen Graphics adapter type. See the mfxMediaAdapterType enumerator for a list of possible values.

mfxU16 reserved[13]

Reserved for future use.

12.8.5.7 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.

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.

12.8.5.8 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;
```

(continues on next page)

(continued from previous page)

```

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 oneVPL. 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 framerate 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 oneVPL. 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 oneVPL. 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 specify 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 framerate by the bitrate.)

The decoder starts decoding after the buffer reaches the initial size InitialDelayInKB, which is equivalent to reaching an initial delay of $\text{InitialDelayInKB} * 8000 / \text{TargetKbpsms}$. Note: 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, level, and so on.

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.

12.8.5.9 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

mfxU16 CropH

Height

Public Members

mfxU32 reserved[4]

Reserbed for future use.

mfxU16 reserved4

Reserbed 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. Ignored as obsolete parameter.

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																0 0 0 0 0 0

12.8.5.10 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.

12.8.5.11 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)

12.8.5.12 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, oneVPL 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, oneVPL 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 surfaceis 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.

12.8.5.13 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.

12.8.5.14 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 presentation time stamp provided by the application in *mfxFrameSurface1* structure and from 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.

12.8.5.15 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.

12.8.5.16 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.

12.8.5.17 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

12.8.5.18 `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.

12.8.5.19 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.

12.8.5.20 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.

12.8.5.21 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.

12.8.5.22 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.

12.8.5.23 mfxAdapterInfo**struct mfxAdapterInfo**

The *mfxAdapterInfo* structure contains description of Intel Gen 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](#)).

12.8.5.24 mfxAdaptersInfo

struct mfxAdaptersInfo

The *mfxAdaptersInfo* structure contains description of all Intel Gen Graphics adapters available on 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.

12.8.5.25 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.

12.8.5.26 VPP Structures

12.8.5.26.1 mfxInfoVPP

struct mfxInfoVPP

Public Members

mfxFrameInfo In

Input format for video processing.

mfxFrameInfo Out

Output format for video processing.

12.8.5.26.2 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.

12.8.5.27 Extension buffers structures

12.8.5.27.1 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.

12.8.5.27.2 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 unconformant 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.

12.8.5.27.3 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 disable 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.

12.8.5.27.4 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_OPTION2.

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 Log2MaxMvLengthHorizontal***mfxU16 Log2MaxMvLengthVertical***

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.

mfxI16 DeblockingAlphaTcOffset***mfxI16 DeblockingBetaOffset******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 ConstrainedIntraPredFlag***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 QuantScaleType***mfxU16 IntraVLCFormat******mfxU16 ScanType******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]

12.8.5.27.5 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.

12.8.5.27.6 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]**

12.8.5.27.7 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.

12.8.5.27.8 `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

12.8.5.27.9 `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`

12.8.5.27.10 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

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.

12.8.5.27.11 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.

12.8.5.27.12 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.

12.8.5.27.13 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]

```

12.8.5.27.14 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.

12.8.5.27.15 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.

12.8.5.27.16 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.

12.8.5.27.17 *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.

12.8.5.27.18 ***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 ROIMode = 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.

12.8.5.27.19 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

12.8.5.27.20 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.

12.8.5.27.21 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]

12.8.5.27.22 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.

12.8.5.27.23 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 **NumQPAlloc**

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.

12.8.5.27.24 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.

12.8.5.27.25 **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.

12.8.5.27.26 **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.

12.8.5.27.27 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.

12.8.5.27.28 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.

12.8.5.27.29 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.

12.8.5.27.30 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.

12.8.5.27.31 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

12.8.5.27.32 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.

12.8.5.27.33 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.

12.8.5.27.34 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.

12.8.5.27.35 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.

12.8.5.27.36 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.

12.8.5.27.37 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.

12.8.5.27.38 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.

12.8.5.27.39 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.

12.8.5.27.40 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.

12.8.5.27.41 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.

12.8.5.27.42 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.

12.8.5.27.43 **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 granulatiry for PartialBitstreamGranularity, valid only for MFX_PARTIAL_BITSTREAM_BLOCK.

mfxU16 **Granularity**

Granulatiry of the partial bitstream: slice/block/any, all types of granularity state in PartialBitstreamOutput enum.

12.8.5.28 VPP Extention buffers

12.8.5.28.1 **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

12.8.5.28.2 **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

12.8.5.29 **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.

12.8.5.29.1 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.

12.8.5.29.2 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.

12.8.5.29.3 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

12.8.5.29.4 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.

12.8.5.29.5 ***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.

12.8.5.29.6 ***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.

12.8.5.29.7 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.

12.8.5.29.8 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.

12.8.5.29.9 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 than 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 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.

12.8.5.29.10 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.

12.8.5.29.11 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.

12.8.5.29.12 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.

12.8.5.29.13 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.

12.8.5.29.14 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.

12.8.5.29.15 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.

12.8.5.29.16 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.

12.8.5.29.17 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_YUV444	The input chroma location
MFX_CHROMAFORMAT_YUV422		
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, and vertical location for input

12.8.5.29.18 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.

12.8.5.30 Bit Rate Control Extension Buffers

12.8.5.30.1 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, frame belongs to.

mfxU16 NumRecode

Number of recodings performed for this frame.

mfxU16 NumExtParam

Reserved for the future use.

mfxExtBuffer ***ExtParam

Reserved for the future use.

Frame spatial complexity calculated according to this formula:

$$R = \frac{16}{WH} \sum_{k=0}^{\frac{W}{4}-1} \sum_{l=0}^{\frac{H}{4}-1} \left[\frac{\sum_{i=0}^3 \sum_{j=0}^3 |P[k * 4 + i][l * 4 + j] - P[k * 4 + i - 1][l * 4 + j]|}{16} \right]$$

$$C = \frac{16}{WH} \sum_{k=0}^{\frac{W}{4}-1} \sum_{l=0}^{\frac{H}{4}-1} \left[\frac{\sum_{i=0}^3 \sum_{j=0}^3 |P[k * 4 + i][l * 4 + j] - P[k * 4 + i][l * 4 + j - 1]|}{16} \right]$$

$$FrameCmplx = \sqrt{R^2 + C^2}$$

12.8.5.30.2 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. This is option for repack feature. Driver calls PAK until current frame size is less or equal maxFrameSize or number of repacking for this frame is equal to maxNumRePak. Repack is available if driver support, MaxFrameSize !=0, MaxNumRePak != 0. Ignored if maxNumRePak == 0.

mfxU8 DeltaQP[8]

This is 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 the future use.

mfxExtBuffer **ExtParam

Reserved for the future use.

12.8.5.30.3 ***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.

12.8.5.30.4 ***mfxExtBRC***

`struct mfxExtBRC`

The *mfxExtBRC* structure contains set of callbacks to perform external bit rate control. Can be attached to *mfxVideoParam* structure during encoder initialization. Turn *mfxExtCodingOption2::ExtBRC* option ON to make encoder use external BRC instead of 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 BRC session according to parameters from input *mfxVideoParam* and attached

structures. It does not modify in any way the input *mfxVideoParam* and attached structures. Invoked during MFXVideoENCODE_Init.

Return MFX_ERR_NONE if no error. 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 in any way the input *mfxVideoParam* and attached structures. Invoked during MFXVideoENCODE_Reset.

Return MFX_ERR_NONE if no error. MFX_ERR_UNSUPPORTED The function detected unsupported video parameters.

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.

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 if no error.

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 if no error.

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 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 if no error.

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.

12.8.5.31 VP8 Extension Buffers

12.8.5.31.1 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.

12.8.5.32 JPEG Extension Buffers

12.8.5.32.1 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

12.8.5.32.2 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

12.8.5.33 MVC Extension Buffers

12.8.5.33.1 ***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.

12.8.5.33.2 `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.

12.8.5.33.3 `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.

12.8.5.33.4 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.

12.8.5.33.5 mfxExtEncToolsConfig

struct mfxExtEncToolsConfig

The *mfxExtEncToolsConfig* structure configures EncTools for SDK encoders. It can be attached to the *mfxVideoParam* structure during MFXVideoENCODE_Init or MFXVideoENCODE_Reset call. If mfxEncToolsConfig buffer isn't attached during initialization, EncTools is disabled. If the buffer isn't attached for MFXVideoENCODE_Reset call, encoder continues to use mfxEncToolsConfig which was actual before.

If the EncTools are unsupported in encoder, MFX_ERR_UNSUPPORTED is returned from MFXVideoENCODE_Query, MFX_ERR_INVALID_VIDEO_PARAM is returned from MFXVideoENCODE_Init. If any EncTools feature is on and not compatible with other video parameters, MFX_WRN_INCOMPATIBLE_VIDEO_PARAM is returned from Init and Query functions.

Some features can require delay before encoding can start. Parameter mfxExtCodingOption2::LookaheadDepth can be used to limit the delay. EncTools features requiring longer delay will be disabled.

If a field in mfxEncToolsConfig is set to MFX_CODINGOPTION_UNKNOWN, the corresponding feature will be enabled if it is compatible with other video parameters .

Actual EncTools configuration can be obtained using MFXVideoENCODE_GetVideoParam function with attached mfxEncToolsConfig buffer.

Public Members

mfxExtBuffer **Header**

Extension buffer header. Header.BufferId must be equal to MFX_EXTBUFF_ENCTOOLS_CONFIG.

mfxStructVersion **Version**

The version of the structure.

mfxU16 **SceneChange**

Tri-state flag for enabling/disabling “Scene change analysis” feature.

mfxU16 **AdaptiveI**

Tri-state flag for configuring “Frame type calculation” feature. Distance between Intra frames depends on the content.

mfxU16 **AdaptiveB**

Tri-state flag for configuring “Frame type calculation” feature. Distance between nearest P (or I) frames depends on the content.

mfxU16 **AdaptiveRefP**

Tri-state flag for configuring “Reference frame list calculation” feature. The most useful reference frames are calculated for P frames.

mfxU16 **AdaptiveRefB**

Tri-state flag for configuring “Reference frame list calculation” feature. The most useful reference frames are calculated for B frames.

mfxU16 **AdaptiveLTR**

Tri-state flag for configuring “Reference frame list calculation” feature. The most useful reference frames are calculated as LTR.

mfxU16 **AdaptivePyramidQuantP**

Tri-state flag for configuring “Delta QP hints” feature. Delta QP is calculated for P frames.

mfxU16 **AdaptivePyramidQuantB**

Tri-state flag for configuring “Delta QP hints” feature. Delta QP is calculated for B frames.

mfxU16 **AdaptiveQuantMatrices**

Tri-state flag for configuring “Adaptive quantization matrix” feature.

mfxU16 **BRCBufferHints**

Tri-state flag for enabling/disabling “BRC buffer hints” feature: calculation of optimal frame size, HRD buffer fullness, etc.

mfxU16 **BRC**

Tri-state flag for enabling/disabling “BRC” functionality: QP calculation for frame encoding, encoding status calculation after frame encoding.

12.8.5.34 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.

12.8.6 Functions

12.8.6.1 Implementation Capabilities

mfxHDL MFXQueryImplDescription (*mfxImplCapsDeliveryFormat* format)

This function delivers implementation capabilities in the requested format according to the format value.

Return Handle to the capability report or NULL in case of unsupported format.

Parameters

- [in] format: Format in which capabilities must be delivered. See *mfxImplCapsDeliveryFormat* for more details.

mfxStatus MFXReleaseImplDescription (*mfxHDL* hdl)

This function destroys handle allocated by MFXQueryImplCapabilities function.

Return MFX_ERR_NONE The function completed successfully.

Parameters

- [in] hdl: Handle to destroy. Can be equal to NULL.

12.8.6.2 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 functions. 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 argument ver 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.

mfxStatus MFXClose (mfxSession session)

This function completes and de-initializes 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.

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 MFXDisjoinSession (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.

12.8.6.3 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

12.8.6.4 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.

12.8.6.5 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 HW 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

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 HW 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 HW 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

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.

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

12.8.6.6 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 encoding may be partially accelerated. Only SDK HW 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

*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 HW 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 HW 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

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.

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 *mfxIn*-*foMFX::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

12.8.6.7 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 videoprocessing may be partially accelerated. Only SDK HW 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

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 videoprocessing may be partially accelerated. Only SDK HW 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 HW 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

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.

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

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 BLAS Routines
<code>onemkl::lapack</code>	Dense linear algebra routines from LAPACK and LAPACK like extensions. See LAPACK Routines
<code>onemkl::sparse</code>	Sparse linear algebra routines from Sparse BLAS and Sparse Solvers. See Sparse Linear Algebra
<code>onemkl::dft</code>	Discrete and fast Fourier transformations. See Fourier Transform Functions
<code>onemkl::rng</code>	Random number generator routines. See Random Number Generators
<code>onemkl::vm</code>	Vector mathematics routines, e.g. trigonometric, exponential functions acting on elements of a vector. See Vector Math

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::nonu</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

$$a = [\underbrace{A_{11},*,\dots,*}_{\text{lda}}, \underbrace{A_{12},A_{22},*,\dots,*}_{\text{lda}}, \dots, \underbrace{A_{1n},A_{2n},A_{3n},\dots,A_{mn},*,\dots,*}_{\text{lda}}].$$

$\underbrace{\text{lda} \times n}$

- 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

$$a = [\underbrace{A_{11},A_{21},A_{31},\dots,A_{m1},*,\dots,*}_{\text{lda}}, \underbrace{A_{22},A_{32},\dots,A_{m2},*,\dots,*}_{\text{lda}}, \dots, \underbrace{*,\dots*,A_{mn},*,\dots,*}_{\text{lda}}].$$

$\underbrace{\text{lda} \times n}$

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 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 $A =$

$$A = \begin{array}{ccccccccc|c} A_{11} & A_{12} & A_{13} & \cdots & A_{1,ku+1} & * & \cdots & \cdots & \cdots & \cdots & * \\ A_{21} & A_{22} & A_{23} & A_{24} & \cdots & A_{2,ku+2} & * & \cdots & \cdots & \cdots & * \\ A_{31} & A_{32} & A_{33} & A_{34} & A_{35} & \cdots & A_{3,ku+3} & * & \cdots & \cdots & * \\ \vdots & & A_{42} & A_{43} & \ddots & \ddots & \ddots & \ddots & * & \cdots & \ddots \\ A_{kl+1,1} & \vdots & & A_{53} & \ddots \\ * & A_{kl+2,2} & \vdots & & \ddots \\ \vdots & * & A_{kl+3,3} & & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & * \\ \vdots & \vdots & * & & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & A_{n-ku,n} \\ \vdots & \vdots & \vdots & * & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & * & \ddots & \ddots & \ddots & \ddots & A_{m-2,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & A_{m-1,n} \\ * & * & * & \cdots & \cdots & \cdots & * & A_{m,m-kl} & \cdots & A_{m,n-2} & A_{m,n-1} & A_{mn} \end{array}$$

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 `l_dm`) to band storage (variable `a`, with leading dimension `l_da`):

```

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 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
<code>asum</code>	Sum of vector magnitudes
<code>axpy</code>	Scalar-vector product
<code>copy</code>	Copy vector
<code>dot</code>	Dot product
<code>sdsdot</code>	Dot product with double precision
<code>dotc</code>	Dot product conjugated
<code>dotu</code>	Dot product unconjugated
<code>nrm2</code>	Vector 2-norm (Euclidean norm)
<code>rot</code>	Plane rotation of points
<code>rotg</code>	Generate Givens rotation of points
<code>rotm</code>	Modified Givens plane rotation of points
<code>rotmg</code>	Generate modified Givens plane rotation of points
<code>scal</code>	Vector-scalar product
<code>swap</code>	Vector-vector swap
<code>iamax</code>	Index of the maximum absolute value element of a vector
<code>iamin</code>	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<float></code>	float
<code>std::complex<double></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<float></code>
<code>std::complex<double></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,

α 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 α .

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<float></code>
<code>std::complex<double></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 $(1 + (n - 1) * \text{abs}(\text{incx}))$. 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 $(1 + (n - 1) * \text{abs}(\text{incxy}))$. 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.

T
<code>std::complex<float></code>
<code>std::complex<double></code>

Description

The `dotc` routines perform a dot product between two complex vectors, conjugating the first of them:

$$\text{result} = \sum_{i=1}^n \overline{X}_l Y_i$$

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 $(1 + (n - 1) * \text{abs}(\text{incx}))$. 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 $(1 + (n - 1) * \text{abs}(\text{incy}))$. 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> &dependencies = {})
```

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<float></code>
<code>std::complex<double></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 $(1 + (n - 1) * \text{abs}(incx))$. 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<float></code>	float
<code>std::complex<double></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<float></code>	<code>std::complex<float></code>
<code>std::complex<double></code>	<code>std::complex<double></code>
<code>std::complex<float></code>	<code>float</code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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 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<float></code>
<code>std::complex<double></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 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 $(1 + (n - 1) * \text{abs}(\text{incx}))$. 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 $(1 + (n - 1) * \text{abs}(\text{incx}))$. 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<float></code>
<code>std::complex<double></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 $\text{op}(A) = A^H$,
- alpha and 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 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 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 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<float></code>
<code>std::complex<double></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 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 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 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:

α 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<float></code>
<code>std::complex<double></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:

α 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 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<float></code>
<code>std::complex<double></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:

α 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 $(1 + (m - 1) * \text{abs}(\text{incx}))$. 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.

13.2.1.2.2.15 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 $(1 + (m - 1) * \text{abs}(\text{incx}))$. 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 $\text{l da} * n$. 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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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:

α 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<float></code>
<code>std::complex<double></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:

α and β 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 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<float></code>
<code>std::complex<double></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:

α 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:

α 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 $lda * n$. 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 $(1 + (n - 1) * \text{abs}(incx))$. 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.

13.2.1.2.2.39 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 $lda * n$. 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 $(1 + (n - 1) * \text{abs}(incx))$. 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 $(1 + (n - 1) * \text{abs}(\text{incy}))$. 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](#)

13.2.1.2.2.40 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:

α and β 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:

α 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:

α 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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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.

tpmv supports the following precisions.

T
float
double
<code>std::complex<float></code>
<code>std::complex<double></code>

Description

The 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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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<float></code>	<code>std::complex<float></code>	<code>std::complex<float></code>	<code>std::complex<float></code>
<code>std::complex<double></code>	<code>std::complex<double></code>	<code>std::complex<double></code>	<code>std::complex<double></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<float></code>
<code>std::complex<double></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<float></code>	<code>float</code>
<code>std::complex<double></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)^T + 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<float></code>	<code>float</code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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 $\text{op}(A) = A^H$,

α 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<float></code>
<code>std::complex<double></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 α .

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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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 $\text{op}(A) = A^H$

α 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 `stridet*batch_size`.

ldb Leading dimension of the matrices B. Must be at least m. Must be positive.

stridet 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<float></code>
<code>std::complex<double></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_{dc} 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 $\text{op}(X) = X^H$

α and β 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 * (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`.

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<float></code>
<code>std::complex<double></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 $1 \cdot da * n$.

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<float></code>
<code>std::complex<double></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`, 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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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.

T
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 τ 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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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 τ 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 τ 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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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 τ 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 τ 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<float></code>
<code>std::complex<double></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
<code>ge-brd</code>	<code>ge-brd_scratchpad_size</code>	Reduces a general matrix to bidiagonal form.
<code>gesvd</code>	<code>gesvd_scratchpad</code>	Computes the singular value decomposition of a general rectangular matrix.
<code>heevd</code>	<code>heevd_scratchpad</code>	Computes all eigenvalues and, optionally, all eigenvectors of a complex Hermitian matrix using divide and conquer algorithm.
<code>hegvd</code>	<code>hegvd_scratchpad</code>	Computes all eigenvalues and, optionally, all eigenvectors of a complex generalized Hermitian definite eigenproblem using divide and conquer algorithm.
<code>hetrd</code>	<code>hetrd_scratchpad</code>	Reduces a complex Hermitian matrix to tridiagonal form.
<code>orgbr</code>	<code>orgbr_scratchpad</code>	Generates the real orthogonal matrix Q or P^T determined by <code>gebrd</code> .
<code>orgtr</code>	<code>orgtr_scratchpad</code>	Generates the real orthogonal matrix Q determined by <code>sytrd</code> .
<code>ormtr</code>	<code>ormtr_scratchpad</code>	Multiplies a real matrix by the orthogonal matrix Q determined by <code>sytrd</code> .
<code>syevd</code>	<code>syevd_scratchpad</code>	Computes all eigenvalues and, optionally, all eigenvectors of a real symmetric matrix using divide and conquer algorithm.
<code>sygvd</code>	<code>sygvd_scratchpad</code>	Computes all eigenvalues and, optionally, all eigenvectors of a real generalized symmetric definite eigenproblem using divide and conquer algorithm.
<code>sytrd</code>	<code>sytrd_scratchpad</code>	Reduces a real symmetric matrix to tridiagonal form.
<code>ungbr</code>	<code>ungbr_scratchpad</code>	Generates the complex unitary matrix Q or P^T determined by <code>gebrd</code> .
<code>ungtr</code>	<code>ungtr_scratchpad</code>	Generates the complex unitary matrix Q determined by <code>hetrd</code> .
<code>un-mtr</code>	<code>un-mtr_scratchpad_size</code>	Multiplies a complex matrix by the unitary matrix Q determined by <code>hetrd</code> .

13.2.1.3.2.1 `gebrd`

Reduces a general matrix to bidiagonal form.

`gebrd` supports the following precisions.

T
<code>float</code>
<code>double</code>
<code>std::complex<float></code>
<code>std::complex<double></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 tau_Q , represent the orthogonal matrix Q as a product of elementary reflectors, and the elements above the first superdiagonal, with the buffer 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 tau_Q , represent the orthogonal matrix Q as a product of elementary reflectors, and the elements above the diagonal, with the buffer 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_Q** 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_P** 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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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 Σ 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 Σ 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 Σ 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 Σ 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<float></code>
<code>std::complex<double></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 Λ is a real diagonal matrix whose diagonal elements are the eigenvalues λ_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.

T
<code>std::complex<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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 $A \times = \lambda B \times$;
- if `itype = 2`, the problem type is $A \times B \times = \lambda A \times$;
- if `itype = 3`, the problem type is $B \times A \times = \lambda B \times$.

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<float></code>
<code>zhetrd</code>	<code>std::complex<double></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 τ , 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 τ , 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 τ , 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 τ , 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<float></code>
<code>std::complex<double></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.

T
<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 τ as returned by `sytrd`. The dimension of τ 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 τ as returned by `sytrd`. The dimension of τ 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 τ as returned by [sytrd](#). The dimension of τ 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 Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_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(1..lda, 1..*)` 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^T determined by `gebrd`.

`ungbr` supports the following precisions.

T
<code>std::complex<float></code>
<code>std::complex<double></code>

Description

The routine generates the whole or part of the unitary matrices Q and P^H 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^T, 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^T 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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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 τ as returned by `hetrd`. The dimension of τ 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 τ as returned by `hetrd`. The dimension of τ 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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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 τ as returned by [hetrd](#). The dimension of τ 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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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 $l_{da}[i] * \max(1, n[i])$.

lda A vector, $l_{da}[i]$ is the leading dimension of $a[i]$ ($m[i] \leq l_{da}[i]$).

Output Parameters

a $a[i]$ 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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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 &lda, 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<float></code>
<code>std::complex<double></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 _i
n [i]	m _i
k [i]	k _i

To compute the leading k_i columns of Qk_i (which form an orthonormal basis in the space spanned by the leading k_i columns of the matrix A_i):

m [i]	m _i
n [i]	k _i
k [i]	k _i

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_i (0≤m [i]).

n A vector, n [i] is the number of columns of Q_i 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_i (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_i.

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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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 , α 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<float></code>	
<code>std::complex<double></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
dimension	<code>std::int64_t</code>	Dimension of the transform 1-D transform.

Input Parameters: N-Dimensional

Name	Type	Description
dimensions	<code>std::vector<std::int64_t></code>	Dimensions of the transform.

Output Parameters

Name	Type	Description
status	<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
param	<code>onemkl::dft::ConfigParam</code>	Configuration parameter.
value	Depends on the configuration parameter	Configuration value.

Output Parameters

Name	Type	Description
status	<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<IOType, 1></code>	<code>cl::sycl::buffer</code> containing an array of length no less than specified at <code>onemkl::dft::Descriptor<onemkl::dft::Precision, onemkl::dft::Domain>::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, μ, f, U, g) , where:

- S is a finite set of states (the state space)
- μ 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 μ 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 Distributions for details.
engine	EngineType	Engine object. See Engines 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 ¹³ , 2 ⁵⁹) 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(13¹³, 2⁵⁹) 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>{
    std::initializer_list<std::uint32_t> seed)
    mrg32k3a (cl::sycl::queue& queue,
    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)                                     std::initializer_list<std::uint32_t>_
    ↵wichmann_hill (cl::sycl::queue& queue,           std::uint32_t engine_idx)
    ↵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`

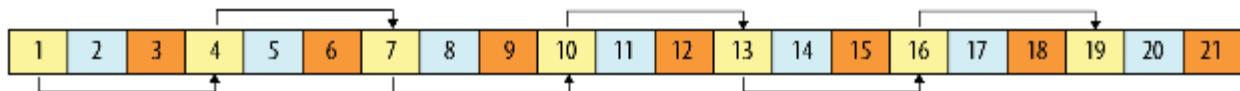
Input Parameters

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

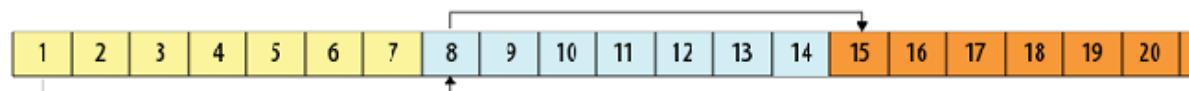
Name	Type	Description
engine	EngineType	Object of engine class, which supports the block-splitting method.
num_to_skip	std::uint64_t	Number of skipped elements.

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 3nd 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::negative_binomial	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_mn	No Yes	gaussian lognormal	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_mn	Yes	2gaussian 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_f	No Yes Yes No Yes No 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 ν 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 No	inverse 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 (stddev , σ), 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 Distributions Template Parameter <code>onemkl::rng::method</code> Values .
mean	T (<code>float</code> , <code>double</code>)	Mean value a .
std-dev	T (<code>float</code> , <code>double</code>)	Standard deviation σ .

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 β , 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 Distributions Template Parameter onemkl::rng::method Values .
a	T (float, double)	Displacement a .
beta	T (float, double)	Scalefactor β .

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, β), 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 Distributions Template Parameter onemkl::rng::method Values .
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 β , and shape α , 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 Distributions Template Parameter onemkl::rng::method Values .
al-pha	<code>T (float, double)</code>	Shape α
a	<code>T (float, double)</code>	Displacement a .
beta	<code>T (float, double)</code>	Scalefactor β .

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, β), 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 Distributions Template Parameter <code>onemkl::rng::method</code> Values .
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, β), 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 Distributions Template Parameter onemkl::rng::method Values .
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 (m , a) and standard deviation (s , σ) of subject normal distribution, displacement ($displ$, b), and scalefactor ($scale$, β), 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	onemkl::rng::method	Generation method. The specific values are as follows: onemkl::rng::box_muller2 or onemkl::rng::inverse_function See brief descriptions of the methods in Distributions Template Parameter onemkl::rng::method Values .
m	T (float, double)	Average a of the subject normal distribution.
s	T (float, double)	Standard deviation σ of the subject normal distribution.
displ	T (float, double)	Displacement $displ$.
scale	T (float, double)	Scalefactor $scale$.

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, β), 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 Distributions Template Parameter onemkl::rng::method Values .
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 α , displacement a , and scale parameter β , 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 Distributions Template Parameter onemkl::rng::method Values .
alpha	T (float, double)	Shape α
a	T (float, double)	Displacement a.
beta	T (float, double)	Scalefactor β .

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, β) , 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 Distributions Template Parameter onemkl::rng::method Values .
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 ν 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 Distributions Template Parameter onemkl::rng::method Values .
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 Distributions Template Parameter onemkl::rng::method Values .
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:

$$P(X = k) = p \cdot (1 - p)^k, \quad k \in \{0, 1, 2, \dots\}.$$

The cumulative distribution function is as follows:

$$F_p(x) = \begin{cases} 0, & x < 0 \\ 1 - (1 - p)^{\lfloor x + 1 \rfloor}, & 0 \geq x \end{cases} \quad x \in \mathbb{R}.$$

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 Distributions Template Parameter onemkl::rng::method Values .
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 Distributions Template Parameter onemkl::rng::method Values .
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 λ , 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 λ .

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 λ_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	onemkl::rng::method	Generation method. The specific values are as follows: onemkl::rng::gaussian_inverse See brief descriptions of the methods in Distributions Template Parameter onemkl::rng::method Values .
lambda	std::vector<double>	Array of the distribution parameters λ .

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 Distributions Template Parameter onemkl::rng::method Values .
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
- [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
Arithmetic Functions:		
<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
Power and Root Functions:		
<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
Exponential and Logarithmic Functions:		
<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
Trigonometric Functions:		

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 π
sinpi	s, d	Computes the sine of vector elements multiplied by π
tanpi	s, d	Computes the tangent of vector elements multiplied by π
acospi	s, d	Computes the inverse cosine of vector elements divided by π
asinpi	s, d	Computes the inverse sine of vector elements divided by π
atanpi	s, d	Computes the inverse tangent of vector elements divided by π
atan2pi	s, d	Computes the four-quadrant inverse tangent of the ratios of the corresponding elements of two vectors multiplied by π
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 $\pi/180$
Hyperbolic Functions:		
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.
Special Functions:		
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
Rounding Functions:		
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
Miscellaneous Functions:		
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<float></code>
<code>std::complex<double></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 ∞ 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<float></code>
<code>std::complex<double></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 ∞ 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	
$+\infty$	$+\infty$	
$-\infty$	$+\infty$	
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 ∞ 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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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$
$2EMIN/2 \leq a[i] \leq 2(EMAX-2)/2$
$2EMIN/2 \leq b[i] \leq 2(EMAX-2)/2$
$a[i] \neq - (shifta/scalea) * (1-\delta_1), \delta_1 \leq 2^{1-(p-1)/2}$
$b[i] \neq - (shiftb/scaleb) * (1-\delta_2), \delta_2 \leq 2^{1-(p-1)/2}$

`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	± 0	NAN	status::sing
$\pm\infty$	b not NAN	NAN	status::sing
± 0	$b \neq 0$, not NAN	± 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	± 0	NAN	status::errdom
$\pm\infty$	b not NAN	NAN	
± 0	$b \neq 0$, not NAN	± 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 ∞ 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 ∞ 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 ∞ 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	$a_i < (\text{FLT_MAX})^{1/b}$ i
double precision	$a_i < (\text{DBL_MAX})^{1/b}$ i

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
$0 < a < \infty$	± 0	1	
± 0	$-\infty < b < 0$	$+\infty$	
± 0	$-\infty$	$+\infty$	
± 0	$b > 0$	$+0$	
1	$-\infty < b < \infty$	1	
± 0	± 0	NAN	
$+\infty$	± 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	$\text{abs}(a[i]) < \sqrt{\text{FLT_MAX}}$ $\text{abs}(b[i]) < \sqrt{\text{FLT_MAX}}$
double precision	$\text{abs}(a[i]) < \sqrt{\text{DBL_MAX}}$ $\text{abs}(b[i]) < \sqrt{\text{DBL_MAX}}$

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	$a[i] < \text{Log}(\text{FLT_MAX})$
double precision	$a[i] < \text{Log}(\text{DBL_MAX})$

Argument	Result	Error Code
+0	+1	
-0	+1	
a > overflow	+∞	status::overflow
a < underflow	+0	status::overflow
+∞	+∞	
-∞	+0	
QNAN	QNAN	
SNAN	QNAN	

+i·∞					
+i·Y					
+i·0					
-i·0					
-i·Y					
-i·∞					
+i·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<float></code>
<code>std::complex<double></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 = {})
```

log2 supports the following precisions.

T
float
double

Description

The *log2(a)* function computes the base 2 logarithm of vector elements.

Argument	Result	Error Code
+1	+0	
<i>a</i> < +0	QNAN	status::errdom
+0	-∞	status::sing
-0	-∞	status::sing
-∞	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 `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 π .
- [sinpi](#) Computes the sine of vector elements multiplied by π .
- [tanpi](#) Computes the tangent of vector elements multiplied by π .
- [acospi](#) Computes the inverse cosine of vector elements divided by π .
- [asinpi](#) Computes the inverse sine of vector elements divided by π .

- `atanpi` Computes the inverse tangent of vector elements divided by π .
- `atan2pi` Computes the four-quadrant inverse tangent of the ratios of the corresponding elements of two vectors divided by π .
- `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<float></code>
<code>double</code>	<code>std::complex<double></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	$+1-i\cdot 0$	
• ∞	$\text{QNAN}+i\cdot \text{QNAN}$	<code>status::errdom</code>
• ∞	$\text{QNAN}+i\cdot \text{QNAN}$	<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 + $i\cdot\text{QNAN}$
$+i\cdot 0$	$+i\cdot\infty$		$+i\cdot 0$	$+i\cdot 0$		$+0 - i\cdot\infty$	QNAN + $i\cdot\text{QNAN}$
$-i\cdot 0$	$-i\cdot\infty$		$-i\cdot 0$	$-i\cdot 0$		$+0 + i\cdot\infty$	QNAN + $i\cdot\text{QNAN}$
$-i\cdot Y$	$-i\cdot\infty$					$+0 + i\cdot\infty$	QNAN + $i\cdot\text{QNAN}$
$-i\cdot\infty$	$+3\pi/4 + i\cdot\infty$	$+\pi/2 + i\cdot\infty$	$+\pi/2 + i\cdot\infty$	$+\pi/2 + i\cdot\infty$	$+\pi/2 + i\cdot\infty$	$+\pi/4 + i\cdot\infty$	QNAN + $i\cdot\infty$
$+i\cdot\text{NAN}$	QNAN + $i\cdot\infty$	QNAN + $i\cdot\text{QNAN}$	QNAN + $i\cdot\text{QNAN}$	QNAN + $i\cdot\text{QNAN}$	QNAN + $i\cdot\text{QNAN}$	QNAN + $i\cdot\infty$	QNAN + $i\cdot\text{QNAN}$

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	
+∞	$+\pi/2$	
-∞	$-\pi/2$	
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 [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 π .

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 π . 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/vcosp�.cpp
```

Parent topic: Trigonometric Functions

13.2.5.2.4.11 sinpi

Computes the sine of vector elements multiplied by π .

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 π . 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 π .

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 π . 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 π .

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 π . 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
$\bullet \infty$	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 π .

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 π . For an argument a, the function computes asinpi(a)/ π .

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 π .

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 π . 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 π .

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 π .

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 π .

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<float></code>
<code>std::complex<double></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·QNAN1+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·QNAN1-i·0	QNAN+i·QNAN
+i·NAN	-1+i·0	QNAN+i·QNANQNAN+i·QNANQNAN+i·QNANQNAN+i·QNANQNAN+i·QNAN1+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<float></code>
<code>std::complex<double></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<float></code>
<code>std::complex<double></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	QNAN+i·QNAN

Notes:

- $\text{atanh}(\pm 1 \pm i \cdot 0) = \pm \infty \pm i \cdot 0$, and `status::sing` error is generated
- $\text{atanh}(\text{CONJ}(a)) = \text{CONJ}(\text{atanh}(a))$
- $\text{atanh}(-a) = -\text{atanh}(a)$.

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 erfc is the complementary error function.

$$2. \quad \Phi(x) = \frac{1}{2} \text{erf}(x/\sqrt{2}),$$

where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp(-t^2/2) dt$$

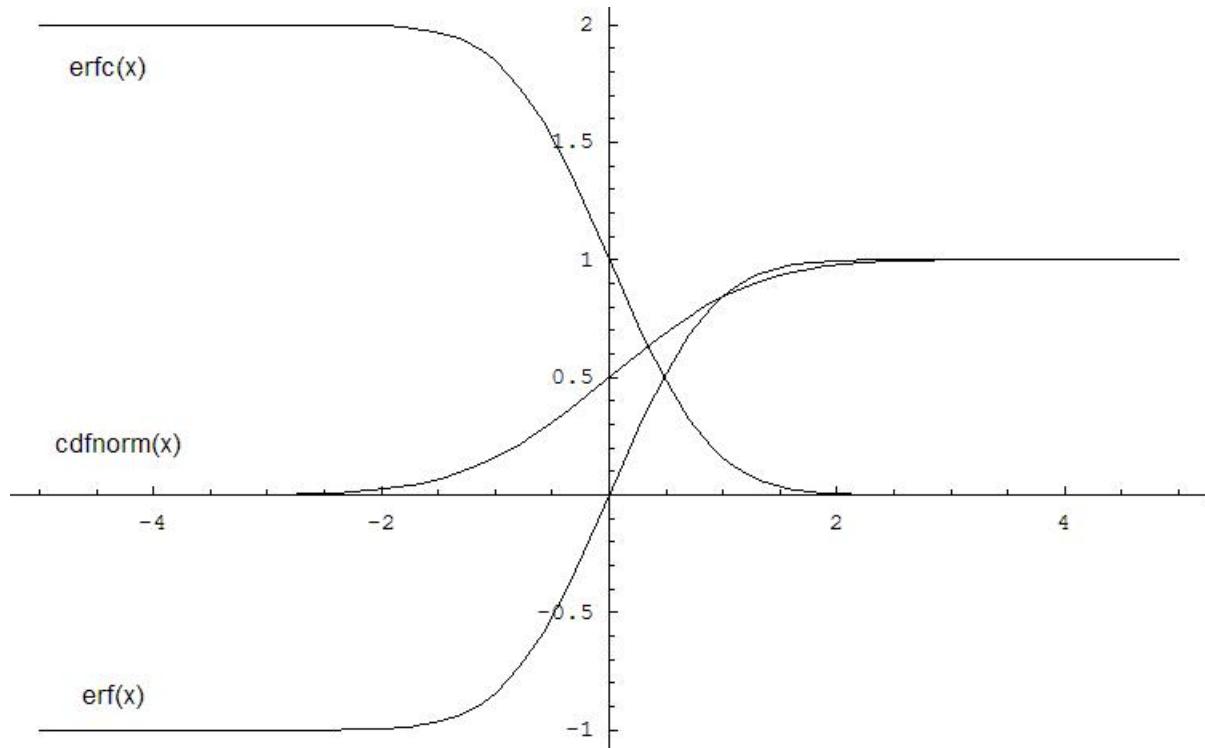
is the cumulative normal distribution function.

$$3. \quad \Phi^{-1}(x) = \sqrt{2} \text{erf}^{-1}(2x - 1),$$

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

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp(-t^2/2) dt$$

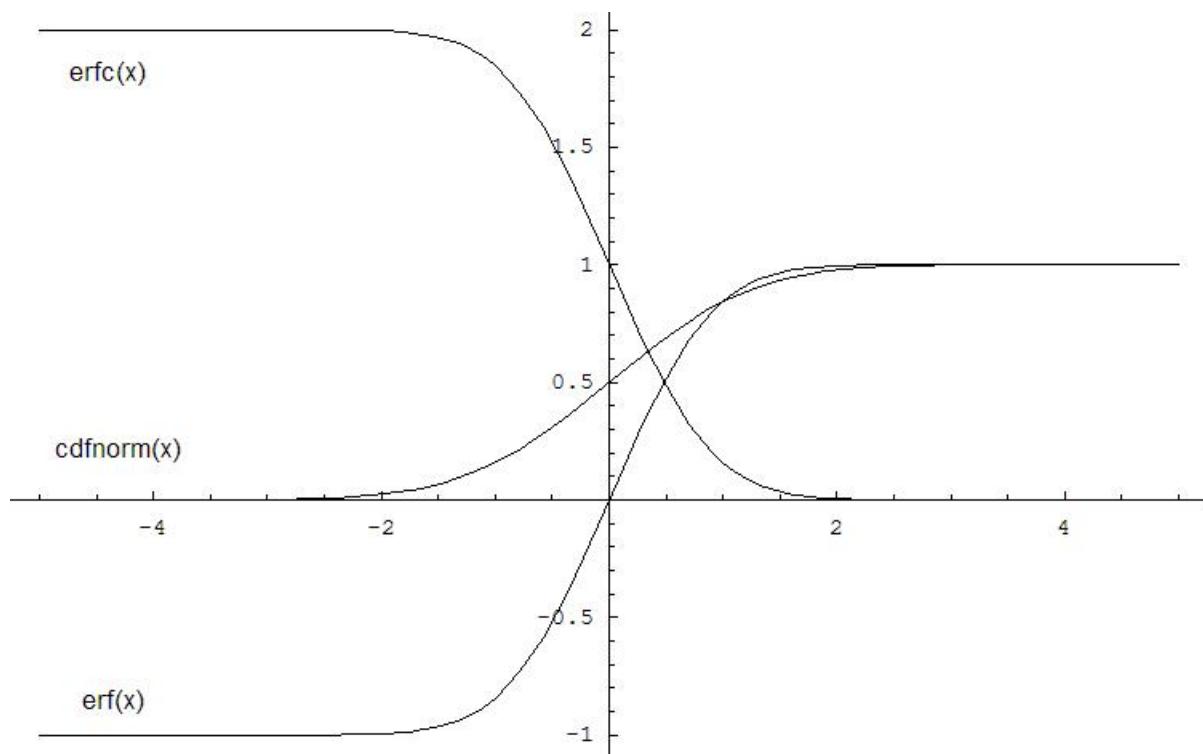
is the cumulative normal distribution function.

$$3. \quad \Phi^{-1}(x) = \sqrt{2} \operatorname{erf}^{-1}(2x - 1),$$

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:

$$\text{CdfNorm}(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{t^2}{2}} dt .$$

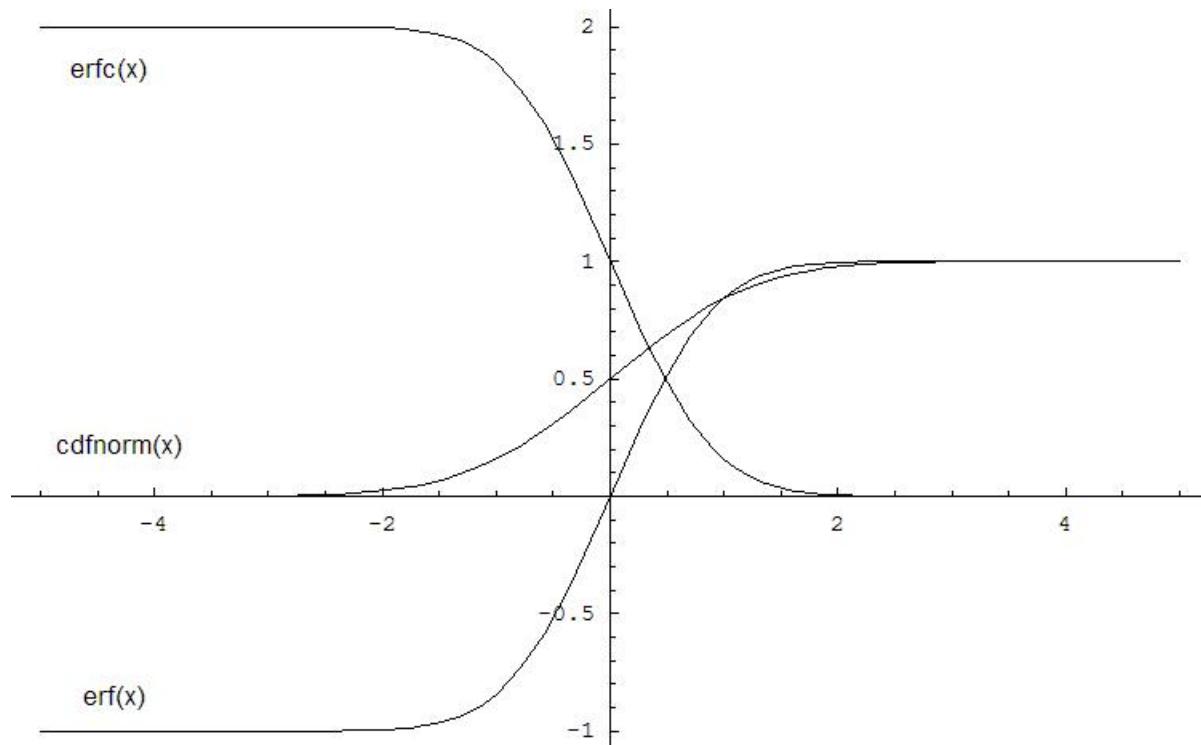
Useful relations:

$$\text{cdfnorm}(x) = \frac{1}{2} \left(1 + \text{erf} \left(\frac{x}{\sqrt{2}} \right) \right) = 1 - \frac{1}{2} \text{erfc} \left(\frac{x}{\sqrt{2}} \right)$$

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

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp(-t^2/2) dt$$

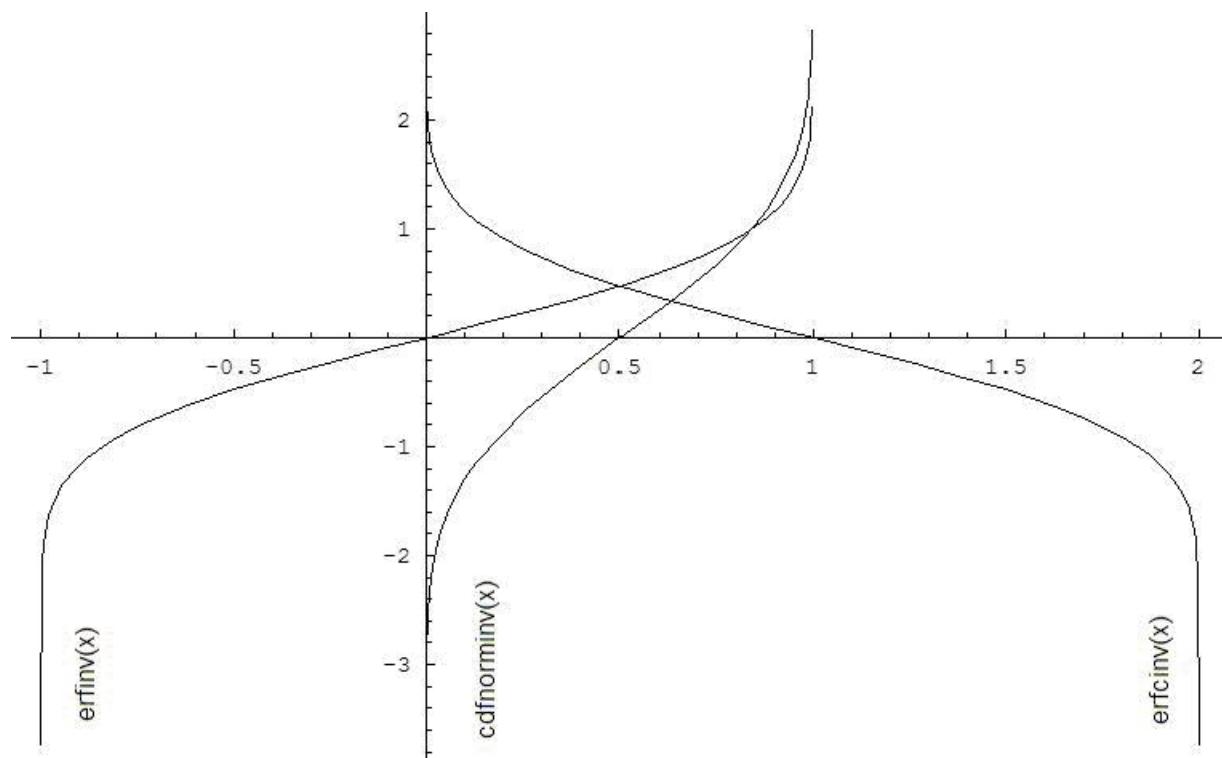
is the cumulative normal distribution function.

$$3. \quad \Phi^{-1}(x) = \sqrt{2} \operatorname{erf}^{-1}(2x - 1),$$

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 (erfinv , $\operatorname{erfcinv}$, $\operatorname{cdfnorminv}$).

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:

$$\text{erfcinv}(x) = \text{erfinv}(1 - x)$$

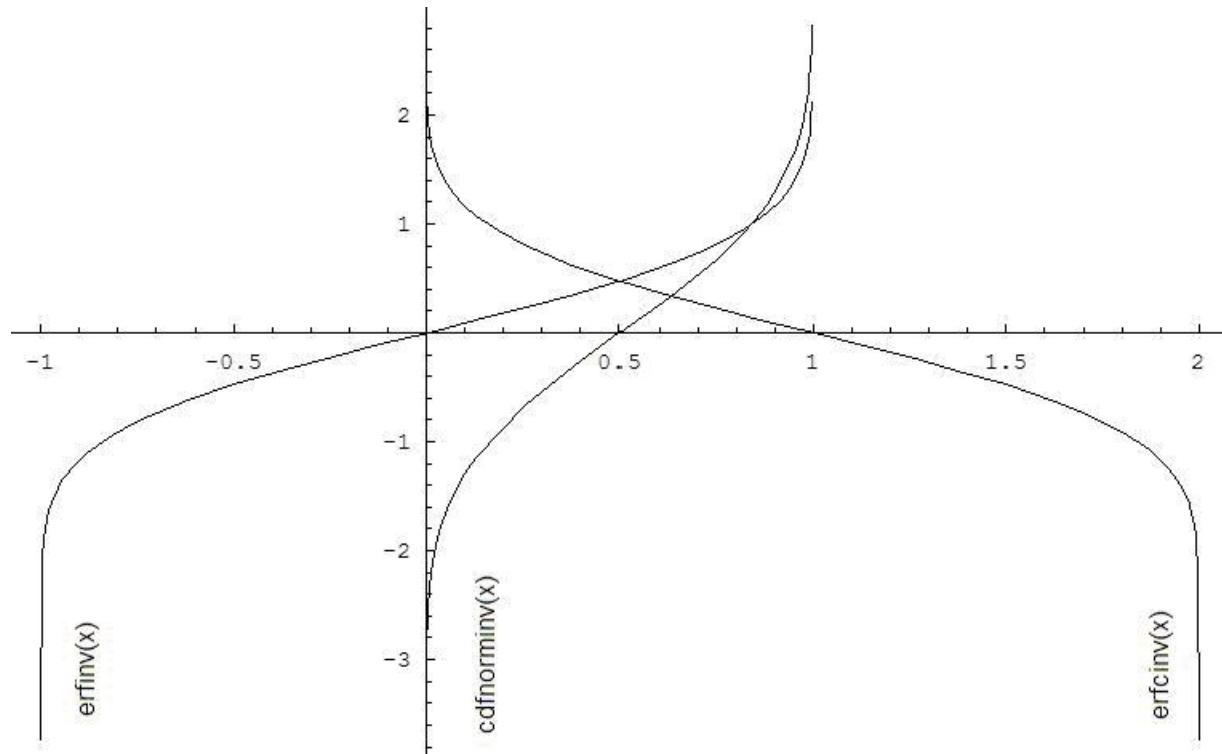
$$\text{erfinv}(x) = \text{erf}^{-1}(x)$$

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

where $\text{erf}(x)$ denotes the error function and $\text{erfinv}(x)$ denotes the inverse error function.

The following figure illustrates the relationships among erfinv family functions (erfinv , erfcinv , 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.

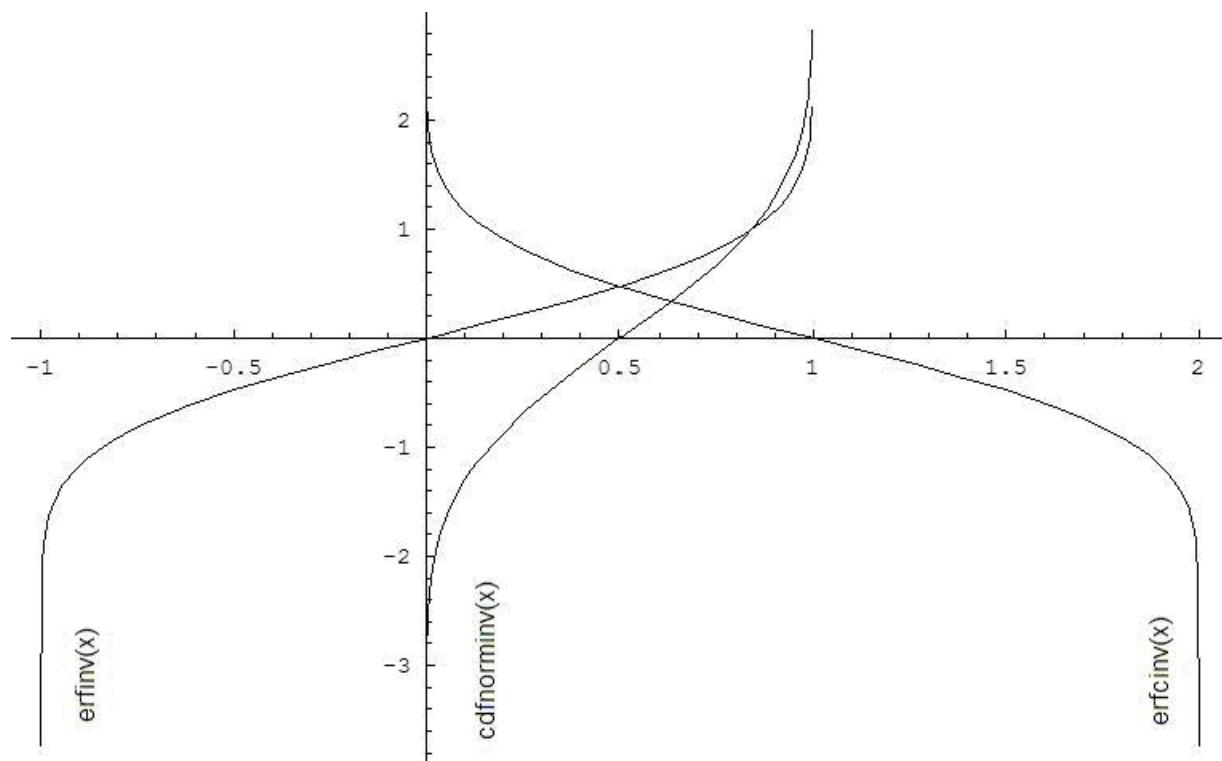
Useful relations:

$$\text{cdfnorminv}(x) = \sqrt{2}\text{erfinv}(2x - 1) = \sqrt{2}\text{erfcinv}(2 - 2x)$$

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
$+\infty$	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 `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 > 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 > 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 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 `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
0.85	06/29/2020	HTML PDF
0.8	05/29/2020	HTML PDF
0.7	03/26/2020	HTML PDF
0.5	11/17/2019	HTML

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

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*), 822
_mfxExtCencParam::Header (*C++ member*), 823
_mfxExtCencParam::StatusReportIndex (*C++ member*), 823
~combinable (*C++ function*), 609
~global_control (*C++ function*), 370
~graph (*C++ function*), 317
~null_mutex (*C++ function*), 634
~null_rw_mutex (*C++ function*), 635
~overwrite_node (*C++ function*), 338
~queuing_mutex (*C++ function*), 632
~queuing_rw_mutex (*C++ function*), 633
~speculative_spin_mutex (*C++ function*), 630
~speculative_spin_rw_mutex (*C++ function*), 631
~spin_mutex (*C++ function*), 628
~spin_rw_mutex (*C++ function*), 629
~split_node (*C++ function*), 354
~task_arena (*C++ function*), 375
~task_group (*C++ function*), 372
~task_group_context (*C++ function*), 369
~task_scheduler_observer (*C++ function*), 378
~write_once_node (*C++ function*), 340

A

abs (*C++ function*), 1196
Accessor, 219
acos (*C++ function*), 1254
acosh (*C++ function*), 1287
acospi (*C++ function*), 1267
activate (*C++ function*), 327
active_value (*C++ function*), 370
add (*C++ function*), 297, 1185
allocate (*C++ function*), 619, 620, 622
allocator_type (*C++ function*), 619
API, 220
arg (*C++ function*), 1197
asin (*C++ function*), 1256
asinh (*C++ function*), 1289
asinpi (*C++ function*), 1269
atan (*C++ function*), 1258

atan2 (*C++ function*), 1259
atan2pi (*C++ function*), 1272
atanh (*C++ function*), 1291
atanpi (*C++ function*), 1271
attach (*C++ struct*), 374
automatic (*C++ member*), 374

B

Batch mode, 219
begin (*C++ function*), 306, 615, 617
blocked_range (*C++ function*), 305
Body::~Body (*C++ function*), 270, 280–282
Body::assign (*C++ function*), 272
Body::Body (*C++ function*), 270, 272, 280–282
Body::join (*C++ function*), 270
Body::operator() (*C++ function*), 270–272, 274, 280–282
Body::reverse_join (*C++ function*), 272
broadcast_node (*C++ function*), 349
buffer_node (*C++ function*), 342
Builder, 219

C

cache_aligned_resource (*C++ function*), 623
cancel (*C++ function*), 373
cancel_group_execution (*C++ function*), 369
canceled (*C macro*), 373
capture_fp_settings (*C++ function*), 369
cast_to (*C++ function*), 362
Categorical feature, 218
cbrt (*C++ function*), 1213
cdfnorm (*C++ function*), 1299
cdfnorminv (*C++ function*), 1308
ceil (*C++ function*), 1318
cis (*C++ function*), 1250
Classification, 218
clear (*C++ function*), 302, 338, 340, 609
clear_status (*C++ function*), 1335
Clustering, 218
combinable (*C++ function*), 608, 609
combine (*C++ function*), 609, 615
Combine::operator() (*C++ function*), 273

combine_each (*C++ function*), 609, 615
 Commit (*C++ function*), 1122
 compete (*C macro*), 373
 composite_node (*C++ function*), 358
 computeBackward (*C++ function*), 1125
 computeForward (*C++ function*), 1123
 conj (*C++ function*), 1194
 const_iterator (*C++ type*), 305
 Contiguous data, 219
 Continuous feature, 218
 copysign (*C++ function*), 1340
 cos (*C++ function*), 1245
 cosd (*C++ function*), 1275
 cosh (*C++ function*), 1280
 cospi (*C++ function*), 1262
 CR::begin (*C++ function*), 279
 CR::const_reference (*C++ type*), 279
 CR::difference_type (*C++ type*), 279
 CR::end (*C++ function*), 279
 CR::grainsize (*C++ function*), 279
 CR::iterator (*C++ type*), 279
 CR::reference (*C++ type*), 279
 CR::size_type (*C++ type*), 279
 CR::value_type (*C++ type*), 279
 create_error_handler (*C++ function*), 1336
 current_thread_index (*C++ function*), 377

D

Data format, 219
 Data layout, 219
 Data type, 219
 Dataset, 218
 deallocate (*C++ function*), 619, 620, 622
 descriptor (*C++ member*), 1117
 div (*C++ function*), 1207
 dnnl::algorithm (*C++ enum*), 53
 dnnl::algorithm::binary_add (*C++ enumerator*), 55
 dnnl::algorithm::binary_max (*C++ enumerator*), 55
 dnnl::algorithm::binary_min (*C++ enumerator*), 55
 dnnl::algorithm::binary_mul (*C++ enumerator*), 55
 dnnl::algorithm::convolution_auto (*C++ enumerator*), 53
 dnnl::algorithm::convolution_direct (*C++ enumerator*), 53
 dnnl::algorithm::convolution_winograd (*C++ enumerator*), 53
 dnnl::algorithm::deconvolution_direct (*C++ enumerator*), 53
 dnnl::algorithm::deconvolution_winograd (*C++ enumerator*), 53

dnnl::algorithm::eltwise_abs (*C++ enumerator*), 54
 dnnl::algorithm::eltwise_bounded_relu (*C++ enumerator*), 54
 dnnl::algorithm::eltwise_clip (*C++ enumerator*), 54
 dnnl::algorithm::eltwise_elu (*C++ enumerator*), 53
 dnnl::algorithm::eltwise_elu_use_dst_for_bwd (*C++ enumerator*), 54
 dnnl::algorithm::eltwise_exp (*C++ enumerator*), 54
 dnnl::algorithm::eltwise_exp_use_dst_for_bwd (*C++ enumerator*), 54
 dnnl::algorithm::eltwise_gelu (*C++ enumerator*), 54
 dnnl::algorithm::eltwise_gelu_erf (*C++ enumerator*), 54
 dnnl::algorithm::eltwise_gelu_tanh (*C++ enumerator*), 54
 dnnl::algorithm::eltwise_linear (*C++ enumerator*), 54
 dnnl::algorithm::eltwise_log (*C++ enumerator*), 54
 dnnl::algorithm::eltwise_logistic (*C++ enumerator*), 54
 dnnl::algorithm::eltwise_logistic_use_dst_for_bwd (*C++ enumerator*), 54
 dnnl::algorithm::eltwise_pow (*C++ enumerator*), 54
 dnnl::algorithm::eltwise_relu (*C++ enumerator*), 53
 dnnl::algorithm::eltwise_relu_use_dst_for_bwd (*C++ enumerator*), 54
 dnnl::algorithm::eltwise_soft_relu (*C++ enumerator*), 54
 dnnl::algorithm::eltwise_sqrt (*C++ enumerator*), 54
 dnnl::algorithm::eltwise_sqrt_use_dst_for_bwd (*C++ enumerator*), 54
 dnnl::algorithm::eltwise_square (*C++ enumerator*), 54
 dnnl::algorithm::eltwise_swish (*C++ enumerator*), 54
 dnnl::algorithm::eltwise_tanh (*C++ enumerator*), 53
 dnnl::algorithm::eltwise_tanh_use_dst_for_bwd (*C++ enumerator*), 54
 dnnl::algorithm::lbr_gru (*C++ enumerator*), 55
 dnnl::algorithm::lrn_across_channels (*C++ enumerator*), 55
 dnnl::algorithm::lrn_within_channel (*C++ enumerator*), 55

dnnl::algorithm::pooling_avg (*C++ enumerator*), 55
dnnl::algorithm::pooling_avg_exclude_padding (*C++ enumerator*), 55
dnnl::algorithm::pooling_avg_include_padding (*C++ enumerator*), 55
dnnl::algorithm::pooling_max (*C++ enumerator*), 55
dnnl::algorithm::resampling_linear (*C++ enumerator*), 55
dnnl::algorithm::resampling_nearest (*C++ enumerator*), 55
dnnl::algorithm::undef (*C++ enumerator*), 53
dnnl::algorithm::vanilla_gru (*C++ enumerator*), 55
dnnl::algorithm::vanilla_lstm (*C++ enumerator*), 55
dnnl::algorithm::vanilla_rnn (*C++ enumerator*), 55
dnnl::batch_normalization_backward (*C++ struct*), 78
dnnl::batch_normalization_backward::batch_norm_backward_desc (*C++ function*), 78
dnnl::batch_normalization_backward::descdnnl::binary (*C++ struct*), 82
dnnl::batch_normalization_backward::descdnnl::binary::binary (*C++ function*), 82
dnnl::batch_normalization_backward::descdnnl::binary::desc (*C++ struct*), 82
dnnl::batch_normalization_backward::descdnnl::binary::primitive_desc (*C++ function*), 83
dnnl::batch_normalization_backward::primdnnl::binary::primitive_desc::primitive_desc (*C++ struct*), 79
dnnl::batch_normalization_backward::primdnnl::binary::primitive_desc::src0_desc (*C++ function*), 80
dnnl::batch_normalization_backward::primdnnl::binary::primitive_desc::src1_desc (*C++ function*), 80
dnnl::batch_normalization_backward::primdnnl::binary::primitive_desc::src_desc (*C++ function*), 80
dnnl::batch_normalization_backward::primdnnl::descat (*C++ struct*), 84
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*), 79
dnnl::batch_normalization_backward::primdnnl::descat::primitive_desc::src_desc (*C++ function*), 80
dnnl::batch_normalization_backward::primdnnl::descat::primitive_desc::src_desc (*C++ function*), 80
dnnl::batch_normalization_backward::primdnnl::descat::rprimitive_desc::primitive_desc (*C++ function*), 85
dnnl::batch_normalization_backward::primdnnl::descat::vprimitive_desc::src_desc (*C++ function*), 85
dnnl::batch_normalization_backward::primdnnl::descat::weight_backward_data (*C++ struct*), 94
dnnl::batch_normalization_backward::primdnnl::descat::weight_backward_data::convolution_backward_data (*C++ function*), 94
dnnl::batch_normalization_forward (*C++ struct*), 77
dnnl::batch_normalization_forward::batch_norm_forward_desc (*C++ function*), 77
dnnl::batch_normalization_forward::desc dnnl::convolution_backward_data::desc (*C++ struct*), 94
dnnl::batch_normalization_forward::desc::desc dnnl::convolution_backward_data::primitive_desc (*C++ function*), 94, 95

(C++ struct), 95 dnnl::deconvolution_backward_data::desc::desc
dnnl::convolution_backward_data::primitive_desc(C++ function), 103
(C++ function), 96 dnnl::deconvolution_backward_data::primitive_desc
dnnl::convolution_backward_data::primitive_desc(C++ function), 104
(C++ function), 96 dnnl::deconvolution_backward_data::primitive_desc
dnnl::convolution_backward_data::primitive_desc(C++ function), 105
(C++ function), 95, 96 dnnl::deconvolution_backward_data::primitive_desc
dnnl::convolution_backward_data::primitive_desc(C++ function), 104
(C++ function), 96 dnnl::deconvolution_backward_data::primitive_desc:
dnnl::convolution_backward_weights(C++ struct), 96 (C++ function), 104
dnnl::convolution_backward_weights::convolution(C++ function), 104
(C++ function), 96 dnnl::deconvolution_backward_weights
dnnl::convolution_backward_weights::desc (C++ struct), 105
(C++ struct), 96 dnnl::deconvolution_backward_weights::deconvolution
dnnl::convolution_backward_weights::desc::desc(C++ function), 105
(C++ function), 97, 98 dnnl::deconvolution_backward_weights::desc
dnnl::convolution_backward_weights::primitive_desc(C++ struct), 105
(C++ struct), 98 dnnl::deconvolution_backward_weights::desc::desc
dnnl::convolution_backward_weights::primitive_desc(C++ function), 105
(C++ function), 99 dnnl::deconvolution_backward_weights::primitive_desc
dnnl::convolution_backward_weights::primitive_desc(C++ struct), 107
(C++ function), 99 dnnl::deconvolution_backward_weights::primitive_desc
dnnl::convolution_backward_weights::primitive_desc(C++ function), 108
(C++ function), 99 dnnl::deconvolution_backward_weights::primitive_desc
dnnl::convolution_backward_weights::primitive_desc(C++ function), 108
(C++ function), 99 dnnl::deconvolution_backward_weights::primitive_desc
dnnl::convolution_forward(C++ struct), 91 (C++ function), 107
dnnl::convolution_forward::convolution_foward::deconvolution_backward_weights::primitive_desc
(C++ function), 91 (C++ function), 107
dnnl::convolution_forward::desc(C++ struct), 91 dnnl::deconvolution_forward(C++ struct),
dnnl::convolution_forward::desc(C++ function), 91-93 (C++ function), 100
dnnl::convolution_forward::primitive_desdnnl::deconvolution_forward::desc(C++
struct), 93 (C++ struct), 100
dnnl::convolution_forward::primitive_desdnnl::bias_desvolution_forward::desc(C++
function), 94 (C++ function), 100, 101
dnnl::convolution_forward::primitive_desdnnl::dst_desavolution_forward::primitive_desc
(C++ function), 94 (C++ struct), 102
dnnl::convolution_forward::primitive_desdnnl::primetime_desation_forward::primitive_desc::bias_
(C++ function), 93 (C++ function), 102
dnnl::convolution_forward::primitive_desdnnl::src_desavolution_forward::primitive_desc::dst_de
(C++ function), 94 (C++ function), 102
dnnl::convolution_forward::primitive_desdnnl::weights_desuton_forward::primitive_desc::primit
(C++ function), 94 (C++ function), 102
dnnl::deconvolution_backward_data(C++ struct), 103 dnnl::deconvolution_forward::primitive_desc::src_d
dnnl::deconvolution_backward_data::deconvolution_backward_data::forward::primitive_desc::weight
(C++ function), 103 (C++ function), 102
dnnl::deconvolution_backward_data::desc dnnl::eltwise_backward(C++ struct), 113
(C++ struct), 103 dnnl::eltwise_backward::desc(C++ struct),

```

    113
dnnl::eltwise_backward::desc::desc (C++ function), 114
dnnl::eltwise_backward::eltwise_backward (C++ function), 113
dnnl::eltwise_backward::primitive_desc (C++ struct), 114
dnnl::eltwise_backward::primitive_desc (C++ function), 115
dnnl::eltwise_backward::primitive_desc::diff_desc (C++ function), 114
dnnl::eltwise_backward::primitive_desc::dst_iter_desc (C++ function), 113
dnnl::eltwise_backward::primitive_desc::dst_layer_desc (C++ function), 113
dnnl::eltwise_backward::primitive_desc::src_desc (C++ function), 113
dnnl::eltwise_backward::primitive_desc::src_layer_desc (C++ function), 113
dnnl::eltwise_forward (C++ struct), 112
dnnl::eltwise_forward::desc (C++ struct), 112
dnnl::eltwise_forward::desc (C++ function), 112
dnnl::eltwise_forward::eltwise_forward (C++ function), 112
dnnl::eltwise_forward::primitive_desc (C++ struct), 112
dnnl::eltwise_forward::primitive_desc (C++ function), 113
dnnl::eltwise_forward::primitive_desc::dst_desc (C++ function), 113
dnnl::eltwise_forward::primitive_desc::dst_desc (C++ function), 113
dnnl::eltwise_forward::primitive_desc::src_desc (C++ function), 113
dnnl::eltwise_forward::primitive_desc::src_desc (C++ function), 113
dnnl::engine (C++ struct), 24
dnnl::engine::engine (C++ function), 24
dnnl::engine::get_count (C++ function), 25
dnnl::engine::get_kind (C++ function), 24
dnnl::engine::get_sycl_context (C++ function), 25
dnnl::engine::get_sycl_device (C++ function), 25
dnnl::engine::kind (C++ enum), 24
dnnl::engine::kind::any (C++ enumerator), 24
dnnl::engine::kind::cpu (C++ enumerator), 24
dnnl::engine::kind::gpu (C++ enumerator), 24
dnnl::error (C++ struct), 21
dnnl::gru_backward (C++ struct), 181
dnnl::gru_backward::desc (C++ struct), 181
dnnl::gru_backward::desc::desc (C++ function), 181
dnnl::gru_backward::gru_backward (C++ function), 181
dnnl::gru_backward::primitive_desc (C++ struct), 182
dnnl::gru_backward::primitive_desc::bias_desc (C++ function), 183
dnnl::gru_backward::primitive_desc::diff_bias_desc (C++ function), 183
dnnl::gru_backward::primitive_desc::diff_dst_iter_desc (C++ function), 183
dnnl::gru_backward::primitive_desc::diff_dst_layer_desc (C++ function), 183
dnnl::gru_backward::primitive_desc::diff_src_iter_desc (C++ function), 183
dnnl::gru_backward::primitive_desc::diff_src_layer_desc (C++ function), 183
dnnl::gru_backward::primitive_desc::dst_desc (C++ function), 183
dnnl::gru_backward::primitive_desc::dst_layer_desc (C++ function), 183
dnnl::gru_backward::primitive_desc::src_desc (C++ function), 183
dnnl::gru_backward::primitive_desc::src_layer_desc (C++ function), 183
dnnl::gru_backward::primitive_desc::weights_desc (C++ function), 183
dnnl::gru_backward::primitive_desc::weights_iter_desc (C++ function), 183
dnnl::gru_backward::primitive_desc::workspace_desc (C++ function), 183
dnnl::gru_forward (C++ struct), 179
dnnl::gru_forward::desc (C++ struct), 179
dnnl::gru_forward::desc::desc (C++ function), 179
dnnl::gru_forward::gru_forward (C++ function), 179
dnnl::gru_forward::primitive_desc (C++ struct), 179
dnnl::gru_forward::primitive_desc::bias_desc (C++ function), 180
dnnl::gru_forward::primitive_desc::dst_iter_desc (C++ function), 180
dnnl::gru_forward::primitive_desc::dst_layer_desc (C++ function), 180
dnnl::gru_forward::primitive_desc::src_desc (C++ function), 180
dnnl::gru_forward::primitive_desc::src_layer_desc (C++ function), 180
dnnl::gru_forward::primitive_desc::weights_desc (C++ function), 180

```

```

dnnl::gru_forward::primitive_desc::weights_dnnl::layer::product_forward::primitive_desc::dst_desc
    (C++ function), 180                               (C++ function), 119
dnnl::gru_forward::primitive_desc::workspans_dnnl::inner_product_forward::primitive_desc::primitive_desc
    (C++ function), 180                               (C++ function), 118
dnnl::inner_product_backward_data (C++ dnnl::inner_product_forward::primitive_desc::src_desc
    struct), 119                                     (C++ function), 118
dnnl::inner_product_backward_data::desc dnnl::inner_product_forward::primitive_desc::weights_dnnl::layer::product_forward::primitive_desc::dst_desc
    (C++ struct), 119                               (C++ function), 119
dnnl::inner_product_backward_data::desc::desc dnnl::layer_normalization_backward (C++ struct)
    (C++ function), 119                               (C++ struct), 127
dnnl::inner_product_backward_data::inner_product_backward::backward::desc
    (C++ function), 119                               (C++ struct), 127
dnnl::inner_product_backward_data::primitive_desc::desc dnnl::layer_normalization_backward::desc::desc
    (C++ struct), 119                               (C++ function), 127, 128
dnnl::inner_product_backward_data::primitive_desc::desc dnnl::layer_normalization_backward::desc::desc
    (C++ function), 120                               (C++ function), 127
dnnl::inner_product_backward_data::primitive_desc::desc dnnl::layer_normalization_backward::desc::desc
    (C++ struct), 120                               (C++ function), 128
dnnl::inner_product_backward_data::primitive_desc::desc dnnl::layer_normalization_backward::desc::desc
    (C++ function), 120                               (C++ function), 129
dnnl::inner_product_backward_data::primitive_desc::desc dnnl::layer_normalization_backward::desc::desc
    (C++ struct), 120                               (C++ function), 129
dnnl::inner_product_backward_weights      dnnl::layer_normalization_backward::primitive_desc
    (C++ struct), 120                               (C++ function), 129
dnnl::inner_product_backward_weights::desc dnnl::layer_normalization_backward::primitive_desc
    (C++ struct), 121                               (C++ function), 128
dnnl::inner_product_backward_weights::desc dnnl::layer_normalization_backward::primitive_desc
    (C++ function), 121                               (C++ function), 129
dnnl::inner_product_backward_weights::desc dnnl::layer_normalization_backward::primitive_desc
    (C++ struct), 121                               (C++ function), 128
dnnl::inner_product_backward_weights::desc dnnl::layer_normalization_backward::primitive_desc
    (C++ function), 122                               (C++ function), 129
dnnl::inner_product_backward_weights::desc dnnl::layer_normalization_backward::primitive_desc
    (C++ struct), 122                               (C++ function), 128
dnnl::inner_product_backward_weights::desc dnnl::layer_normalization_backward::primitive_desc
    (C++ function), 122                               (C++ function), 129
dnnl::inner_product_backward_weights::desc dnnl::layer_normalization_backward::primitive_desc
    (C++ struct), 122                               (C++ function), 128
dnnl::inner_product_backward_weights::desc dnnl::layer_normalization_backward::primitive_desc
    (C++ function), 122                               (C++ function), 129
dnnl::inner_product_backward_weights::desc dnnl::layer_normalization_forward (C++ struct)
    (C++ function), 121, 122                         (C++ function), 125
dnnl::inner_product_backward_weights::desc dnnl::layer_normalization_forward::desc
    (C++ function), 122                               (C++ struct), 125
dnnl::inner_product_forward (C++ struct), dnnl::layer_normalization_forward::desc::desc
    117                                         (C++ function), 126
dnnl::inner_product_forward::desc (C++ dnnl::layer_normalization_forward::desc::desc
    struct), 117                                     (C++ function), 125
dnnl::inner_product_forward::desc dnnl::layer_normalization_forward::primitive_desc
    (C++ function), 118                               (C++ struct), 126
dnnl::inner_product_forward::inner_product dnnl::layer_normalization_forward::primitive_desc
    (C++ function), 117                               (C++ function), 127
dnnl::inner_product_forward::primitive_desc dnnl::layer_normalization_forward::primitive_desc
    (C++ struct), 118                               (C++ function), 127
dnnl::inner_product_forward::primitive_desc dnnl::layer_normalization_forward::primitive_desc
    (C++ function), 119                               (C++ function), 126

```


dnnl::lrn_backward (*C++ struct*), 137
dnnl::lrn_backward::desc (*C++ struct*), 137
dnnl::lrn_backward::desc::desc (*C++ function*), 137
dnnl::lrn_backward::lrn_backward (*C++ function*), 137
dnnl::lrn_backward::primitive_desc (*C++ struct*), 137
dnnl::lrn_backward::primitive_desc::diff_dst::dst::lstm_backward::primitive_desc::dst_iter_desc (*C++ function*), 138
dnnl::lrn_backward::primitive_desc::diff_dst::dst::lstm_backward::primitive_desc::dst_layer_desc (*C++ function*), 138
dnnl::lrn_backward::primitive_desc::prim_dmv::dst::lstm_backward::primitive_desc::primitive_desc (*C++ function*), 138
dnnl::lrn_backward::primitive_desc::workspace::dst::lstm_backward::primitive_desc::src_iter_c_desc (*C++ function*), 138
dnnl::lrn_forward (*C++ struct*), 135
dnnl::lrn_forward::desc (*C++ struct*), 136
dnnl::lrn_forward::desc::desc (*C++ function*), 136
dnnl::lrn_forward::lrn_forward (*C++ function*), 136
dnnl::lrn_forward::primitive_desc (*C++ struct*), 136
dnnl::lrn_forward::primitive_desc::dst_desc::lstm_backward::primitive_desc::workspace_desc (*C++ function*), 137
dnnl::lrn_forward::primitive_desc::prim_dmv::dst::lstm_forward (*C++ struct*), 169
(C++ function), 136
dnnl::lrn_forward::primitive_desc::src_desc::lstm_forward::desc::desc (*C++ function*), 137
dnnl::lrn_forward::primitive_desc::workspace_desc::dst_desc::lstm_forward::lstm_forward (*C++ function*), 169
dnnl::lstm_backward (*C++ struct*), 173
dnnl::lstm_backward::desc (*C++ struct*), 173
dnnl::lstm_backward::desc::desc (*C++ function*), 173, 174, 176
dnnl::lstm_backward::lstm_backward (*C++ function*), 173
dnnl::lstm_backward::primitive_desc (*C++ struct*), 177
dnnl::lstm_backward::primitive_desc::bias_desc::lstm_forward::primitive_desc::dst_layer_desc (*C++ function*), 177
dnnl::lstm_backward::primitive_desc::diff_bias_desc::forward::primitive_desc::primitive_desc (*C++ function*), 178
dnnl::lstm_backward::primitive_desc::diff_dst::dst::lstm_forward::primitive_desc::src_iter_c_desc (*C++ function*), 178
dnnl::lstm_backward::primitive_desc::diff_dst::dst::lstm_forward::primitive_desc::src_iter_desc (*C++ function*), 178
dnnl::lstm_backward::primitive_desc::diff_dst::layer_desc::forward::primitive_desc::src_layer_desc (*C++ function*), 178
dnnl::lstm_backward::primitive_desc::diff_dst::src_desc::lstm_forward::primitive_desc::weights_iter_desc (*C++ function*), 178
dnnl::lstm_backward::primitive_desc::diff_dst::src_desc::lstm_forward::primitive_desc::weights_layer_desc (*C++ function*), 178
dnnl::lstm_backward::primitive_desc::diff_src_layer_desc::forward::primitive_desc::diff_weights_desc (*C++ function*), 178
dnnl::lstm_backward::primitive_desc::diff_weights_desc::forward::primitive_desc::dst_iter_desc (*C++ function*), 178
dnnl::lstm_backward::primitive_desc::dst_iter_c_desc::forward::primitive_desc::dst_layer_desc (*C++ function*), 178
dnnl::lstm_backward::primitive_desc::src_iter_desc::forward::primitive_desc::src_layer_desc (*C++ function*), 178
dnnl::lstm_forward::desc (*C++ struct*), 169
(C++ function), 170, 171
dnnl::lstm_forward::desc::desc (*C++ function*), 170, 171
dnnl::lstm_forward::primitive_desc::dst_desc::lstm_forward::lstm_forward (*C++ function*), 172
dnnl::lstm_forward::primitive_desc::bias_desc (*C++ function*), 172
dnnl::lstm_forward::primitive_desc::dst_iter_c_desc (*C++ function*), 173
dnnl::lstm_forward::primitive_desc::dst_iter_desc (*C++ function*), 172
dnnl::lstm_forward::primitive_desc::dst_layer_desc (*C++ function*), 172
dnnl::lstm_forward::primitive_desc::forward::primitive_desc::primitive_desc (*C++ function*), 172
dnnl::lstm_forward::primitive_desc::src_iter_c_desc (*C++ function*), 172
dnnl::lstm_forward::primitive_desc::src_iter_desc (*C++ function*), 172
dnnl::lstm_forward::primitive_desc::src_layer_desc (*C++ function*), 172
dnnl::lstm_forward::primitive_desc::weights_iter_desc (*C++ function*), 172
dnnl::lstm_forward::primitive_desc::weights_layer_desc (*C++ function*), 172

dnnl::lstm_forward::primitive_desc::worksp~~ate~~:memory::dim (C++ type), 29
(C++ function), 173

dnnl::matmul (C++ struct), 141

dnnl::matmul::desc (C++ struct), 141

dnnl::matmul::desc (C++ function), 141

dnnl::matmul::matmul (C++ function), 141

dnnl::matmul::primitive_desc (C++ struct), 141

dnnl::matmul::primitive_desc::bias_desc (C++ function), 142

dnnl::matmul::primitive_desc::dst_desc (C++ function), 142

dnnl::matmul::primitive_desc::primitive_desc (C++ function), 142

dnnl::matmul::primitive_desc::src_desc (C++ function), 142

dnnl::matmul::primitive_desc::weights_desc (C++ function), 142

dnnl::memory (C++ struct), 40

dnnl::memory::data_type (C++ enum), 27

dnnl::memory::data_type::bf16 (C++ enumerator), 27

dnnl::memory::data_type::f16 (C++ enumerator), 27

dnnl::memory::data_type::f32 (C++ enumerator), 27

dnnl::memory::data_type::s32 (C++ enumerator), 27

dnnl::memory::data_type::s8 (C++ enumerator), 27

dnnl::memory::data_type::u8 (C++ enumerator), 27

dnnl::memory::data_type::undef (C++ enumerator), 27

dnnl::memory::desc (C++ struct), 37

dnnl::memory::desc::data_type (C++ function), 39

dnnl::memory::desc::desc (C++ function), 37, 38

dnnl::memory::desc::dims (C++ function), 39

dnnl::memory::desc::get_size (C++ function), 40

dnnl::memory::desc::is_zero (C++ function), 40

dnnl::memory::desc::operator!= (C++ function), 40

dnnl::memory::desc::operator== (C++ function), 40

dnnl::memory::desc::permute_axes (C++ function), 39

dnnl::memory::desc::reshape (C++ function), 38

dnnl::memory::desc::submemory_desc (C++ function), 38

dnnl::memory::format_tag::dim (C++ type), 29

dnnl::memory::dims (C++ type), 29

dnnl::memory::format_tag (C++ enum), 32

dnnl::memory::format_tag::a (C++ enumerator), 33

dnnl::memory::format_tag::ab (C++ enumerator), 33

dnnl::memory::format_tag::abc (C++ enumerator), 33

dnnl::memory::format_tag::abcd (C++ enumerator), 33

dnnl::memory::format_tag::abcde (C++ enumerator), 34

dnnl::memory::format_tag::abcdef (C++ enumerator), 34

dnnl::memory::format_tag::abdc (C++ enumerator), 33

dnnl::memory::format_tag::abdec (C++ enumerator), 34

dnnl::memory::format_tag::acb (C++ enumerator), 33

dnnl::memory::format_tag::acbde (C++ enumerator), 34

dnnl::memory::format_tag::acbdef (C++ enumerator), 34

dnnl::memory::format_tag::acdb (C++ enumerator), 33

dnnl::memory::format_tag::acdeb (C++ enumerator), 34

dnnl::memory::format_tag::any (C++ enumerator), 33

dnnl::memory::format_tag::ba (C++ enumerator), 33

dnnl::memory::format_tag::bac (C++ enumerator), 33

dnnl::memory::format_tag::bacd (C++ enumerator), 33

dnnl::memory::format_tag::bca (C++ enumerator), 33

dnnl::memory::format_tag::bcda (C++ enumerator), 33

dnnl::memory::format_tag::bcdea (C++ enumerator), 34

dnnl::memory::format_tag::cba (C++ enumerator), 33

dnnl::memory::format_tag::cdab (C++ enumerator), 34

dnnl::memory::format_tag::cdeba (C++ enumerator), 34

dnnl::memory::format_tag::chwn (C++ enumerator), 35

dnnl::memory::format_tag::cn (C++ enumerator), 34

dnnl::memory::format_tag::dcab (C++ enum-

<i>merator), 34</i>		<i>enumerator), 35</i>
dnnl::memory::format_tag::decab <i>(C++ enumerator), 34</i>		dnnl::memory::format_tag::nhwc (<i>C++ enumerator</i>), 34
dnnl::memory::format_tag::defcab <i>(C++ enumerator), 34</i>		dnnl::memory::format_tag::nt (<i>C++ enumerator</i>), 34
dnnl::memory::format_tag::dhwigo <i>(C++ enumerator), 36</i>		dnnl::memory::format_tag::ntc (<i>C++ enumerator</i>), 36
dnnl::memory::format_tag::dhwio <i>(C++ enumerator), 35</i>		dnnl::memory::format_tag::nwc (<i>C++ enumerator</i>), 34
dnnl::memory::format_tag::giodhw <i>(C++ enumerator), 36</i>		dnnl::memory::format_tag::odhwi (<i>C++ enumerator</i>), 35
dnnl::memory::format_tag::giohw <i>(C++ enumerator), 36</i>		dnnl::memory::format_tag::ohwi (<i>C++ enumerator</i>), 35
dnnl::memory::format_tag::goidhw <i>(C++ enumerator), 36</i>		dnnl::memory::format_tag::oi (<i>C++ enumerator</i>), 35
dnnl::memory::format_tag::goihw <i>(C++ enumerator), 35</i>		dnnl::memory::format_tag::oidhw (<i>C++ enumerator</i>), 35
dnnl::memory::format_tag::goiw (<i>C++ enumerator</i>), 35		dnnl::memory::format_tag::oihw (<i>C++ enumerator</i>), 35
dnnl::memory::format_tag::hwigo <i>(C++ enumerator), 36</i>		dnnl::memory::format_tag::oiw (<i>C++ enumerator</i>), 35
dnnl::memory::format_tag::hwio (<i>C++ enumerator</i>), 35		dnnl::memory::format_tag::owi (<i>C++ enumerator</i>), 35
dnnl::memory::format_tag::idhwo <i>(C++ enumerator), 35</i>		dnnl::memory::format_tag::tn (<i>C++ enumerator</i>), 34
dnnl::memory::format_tag::ihwo (<i>C++ enumerator</i>), 35		dnnl::memory::format_tag::tnc (<i>C++ enumerator</i>), 36
dnnl::memory::format_tag::io (<i>C++ enumerator</i>), 35		dnnl::memory::format_tag::undef (<i>C++ enumerator</i>), 33
dnnl::memory::format_tag::iohw (<i>C++ enumerator</i>), 35		dnnl::memory::format_tag::wigo (<i>C++ enumerator</i>), 35
dnnl::memory::format_tag::iwo (<i>C++ enumerator</i>), 35		dnnl::memory::format_tag::wio (<i>C++ enumerator</i>), 35
dnnl::memory::format_tag::ldgo (<i>C++ enumerator</i>), 36		dnnl::memory::format_tag::x (<i>C++ enumerator</i>), 34
dnnl::memory::format_tag::ldgoi <i>(C++ enumerator), 36</i>		dnnl::memory::get_data_handle (<i>C++ function</i>), 41
dnnl::memory::format_tag::ldigo <i>(C++ enumerator), 36</i>		dnnl::memory::get_desc (<i>C++ function</i>), 41
dnnl::memory::format_tag::ldio (<i>C++ enumerator</i>), 36		dnnl::memory::get_engine (<i>C++ function</i>), 41
dnnl::memory::format_tag::ldnc (<i>C++ enumerator</i>), 36		dnnl::memory::get_sycl_buffer (<i>C++ function</i>), 42
dnnl::memory::format_tag::ldoi (<i>C++ enumerator</i>), 36		dnnl::memory::map_data (<i>C++ function</i>), 42
dnnl::memory::format_tag::nc (<i>C++ enumerator</i>), 34		dnnl::memory::memory (<i>C++ function</i>), 40, 41
dnnl::memory::format_tag::ncdhw <i>(C++ enumerator), 35</i>		dnnl::memory::set_data_handle (<i>C++ function</i>), 41, 42
dnnl::memory::format_tag::nchw (<i>C++ enumerator</i>), 34		dnnl::memory::set_sycl_buffer (<i>C++ function</i>), 43
dnnl::memory::format_tag::ncw (<i>C++ enumerator</i>), 34		dnnl::memory::unmap_data (<i>C++ function</i>), 42
dnnl::memory::format_tag::ndhwc <i>(C++</i>		dnnl::normalization_flags (<i>C++ enum</i>), 56
		dnnl::normalization_flags::fuse_norm_relu <i>(C++ enumerator), 56</i>
		dnnl::normalization_flags::none <i>(C++ enumerator), 56</i>
		dnnl::normalization_flags::use_global_stats

(C++ enumerator), 56
 dnnl::normalization_flags::use_scale_shift (C++ enumerator), 56
 dnnl::pooling_backward (C++ struct), 146
 dnnl::pooling_backward::desc (C++ struct), 146
 dnnl::pooling_backward::desc::desc (C++ function), 147
 dnnl::pooling_backward::pooling_backward (C++ function), 146
 dnnl::pooling_backward::primitive_desc (C++ struct), 147
 dnnl::pooling_backward::primitive_desc::diff_desc (C++ function), 148
 dnnl::pooling_backward::primitive_desc::diff_desc::diff_desc (C++ function), 148
 dnnl::pooling_backward::primitive_desc::diff_desc::diff_desc::diff_desc (C++ function), 148
 dnnl::pooling_forward (C++ struct), 144
 dnnl::pooling_forward::desc (C++ struct), 145
 dnnl::pooling_forward::desc::desc (C++ function), 145
 dnnl::pooling_forward::pooling_forward (C++ function), 145
 dnnl::pooling_forward::primitive_desc (C++ struct), 145
 dnnl::pooling_forward::primitive_desc::diff_desc (C++ function), 146
 dnnl::pooling_forward::primitive_desc::diff_desc::diff_desc (C++ function), 146
 dnnl::pooling_forward::primitive_desc::diff_desc::diff_desc::diff_desc (C++ function), 146
 dnnl::pooling_forward::primitive_desc::diff_desc::diff_desc::diff_desc (C++ function), 146
 dnnl::pooling_forward::primitive_desc::diff_desc::diff_desc::diff_desc (C++ function), 146
 dnnl::post_ops (C++ struct), 61
 dnnl::post_ops::append_eltwise (C++ function), 62
 dnnl::post_ops::append_sum (C++ function), 62
 dnnl::post_ops::get_params_eltwise (C++ function), 63
 dnnl::post_ops::get_params_sum (C++ function), 62
 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), 45
 dnnl::primitive::execute (C++ function), 46
 dnnl::primitive::execute_sycl (C++ function), 47
 dnnl::primitive::get_kind (C++ function), 46
 dnnl::primitive::kind (C++ enum), 45
 dnnl::primitive::kind::batch_normalization (C++ enumerator), 46
 dnnl::primitive::kind::binary (C++ enumerator), 46
 dnnl::primitive::kind::concat (C++ enumerator), 45
 dnnl::primitive::kind::convolution (C++ enumerator), 45
 dnnl::primitive::kind::deconvolution (C++ enumerator), 46
 dnnl::primitive::kind::eltwise (C++ enumerator), 46
 dnnl::primitive::kind::inner_product (C++ enumerator), 46
 dnnl::primitive::kind::layer_normalization (C++ enumerator), 46
 dnnl::primitive::kind::logsoftmax (C++ enumerator), 46
 dnnl::primitive::kind::lrn (C++ enumerator), 46
 dnnl::primitive::kind::matmul (C++ enumerator), 46
 dnnl::primitive::kind::pooling (C++ enumerator), 46
 dnnl::primitive::kind::reorder (C++ enumerator), 45
 dnnl::primitive::kind::resampling (C++ enumerator), 46
 dnnl::primitive::kind::rnn (C++ enumerator), 46
 dnnl::primitive::kind::shuffle (C++ enumerator), 45
 dnnl::primitive::kind::softmax (C++ enumerator), 46
 dnnl::primitive::kind::sum (C++ enumerator), 45
 dnnl::primitive::kind::undef (C++ enumerator), 45
 dnnl::primitive::operator= (C++ function), 47
 dnnl::primitive::primitive (C++ function), 46
 dnnl::primitive_attr (C++ struct), 70
 dnnl::primitive_attr::get_output_scales (C++ function), 70
 dnnl::primitive_attr::get_post_ops (C++ function), 72
 dnnl::primitive_attr::get_scales (C++ function), 71
 dnnl::primitive_attr::get_scratchpad_mode (C++ function), 70
 dnnl::primitive_attr::get_zero_points (C++ function), 71

```

dnnl::primitive_attr::primitive_attr      dnnl::prop_kind(C++ enum), 53
    (C++ function), 70                  dnnl::prop_kind::backward(C++ enumerator),
dnnl::primitive_attr::set_output_scales   53
    (C++ function), 70                  dnnl::prop_kind::backward_bias (C++ enu-
dnnl::primitive_attr::set_post_ops (C++     merator), 53
    function), 72                      dnnl::prop_kind::backward_data (C++ enu-
dnnl::primitive_attr::set_rnn_data_qparams  merator), 53
    (C++ function), 72                  dnnl::prop_kind::backward_weights (C++ enu-
dnnl::primitive_attr::set_rnn_weights_qparams  merator), 53
    (C++ function), 73                  dnnl::prop_kind::forward (C++ enumerator),
dnnl::primitive_attr::set_scales   (C++      53
    function), 71                      dnnl::prop_kind::forward_inference (C++ enu-
dnnl::primitive_attr::set_scratchpad_mode  merator), 53
    (C++ function), 70                  dnnl::prop_kind::forward_scoring (C++ enu-
dnnl::primitive_attr::set_zero_points     merator), 53
    (C++ function), 71                  dnnl::prop_kind::forward_training (C++ enu-
dnnl::primitive_desc (C++ struct), 50      merator), 53
dnnl::primitive_desc::next_impl   (C++      dnnl::prop_kind::undef (C++ enumerator), 53
    function), 50                      dnnl::reorder (C++ struct), 150
dnnl::primitive_desc::primitive_desc  dnnl::reorder::execute (C++ function), 150
    (C++ function), 50                  dnnl::reorder::execute_sycl (C++ function),
dnnl::primitive_desc_base (C++ struct), 47      150
dnnl::primitive_desc_base::diff_dst_desc dnnl::reorder::primitive_desc (C++ struct), 151
    dnnl::reorder::primitive_desc::dst_desc
    (C++ function), 49                  (C++ function), 151
dnnl::primitive_desc_base::diff_src_desc dnnl::reorder::primitive_desc::dst_desc
    dnnl::reorder::primitive_desc::get_dst_engine
    (C++ function), 48, 49              (C++ function), 151
dnnl::primitive_desc_base::diff_weights_desc dnnl::reorder::primitive_desc::get_src_engine
    dnnl::reorder::primitive_desc::get_src_engine
    (C++ function), 49                  (C++ function), 151
dnnl::primitive_desc_base::dst_desc      dnnl::reorder::primitive_desc::primitive_desc
    dnnl::reorder::primitive_desc::src_desc
    (C++ function), 48, 49              (C++ function), 151
dnnl::primitive_desc_base::get_engine   dnnl::reorder::primitive_desc::src_desc
    dnnl::reorder::reorder (C++ function), 150
    (C++ function), 47                  dnnl::resampling_backward (C++ struct), 156
dnnl::primitive_desc_base::impl_info_str dnnl::reorder::resampling_backward::desc (C++ struct), 156
    dnnl::reorder::resampling_backward::desc
    (C++ function), 47                  (C++ function), 156
dnnl::primitive_desc_base::primitive_desc dnnl::resampling_backward::desc::desc
    dnnl::resampling_backward::desc
    (C++ function), 47                  (C++ function), 156
dnnl::primitive_desc_base::query_md     dnnl::resampling_backward::primitive_desc
    dnnl::resampling_backward::primitive_desc
    (C++ function), 48                  (C++ struct), 156
dnnl::primitive_desc_base::query_s64     dnnl::resampling_backward::primitive_desc::diff_dst_desc
    dnnl::resampling_backward::primitive_desc::diff_dst_desc
    (C++ function), 47                  (C++ function), 157
dnnl::primitive_desc_base::scratchpad_d dnnl::resampling_backward::primitive_desc::diff_src_desc
    dnnl::resampling_backward::primitive_desc::diff_src_desc
    (C++ function), 50                  (C++ function), 157
dnnl::primitive_desc_base::scratchpad_eng dnnl::resampling_backward::primitive_desc::primitive_desc
    dnnl::resampling_backward::primitive_desc::primitive_desc
    (C++ function), 50                  (C++ function), 157
dnnl::primitive_desc_base::src_desc      dnnl::resampling_backward::resampling_backward
    dnnl::resampling_backward::resampling_backward
    (C++ function), 48, 49              (C++ function), 156
dnnl::primitive_desc_base::weights_desc  dnnl::resampling_forward (C++ struct), 154
    dnnl::resampling_forward::desc
    (C++ function), 48, 49              (C++ struct), 154
dnnl::primitive_desc_base::workspace_desc dnnl::resampling_forward::desc
    dnnl::resampling_forward::desc
    (C++ function), 50                  (C++ function), 154

```

dnnl::resampling_forward::primitive_desc (C++ function), 154, 155
dnnl::resampling_forward::primitive_desc (C++ struct), 155
dnnl::resampling_forward::primitive_desc::dst_desc (C++ function), 51
(C++ function), 156
dnnl::resampling_forward::primitive_desc::dst_desc (C++ function), 51
(C++ function), 155
dnnl::resampling_forward::primitive_desc::src_desc (C++ function), 51
(C++ function), 155
dnnl::resampling_forward::primitive_desc::src_desc (C++ function), 51
(C++ function), 155
dnnl::resampling_forward::resampling_forward (C++ function), 51
(C++ function), 154
dnnl::rnn_direction (C++ enum), 164
dnnl::rnn_direction::bidirectional_concat (C++ enumerator), 63
(C++ enumerator), 164
dnnl::rnn_direction::bidirectional_sum (C++ enumerator), 164
dnnl::rnn_direction::unidirectional (C++ enumerator), 164
dnnl::rnn_direction::unidirectional_left (C++ enumerator), 164
dnnl::rnn_direction::unidirectional_right (C++ enumerator), 164
dnnl::rnn_flags (C++ enum), 164
dnnl::rnn_flags::undef (C++ enumerator), 164
dnnl::rnn_primitive_desc_base (C++ struct), 51
dnnl::rnn_primitive_desc_base::bias_desc (C++ function), 51
dnnl::rnn_primitive_desc_base::diff_bias_desc (C++ function), 52
dnnl::rnn_primitive_desc_base::diff_dst_desc (C++ function), 52
dnnl::rnn_primitive_desc_base::diff_dst_iter_desc (C++ function), 52
dnnl::rnn_primitive_desc_base::diff_dst_layer_desc (C++ function), 52
dnnl::rnn_primitive_desc_base::diff_src_iter_c_desc (C++ function), 52
dnnl::rnn_primitive_desc_base::diff_src_iter_desc (C++ function), 52
dnnl::rnn_primitive_desc_base::diff_src_layer_desc (C++ function), 52
dnnl::rnn_primitive_desc_base::diff_weights_iter_desc (C++ function), 52
dnnl::rnn_primitive_desc_base::diff_weights_layer_desc (C++ function), 52
dnnl::rnn_primitive_desc_base::dst_iter_desc (C++ function), 51
dnnl::rnn_primitive_desc_base::dst_iter_desc (C++ function), 51
dnnl::rnn_primitive_desc_base::dst_layer_desc (C++ function), 51
dnnl::rnn_primitive_desc_base::rnn_primitive_desc (C++ function), 51
dnnl::rnn_primitive_desc_base::src_iter_c_desc (C++ function), 51
dnnl::rnn_primitive_desc_base::src_iter_desc (C++ function), 51
dnnl::rnn_primitive_desc_base::src_layer_desc (C++ function), 51
dnnl::rnn_primitive_desc_base::src_desc (C++ function), 51
dnnl::rnn_primitive_desc_base::weights_iter_desc (C++ function), 51
dnnl::rnn_primitive_desc_base::weights_layer_desc (C++ function), 51
dnnl::scratchpad_mode (C++ enum), 63
dnnl::scratchpad_mode::library (C++ enum), 63
dnnl::scratchpad_mode::user (C++ enumerator), 63
dnnl::shuffle_backward (C++ struct), 191
dnnl::shuffle_backward::desc (C++ struct), 192
dnnl::shuffle_backward::desc (C++ function), 192
dnnl::shuffle_backward::primitive_desc (C++ struct), 192
dnnl::shuffle_backward::primitive_desc (C++ function), 192
dnnl::shuffle_backward::primitive_desc::diff_dst_desc (C++ function), 192
dnnl::shuffle_backward::primitive_desc::diff_src_desc (C++ function), 192
dnnl::shuffle_backward::primitive_desc::primitive_desc (C++ function), 192
dnnl::shuffle_backward::primitive_desc::primitive_desc (C++ function), 192
dnnl::shuffle_backward::shuffle_backward (C++ function), 192
dnnl::shuffle_backward::shuffle_backward::desc (C++ function), 192
dnnl::shuffle_forward (C++ struct), 190
dnnl::shuffle_forward::desc (C++ struct), 190
dnnl::shuffle_forward::desc (C++ function), 190
dnnl::shuffle_forward::desc (C++ function), 191
dnnl::softmax_backward (C++ struct), 195
dnnl::softmax_backward::desc (C++ struct), 196
dnnl::softmax_backward::desc (C++ function), 196
dnnl::softmax_backward::desc (C++ function), 196
dnnl::softmax_backward::desc (C++ struct), 196
dnnl::softmax_backward::desc (C++ function), 197

dnnl::softmax_backward::primitive_desc::dnnl::svnndsa_rnn_backward::primitive_desc
 (*C++ function*), 197
 (*C++ struct*), 167

dnnl::softmax_backward::primitive_desc::dnnl::devanilla_rnn_backward::primitive_desc::bias_desc
 (*C++ function*), 197
 (*C++ struct*), 168

dnnl::softmax_backward::primitive_desc::dnnl::devacrnn_backward::primitive_desc::diff_desc
 (*C++ function*), 196
 (*C++ struct*), 169

dnnl::softmax_backward::primitive_desc::softmax_backward::vanilla_rnn_backward::primitive_desc::diff_desc
 (*C++ function*), 196
 (*C++ struct*), 169

dnnl::softmax_forward (*C++ struct*), 194
 (*C++ struct*), 194

dnnl::softmax_forward::desc (*C++ struct*),
 194

dnnl::softmax_forward::desc::desc (*C++ function*), 195

dnnl::softmax_forward::primitive_desc
 (*C++ struct*), 195

dnnl::softmax_forward::primitive_desc::dst_desc (*C++ function*), 195

dnnl::softmax_forward::primitive_desc::dst_desc
 (*C++ struct*), 169

dnnl::softmax_forward::primitive_desc::dst_desc
 (*C++ function*), 169

dnnl::softmax_forward::primitive_desc::dst_desc
 (*C++ struct*), 169

dnnl::softmax_forward::softmax_forward
 (*C++ function*), 194

dnnl::stream (*C++ struct*), 25

dnnl::stream::flags (*C++ enum*), 26

dnnl::stream::flags::default_flags (*C++ enumerator*), 26

dnnl::stream::flags::default_order (*C++ enumerator*), 26

dnnl::stream::flags::in_order (*C++ enumerator*), 26

dnnl::stream::flags::out_of_order (*C++ enumerator*), 26

dnnl::stream::get_sycl_queue (*C++ function*), 26

dnnl::stream::stream (*C++ function*), 26

dnnl::stream::wait (*C++ function*), 26

dnnl::stream_attr (*C++ struct*), 25

dnnl::stream_attr::stream_attr (*C++ function*), 25

dnnl::sum (*C++ struct*), 198

dnnl::sum::primitive_desc (*C++ struct*), 198

dnnl::sum::primitive_desc::dst_desc
 (*C++ function*), 199

dnnl::sum::primitive_desc::primitive_desc
 (*C++ function*), 198, 199

dnnl::sum::primitive_desc::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*), 167

dnnl::vanilla_rnn_backward::desc::desc
 (*C++ function*), 167

dnnl::vanilla_rnn_backward::vanilla_rnn_backward
 (*C++ function*), 166

dnnl::vanilla_rnn_forward (*C++ struct*), 164

dnnl::vanilla_rnn_forward::desc
 (*C++ struct*), 164

dnnl::vanilla_rnn_forward::desc::desc
 (*C++ function*), 165

dnnl::vanilla_rnn_forward::primitive_desc
 (*C++ struct*), 165

dnnl::vanilla_rnn_forward::primitive_desc::bias_desc
 (*C++ function*), 166

dnnl::vanilla_rnn_forward::primitive_desc::dst_iter_desc
 (*C++ function*), 166

dnnl::vanilla_rnn_forward::primitive_desc::dst_layer_desc
 (*C++ function*), 166

dnnl::vanilla_rnn_forward::primitive_desc::primitiv
 (*C++ function*), 165

dnnl::vanilla_rnn_forward::primitive_desc::src_iter_desc
 (*C++ function*), 166

dnnl::vanilla_rnn_forward::primitive_desc::src_layer_desc
 (*C++ function*), 166

(*C++ function*), 166
dnnl::vanilla_rnn_forward::primitive_desc DNNL_ARG_WEIGHTS (*C macro*), 57
(C++ function), 166
dnnl::vanilla_rnn_forward::primitive_desc DNNL_ARG_WEIGHTS_0 (*C macro*), 57
(C++ function), 166
dnnl::vanilla_rnn_forward::primitive_desc DNNL_ARG_WEIGHTS_1 (*C macro*), 57
(C++ function), 166
dnnl::vanilla_rnn_forward::primitive_desc DNNL_ARG_WEIGHTS_LAYER (*C macro*), 57
(C++ function), 166
dnnl::vanilla_rnn_forward::vanilla_rnn_f DNNL_ARG_MEMORY_ALLOCATE (*C macro*), 43
(C++ function), 164
DNNL_ARG_ATTR_OUTPUT_SCALES (*C macro*), 59
DNNL_ARG_ATTR_ZERO_POINTS (*C macro*), 59
DNNL_ARG_BIAS (*C macro*), 57
DNNL_ARG_DIFF_BIAS (*C macro*), 58
DNNL_ARG_DIFF_DST (*C macro*), 58
DNNL_ARG_DIFF_DST_0 (*C macro*), 58
DNNL_ARG_DIFF_DST_1 (*C macro*), 58
DNNL_ARG_DIFF_DST_2 (*C macro*), 58
DNNL_ARG_DIFF_DST_ITER (*C macro*), 58
DNNL_ARG_DIFF_DST_ITER_C (*C macro*), 58
DNNL_ARG_DIFF_DST_LAYER (*C macro*), 58
DNNL_ARG_DIFF_SCALE_SHIFT (*C macro*), 58
DNNL_ARG_DIFF_SRC (*C macro*), 57
DNNL_ARG_DIFF_SRC_0 (*C macro*), 57
DNNL_ARG_DIFF_SRC_1 (*C macro*), 58
DNNL_ARG_DIFF_SRC_2 (*C macro*), 58
DNNL_ARG_DIFF_SRC_ITER (*C macro*), 58
DNNL_ARG_DIFF_SRC_ITER_C (*C macro*), 58
DNNL_ARG_DIFF_SRC_LAYER (*C macro*), 57
DNNL_ARG_DIFF_WEIGHTS (*C macro*), 58
DNNL_ARG_DIFF_WEIGHTS_0 (*C macro*), 58
DNNL_ARG_DIFF_WEIGHTS_1 (*C macro*), 58
DNNL_ARG_DIFF_WEIGHTS_ITER (*C macro*), 58
DNNL_ARG_DIFF_WEIGHTS_LAYER (*C macro*), 58
DNNL_ARG_DST (*C macro*), 56
DNNL_ARG_DST_0 (*C macro*), 56
DNNL_ARG_DST_1 (*C macro*), 57
DNNL_ARG_DST_2 (*C macro*), 57
DNNL_ARG_DST_ITER (*C macro*), 57
DNNL_ARG_DST_ITER_C (*C macro*), 57
DNNL_ARG_DST_LAYER (*C macro*), 57
DNNL_ARG_FROM (*C macro*), 56
DNNL_ARG_MEAN (*C macro*), 57
DNNL_ARG_MULTIPLE_DST (*C macro*), 59
DNNL_ARG_MULTIPLE_SRC (*C macro*), 59
DNNL_ARG_SCALE_SHIFT (*C macro*), 57
DNNL_ARG_SCRATCHPAD (*C macro*), 57
DNNL_ARG_SRC (*C macro*), 56
DNNL_ARG_SRC_0 (*C macro*), 56
DNNL_ARG_SRC_1 (*C macro*), 56
DNNL_ARG_SRC_2 (*C macro*), 56
DNNL_ARG_SRC_ITER (*C macro*), 56
DNNL_ARG_SRC_ITER_C (*C macro*), 56
DNNL_ARG_SRC_LAYER (*C macro*), 56
DNNL_ARG_TO (*C macro*), 57
DNNL_ARG_VARIANCE (*C macro*), 57
DNNL_WEIGHTS (*C macro*), 57
DNNL_WEIGHTS_0 (*C macro*), 57
DNNL_WEIGHTS_1 (*C macro*), 57
DNNL_WEIGHTS_LAYER (*C macro*), 57
DNNL_WORKSPACE (*C macro*), 57
DNNL_MEMORY_ALLOCATE (*C macro*), 43
DNNL_MEMORY_NONE (*C macro*), 43
DNNL_RUNTIME_DIM_VAL (*C macro*), 59
DNNL_RUNTIME_F32_VAL (*C macro*), 59
DNNL_RUNTIME_S32_VAL (*C macro*), 59
DNNL_RUNTIME_SIZE_VAL (*C macro*), 59
do_allocate (*C++ function*), 623
do_deallocate (*C++ function*), 623
do_is_equal (*C++ function*), 623
DPC++, 220

E

empty (*C++ function*), 306, 614
end (*C++ function*), 306, 615, 617
enqueue (*C++ function*), 375
erf (*C++ function*), 1293
erfc (*C++ function*), 1296
erfcinv (*C++ function*), 1305
erfinv (*C++ function*), 1302
ets_key_usage_type::ets_key_per_instance (*C++ enum*), 616
ets_key_usage_type::ets_no_key (*C++ enum*), 616
ets_key_usage_type::ets_suspend_aware (*C++ enum*), 616
exception_thrown (*C++ function*), 317
exp (*C++ function*), 1228
exp10 (*C++ function*), 1232
exp2 (*C++ function*), 1230
expint1 (*C++ function*), 1314
expml (*C++ function*), 1234

F

fdim (*C++ function*), 1343
Feature, 218
Feature vector, 218
filter (*C++ function*), 302
Flat data, 219
flatten2d (*C++ function*), 618
flattened2d (*C++ function*), 617, 618
floor (*C++ function*), 1316
fmax (*C++ function*), 1345
fmin (*C++ function*), 1346
fmod (*C++ function*), 1201
frac (*C++ function*), 1328
Func::~Func (*C++ function*), 282
Func::Func (*C++ function*), 279, 281

`Func::operator()` (*C++ function*), 270, 279, 282

G

`generate` (*C++ function*), 1130
`get_mode` (*C++ function*), 1331
`get_status` (*C++ function*), 1334
`Getter`, 219
`getValue` (*C++ function*), 1120
`global_control` (*C++ function*), 370
`grainsize` (*C++ function*), 306
`graph` (*C++ function*), 317

H

`H::~H` (*C++ function*), 278
`H::equal` (*C++ function*), 278
`H::H` (*C++ function*), 278
`H::hash` (*C++ function*), 278
 Heterogeneous data, 220
 Homogeneous data, 220
`Host/Device`, 220
`hypot` (*C++ function*), 1226

I

Immutability, 220
`indexer_node` (*C++ function*), 356
`Inference`, 218
`Inference set`, 218
`init` (*C++ function*), 1118
`initialize` (*C++ function*), 375
`input_node` (*C++ function*), 326
`input_ports` (*C++ function*), 356, 358
 Interval feature, 218
`inv` (*C++ function*), 1205
`invcbrt` (*C++ function*), 1214
`invsqrt` (*C++ function*), 1211
`is_a` (*C++ function*), 363
`is_active` (*C++ function*), 375
`is_canceling` (*C++ function*), 373
`is_cancelled` (*C++ function*), 317
`is_current_task_group_canceling` (*C++ function*), 373
`is_divisible` (*C++ function*), 306
`is_final_scan` (*C++ function*), 294
`is_group_execution_cancelled` (*C++ function*), 369
`is_observing` (*C++ function*), 378
`is_valid` (*C++ function*), 338, 340

J

JIT, 220

K

Kernel, 220

`kind_t::bound` (*C++ enum*), 368
`kind_t::isolated` (*C++ enum*), 368

L

`Label`, 218
`leapfrog` (*C++ function*), 1143
`left` (*C++ function*), 315
`lgamma` (*C++ function*), 1311
`limiter_node` (*C++ function*), 348
`linearfrac` (*C++ function*), 1199
`ln` (*C++ function*), 1235
`local` (*C++ function*), 609, 614
`lock` (*C++ function*), 628, 629, 634, 635
`lock_shared` (*C++ function*), 629, 635
`log10` (*C++ function*), 1239
`log1p` (*C++ function*), 1241
`log2` (*C++ function*), 1237
`logb` (*C++ function*), 1243

M

`M::~scoped_lock` (*C++ function*), 275
`M::is_fair_mutex` (*C++ member*), 275, 277
`M::is_recursive_mutex` (*C++ member*), 275, 277
`M::is_rw_mutex` (*C++ member*), 275, 277
`M::scoped_lock` (*C++ function*), 275
`M::scoped_lock` (*C++ type*), 275
`M::scoped_lock::acquire` (*C++ function*), 275
`M::scoped_lock::release` (*C++ function*), 275
`M::scoped_lock::try_acquire` (*C++ function*), 275
`make_filter` (*C++ function*), 302
`max_concurrency` (*C++ function*), 375, 377
`max_size` (*C++ function*), 622
`maxmag` (*C++ function*), 1348
`Metadata`, 220
`MFX_ANGLE_0` (*C++ enumerator*), 739
`MFX_ANGLE_180` (*C++ enumerator*), 739
`MFX_ANGLE_270` (*C++ enumerator*), 739
`MFX_ANGLE_90` (*C++ enumerator*), 739
`MFX_B_REF_OFF` (*C++ enumerator*), 723
`MFX_B_REF_PYRAMID` (*C++ enumerator*), 723
`MFX_B_REF_UNKNOWN` (*C++ enumerator*), 723
`MFX_BITSTREAM_COMPLETE_FRAME` (*C++ enumerator*), 726
`MFX_BITSTREAM_EOS` (*C++ enumerator*), 726
`MFX_BLOCKSIZE_MIN_16X16` (*C++ enumerator*), 726
`MFX_BLOCKSIZE_MIN_4X4` (*C++ enumerator*), 726
`MFX_BLOCKSIZE_MIN_8X8` (*C++ enumerator*), 726
`MFX_BLOCKSIZE_UNKNOWN` (*C++ enumerator*), 726
`MFX_BPSEI_DEFAULT` (*C++ enumerator*), 724
`MFX_BPSEI_IFRAME` (*C++ enumerator*), 724
`MFX_BRC_BIG_FRAME` (*C++ enumerator*), 742
`MFX_BRC_OK` (*C++ enumerator*), 742

MFX_BRC_PANIC_BIG_FRAME (*C++ enumerator*), 742
MFX_BRC_PANIC_SMALL_FRAME (*C++ enumerator*), 743
MFX_BRC_SMALL_FRAME (*C++ enumerator*), 742
MFX_CHROMA_SITING_HORIZONTAL_CENTER (*C++ enumerator*), 740
MFX_CHROMA_SITING_HORIZONTAL_LEFT (*C++ enumerator*), 740
MFX_CHROMA_SITING_UNKNOWN (*C++ enumerator*), 740
MFX_CHROMA_SITING_VERTICAL_BOTTOM (*C++ enumerator*), 740
MFX_CHROMA_SITING_VERTICAL_CENTER (*C++ enumerator*), 740
MFX_CHROMA_SITING_VERTICAL_TOP (*C++ enumerator*), 740
MFX_CHROMAFORMAT_JPEG_SAMPLING (*C++ enumerator*), 715
MFX_CHROMAFORMAT_MONOCHROME (*C++ enumerator*), 715
MFX_CHROMAFORMAT_RESERVED1 (*C++ enumerator*), 715
MFX_CHROMAFORMAT_YUV400 (*C++ enumerator*), 715
MFX_CHROMAFORMAT_YUV411 (*C++ enumerator*), 715
MFX_CHROMAFORMAT_YUV420 (*C++ enumerator*), 715
MFX_CHROMAFORMAT_YUV422 (*C++ enumerator*), 715
MFX_CHROMAFORMAT_YUV422H (*C++ enumerator*), 715
MFX_CHROMAFORMAT_YUV422V (*C++ enumerator*), 715
MFX_CHROMAFORMAT_YUV444 (*C++ enumerator*), 715
MFX_CODEC_AV1 (*C++ enumerator*), 717
MFX_CODEC_AVC (*C++ enumerator*), 717
MFX_CODEC_HEVC (*C++ enumerator*), 717
MFX_CODEC_JPEG (*C++ enumerator*), 718
MFX_CODEC_MPEG2 (*C++ enumerator*), 717
MFX_CODEC_VC1 (*C++ enumerator*), 717
MFX_CODEC_VP9 (*C++ enumerator*), 717
MFX_CODINGOPTION_ADAPTIVE (*C++ enumerator*), 726
MFX_CODINGOPTION_OFF (*C++ enumerator*), 726
MFX_CODINGOPTION_ON (*C++ enumerator*), 726
MFX_CODINGOPTION_UNKNOWN (*C++ enumerator*), 726
MFX_CONTENT_FULL_SCREEN_VIDEO (*C++ enumerator*), 725
MFX_CONTENT_NON_VIDEO_SCREEN (*C++ enumerator*), 725
MFX_CONTENT_UNKNOWN (*C++ enumerator*), 725
MFX_CORRUPTION_ABSENT_BOTTOM_FIELD (*C++ enumerator*), 716
MFX_CORRUPTION_ABSENT_TOP_FIELD (*C++ enumerator*), 716
MFX_CORRUPTION_MAJOR (*C++ enumerator*), 716
MFX_CORRUPTION_MINOR (*C++ enumerator*), 716
MFX_CORRUPTION_REFERENCE_FRAME (*C++ enumerator*), 716
MFX_CORRUPTION_REFERENCE_LIST (*C++ enumerator*), 716
MFX_DEINTERLACING_24FPS_OUT (*C++ enumerator*), 736
MFX_DEINTERLACING_30FPS_OUT (*C++ enumerator*), 736
MFX_DEINTERLACING_ADVANCED (*C++ enumerator*), 736
MFX_DEINTERLACING_ADVANCED_NOREF (*C++ enumerator*), 736
MFX_DEINTERLACING_ADVANCED_SCD (*C++ enumerator*), 737
MFX_DEINTERLACING_AUTO_DOUBLE (*C++ enumerator*), 736
MFX_DEINTERLACING_AUTO_SINGLE (*C++ enumerator*), 736
MFX_DEINTERLACING_BOB (*C++ enumerator*), 736
MFX_DEINTERLACING_DETECT_INTERLACE (*C++ enumerator*), 736
MFX_DEINTERLACING_FIELD_WEAVING (*C++ enumerator*), 737
MFX_DEINTERLACING_FIXED_TELECINE_PATTERN (*C++ enumerator*), 736
MFX_DEINTERLACING_FULL_FR_OUT (*C++ enumerator*), 736
MFX_DEINTERLACING_HALF_FR_OUT (*C++ enumerator*), 736
MFX_ERROR_FRAME_GAP (*C++ enumerator*), 739
MFX_ERROR_PPS (*C++ enumerator*), 739
MFX_ERROR_SLICEDATA (*C++ enumerator*), 739
MFX_ERROR_SLICEHEADER (*C++ enumerator*), 739
MFX_ERROR_SPS (*C++ enumerator*), 739
MFX_EXTBUFF_AVC_REFLIST_CTRL (*C++ enumerator*), 727
MFX_EXTBUFF_AVC_REFLISTS (*C++ enumerator*), 728
MFX_EXTBUFF_AVC_ROUNDING_OFFSET (*C++ enumerator*), 730
MFX_EXTBUFF_AVC_TEMPORAL_LAYERS (*C++ enumerator*), 728
MFX_EXTBUFF_BRC (*C++ enumerator*), 731
MFX_EXTBUFF_CENC_PARAM (*C++ enumerator*), 731
MFX_EXTBUFF_CHROMA_LOC_INFO (*C++ enumerator*), 729
MFX_EXTBUFF_CODING_OPTION (*C++ enumerator*),

MFX_EXTBUFF_CODING_OPTION2 (<i>C++ enumerator</i>), 728	MFX_EXTBUFF_MB_FORCE_INTRA (<i>C++ enumerator</i>), 729
MFX_EXTBUFF_CODING_OPTION3 (<i>C++ enumerator</i>), 729	MFX_EXTBUFF_MBQP (<i>C++ enumerator</i>), 729
MFX_EXTBUFF_CODING_OPTION_SPSPPS (<i>C++ enumerator</i>), 727	MFX_EXTBUFF_MOVING_RECTANGLES (<i>C++ enumerator</i>), 729
MFX_EXTBUFF_CODING_OPTION_VPS (<i>C++ enumerator</i>), 729	MFX_EXTBUFF_MULTI_FRAME_CONTROL (<i>C++ enumerator</i>), 730
MFX_EXTBUFF_CONTENT_LIGHT_LEVEL_INFO (<i>C++ enumerator</i>), 730	MFX_EXTBUFF_MULTI_FRAME_PARAM (<i>C++ enumerator</i>), 730
MFX_EXTBUFF_DEC_VIDEO_PROCESSING (<i>C++ enumerator</i>), 728	MFX_EXTBUFF_MV_OVER_PIC_BOUNDARIES (<i>C++ enumerator</i>), 730
MFX_EXTBUFF_DECODE_ERROR_REPORT (<i>C++ enumerator</i>), 730	MFX_EXTBUFF_MVC_SEQ_DESC (<i>C++ enumerator</i>), 731
MFX_EXTBUFF_DECODED_FRAME_INFO (<i>C++ enumerator</i>), 729	MFX_EXTBUFF_MVC_TARGET_VIEWS (<i>C++ enumerator</i>), 731
MFX_EXTBUFF_DIRTY_RECTANGLES (<i>C++ enumerator</i>), 729	MFX_EXTBUFF_PARTIAL_BITSTREAM_PARAM (<i>C++ enumerator</i>), 730
MFX_EXTBUFF_ENCODED_FRAME_INFO (<i>C++ enumerator</i>), 728	MFX_EXTBUFF_PICTURE_TIMING_SEI (<i>C++ enumerator</i>), 728
MFX_EXTBUFF_ENCODED_SLICES_INFO (<i>C++ enumerator</i>), 729	MFX_EXTBUFF_PRED_WEIGHT_TABLE (<i>C++ enumerator</i>), 729
MFX_EXTBUFF_ENCODED_UNITS_INFO (<i>C++ enumerator</i>), 730	MFX_EXTBUFF_THREADS_PARAM (<i>C++ enumerator</i>), 727
MFX_EXTBUFF_ENCODER_CAPABILITY (<i>C++ enumerator</i>), 728	MFX_EXTBUFF_TIME_CODE (<i>C++ enumerator</i>), 729
MFX_EXTBUFF_ENCODER_IPCM_AREA (<i>C++ enumerator</i>), 731	MFX_EXTBUFF_VIDEO_SIGNAL_INFO (<i>C++ enumerator</i>), 727
MFX_EXTBUFF_ENCODER_RESET_OPTION (<i>C++ enumerator</i>), 728	MFX_EXTBUFF_VP8_CODING_OPTION (<i>C++ enumerator</i>), 731
MFX_EXTBUFF_ENCODER_ROI (<i>C++ enumerator</i>), 728	MFX_EXTBUFF_VP9_PARAM (<i>C++ enumerator</i>), 730
MFX_EXTBUFF_ENCTOOLS_CONFIG (<i>C++ enumerator</i>), 731	MFX_EXTBUFF_VP9_SEGMENTATION (<i>C++ enumerator</i>), 730
MFX_EXTBUFF_HEVC_PARAM (<i>C++ enumerator</i>), 729	MFX_EXTBUFF_VP9_TEMPORAL_LAYERS (<i>C++ enumerator</i>), 730
MFX_EXTBUFF_HEVC_REFLIST_CTRL (<i>C++ enumerator</i>), 729	MFX_EXTBUFF_VPP_AUXDATA (<i>C++ enumerator</i>), 727
MFX_EXTBUFF_HEVC_REFLISTS (<i>C++ enumerator</i>), 730	MFX_EXTBUFF_VPP_COLOR_CONVERSION (<i>C++ enumerator</i>), 730
MFX_EXTBUFF_HEVC_REGION (<i>C++ enumerator</i>), 729	MFX_EXTBUFF_VPP_COLORFILL (<i>C++ enumerator</i>), 730
MFX_EXTBUFF_HEVC_TEMPORAL_LAYERS (<i>C++ enumerator</i>), 730	MFX_EXTBUFF_VPP_COMPOSITE (<i>C++ enumerator</i>), 728
MFX_EXTBUFF_HEVC_TILES (<i>C++ enumerator</i>), 729	MFX_EXTBUFF_VPP_DEINTERLACING (<i>C++ enumerator</i>), 728
MFX_EXTBUFF_INSERT_HEADERS (<i>C++ enumerator</i>), 731	MFX_EXTBUFF_VPP_DENOISE (<i>C++ enumerator</i>), 727
MFX_EXTBUFF_JPEG_HUFFMAN (<i>C++ enumerator</i>), 731	MFX_EXTBUFF_VPP_DETAIL (<i>C++ enumerator</i>), 727
MFX_EXTBUFF_JPEG_QT (<i>C++ enumerator</i>), 731	MFX_EXTBUFF_VPP_DONOTUSE (<i>C++ enumerator</i>), 727
MFX_EXTBUFF_MASTERING_DISPLAY_COLOUR_VOLUME (<i>C++ enumerator</i>), 730	MFX_EXTBUFF_VPP_DOUSE (<i>C++ enumerator</i>), 727
MFX_EXTBUFF_MB_DISABLE_SKIP_MAP (<i>C++ enumerator</i>), 729	MFX_EXTBUFF_VPP_FIELD_PROCESSING (<i>C++ enumerator</i>), 728
	MFX_EXTBUFF_VPP_FRAME_RATE_CONVERSION (<i>C++ enumerator</i>), 727
	MFX_EXTBUFF_VPP_IMAGE_STABILIZATION

(*C++ enumerator*), 728
MFX_EXTBUFF_VPP_MCTF (*C++ enumerator*), 730
MFX_EXTBUFF_VPP_MIRRORING (*C++ enumerator*), 730
MFX_EXTBUFF_VPP_PROCAMP (*C++ enumerator*), 727
MFX_EXTBUFF_VPP_ROTATION (*C++ enumerator*), 729
MFX_EXTBUFF_VPP_SCALING (*C++ enumerator*), 729
MFX_EXTBUFF_VPP_SCENE_ANALYSIS (*C++ enumerator*), 727
MFX_EXTBUFF_VPP_VIDEO_SIGNAL_INFO (*C++ enumerator*), 728
MFX_FOURCC_A2RGB10 (*C++ enumerator*), 714
MFX_FOURCC_ABGR16 (*C++ enumerator*), 714
MFX_FOURCC_ARGB16 (*C++ enumerator*), 714
MFX_FOURCC_AYUV (*C++ enumerator*), 714
MFX_FOURCC_AYUV_RGB4 (*C++ enumerator*), 714
MFX_FOURCC_BGR4 (*C++ enumerator*), 714
MFX_FOURCC_I010 (*C++ enumerator*), 714
MFX_FOURCC_IYUV (*C++ enumerator*), 713
MFX_FOURCC_NV12 (*C++ enumerator*), 713
MFX_FOURCC_NV16 (*C++ enumerator*), 713
MFX_FOURCC_NV21 (*C++ enumerator*), 713
MFX_FOURCC_P010 (*C++ enumerator*), 714
MFX_FOURCC_P016 (*C++ enumerator*), 714
MFX_FOURCC_P210 (*C++ enumerator*), 714
MFX_FOURCC_P8 (*C++ enumerator*), 714
MFX_FOURCC_P8_TEXTURE (*C++ enumerator*), 714
MFX_FOURCC_R16 (*C++ enumerator*), 714
MFX_FOURCC_RGB4 (*C++ enumerator*), 713
MFX_FOURCC_RGB565 (*C++ enumerator*), 713
MFX_FOURCC_RGBP (*C++ enumerator*), 713
MFX_FOURCC_UYVY (*C++ enumerator*), 714
MFX_FOURCC_Y210 (*C++ enumerator*), 714
MFX_FOURCC_Y216 (*C++ enumerator*), 715
MFX_FOURCC_Y410 (*C++ enumerator*), 715
MFX_FOURCC_Y416 (*C++ enumerator*), 715
MFX_FOURCC_YUY2 (*C++ enumerator*), 713
MFX_FOURCC_YV12 (*C++ enumerator*), 713
MFX_FRAMEDATA_ORIGINAL_TIMESTAMP (*C++ enumerator*), 716
MFX_FRAMEORDER_UNKNOWN (*C++ enumerator*), 716
MFX_FRAMETYPE_B (*C++ enumerator*), 732
MFX_FRAMETYPE_I (*C++ enumerator*), 732
MFX_FRAMETYPE_IDR (*C++ enumerator*), 732
MFX_FRAMETYPE_P (*C++ enumerator*), 732
MFX_FRAMETYPE_REF (*C++ enumerator*), 732
MFX_FRAMETYPE_S (*C++ enumerator*), 732
MFX_FRAMETYPE_UNKNOWN (*C++ enumerator*), 732
MFX_FRAMETYPE_xB (*C++ enumerator*), 733
MFX_FRAMETYPE_xI (*C++ enumerator*), 732
MFX_FRAMETYPE_xIDR (*C++ enumerator*), 733
MFX_FRAMETYPE_xP (*C++ enumerator*), 733
MFX_FRAMETYPE_xREF (*C++ enumerator*), 733
MFX_FRAMETYPE_xs (*C++ enumerator*), 733
MFX_FRCALGM_DISTRIBUTED_TIMESTAMP (*C++ enumerator*), 735
MFX_FRCALGM_FRAME_INTERPOLATION (*C++ enumerator*), 735
MFX_FRCALGM_PRESERVE_TIMESTAMP (*C++ enumerator*), 735
MFX_GOP_CLOSED (*C++ enumerator*), 721
MFX_GOP_STRICT (*C++ enumerator*), 721
MFX_GPUCOPY_DEFAULT (*C++ enumerator*), 711
MFX_GPUCOPY_OFF (*C++ enumerator*), 711
MFX_GPUCOPY_ON (*C++ enumerator*), 711
MFX_HEVC_CONSTR_REXT_INTRA (*C++ enumerator*), 738
MFX_HEVC_CONSTR_REXT_LOWER_BIT_RATE (*C++ enumerator*), 738
MFX_HEVC_CONSTR_REXT_MAX_10BIT (*C++ enumerator*), 738
MFX_HEVC_CONSTR_REXT_MAX_12BIT (*C++ enumerator*), 738
MFX_HEVC_CONSTR_REXT_MAX_420CHROMA (*C++ enumerator*), 738
MFX_HEVC_CONSTR_REXT_MAX_422CHROMA (*C++ enumerator*), 738
MFX_HEVC_CONSTR_REXT_MAX_8BIT (*C++ enumerator*), 738
MFX_HEVC_CONSTR_REXT_MAX_MONOCHROME (*C++ enumerator*), 738
MFX_HEVC_CONSTR_REXT_ONE_PICTURE_ONLY (*C++ enumerator*), 738
MFX_HEVC_NALU_TYPE_CRA_NUT (*C++ enumerator*), 733
MFX_HEVC_NALU_TYPE_IDR_N_LP (*C++ enumerator*), 733
MFX_HEVC_NALU_TYPE_IDR_W_RADL (*C++ enumerator*), 733
MFX_HEVC_NALU_TYPE_RADL_N (*C++ enumerator*), 733
MFX_HEVC_NALU_TYPE_RADL_R (*C++ enumerator*), 733
MFX_HEVC_NALU_TYPE_RASL_N (*C++ enumerator*), 733
MFX_HEVC_NALU_TYPE_RASL_R (*C++ enumerator*), 733
MFX_HEVC_NALU_TYPE_TRAIL_N (*C++ enumerator*), 733
MFX_HEVC_NALU_TYPE_TRAIL_R (*C++ enumerator*), 733
MFX_HEVC_NALU_TYPE_UNKNOWN (*C++ enumerator*), 733
MFX_HEVC_REGION_ENCODING_OFF (*C++ enumerator*), 739

MFX_HEVC_REGION_ENCODING_ON (*C++ enumerator*), 739
MFX_HEVC_REGION_SLICE (*C++ enumerator*), 739
MFX_IMAGESTAB_MODE_BOXING (*C++ enumerator*), 735
MFX_IMAGESTAB_MODE_UPSCALE (*C++ enumerator*), 735
MFX_IMPL_AUTO_ANY (*C++ enumerator*), 710
MFX_IMPL_BASETYPE (*C macro*), 710
MFX_IMPL_EXTERNAL_THREADING (*C++ enumerator*), 710
MFX_IMPL_HARDWARE (*C++ enumerator*), 710
MFX_IMPL_HARDWARE2 (*C++ enumerator*), 710
MFX_IMPL_HARDWARE3 (*C++ enumerator*), 710
MFX_IMPL_HARDWARE4 (*C++ enumerator*), 710
MFX_IMPL_HARDWARE_ANY (*C++ enumerator*), 710
MFX_IMPL_NAME (*C macro*), 698
MFX_IMPL_RUNTIME (*C++ enumerator*), 710
MFX_IMPL_SINGLE_THREAD (*C++ enumerator*), 710
MFX_IMPL_SOFTWARE (*C++ enumerator*), 710
MFX_IMPL_UNSUPPORTED (*C++ enumerator*), 710
MFX_IMPL_VIA_ANY (*C++ enumerator*), 710
MFX_IMPL_VIA_D3D11 (*C++ enumerator*), 710
MFX_IMPL_VIA_D3D9 (*C++ enumerator*), 710
MFX_IMPL_VIA_VAAPI (*C++ enumerator*), 710
MFX_INTERPOLATION_ADVANCED (*C++ enumerator*), 740
MFX_INTERPOLATION_BILINEAR (*C++ enumerator*), 740
MFX_INTERPOLATION_DEFAULT (*C++ enumerator*), 740
MFX_INTERPOLATION_NEAREST_NEIGHBOR (*C++ enumerator*), 740
MFX_IOPATTERN_IN_SYSTEM_MEMORY (*C++ enumerator*), 717
MFX_IOPATTERN_IN_VIDEO_MEMORY (*C++ enumerator*), 717
MFX_IOPATTERN_OUT_SYSTEM_MEMORY (*C++ enumerator*), 717
MFX_IOPATTERN_OUT_VIDEO_MEMORY (*C++ enumerator*), 717
MFX_JPEG_COLORFORMAT_RGB (*C++ enumerator*), 743
MFX_JPEG_COLORFORMAT_UNKNOWN (*C++ enumerator*), 743
MFX_JPEG_COLORFORMAT_YCbCr (*C++ enumerator*), 743
MFX_LEVEL_AVC_1 (*C++ enumerator*), 719
MFX_LEVEL_AVC_11 (*C++ enumerator*), 719
MFX_LEVEL_AVC_12 (*C++ enumerator*), 719
MFX_LEVEL_AVC_13 (*C++ enumerator*), 719
MFX_LEVEL_AVC_1b (*C++ enumerator*), 719
MFX_LEVEL_AVC_2 (*C++ enumerator*), 719
MFX_LEVEL_AVC_21 (*C++ enumerator*), 719
MFX_LEVEL_AVC_22 (*C++ enumerator*), 719
MFX_LEVEL_AVC_3 (*C++ enumerator*), 719
MFX_LEVEL_AVC_31 (*C++ enumerator*), 719
MFX_LEVEL_AVC_32 (*C++ enumerator*), 720
MFX_LEVEL_AVC_4 (*C++ enumerator*), 720
MFX_LEVEL_AVC_41 (*C++ enumerator*), 720
MFX_LEVEL_AVC_42 (*C++ enumerator*), 720
MFX_LEVEL_AVC_5 (*C++ enumerator*), 720
MFX_LEVEL_AVC_51 (*C++ enumerator*), 720
MFX_LEVEL_AVC_52 (*C++ enumerator*), 720
MFX_LEVEL_HEVC_1 (*C++ enumerator*), 720
MFX_LEVEL_HEVC_2 (*C++ enumerator*), 720
MFX_LEVEL_HEVC_21 (*C++ enumerator*), 720
MFX_LEVEL_HEVC_3 (*C++ enumerator*), 720
MFX_LEVEL_HEVC_31 (*C++ enumerator*), 720
MFX_LEVEL_HEVC_4 (*C++ enumerator*), 720
MFX_LEVEL_HEVC_41 (*C++ enumerator*), 720
MFX_LEVEL_HEVC_5 (*C++ enumerator*), 720
MFX_LEVEL_HEVC_51 (*C++ enumerator*), 720
MFX_LEVEL_HEVC_52 (*C++ enumerator*), 720
MFX_LEVEL_HEVC_6 (*C++ enumerator*), 720
MFX_LEVEL_HEVC_61 (*C++ enumerator*), 721
MFX_LEVEL_HEVC_62 (*C++ enumerator*), 721
MFX_LEVEL_MPEG2_HIGH (*C++ enumerator*), 720
MFX_LEVEL_MPEG2_HIGH1440 (*C++ enumerator*), 720
MFX_LEVEL_MPEG2_LOW (*C++ enumerator*), 720
MFX_LEVEL_MPEG2_MAIN (*C++ enumerator*), 720
MFX_LEVEL_UNKNOWN (*C++ enumerator*), 719
MFX_LEVEL_VC1_0 (*C++ enumerator*), 720
MFX_LEVEL_VC1_1 (*C++ enumerator*), 720
MFX_LEVEL_VC1_2 (*C++ enumerator*), 720
MFX_LEVEL_VC1_3 (*C++ enumerator*), 720
MFX_LEVEL_VC1_4 (*C++ enumerator*), 720
MFX_LEVEL_VC1_HIGH (*C++ enumerator*), 720
MFX_LEVEL_VC1_LOW (*C++ enumerator*), 720
MFX_LEVEL_VC1_MEDIAN (*C++ enumerator*), 720
MFX_LONGTERM_IDX_NO_IDX (*C++ enumerator*), 735
MFX_LOOKAHEAD_DS_2x (*C++ enumerator*), 723
MFX_LOOKAHEAD_DS_4x (*C++ enumerator*), 723
MFX_LOOKAHEAD_DS_OFF (*C++ enumerator*), 723
MFX_LOOKAHEAD_DS_UNKNOWN (*C++ enumerator*), 723
MFX_MBQP_MODE_QP_ADAPTIVE (*C++ enumerator*), 738
MFX_MBQP_MODE_QP_DELTA (*C++ enumerator*), 738
MFX_MBQP_MODE_QP_VALUE (*C++ enumerator*), 738
MFX_MCTF_TEMPORAL_MODE_1REF (*C++ enumerator*), 742
MFX_MCTF_TEMPORAL_MODE_2REF (*C++ enumerator*), 742
MFX_MCTF_TEMPORAL_MODE_4REF (*C++ enumerator*), 742

MFX_MCTF_TEMPORAL_MODE_SPATIAL (*C++ enumerator*), 742
MFX_MCTF_TEMPORAL_MODE_UNKNOWN (*C++ enumerator*), 742
MFX_MEMTYPE_DXVA2_DECODER_TARGET (*C++ enumerator*), 731
MFX_MEMTYPE_DXVA2_PROCESSOR_TARGET (*C++ enumerator*), 731
MFX_MEMTYPE_EXPORT_FRAME (*C++ enumerator*), 732
MFX_MEMTYPE_EXTERNAL_FRAME (*C++ enumerator*), 732
MFX_MEMTYPE_FROM_DECODE (*C++ enumerator*), 732
MFX_MEMTYPE_FROM_ENC (*C++ enumerator*), 732
MFX_MEMTYPE_FROM_ENCODE (*C++ enumerator*), 732
MFX_MEMTYPE_FROM_VPPIN (*C++ enumerator*), 732
MFX_MEMTYPE_FROM_VPPOUT (*C++ enumerator*), 732
MFX_MEMTYPE_INTERNAL_FRAME (*C++ enumerator*), 732
MFX_MEMTYPE_PERSISTENT_MEMORY (*C++ enumerator*), 731
MFX_MEMTYPE_RESERVED1 (*C++ enumerator*), 732
MFX_MEMTYPE_SHARED_RESOURCE (*C++ enumerator*), 732
MFX_MEMTYPE_SYSTEM_MEMORY (*C++ enumerator*), 732
MFX_MEMTYPE_VIDEO_MEMORY_DECODER_TARGET (*C++ enumerator*), 731
MFX_MEMTYPE_VIDEO_MEMORY_ENCODER_TARGET (*C++ enumerator*), 732
MFX_MEMTYPE_VIDEO_MEMORY_PROCESSOR_TARGET (*C++ enumerator*), 731
MFX_MIRRORING_DISABLED (*C++ enumerator*), 740
MFX_MIRRORING_HORIZONTAL (*C++ enumerator*), 740
MFX_MIRRORING_VERTICAL (*C++ enumerator*), 740
MFX_MVPRECISION_HALFPEL (*C++ enumerator*), 726
MFX_MVPRECISION_INTEGER (*C++ enumerator*), 726
MFX_MVPRECISION_QUARTERPEL (*C++ enumerator*), 726
MFX_MVPRECISION_UNKNOWN (*C++ enumerator*), 726
MFX_NOMINALRANGE_0_255 (*C++ enumerator*), 736
MFX_NOMINALRANGE_16_235 (*C++ enumerator*), 736
MFX_NOMINALRANGE_UNKNOWN (*C++ enumerator*), 736
MFX_P_REF_DEFAULT (*C++ enumerator*), 725
MFX_P_REF_PYRAMID (*C++ enumerator*), 725
MFX_P_REF_SIMPLE (*C++ enumerator*), 725
MFX_PARTIAL_BITSTREAM_ANY (*C++ enumerator*), 742
MFX_PARTIAL_BITSTREAM_BLOCK (*C++ enumerator*), 742
MFX_PARTIAL_BITSTREAM_NONE (*C++ enumerator*), 742
MFX_PARTIAL_BITSTREAM_SLICE (*C++ enumerator*), 742
MFX_PAYLOAD_CTRL_SUFFIX (*C++ enumerator*), 731
MFX_PAYLOAD_IDR (*C++ enumerator*), 735
MFX_PAYLOAD_OFF (*C++ enumerator*), 735
MFX_PICSTRUCT_FIELD_BFF (*C++ enumerator*), 715
MFX_PICSTRUCT_FIELD_BOTTOM (*C++ enumerator*), 716
MFX_PICSTRUCT_FIELD_PAIRED_NEXT (*C++ enumerator*), 716
MFX_PICSTRUCT_FIELD_PAIRED_PREV (*C++ enumerator*), 716
MFX_PICSTRUCT_FIELD_REPEATED (*C++ enumerator*), 716
MFX_PICSTRUCT_FIELD_SINGLE (*C++ enumerator*), 716
MFX_PICSTRUCT_FIELD_TFF (*C++ enumerator*), 715
MFX_PICSTRUCT_FIELD_TOP (*C++ enumerator*), 716
MFX_PICSTRUCT_FRAME_DOUBLING (*C++ enumerator*), 716
MFX_PICSTRUCT_FRAME_TRIPLING (*C++ enumerator*), 716
MFX_PICSTRUCT_PROGRESSIVE (*C++ enumerator*), 715
MFX_PICSTRUCT_UNKNOWN (*C++ enumerator*), 715
MFX_PICTYPE_BOTTOMFIELD (*C++ enumerator*), 737
MFX_PICTYPE_FRAME (*C++ enumerator*), 737
MFX_PICTYPE_TOPFIELD (*C++ enumerator*), 737
MFX_PICTYPE_UNKNOWN (*C++ enumerator*), 737
MFX_PLATFORM_APOLLOLAKE (*C++ enumerator*), 712
MFX_PLATFORM_BAYTRAIL (*C++ enumerator*), 711
MFX_PLATFORM_BROADWELL (*C++ enumerator*), 711
MFX_PLATFORM_CANNONLAKE (*C++ enumerator*), 712
MFX_PLATFORM_CHERRYTRAIL (*C++ enumerator*), 711
MFX_PLATFORM_COFFEELAKE (*C++ enumerator*), 712
MFX_PLATFORM_ELKHARTLAKE (*C++ enumerator*), 712
MFX_PLATFORM_GEMINILAKE (*C++ enumerator*),

MFX_PROFILE_UNKNOWN (<i>C++ enumerator</i>),	718
MFX_PROFILE_VC1_ADVANCED (<i>C++ enumerator</i>),	718
MFX_PROFILE_VC1_MAIN (<i>C++ enumerator</i>),	718
MFX_PROFILE_VC1_SIMPLE (<i>C++ enumerator</i>),	718
MFX_PROFILE_VP8_0 (<i>C++ enumerator</i>),	719
MFX_PROFILE_VP8_1 (<i>C++ enumerator</i>),	719
MFX_PROFILE_VP8_2 (<i>C++ enumerator</i>),	719
MFX_PROFILE_VP8_3 (<i>C++ enumerator</i>),	719
MFX_PROFILE_VP9_0 (<i>C++ enumerator</i>),	719
MFX_PROFILE_VP9_1 (<i>C++ enumerator</i>),	719
MFX_PROFILE_VP9_2 (<i>C++ enumerator</i>),	719
MFX_PROFILE_VP9_3 (<i>C++ enumerator</i>),	719
MFX_PROTECTION_CENC_WV_CLASSIC (<i>C++ enumerator</i>),	744
MFX_PROTECTION_CENC_WV_GOOGLE_DASH (<i>C++ enumerator</i>),	744
MFX_RATECONTROL_AVBR (<i>C++ enumerator</i>),	722
MFX_RATECONTROL_CBR (<i>C++ enumerator</i>),	722
MFX_RATECONTROL_CQP (<i>C++ enumerator</i>),	722
MFX_RATECONTROL_ICQ (<i>C++ enumerator</i>),	722
MFX_RATECONTROL_LA (<i>C++ enumerator</i>),	722
MFX_RATECONTROL_LA_HRD (<i>C++ enumerator</i>),	722
MFX_RATECONTROL_LA_ICQ (<i>C++ enumerator</i>),	722
MFX_RATECONTROL_QVBR (<i>C++ enumerator</i>),	723
MFX_RATECONTROL_VBR (<i>C++ enumerator</i>),	722
MFX_RATECONTROL_VCM (<i>C++ enumerator</i>),	722
MFX_REFRESH_HORIZONTAL (<i>C++ enumerator</i>),	724
MFX_REFRESH_NO (<i>C++ enumerator</i>),	724
MFX_REFRESH_SLICE (<i>C++ enumerator</i>),	724
MFX_REFRESH_VERTICAL (<i>C++ enumerator</i>),	724
MFX_ROI_MODE_PRIORITY (<i>C++ enumerator</i>),	736
MFX_ROI_MODE_QP_DELTA (<i>C++ enumerator</i>),	736
MFX_ROI_MODE_QP_VALUE (<i>C++ enumerator</i>),	736
MFX_ROTATION_0 (<i>C++ enumerator</i>),	743
MFX_ROTATION_180 (<i>C++ enumerator</i>),	743
MFX_ROTATION_270 (<i>C++ enumerator</i>),	743
MFX_ROTATION_90 (<i>C++ enumerator</i>),	743
MFX_SAO_DISABLE (<i>C++ enumerator</i>),	738
MFX_SAO_ENABLE_CHROMA (<i>C++ enumerator</i>),	738
MFX_SAO_ENABLE_LUMA (<i>C++ enumerator</i>),	738
MFX_SAO_UNKNOWN (<i>C++ enumerator</i>),	738
MFX_SCALING_MODE_DEFAULT (<i>C++ enumerator</i>),	740
MFX_SCALING_MODE_LOWPOWER (<i>C++ enumerator</i>),	740
MFX_SCALING_MODE_QUALITY (<i>C++ enumerator</i>),	740
MFX_SCANTYPE_INTERLEAVED (<i>C++ enumerator</i>),	743
MFX_SCANTYPE_NONINTERLEAVED (<i>C++ enumerator</i>),	743
MFX_SCANTYPE_UNKNOWN (<i>C++ enumerator</i>),	743
MFX_SCENARIO_ARCHIVE (<i>C++ enumerator</i>),	725
MFX_PROFILE_PLATFORM_HASWELL (<i>C++ enumerator</i>),	711
MFX_PROFILE_PLATFORM_ICELAKE (<i>C++ enumerator</i>),	712
MFX_PROFILE_PLATFORM_IVYBRIDGE (<i>C++ enumerator</i>),	711
MFX_PROFILE_PLATFORM_JASPERLAKE (<i>C++ enumerator</i>),	712
MFX_PROFILE_PLATFORM_KABYLAKE (<i>C++ enumerator</i>),	712
MFX_PROFILE_PLATFORM_SANDYBRIDGE (<i>C++ enumerator</i>),	711
MFX_PROFILE_SKYLAKE (<i>C++ enumerator</i>),	711
MFX_PROFILE_TIGERLAKE (<i>C++ enumerator</i>),	712
MFX_PROFILE_UNKNOWN (<i>C++ enumerator</i>),	711
MFX_PROFILE_AVC_BASELINE (<i>C++ enumerator</i>),	718
MFX_PROFILE_AVC_CONSTRAINED_BASELINE (<i>C++ enumerator</i>),	718
MFX_PROFILE_AVC_CONSTRAINED_HIGH (<i>C++ enumerator</i>),	718
MFX_PROFILE_AVC_CONSTRAINT_SET0 (<i>C++ enumerator</i>),	718
MFX_PROFILE_AVC_CONSTRAINT_SET1 (<i>C++ enumerator</i>),	718
MFX_PROFILE_AVC_CONSTRAINT_SET2 (<i>C++ enumerator</i>),	718
MFX_PROFILE_AVC_CONSTRAINT_SET3 (<i>C++ enumerator</i>),	718
MFX_PROFILE_AVC_CONSTRAINT_SET4 (<i>C++ enumerator</i>),	718
MFX_PROFILE_AVC_CONSTRAINT_SET5 (<i>C++ enumerator</i>),	718
MFX_PROFILE_AVC_EXTENDED (<i>C++ enumerator</i>),	718
MFX_PROFILE_AVC_HIGH (<i>C++ enumerator</i>),	718
MFX_PROFILE_AVC_HIGH10 (<i>C++ enumerator</i>),	718
MFX_PROFILE_AVC_HIGH_422 (<i>C++ enumerator</i>),	718
MFX_PROFILE_AVC_MAIN (<i>C++ enumerator</i>),	718
MFX_PROFILE_AVC_MULTIVIEW_HIGH (<i>C++ enumerator</i>),	718
MFX_PROFILE_AVC_STEREO_HIGH (<i>C++ enumerator</i>),	718
MFX_PROFILE_HEVC_MAIN (<i>C++ enumerator</i>),	719
MFX_PROFILE_HEVC_MAIN10 (<i>C++ enumerator</i>),	719
MFX_PROFILE_HEVC_MAINSP (<i>C++ enumerator</i>),	719
MFX_PROFILE_HEVC_REXT (<i>C++ enumerator</i>),	719
MFX_PROFILE_HEVC_SCC (<i>C++ enumerator</i>),	719
MFX_PROFILE_JPEG_BASELINE (<i>C++ enumerator</i>),	719
MFX_PROFILE_MPEG2_HIGH (<i>C++ enumerator</i>),	718
MFX_PROFILE_MPEG2_MAIN (<i>C++ enumerator</i>),	718
MFX_PROFILE_MPEG2_SIMPLE (<i>C++ enumerator</i>),	718

MFX_SCENARIO_CAMERA_CAPTURE (*C++ enumerator*), 725
MFX_SCENARIO_DISPLAY_Remoting (*C++ enumerator*), 725
MFX_SCENARIO_GAME_STREAMING (*C++ enumerator*), 725
MFX_SCENARIO_LIVE_STREAMING (*C++ enumerator*), 725
MFX_SCENARIO_REMOTE_GAMING (*C++ enumerator*), 725
MFX_SCENARIO_UNKNOWN (*C++ enumerator*), 725
MFX_SCENARIO_VIDEO_CONFERENCE (*C++ enumerator*), 725
MFX_SCENARIO_VIDEO_SURVEILLANCE (*C++ enumerator*), 725
MFX_SKIPFRAME_BRC_ONLY (*C++ enumerator*), 724
MFX_SKIPFRAME_INSERT_DUMMY (*C++ enumerator*), 724
MFX_SKIPFRAME_INSERT_NOTHING (*C++ enumerator*), 724
MFX_SKIPFRAME_NO_SKIP (*C++ enumerator*), 724
MFX_STRFIELD_LEN (*C macro*), 698
MFX_TARGETUSAGE_1 (*C++ enumerator*), 721
MFX_TARGETUSAGE_2 (*C++ enumerator*), 721
MFX_TARGETUSAGE_3 (*C++ enumerator*), 721
MFX_TARGETUSAGE_4 (*C++ enumerator*), 721
MFX_TARGETUSAGE_5 (*C++ enumerator*), 721
MFX_TARGETUSAGE_6 (*C++ enumerator*), 721
MFX_TARGETUSAGE_7 (*C++ enumerator*), 721
MFX_TARGETUSAGE_BALANCED (*C++ enumerator*), 722
MFX_TARGETUSAGE_BEST_QUALITY (*C++ enumerator*), 722
MFX_TARGETUSAGE_BEST_SPEED (*C++ enumerator*), 722
MFX_TARGETUSAGE_UNKNOWN (*C++ enumerator*), 721
MFX_TELECINE_PATTERN_2332 (*C++ enumerator*), 737
MFX_TELECINE_PATTERN_32 (*C++ enumerator*), 737
MFX_TELECINE_PATTERN_41 (*C++ enumerator*), 737
MFX_TELECINE_PATTERN_FRAME_REPEAT (*C++ enumerator*), 737
MFX_TELECINE_POSITION_PROVIDED (*C++ enumerator*), 737
MFX_TIER_HEVC_HIGH (*C++ enumerator*), 721
MFX_TIER_HEVC_MAIN (*C++ enumerator*), 721
MFX_TIMESTAMP_UNKNOWN (*C++ enumerator*), 716
MFX_TIMESTAMPCALC_TELECINE (*C++ enumerator*), 717
MFX_TIMESTAMPCALC_UNKNOWN (*C++ enumerator*), 717
MFX_TRANSFERMATRIX_BT601 (*C++ enumerator*), 735
MFX_TRANSFERMATRIX_BT709 (*C++ enumerator*), 735
MFX_TRANSFERMATRIX_UNKNOWN (*C++ enumerator*), 735
MFX_TRELLIS_B (*C++ enumerator*), 723
MFX_TRELLIS_I (*C++ enumerator*), 723
MFX_TRELLIS_OFF (*C++ enumerator*), 723
MFX_TRELLIS_P (*C++ enumerator*), 723
MFX_TRELLIS_UNKNOWN (*C++ enumerator*), 723
MFX_VP9_REF_ALTREF (*C++ enumerator*), 741
MFX_VP9_REF_GOLDEN (*C++ enumerator*), 741
MFX_VP9_REF_INTRA (*C++ enumerator*), 741
MFX_VP9_REF_LAST (*C++ enumerator*), 741
MFX_VP9_SEGMENT_FEATURE_LOOP_FILTER (*C++ enumerator*), 741
MFX_VP9_SEGMENT_FEATURE_QINDEX (*C++ enumerator*), 741
MFX_VP9_SEGMENT_FEATURE_REFERENCE (*C++ enumerator*), 741
MFX_VP9_SEGMENT_FEATURE_SKIP (*C++ enumerator*), 741
MFX_VP9_SEGMENT_ID_BLOCK_SIZE_16x16 (*C++ enumerator*), 741
MFX_VP9_SEGMENT_ID_BLOCK_SIZE_32x32 (*C++ enumerator*), 741
MFX_VP9_SEGMENT_ID_BLOCK_SIZE_64x64 (*C++ enumerator*), 741
MFX_VP9_SEGMENT_ID_BLOCK_SIZE_8x8 (*C++ enumerator*), 741
MFX_VP9_SEGMENT_ID_BLOCK_SIZE_UNKNOWN (*C++ enumerator*), 741
MFX_VPP_COPY_FIELD (*C++ enumerator*), 737
MFX_VPP_COPY_FRAME (*C++ enumerator*), 737
MFX_VPP_SWAP_FIELDS (*C++ enumerator*), 737
MFX_WEIGHTED_PRED_DEFAULT (*C++ enumerator*), 724
MFX_WEIGHTED_PRED_EXPLICIT (*C++ enumerator*), 724
MFX_WEIGHTED_PRED_IMPLICIT (*C++ enumerator*), 724
MFX_WEIGHTED_PRED_UNKNOWN (*C++ enumerator*), 724
mfxA2RGB10 (*C++ struct*), 754
mfxA2RGB10::A (*C++ member*), 754
mfxA2RGB10::B (*C++ member*), 754
mfxA2RGB10::G (*C++ member*), 754
mfxA2RGB10::R (*C++ member*), 754
mfxAdapterInfo (*C++ struct*), 766
mfxAdapterInfo::Number (*C++ member*), 766
mfxAdapterInfo::Platform (*C++ member*), 766
mfxAdaptersInfo (*C++ struct*), 767
mfxAdaptersInfo::Adapters (*C++ member*),

767
mfxAdaptersInfo::NumActual (*C++ member*),
 767
mfxAdaptersInfo::NumAlloc (*C++ member*),
 767
mfxBitstream (*C++ struct*), 760
mfxBitstream::Data (*C++ member*), 760
mfxBitstream::DataFlag (*C++ member*), 761
mfxBitstream::DataLength (*C++ member*), 761
mfxBitstream::DataOffset (*C++ member*), 760
mfxBitstream::DecodeTimeStamp (*C++ member*), 760
mfxBitstream::EncryptedData (*C++ member*),
 760
mfxBitstream::ExtParam (*C++ member*), 760
mfxBitstream::FrameType (*C++ member*), 761
mfxBitstream::MaxLength (*C++ member*), 761
mfxBitstream::NumExtParam (*C++ member*),
 760
mfxBitstream::PicStruct (*C++ member*), 761
mfxBitstream::reserved2 (*C++ member*), 761
mfxBitstream::TimeStamp (*C++ member*), 760
mfxBRCFrameCtrl (*C++ struct*), 814
mfxBRCFrameCtrl::DeltaQP (*C++ member*), 815
mfxBRCFrameCtrl::ExtParam (*C++ member*),
 815
mfxBRCFrameCtrl::InitialCpbRemovalDelay
 (*C++ member*), 814
mfxBRCFrameCtrl::InitialCpbRemovalOffset
 (*C++ member*), 814
mfxBRCFrameCtrl::MaxFrameSize (*C++ mem-
 ber*), 814
mfxBRCFrameCtrl::MaxNumRepak (*C++ mem-
 ber*), 815
mfxBRCFrameCtrl::NumExtParam (*C++ mem-
 ber*), 815
mfxBRCFrameCtrl::QpY (*C++ member*), 814
mfxBRCFrameParam (*C++ struct*), 813
mfxBRCFrameParam::CodedFrameSize (*C++ mem-
 ber*), 814
mfxBRCFrameParam::DisplayOrder (*C++ mem-
 ber*), 814
mfxBRCFrameParam::EncodedOrder (*C++ mem-
 ber*), 813
mfxBRCFrameParam::ExtParam (*C++ member*),
 814
mfxBRCFrameParam::FrameCmplx (*C++ mem-
 ber*), 813
mfxBRCFrameParam::FrameType (*C++ member*),
 814
mfxBRCFrameParam::LongTerm (*C++ member*),
 813
mfxBRCFrameParam::NumExtParam (*C++ mem-
 ber*), 814
mfxBRCFrameParam::NumRecode (*C++ member*),
 814
mfxBRCFrameParam::PyramidLayer (*C++ mem-
 ber*), 814
mfxBRCFrameParam::SceneChange (*C++ mem-
 ber*), 813
mfxBRCFrameStatus (*C++ struct*), 815
mfxBRCFrameStatus::BRCStatus (*C++ mem-
 ber*), 815
mfxBRCFrameStatus::MinFrameSize (*C++ mem-
 ber*), 815
mfxChar (*C++ type*), 697
MFXCloneSession (*C++ function*), 825
MFXClose (*C++ function*), 824
mfxComponentInfo (*C++ struct*), 766
mfxComponentInfo::Requirements (*C++ mem-
 ber*), 766
mfxComponentInfo::Type (*C++ member*), 766
mfxComponentType (*C++ enum*), 742
mfxComponentType::MFX_COMPONENT_DECODE
 (*C++ enumerator*), 742
mfxComponentType::MFX_COMPONENT_ENCODE
 (*C++ enumerator*), 742
mfxComponentType::MFX_COMPONENT_VPP
 (*C++ enumerator*), 742
mfxConfig (*C++ type*), 698
MFXCreateConfig (*C++ function*), 705
MFXCreateSession (*C++ function*), 707
mfxDecoderDescription (*C++ struct*), 700
mfxDecoderDescription::Codecs (*C++ mem-
 ber*), 700
mfxDecoderDescription::decoder (*C++
 struct*), 700
mfxDecoderDescription::decoder::CodecID
 (*C++ member*), 700
mfxDecoderDescription::decoder::decprofile
 (*C++ struct*), 700
mfxDecoderDescription::decoder::decprofile::dec-
 mem (*C++ struct*), 700
mfxDecoderDescription::decoder::decprofile::dec-
 mem (*C++ member*), 701
mfxDecoderDescription::decoder::decprofile::MemDes-
 criptor (*C++ member*), 700
mfxDecoderDescription::decoder::decprofile::NumMem-

(C++ member), 786
`mfxExtAVCEncodedFrameInfo::MAD` (C++ member), 786
`mfxExtAVCEncodedFrameInfo::PicStruct` (C++ member), 786
`mfxExtAVCEncodedFrameInfo::QP` (C++ member), 787
`mfxExtAVCEncodedFrameInfo::SecondFieldOffset` (C++ member), 787
`mfxExtAVCEncodedFrameInfo::UsedRefListL1`
`mfxExtBRC` (C++ struct), 815
(C++ member), 787
`mfxExtAVCRefListCtrl` (C++ struct), 781
`mfxExtAVCRefListCtrl::ApplyLongTermIdx` (C++ member), 781
`mfxExtAVCRefListCtrl::FrameOrder` (C++ member), 781
`mfxExtAVCRefListCtrl::Header` (C++ member), 781
`mfxExtAVCRefListCtrl::LongTermIdx` (C++ member), 781
`mfxExtAVCRefListCtrl::NumRefIdxL0Active` (C++ member), 781
`mfxExtAVCRefListCtrl::NumRefIdxL1Active` (C++ member), 781
`mfxExtAVCRefListCtrl::PicStruct` (C++ member), 781
`mfxExtAVCRefListCtrl::reserved` (C++ member), 781
`mfxExtAVCRefListCtrl::ViewId` (C++ member), 781
`mfxExtAVCRefLists` (C++ struct), 788
`mfxExtAVCRefLists::Header` (C++ member), 788
`mfxExtAVCRefLists::mfxRefPic` (C++ struct), 788
`mfxExtAVCRefLists::mfxRefPic::FrameOrder`
`mfxExtCodingOption2::AdaptiveI` (C++ member), 772
`mfxExtAVCRefLists::mfxRefPic::PicStruct`
`mfxExtCodingOption2::BitrateLimit` (C++ member), 771
`mfxExtAVCRefLists::NumRefIdxL0Active` (C++ member), 788
`mfxExtAVCRefLists::NumRefIdxL1Active` (C++ member), 788
`mfxExtAVCRefLists::RefPicList1` (C++ member), 788
`mfxExtAVCRoundingOffset` (C++ struct), 794
`mfxExtAVCRoundingOffset::EnableRoundingInter` (C++ member), 794
`mfxExtAVCRoundingOffset::EnableRoundingIntra` (C++ member), 794
`mfxExtAVCRoundingOffset::Header` (C++ member), 794
`mfxExtAVCRoundingOffset::RoundingOffsetInter` (C++ member), 794
`mfxExtAVCRoundingOffset::RoundingOffsetIntra` (C++ member), 794
`mfxExtAVCRoundingOffset::RoundingOffsetIntra` (C++ member), 794
`mfxExtAvcTemporalLayers` (C++ struct), 784
`mfxExtAvcTemporalLayers::BaseLayerPID` (C++ member), 784
`mfxExtAvcTemporalLayers::Header` (C++ member), 784
`mfxExtAvcTemporalLayers::Scale` (C++ member), 784
`mfxExtBRC::Close` (C++ member), 816
`mfxExtBRC::GetFrameCtrl` (C++ member), 816
`mfxExtBRC::Header` (C++ member), 815
`mfxExtBRC::Init` (C++ member), 815
`mfxExtBRC::pthis` (C++ member), 815
`mfxExtBRC::Reset` (C++ member), 816
`mfxExtBRC::Update` (C++ member), 816
`mfxExtBuffer` (C++ struct), 768
`mfxExtBuffer::BufferId` (C++ member), 768
`mfxExtBuffer::BufferSz` (C++ member), 768
`mfxExtChromaLocInfo` (C++ struct), 789
`mfxExtChromaLocInfo::ChromaLocInfoPresentFlag` (C++ member), 789
`mfxExtChromaLocInfo::ChromaSampleLocTypeBottomField` (C++ member), 789
`mfxExtChromaLocInfo::ChromaSampleLocTypeTopField` (C++ member), 789
`mfxExtChromaLocInfo::Header` (C++ member), 789
`mfxExtChromaLocInfo::reserved` (C++ member), 789
`mfxExtCodingOption` (C++ struct), 768
`mfxExtCodingOption2` (C++ struct), 770
`mfxExtCodingOption2::AdaptiveB` (C++ member), 772
`mfxExtCodingOption2::AdaptiveI` (C++ member), 772
`mfxExtCodingOption2::BitrateLimit` (C++ member), 771
`mfxExtCodingOption2::BRefType` (C++ member), 772
`mfxExtCodingOption2::BufferingPeriodSEI` (C++ member), 773
`mfxExtCodingOption2::DisableDeblockingIdc` (C++ member), 773
`mfxExtCodingOption2::DisableVUI` (C++ member), 773
`mfxExtCodingOption2::EnableMAD` (C++ member), 773
`mfxExtCodingOption2::ExtBRC` (C++ member), 771
`mfxExtCodingOption2::FixedFrameRate`
`mfxExtCodingOption2::RoundingOffsetInter` (C++ member), 773
`mfxExtCodingOption2::Header` (C++ member),

771
`mfxExtCodingOption2::IntRefCycleSize (C++ member), 771`
`mfxExtCodingOption2::IntRefQPDelta (C++ member), 771`
`mfxExtCodingOption2::IntRefType (C++ member), 771`
`mfxExtCodingOption2::LookAheadDepth (C++ member), 771`
`mfxExtCodingOption2::LookAheadDS (C++ member), 772`
`mfxExtCodingOption2::MaxFrameSize (C++ member), 771`
`mfxExtCodingOption2::MaxQPB (C++ member), 773`
`mfxExtCodingOption2::MaxQPI (C++ member), 772`
`mfxExtCodingOption2::MaxQPP (C++ member), 773`
`mfxExtCodingOption2::MaxSliceSize (C++ member), 771`
`mfxExtCodingOption2::MBBRC (C++ member), 771`
`mfxExtCodingOption2::MinQPB (C++ member), 773`
`mfxExtCodingOption2::MinQPI (C++ member), 772`
`mfxExtCodingOption2::MinQPP (C++ member), 772`
`mfxExtCodingOption2::NumMbPerSlice (C++ member), 772`
`mfxExtCodingOption2::RepeatPPS (C++ member), 772`
`mfxExtCodingOption2::SkipFrame (C++ member), 772`
`mfxExtCodingOption2::Trellis (C++ member), 772`
`mfxExtCodingOption2::UseRawRef (C++ member), 773`
`mfxExtCodingOption3 (C++ struct), 774`
`mfxExtCodingOption3::AdaptiveMaxFrameSize (C++ member), 777`
`mfxExtCodingOption3::AspectRatioInfoPresent (C++ member), 775`
`mfxExtCodingOption3::BitstreamRestriction (C++ member), 775`
`mfxExtCodingOption3::BRCPanicMode (C++ member), 777`
`mfxExtCodingOption3::ConstrainedIntraPredFlag (C++ member), 776`
`mfxExtCodingOption3::ContentInfo (C++ member), 776`
`mfxExtCodingOption3::DeblockingAlphaTcOffset (C++ member), 776`
`mfxExtCodingOption3::DeblockingBetaOffset (C++ member), 776`
`mfxExtCodingOption3::DirectBiasAdjustment (C++ member), 774`
`mfxExtCodingOption3::EnableMBForceIntra (C++ member), 777`
`mfxExtCodingOption3::EnableMBQP (C++ member), 774`
`mfxExtCodingOption3::EnableNalUnitType (C++ member), 777`
`mfxExtCodingOption3::EnableQPOffset (C++ member), 776`
`mfxExtCodingOption3::EncodedUnitsInfo (C++ member), 777`
`mfxExtCodingOption3::ExtBrcAdaptiveLTR (C++ member), 777`
`mfxExtCodingOption3::FadeDetection (C++ member), 776`
`mfxExtCodingOption3::GlobalMotionBiasAdjustment (C++ member), 774`
`mfxExtCodingOption3::GPB (C++ member), 776`
`mfxExtCodingOption3::Header (C++ member), 774`
`mfxExtCodingOption3::IntraVLCFormat (C++ member), 777`
`mfxExtCodingOption3::IntRefCycleDist (C++ member), 774`
`mfxExtCodingOption3::Log2MaxMvLengthHorizontal (C++ member), 775`
`mfxExtCodingOption3::Log2MaxMvLengthVertical (C++ member), 775`
`mfxExtCodingOption3::LowDelayBRC (C++ member), 777`
`mfxExtCodingOption3::LowDelayHrd (C++ member), 775`
`mfxExtCodingOption3::MaxFrameSizeI (C++ member), 776`
`mfxExtCodingOption3::MaxFrameSizeP (C++ member), 776`
`mfxExtCodingOption3::MBDisableSkipMap (C++ member), 775`
`mfxExtCodingOption3::MotionVectorsOverPicBoundaries (C++ member), 775`
`mfxExtCodingOption3::MVCostScalingFactor (C++ member), 775`
`mfxExtCodingOption3::NumRefActiveBL0 (C++ member), 776`
`mfxExtCodingOption3::NumRefActiveBL1 (C++ member), 776`
`mfxExtCodingOption3::NumRefActiveP (C++ member), 776`
`mfxExtCodingOption3::NumSliceB (C++ member), 774`
`mfxExtCodingOption3::NumSliceI (C++ member), 776`

ber), 774 *(C++ member), 770*
mfxExtCodingOption3::NumSliceP (C++ mem- mfxExtCodingOption::IntraPredBlockSize ber), 774 *(C++ member), 770*
mfxExtCodingOption3::OverscanAppropriate mfxExtCodingOption::MaxDecFrameBuffering (C++ member), 775 *(C++ member), 770*
mfxExtCodingOption3::OverscanInfoPresent mfxExtCodingOption::MECostType (C++ mem- (C++ member), 775 *ber), 769*
mfxExtCodingOption3::PRefType (C++ mem- mfxExtCodingOption::MSESearchType (C++ ber), 776 *member), 769*
mfxExtCodingOption3::QPOffset (C++ mem- mfxExtCodingOption::MVPrecision (C++ ber), 776 *member), 770*
mfxExtCodingOption3::QuantScaleType mfxExtCodingOption::MVSearchWindow (C++ (C++ member), 777 *member), 769*
mfxExtCodingOption3::QVBRQuality (C++ mfxExtCodingOption::NalHrdConformance member), 774 *(C++ member), 769*
mfxExtCodingOption3::RepartitionCheckEna- mfxExtCodingOption::PicTimingSEI (C++ ble), 777 *member), 770*
mfxExtCodingOption3::reserved (C++ mem- mfxExtCodingOption::RateDistortionOpt ber), 778 *(C++ member), 769*
mfxExtCodingOption3::reserved3 (C++ mem- mfxExtCodingOption::RecoveryPointSEI ber), 776 *(C++ member), 769*
mfxExtCodingOption3::ScanType (C++ mem- mfxExtCodingOption::RefPicListReordering ber), 777 *(C++ member), 770*
mfxExtCodingOption3::ScenarioInfo (C++ mfxExtCodingOption::RefPicMarkRep (C++ member), 775 *member), 770*
mfxExtCodingOption3::TargetBitDepthChroma mfxExtCodingOption::ResetRefList (C++ (C++ member), 777 *member), 770*
mfxExtCodingOption3::TargetBitDepthLuma mfxExtCodingOption::SingleSeiNalUnit (C++ member), 777 *(C++ member), 769*
mfxExtCodingOption3::TargetChromaFormatPmfxExtCodingOption::ViewOutput (C++ mem- ber), 776 *(C++ member), 769*
mfxExtCodingOption3::TimingInfoPresent mfxExtCodingOption::VuiNalHrdParameters (C++ member), 775 *(C++ member), 770*
mfxExtCodingOption3::TransformSkip (C++ mfxExtCodingOption::VuiVclHrdParameters member), 776 *(C++ member), 769*
mfxExtCodingOption3::WeightedBiPred mfxExtCodingOptionSPSPPS (C++ struct), 775
(C++ member), 775 *mfxExtCodingOptionSPSPPS::Header (C++ member), 778*
mfxExtCodingOption3::WeightedPred (C++ mfxExtCodingOptionSPSPPS::PPSBuffer member), 775 *(C++ member), 778*
mfxExtCodingOption3::WinBRCMaxAvgKbps mfxExtCodingOptionSPSPPS::PPSBufSize (C++ member), 774 *(C++ member), 778*
mfxExtCodingOption3::WinBRCSize (C++ mfxExtCodingOptionSPSPPS::PPSID (C++ member), 774 *member), 778*
mfxExtCodingOption::AUDelimiter (C++ mfxExtCodingOptionSPSPPS::SPSBuffer member), 770 *(C++ member), 778*
mfxExtCodingOption::CAVLC (C++ member), 769 *mfxExtCodingOptionSPSPPS::SPSBufSize (C++ member), 778*
mfxExtCodingOption::FieldOutput (C++ member), 770 *mfxExtCodingOptionSPSPPS::SPSID (C++ member), 778*
mfxExtCodingOption::FramePicture (C++ member), 769 *mfxExtCodingOptionVPS (C++ struct), 779*
mfxExtCodingOption::Header (C++ member), 769 *mfxExtCodingOptionVPS::Header (C++ mem- ber), 779*
mfxExtCodingOption::InterPredBlockSize mfxExtCodingOptionVPS::VPSBuffer (C++

member), 779

`mfxExtCodingOptionVPS::VPSSBufSize (C++ member), 779`

`mfxExtCodingOptionVPS::VPSId (C++ member), 779`

`mfxExtColorConversion (C++ struct), 812`

`mfxExtColorConversion::ChromaSiting (C++ member), 812`

`mfxExtColorConversion::Header (C++ member), 812`

`mfxExtContentLightLevelInfo (C++ struct), 782`

`mfxExtContentLightLevelInfo::Header (C++ member), 783`

`mfxExtContentLightLevelInfo::InsertPayload (C++ member), 783`

`mfxExtContentLightLevelInfo::MaxContentLength (C++ member), 783`

`mfxExtContentLightLevelInfo::MaxPicAverageLength (C++ member), 783`

`mfxExtDecodedFrameInfo (C++ struct), 792`

`mfxExtDecodedFrameInfo::FrameType (C++ member), 792`

`mfxExtDecodedFrameInfo::Header (C++ member), 792`

`mfxExtDecodeErrorReport (C++ struct), 792`

`mfxExtDecodeErrorReport::ErrorTypes (C++ member), 792`

`mfxExtDecodeErrorReport::Header (C++ member), 792`

`mfxExtDecVideoProcessing (C++ struct), 810`

`mfxExtDecVideoProcessing::Header (C++ member), 810`

`mfxExtDecVideoProcessing::In (C++ member), 810`

`mfxExtDecVideoProcessing::mfxIn (C++ struct), 810`

`mfxExtDecVideoProcessing::mfxIn::CropH (C++ member), 810`

`mfxExtDecVideoProcessing::mfxIn::CropW (C++ member), 810`

`mfxExtDecVideoProcessing::mfxIn::CropX (C++ member), 810`

`mfxExtDecVideoProcessing::mfxIn::CropY (C++ member), 810`

`mfxExtDecVideoProcessing::mfxOut (C++ struct), 810`

`mfxExtDecVideoProcessing::mfxOut::Chroma (C++ member), 810`

`mfxExtDecVideoProcessing::mfxOut::CropH (C++ member), 811`

`mfxExtDecVideoProcessing::mfxOut::CropW (C++ member), 811`

`mfxExtDecVideoProcessing::mfxOut::CropX (C++ member), 811`

(C++ member), 811

`mfxExtDecVideoProcessing::mfxOut::CropY (C++ member), 811`

`mfxExtDecVideoProcessing::mfxOut::FourCC (C++ member), 810`

`mfxExtDecVideoProcessing::mfxOut::Height (C++ member), 810`

`mfxExtDecVideoProcessing::mfxOut::Width (C++ member), 810`

`mfxExtDecVideoProcessing::Out (C++ member), 810`

`mfxExtDirtyRect (C++ struct), 794`

`mfxExtDirtyRect::Bottom (C++ member), 795`

`mfxExtDirtyRect::Header (C++ member), 795`

`mfxExtDirtyRect::Left (C++ member), 795`

`mfxExtDirtyRect::NumRect (C++ member), 795`

`mfxExtDirtyRect::Right (C++ member), 795`

`mfxExtDirtyRect::Top (C++ member), 795`

`mfxExtEncodedSlicesInfo (C++ struct), 804`

`mfxExtEncodedSlicesInfo::Header (C++ member), 804`

`mfxExtEncodedSlicesInfo::NumEncodedSlice (C++ member), 804`

`mfxExtEncodedSlicesInfo::NumSliceNonCopiant (C++ member), 804`

`mfxExtEncodedSlicesInfo::NumSliceSizeAlloc (C++ member), 804`

`mfxExtEncodedSlicesInfo::SliceSize (C++ member), 804`

`mfxExtEncodedSlicesInfo::SliceSizeOverflow (C++ member), 804`

`mfxExtEncodedUnitsInfo (C++ struct), 800`

`mfxExtEncodedUnitsInfo::Header (C++ member), 801`

`mfxExtEncodedUnitsInfo::NumUnitsAlloc (C++ member), 801`

`mfxExtEncodedUnitsInfo::NumUnitsEncoded (C++ member), 801`

`mfxExtEncodedUnitsInfo::UnitInfo (C++ member), 801`

`mfxExtEncoderCapability (C++ struct), 784`

`mfxExtEncoderCapability::Header (C++ member), 784`

`mfxExtEncoderCapability::MBPerSec (C++ member), 784`

`mfxExtEncoderIPCMArea (C++ struct), 788`

`mfxExtEncoderIPCMArea::Area (C++ member), 788`

`mfxExtEncoderIPCMArea::Bottom (C++ member), 788`

`mfxExtEncoderIPCMArea::Header (C++ member), 788`

`mfxExtEncoderIPCMArea::Left (C++ member),`

788
mfxExtEncoderIPCMArea::Right (C++ member), 788
mfxExtEncoderIPCMArea::Top (C++ member), 788
mfxExtEncoderResetOption (C++ struct), 785
mfxExtEncoderResetOption::Header (C++ member), 786
mfxExtEncoderResetOption::StartNewSequence (C++ member), 786
mfxExtEncoderROI (C++ struct), 787
mfxExtEncoderROI::Bottom (C++ member), 787
mfxExtEncoderROI::DeltaQP (C++ member), 787
mfxExtEncoderROI::Header (C++ member), 787
mfxExtEncoderROI::Left (C++ member), 787
mfxExtEncoderROI::NumROI (C++ member), 787
mfxExtEncoderROI::Right (C++ member), 787
mfxExtEncoderROI::ROI (C++ member), 787
mfxExtEncoderROI::ROIMode (C++ member), 787
mfxExtEncoderROI::Top (C++ member), 787
mfxExtEncToolsConfig (C++ struct), 821
mfxExtEncToolsConfig::AdaptiveB (C++ member), 822
mfxExtEncToolsConfig::AdaptiveI (C++ member), 822
mfxExtEncToolsConfig::AdaptiveLTR (C++ member), 822
mfxExtEncToolsConfig::AdaptivePyramidQuantB (C++ member), 822
mfxExtEncToolsConfig::AdaptivePyramidQuantP (C++ member), 822
mfxExtEncToolsConfig::AdaptiveQuantMatrices (C++ member), 822
mfxExtEncToolsConfig::AdaptiveRefB (C++ member), 822
mfxExtEncToolsConfig::AdaptiveRefP (C++ member), 822
mfxExtEncToolsConfig::BRC (C++ member), 822
mfxExtEncToolsConfig::BRCBufferHints (C++ member), 822
mfxExtEncToolsConfig::Header (C++ member), 822
mfxExtEncToolsConfig::SceneChange (C++ member), 822
mfxExtEncToolsConfig::Version (C++ member), 822
mfxExtHEVCPParam (C++ struct), 791
mfxExtHEVCPParam::GeneralConstraintFlags (C++ member), 791
mfxExtHEVCPParam::Header (C++ member), 791
mfxExtHEVCPParam::LCUSize (C++ member), 791
mfxExtHEVCPParam::PicHeightInLumaSamples (C++ member), 791
mfxExtHEVCPParam::PicWidthInLumaSamples (C++ member), 791
mfxExtHEVCPParam::SampleAdaptiveOffset (C++ member), 791
mfxExtHEVCRegion (C++ struct), 793
mfxExtHEVCRegion::Header (C++ member), 793
mfxExtHEVCRegion::RegionEncoding (C++ member), 793
mfxExtHEVCRegion::RegionId (C++ member), 793
mfxExtHEVCRegion::RegionType (C++ member), 793
mfxExtHEVCTiles (C++ struct), 790
mfxExtHEVCTiles::Header (C++ member), 791
mfxExtHEVCTiles::NumTileColumns (C++ member), 791
mfxExtHEVCTiles::NumTileRows (C++ member), 791
mfxExtInsertHeaders (C++ struct), 779
mfxExtInsertHeaders::Header (C++ member), 779
mfxExtInsertHeaders::PPS (C++ member), 779
mfxExtInsertHeaders::reserved (C++ member), 779
mfxExtInsertHeaders::SPS (C++ member), 779
mfxExtJPEGHuffmanTables (C++ struct), 818
mfxExtJPEGHuffmanTables::ACTables (C++ member), 819
mfxExtJPEGHuffmanTables::Bits (C++ member), 818
mfxExtJPEGHuffmanTables::DCTables (C++ member), 819
mfxExtJPEGHuffmanTables::Header (C++ member), 818
mfxExtJPEGHuffmanTables::NumACTable (C++ member), 818
mfxExtJPEGHuffmanTables::NumDCTable (C++ member), 818
mfxExtJPEGHuffmanTables::Values (C++ member), 818
mfxExtJPEGQuantTables (C++ struct), 818
mfxExtJPEGQuantTables::Header (C++ member), 818
mfxExtJPEGQuantTables::NumTable (C++ member), 818
mfxExtJPEGQuantTables::Qm (C++ member), 818
mfxExtMasteringDisplayColourVolume (C++ struct), 782
mfxExtMasteringDisplayColourVolume::DisplayPrimaries (C++ member), 782
mfxExtMasteringDisplayColourVolume::DisplayPrimaries

(*C++ member*), 782
mfxExtMasteringDisplayColourVolume::Header (*C++ member*), 782
mfxExtMasteringDisplayColourVolume::InsertPayload (*C++ member*), 782
mfxExtMasteringDisplayColourVolume::MaxDisplayLuminance (*C++ member*), 782
mfxExtMasteringDisplayColourVolume::MinDisplayLuminance (*C++ member*), 782
mfxExtMasteringDisplayColourVolume::WhitePoint (*C++ member*), 782
mfxExtMasteringDisplayColourVolume::WhitePointX (*C++ member*), 782
mfxExtMBDisableSkipMap (*C++ struct*), 791
mfxExtMBDisableSkipMap::Header (*C++ member*), 791
mfxExtMBDisableSkipMap::Map (*C++ member*), 791
mfxExtMBDisableSkipMap::MapSize (*C++ member*), 791
mfxExtMBForceIntra (*C++ struct*), 789
mfxExtMBForceIntra::Header (*C++ member*), 789
mfxExtMBForceIntra::Map (*C++ member*), 789
mfxExtMBForceIntra::MapSize (*C++ member*), 789
mfxExtMBQP (*C++ struct*), 790
mfxExtMBQP::BlockSize (*C++ member*), 790
mfxExtMBQP::DeltaQP (*C++ member*), 790
mfxExtMBQP::Header (*C++ member*), 790
mfxExtMBQP::Mode (*C++ member*), 790
mfxExtMBQP::NumQPAlloc (*C++ member*), 790
mfxExtMBQP::QP (*C++ member*), 790
mfxExtMBQP::QPmode (*C++ member*), 790
mfxExtMoveRect (*C++ struct*), 795
mfxExtMoveRect::DestBottom (*C++ member*), 795
mfxExtMoveRect::DestLeft (*C++ member*), 795
mfxExtMoveRect::DestRight (*C++ member*), 795
mfxExtMoveRect::DestTop (*C++ member*), 795
mfxExtMoveRect::Header (*C++ member*), 795
mfxExtMoveRect::NumRect (*C++ member*), 795
mfxExtMoveRect::Rect (*C++ member*), 795
mfxExtMoveRect::SourceLeft (*C++ member*), 795
mfxExtMoveRect::SourceTop (*C++ member*), 795
mfxExtMVCSeqDesc (*C++ struct*), 820
mfxExtMVCSeqDesc::Header (*C++ member*), 820
mfxExtMVCSeqDesc::NumOP (*C++ member*), 820
mfxExtMVCSeqDesc::NumOPAlloc (*C++ member*), 820
mfxExtMVCSeqDesc::NumRefsTotal (*C++ member*), 821
mfxExtMVCSeqDesc::NumView (*C++ member*), 820
mfxExtMVCSeqDesc::InsertPayloadToggle (*C++ member*), 820
mfxExtMVCSeqDesc::NumViewAlloc (*C++ member*), 820
mfxExtMVCSeqDesc::MaxDisplayLuminance (*C++ member*), 820
mfxExtMVCSeqDesc::NumViewId (*C++ member*), 820
mfxExtMVCSeqDesc::MinDisplayLuminance (*C++ member*), 820
mfxExtMVCSeqDesc::NumViewIdAlloc (*C++ member*), 820
mfxExtMVCSeqDesc::OP (*C++ member*), 820
mfxExtMVCSeqDesc::View (*C++ member*), 820
mfxExtMVCSeqDesc::ViewId (*C++ member*), 820
mfxExtMVCTargetViews (*C++ struct*), 821
mfxExtMVCTargetViews::Header (*C++ member*), 821
mfxExtMVCTargetViews::NumView (*C++ member*), 821
mfxExtMVCTargetViews::TemporalId (*C++ member*), 821
mfxExtMVCTargetViews::ViewId (*C++ member*), 821
mfxExtMVOVerPicBoundaries (*C++ struct*), 796
mfxExtMVOVerPicBoundaries::Header (*C++ member*), 796
mfxExtMVOVerPicBoundaries::StickBottom (*C++ member*), 796
mfxExtMVOVerPicBoundaries::StickLeft (*C++ member*), 796
mfxExtMVOVerPicBoundaries::StickRight (*C++ member*), 796
mfxExtMVOVerPicBoundaries::StickTop (*C++ member*), 796
mfxExtPartialBitstreamParam (*C++ struct*), 801
mfxExtPartialBitstreamParam::BlockSize (*C++ member*), 801
mfxExtPartialBitstreamParam::Granularity (*C++ member*), 801
mfxExtPartialBitstreamParam::Header (*C++ member*), 801
mfxExtPictureTimingSEI (*C++ struct*), 783
mfxExtPictureTimingSEI::ClockTimestampFlag (*C++ member*), 783
mfxExtPictureTimingSEI::CntDroppedFlag (*C++ member*), 783
mfxExtPictureTimingSEI::CountingType (*C++ member*), 783
mfxExtPictureTimingSEI::CtType (*C++ member*), 783
mfxExtPictureTimingSEI::DiscontinuityFlag (*C++ member*), 783
mfxExtPictureTimingSEI::FullTimestampFlag (*C++ member*), 783

mfxExtPictureTimingSEI::Header (*C++ member*), 783
mfxExtPictureTimingSEI::HoursFlag (*C++ member*), 783
mfxExtPictureTimingSEI::HoursValue (*C++ member*), 784
mfxExtPictureTimingSEI::MinutesFlag (*C++ member*), 783
mfxExtPictureTimingSEI::MinutesValue (*C++ member*), 784
mfxExtPictureTimingSEI::NFrames (*C++ member*), 783
mfxExtPictureTimingSEI::NuitFieldBasedFlag (*C++ member*), 783
mfxExtPictureTimingSEI::reserved (*C++ member*), 783
mfxExtPictureTimingSEI::SecondsFlag (*C++ member*), 783
mfxExtPictureTimingSEI::SecondsValue (*C++ member*), 783
mfxExtPictureTimingSEI::TimeOffset (*C++ member*), 784
mfxExtPictureTimingSEI::TimeStamp (*C++ member*), 784
mfxExtPredWeightTable (*C++ struct*), 793
mfxExtPredWeightTable::ChromaLog2WeightDenom (*C++ member*), 793
mfxExtPredWeightTable::ChromaWeightFlag (*C++ member*), 793
mfxExtPredWeightTable::Header (*C++ member*), 793
mfxExtPredWeightTable::LumaLog2WeightDenom (*C++ member*), 793
mfxExtPredWeightTable::LumaWeightFlag (*C++ member*), 793
mfxExtPredWeightTable::Weights (*C++ member*), 794
mfxExtThreadsParam (*C++ struct*), 780
mfxExtThreadsParam::Header (*C++ member*), 780
mfxExtThreadsParam::NumThread (*C++ member*), 780
mfxExtThreadsParam::Priority (*C++ member*), 780
mfxExtThreadsParam::reserved (*C++ member*), 780
mfxExtThreadsParam::SchedulingType (*C++ member*), 780
mfxExtTimeCode (*C++ struct*), 792
mfxExtTimeCode::DropFrameFlag (*C++ member*), 792
mfxExtTimeCode::Header (*C++ member*), 792
mfxExtTimeCode::TimeCodeHours (*C++ member*), 792
mfxExtTimeCode::TimeCodeMinutes (*C++ member*), 792
mfxExtTimeCode::TimeCodePictures (*C++ member*), 793
mfxExtTimeCode::TimeCodeSeconds (*C++ member*), 793
mfxExtVideoSignalInfo (*C++ struct*), 780
mfxExtVideoSignalInfo::ColourDescriptionPresent (*C++ member*), 780
mfxExtVideoSignalInfo::ColourPrimaries (*C++ member*), 780
mfxExtVideoSignalInfo::Header (*C++ member*), 780
mfxExtVideoSignalInfo::MatrixCoefficients (*C++ member*), 780
mfxExtVideoSignalInfo::TransferCharacteristics (*C++ member*), 780
mfxExtVideoSignalInfo::VideoFormat (*C++ member*), 780
mfxExtVideoSignalInfo::VideoFullRange (*C++ member*), 780
mfxExtVP8CodingOption (*C++ struct*), 817
mfxExtVP8CodingOption::CoeffTypeQPDelta (*C++ member*), 817
mfxExtVP8CodingOption::EnableMultipleSegments (*C++ member*), 817
mfxExtVP8CodingOption::Header (*C++ member*), 817
mfxExtVP8CodingOption::LoopFilterLevel (*C++ member*), 817
mfxExtVP8CodingOption::LoopFilterMbModeDelta (*C++ member*), 817
mfxExtVP8CodingOption::LoopFilterRefTypeDelta (*C++ member*), 817
mfxExtVP8CodingOption::LoopFilterType (*C++ member*), 817
mfxExtVP8CodingOption::NumFramesForIVFHeader (*C++ member*), 817
mfxExtVP8CodingOption::NumTokenPartitions (*C++ member*), 817
mfxExtVP8CodingOption::SegmentQPDelta (*C++ member*), 817
mfxExtVP8CodingOption::SharpnessLevel (*C++ member*), 817
mfxExtVP8CodingOption::Version (*C++ member*), 817
mfxExtVP8CodingOption::WriteIVFHeaders (*C++ member*), 817
mfxExtVP9Param (*C++ struct*), 799
mfxExtVP9Param::FrameHeight (*C++ member*), 799
mfxExtVP9Param::FrameWidth (*C++ member*), 799
mfxExtVP9Param::Header (*C++ member*), 799

mfxExtVP9Param::NumTileColumns (*C++ member*), 799
 mfxExtVP9Param::NumTileRows (*C++ member*), 799
 mfxExtVP9Param::QIndexDeltaChromaAC (*C++ member*), 799
 mfxExtVP9Param::QIndexDeltaChromaDC (*C++ member*), 799
 mfxExtVP9Param::QIndexDeltaLumaDC (*C++ member*), 799
 mfxExtVP9Param::WriteIVFHeaders (*C++ member*), 799
 mfxExtVP9Segmentation (*C++ struct*), 797
 mfxExtVP9Segmentation::Header (*C++ member*), 797
 mfxExtVP9Segmentation::NumSegmentIdAlloc
 mfxExtVP9Segmentation::NumSegments (*C++ member*), 797
 mfxExtVP9Segmentation::Segment (*C++ member*), 797
 mfxExtVP9Segmentation::SegmentId (*C++ member*), 797
 mfxExtVP9Segmentation::SegmentIdBlockSize
 mfxExtVP9TemporalLayers (*C++ struct*), 798
 mfxExtVP9TemporalLayers::Header (*C++ member*), 798
 mfxExtVP9TemporalLayers::Layer (*C++ member*), 798
 mfxExtVppAuxData (*C++ struct*), 805
 mfxExtVppAuxData::Header (*C++ member*), 805
 mfxExtVppAuxData::PicStruct (*C++ member*), 805
 mfxExtVPPColorFill (*C++ struct*), 812
 mfxExtVPPColorFill::Enable (*C++ member*), 812
 mfxExtVPPColorFill::Header (*C++ member*), 812
 mfxExtVPPComposite (*C++ struct*), 807
 mfxExtVPPComposite::B (*C++ member*), 808
 mfxExtVPPComposite::G (*C++ member*), 808
 mfxExtVPPComposite::Header (*C++ member*), 808
 mfxExtVPPComposite::InputStream (*C++ member*), 809
 mfxExtVPPComposite::NumInputStream (*C++ member*), 808
 mfxExtVPPComposite::NumTiles (*C++ member*), 808
 mfxExtVPPComposite::R (*C++ member*), 808
 mfxExtVPPComposite::U (*C++ member*), 808
 mfxExtVPPComposite::V (*C++ member*), 808
 mfxExtVPPComposite::Y (*C++ member*), 808
 mfxExtVPPDeinterlacing (*C++ struct*), 804
 mfxExtVPPDeinterlacing::Header (*C++ member*), 804
 mfxExtVPPDeinterlacing::Mode (*C++ member*), 804
 mfxExtVPPDeinterlacing::reserved (*C++ member*), 804
 mfxExtVPPDeinterlacing::TelecineLocation
 mfxExtVPPDeinterlacing::TelecinePattern
 mfxExtVPPDenoise (*C++ struct*), 802
 mfxExtVPPDenoise::DenoiseFactor (*C++ member*), 802
 mfxExtVPPDenoise::Header (*C++ member*), 802
 mfxExtVPPDetail (*C++ struct*), 803
 mfxExtVPPDetail::DetailFactor (*C++ member*), 803
 mfxExtVPPDetail::Header (*C++ member*), 803
 mfxExtVPPDoNotUse (*C++ struct*), 801
 mfxExtVPPDoNotUse::AlgList (*C++ member*), 802
 mfxExtVPPDoNotUse::Header (*C++ member*), 802
 mfxExtVPPDoNotUse::NumAlg (*C++ member*), 802
 mfxExtVPPDoUse (*C++ struct*), 802
 mfxExtVPPDoUse::AlgList (*C++ member*), 802
 mfxExtVPPDoUse::Header (*C++ member*), 802
 mfxExtVPPDoUse::NumAlg (*C++ member*), 802
 mfxExtVPPFieldProcessing (*C++ struct*), 809
 mfxExtVPPFieldProcessing::Header (*C++ member*), 809
 mfxExtVPPFieldProcessing::InField (*C++ member*), 809
 mfxExtVPPFieldProcessing::Mode (*C++ member*), 809
 mfxExtVPPFieldProcessing::OutField (*C++ member*), 809
 mfxExtVPPFrameRateConversion (*C++ struct*), 805
 mfxExtVPPFrameRateConversion::Algorithm
 mfxExtVPPFrameRateConversion::Header
 mfxExtVPPImageStab (*C++ struct*), 806
 mfxExtVPPImageStab::Header (*C++ member*), 806
 mfxExtVPPImageStab::Mode (*C++ member*), 806
 mfxExtVppMctf (*C++ struct*), 813
 mfxExtVppMctf::FilterStrength (*C++ member*), 813
 mfxExtVppMctf::Header (*C++ member*), 813
 mfxExtVPPMirroring (*C++ struct*), 812

mfxExtVPPMirroring::Header (*C++ member*), 812
 mfxExtVPPMirroring::Type (*C++ member*), 812
 mfxExtVPPProcAmp (*C++ struct*), 803
 mfxExtVPPProcAmp::Brightness (*C++ member*), 803
 mfxExtVPPProcAmp::Contrast (*C++ member*), 803
 mfxExtVPPProcAmp::Header (*C++ member*), 803
 mfxExtVPPProcAmp::Hue (*C++ member*), 803
 mfxExtVPPProcAmp::Saturation (*C++ member*), 803
 mfxExtVPPRotation (*C++ struct*), 811
 mfxExtVPPRotation::Angle (*C++ member*), 811
 mfxExtVPPRotation::Header (*C++ member*), 811
 mfxExtVPPScaling (*C++ struct*), 811
 mfxExtVPPScaling::Header (*C++ member*), 811
 mfxExtVPPScaling::InterpolationMethod (*C++ member*), 811
 mfxExtVPPScaling::ScalingMode (*C++ member*), 811
 mfxExtVPPVideoSignalInfo (*C++ struct*), 809
 mfxExtVPPVideoSignalInfo::Header (*C++ member*), 809
 mfxExtVPPVideoSignalInfo::NominalRange (*C++ member*), 809
 mfxExtVPPVideoSignalInfo::TransferMatrix
 (*C++ member*), 809
 mfxF32 (*C++ type*), 697
 mfxF64 (*C++ type*), 697
 mfxFrameAllocator (*C++ struct*), 764
 mfxFrameAllocator::Alloc (*C++ member*), 765
 mfxFrameAllocator::Free (*C++ member*), 766
 mfxFrameAllocator::GetHDL (*C++ member*), 765
 mfxFrameAllocator::Lock (*C++ member*), 765
 mfxFrameAllocator::pthis (*C++ member*), 765
 mfxFrameAllocator::Unlock (*C++ member*), 765
 mfxFrameAllocRequest (*C++ struct*), 764
 mfxFrameAllocRequest::AllocId (*C++ member*), 764
 mfxFrameAllocRequest::Info (*C++ member*), 764
 mfxFrameAllocRequest::NumFrameMin (*C++ member*), 764
 mfxFrameAllocRequest::NumFrameSuggested (*C++ member*), 764
 mfxFrameAllocRequest::Type (*C++ member*), 764
 mfxFrameAllocResponse (*C++ struct*), 764
 mfxFrameAllocResponse::AllocId (*C++ member*), 764
 mfxFrameAllocResponse::mids (*C++ member*), 764
 mfxFrameAllocResponse::NumFrameActual
 (*C++ member*), 764
 mfxFrameData (*C++ struct*), 754
 mfxFrameData::A (*C++ member*), 755
 mfxFrameData::A2RGB10 (*C++ member*), 756
 mfxFrameData::B (*C++ member*), 756
 mfxFrameData::Cb (*C++ member*), 756
 mfxFrameData::CbCr (*C++ member*), 756
 mfxFrameData::Corrupted (*C++ member*), 755
 mfxFrameData::Cr (*C++ member*), 756
 mfxFrameData::CrCb (*C++ member*), 756
 mfxFrameData::DataFlag (*C++ member*), 755
 mfxFrameData::ExtParam (*C++ member*), 756
 mfxFrameData::FrameOrder (*C++ member*), 755
 mfxFrameData::G (*C++ member*), 756
 mfxFrameData::Locked (*C++ member*), 755
 mfxFrameData::MemId (*C++ member*), 755
 mfxFrameData::MemType (*C++ member*), 755
 mfxFrameData::NumExtParam (*C++ member*), 755
 mfxFrameData::PitchHigh (*C++ member*), 755
 mfxFrameData::PitchLow (*C++ member*), 756
 mfxFrameData::R (*C++ member*), 756
 mfxFrameData::reserved (*C++ member*), 755
 mfxFrameData::TimeStamp (*C++ member*), 755
 mfxFrameData::U (*C++ member*), 756
 mfxFrameData::U16 (*C++ member*), 756
 mfxFrameData::UV (*C++ member*), 756
 mfxFrameData::V (*C++ member*), 756
 mfxFrameData::V16 (*C++ member*), 756
 mfxFrameData::VU (*C++ member*), 756
 mfxFrameData::Y (*C++ member*), 756
 mfxFrameData::Y16 (*C++ member*), 756
 mfxFrameData::Y410 (*C++ member*), 756
 mfxFrameInfo (*C++ struct*), 751
 mfxFrameInfo::AspectRatioH (*C++ member*), 751
 mfxFrameInfo::AspectRatioW (*C++ member*), 751
 mfxFrameInfo::BitDepthChroma (*C++ member*), 752
 mfxFrameInfo::BitDepthLuma (*C++ member*), 752
 mfxFrameInfo::BufferSize (*C++ member*), 752
 mfxFrameInfo::ChromaFormat (*C++ member*), 752
 mfxFrameInfo::CropH (*C++ member*), 751
 mfxFrameInfo::CropW (*C++ member*), 751
 mfxFrameInfo::CropX (*C++ member*), 751
 mfxFrameInfo::CropY (*C++ member*), 751
 mfxFrameInfo::FourCC (*C++ member*), 752
 mfxFrameInfo::FrameId (*C++ member*), 752

mfxFrameInfo::FrameRateExtD (*C++ member*), 751
mfxFrameInfo::FrameRateExtN (*C++ member*), 751
mfxFrameInfo::Height (*C++ member*), 752
mfxFrameInfo::PicStruct (*C++ member*), 752
mfxFrameInfo::reserved (*C++ member*), 752
mfxFrameInfo::reserved4 (*C++ member*), 752
mfxFrameInfo::Shift (*C++ member*), 752
mfxFrameInfo::Width (*C++ member*), 752
mfxFrameSurface1 (*C++ struct*), 760
mfxFrameSurface1::Data (*C++ member*), 760
mfxFrameSurface1::FrameInterface (*C++ member*), 760
mfxFrameSurface1::Info (*C++ member*), 760
mfxFrameSurfaceInterface (*C++ struct*), 757
mfxFrameSurfaceInterface::AddRef (*C++ member*), 757
mfxFrameSurfaceInterface::Context (*C++ member*), 757
mfxFrameSurfaceInterface::GetDeviceHandle (*C++ member*), 759
mfxFrameSurfaceInterface::GetNativeHandle (*C++ member*), 758
mfxFrameSurfaceInterface::GetRefCount (*C++ member*), 757
mfxFrameSurfaceInterface::Map (*C++ member*), 758
mfxFrameSurfaceInterface::Release (*C++ member*), 757
mfxFrameSurfaceInterface::Synchronize (*C++ member*), 759
mfxFrameSurfaceInterface::Unmap (*C++ member*), 758
mfxFrameSurfaceInterface::Version (*C++ member*), 757
MFXGetPriority (*C++ function*), 826
mfxHandleType (*C++ enum*), 734
mfxHandleType::MFX_HANDLE_CM_DEVICE (*C++ enumerator*), 734
mfxHandleType::MFX_HANDLE_D3D11_DEVICE (*C++ enumerator*), 734
mfxHandleType::MFX_HANDLE_D3D9_DEVICE_MANAGER (*C++ enumerator*), 734
mfxHandleType::MFX_HANDLE_DIRECT3D_DEVICE_MANAGER (*C++ enumerator*), 734
mfxHandleType::MFX_HANDLE_RESERVED1 (*C++ enumerator*), 734
mfxHandleType::MFX_HANDLE_RESERVED3 (*C++ enumerator*), 734
mfxHandleType::MFX_HANDLE_VA_CONFIG_ID (*C++ enumerator*), 734
mfxHandleType::MFX_HANDLE_VA_CONTEXT_ID (*C++ enumerator*), 734
mfxHandleType::MFX_HANDLE_VA_DISPLY (*C++ enumerator*), 734
mfxHDL (*C++ type*), 697
mfxHDLPair (*C++ struct*), 744
mfxHDLPair::first (*C++ member*), 745
mfxHDLPair::second (*C++ member*), 745
mfxI16 (*C++ type*), 697
mfxI16Pair (*C++ struct*), 744
mfxI16Pair::x (*C++ member*), 744
mfxI16Pair::y (*C++ member*), 744
mfxI32 (*C++ type*), 697
mfxI64 (*C++ type*), 697
mfxI8 (*C++ type*), 697
mfxIMPL (*C++ type*), 710
mfxImplCapsDeliveryFormat (*C++ enum*), 711
mfxImplCapsDeliveryFormat::MFX_IMPLCAPS_IMPLDESCST (*C++ enumerator*), 711
mfxImplDescription (*C++ struct*), 704
mfxImplDescription::accelerationMode (*C++ member*), 704
mfxImplDescription::ApiVersion (*C++ member*), 704
mfxImplDescription::Dec (*C++ member*), 704
mfxImplDescription::Enc (*C++ member*), 704
mfxImplDescription::ExtParam (*C++ member*), 704
mfxImplDescription::ExtParams (*C++ member*), 705
mfxImplDescription::Impl (*C++ member*), 704
mfxImplDescription::ImplName (*C++ member*), 704
mfxImplDescription::Keywords (*C++ member*), 704
mfxImplDescription::License (*C++ member*), 704
mfxImplDescription::NumExtParam (*C++ member*), 704
mfxImplDescription::reserved (*C++ member*), 704
mfxImplDescription::Reserved2 (*C++ member*), 705
mfxImplDescription::VendorID (*C++ member*), 704
mfxImplDescription::VendorImplID (*C++ member*), 704
mfxImplDescription::Version (*C++ member*), 704
mfxImplDescription::VPP (*C++ member*), 704
mfxInfoMFx (*C++ struct*), 747
mfxInfoMFx::Accuracy (*C++ member*), 748
mfxInfoMFx::BRCPParamMultiplier (*C++ member*), 747
mfxInfoMFx::BufferSizeInKB (*C++ member*), 748

mfxInfoMFX::CodecId (*C++ member*), 747
 mfxInfoMFX::CodecLevel (*C++ member*), 747
 mfxInfoMFX::CodecProfile (*C++ member*), 747
 mfxInfoMFX::Convergence (*C++ member*), 749
 mfxInfoMFX::DecodedOrder (*C++ member*), 749
 mfxInfoMFX::EnableReallocRequest (*C++ member*), 749
 mfxInfoMFX::EncodedOrder (*C++ member*), 749
 mfxInfoMFX::ExtendedPicStruct (*C++ member*), 749
 mfxInfoMFX::FrameInfo (*C++ member*), 747
 mfxInfoMFX::GopOptFlag (*C++ member*), 748
 mfxInfoMFX::GopPicSize (*C++ member*), 747
 mfxInfoMFX::GopRefDist (*C++ member*), 748
 mfxInfoMFX::ICQQuality (*C++ member*), 749
 mfxInfoMFX::IdrInterval (*C++ member*), 748
 mfxInfoMFX::InitialDelayInKB (*C++ member*), 748
 mfxInfoMFX::Interleaved (*C++ member*), 750
 mfxInfoMFX::InterleavedDec (*C++ member*), 750
 mfxInfoMFX::JPEGChromaFormat (*C++ member*), 749
 mfxInfoMFX::JPEGColorFormat (*C++ member*), 750
 mfxInfoMFX::LowPower (*C++ member*), 747
 mfxInfoMFX::MaxDecFrameBuffering (*C++ member*), 749
 mfxInfoMFX::MaxKbps (*C++ member*), 749
 mfxInfoMFX::NumRefFrame (*C++ member*), 749
 mfxInfoMFX::NumSlice (*C++ member*), 749
 mfxInfoMFX::QPB (*C++ member*), 749
 mfxInfoMFX::QPI (*C++ member*), 748
 mfxInfoMFX::QPP (*C++ member*), 748
 mfxInfoMFX::Quality (*C++ member*), 750
 mfxInfoMFX::reserved (*C++ member*), 747
 mfxInfoMFX::RestartInterval (*C++ member*), 750
 mfxInfoMFX::Rotation (*C++ member*), 749
 mfxInfoMFX::SamplingFactorH (*C++ member*), 750
 mfxInfoMFX::SamplingFactorV (*C++ member*), 750
 mfxInfoMFX::SliceGroupsPresent (*C++ member*), 749
 mfxInfoMFX::TargetKbps (*C++ member*), 748
 mfxInfoMFX::TargetUsage (*C++ member*), 747
 mfxInfoMFX::TimeStampCalc (*C++ member*), 749
 mfxInfoVPP (*C++ struct*), 767
 mfxInfoVPP::In (*C++ member*), 768
 mfxInfoVPP::Out (*C++ member*), 768
 MFXInit (*C++ function*), 823
 MFXInitEx (*C++ function*), 824
 mfxInitParam (*C++ struct*), 746
 mfxInitParam::ExternalThreads (*C++ member*), 746
 mfxInitParam::ExtParam (*C++ member*), 746
 mfxInitParam::GPUCopy (*C++ member*), 746
 mfxInitParam::Implementation (*C++ member*), 746
 mfxInitParam::NumExtParam (*C++ member*), 746
 mfxInitParam::Version (*C++ member*), 746
 MFXJoinSession (*C++ function*), 824
 mfxL32 (*C++ type*), 697
 MFXLoad (*C++ function*), 705
 mfxLoader (*C++ type*), 698
 mfxMediaAdapterType (*C++ enum*), 712
 mfxMediaAdapterType::MFX_MEDIA_DISCRETE
 (*C++ enumerator*), 712
 mfxMediaAdapterType::MFX_MEDIA_INTEGRATED
 (*C++ enumerator*), 712
 mfxMediaAdapterType::MFX_MEDIA_UNKNOWN
 (*C++ enumerator*), 712
 mfxMemId (*C++ type*), 697
 MFXMemory_GetSurfaceForDecode (*C++ function*), 828
 MFXMemory_GetSurfaceForEncode (*C++ function*), 827
 MFXMemory_GetSurfaceForVPP (*C++ function*), 827
 mfxMemoryFlags (*C++ enum*), 712
 mfxMemoryFlags::MFX_MAP_NOWAIT
 (*C++ enumerator*), 712
 mfxMemoryFlags::MFX_MAP_READ
 (*C++ enumerator*), 712
 mfxMemoryFlags::MFX_MAP_READ_WRITE
 (*C++ enumerator*), 712
 mfxMemoryFlags::MFX_MAP_WRITE
 (*C++ enumerator*), 712
 mfxMVCOperationPoint (*C++ struct*), 820
 mfxMVCOperationPoint::LevelIdc (*C++ member*), 820
 mfxMVCOperationPoint::NumTargetViews
 (*C++ member*), 820
 mfxMVCOperationPoint::NumViews (*C++ member*), 820
 mfxMVCOperationPoint::TargetViewId
 (*C++ member*), 820
 mfxMVCOperationPoint::TemporalId
 (*C++ member*), 820
 mfxMVCViewDependency (*C++ struct*), 819
 mfxMVCViewDependency::AnchorRefL0
 (*C++ member*), 819
 mfxMVCViewDependency::AnchorRefL1
 (*C++ member*), 819
 mfxMVCViewDependency::NonAnchorRefL0

(*C++ member*), 819
mfxMVCViewDependency::NumAnchorRefsL0 (*C++ member*), 819
mfxMVCViewDependency::NumAnchorRefsL1 (*C++ member*), 819
mfxMVCViewDependency::NumNonAnchorRefsL0MFXSetConfigFilterProperty (*C++ function*), 705
mfxMVCViewDependency::NumNonAnchorRefsL1MFXSetPriority (*C++ function*), 825
mfxPayload (*C++ struct*), 762
mfxPayload::BufSize (*C++ member*), 762
mfxPayload::CtrlFlags (*C++ member*), 762
mfxPayload::Data (*C++ member*), 762
mfxPayload::NumBit (*C++ member*), 762
mfxPayload::Type (*C++ member*), 762
mfxPlatform (*C++ struct*), 746
mfxPlatform::CodeName (*C++ member*), 746
mfxPlatform::DeviceId (*C++ member*), 746
mfxPlatform::MediaAdapterType (*C++ member*), 746
mfxPlatform::reserved (*C++ member*), 746
mfxPriority (*C++ enum*), 711
mfxPriority::MFX_PRIORITY_HIGH (*C++ enumerator*), 711
mfxPriority::MFX_PRIORITY_LOW (*C++ enumerator*), 711
mfxPriority::MFX_PRIORITY_NORMAL (*C++ enumerator*), 711
mfxQPandMode (*C++ struct*), 767
mfxQPandMode::DeltaQP (*C++ member*), 767
mfxQPandMode::Mode (*C++ member*), 767
mfxQPandMode::QP (*C++ member*), 767
MFXQueryIMPL (*C++ function*), 824
MFXQueryImplDescription (*C++ function*), 823
MFXQueryVersion (*C++ function*), 824
mfxRange32U (*C++ struct*), 744
mfxRange32U::Max (*C++ member*), 744
mfxRange32U::Min (*C++ member*), 744
mfxRange32U::Step (*C++ member*), 744
MFXReleaseImplDescription (*C++ function*), 823
mfx ResourceType (*C++ enum*), 713
mfx ResourceType::MFX_RESOURCE_DMA_RESOURCEExStatus::MFX_ERR_NONE_PARTIAL_OUTPUT (*C++ enumerator*), 709
mfx ResourceType::MFX_RESOURCE_DX11_TEXTUREExStatus::MFX_ERR_NOT_ENOUGH_BUFFER (*C++ enumerator*), 708
mfx ResourceType::MFX_RESOURCE_DX12_RESOURCEExStatus::MFX_ERR_NOT_FOUND (*C++ enumerator*), 708
mfx ResourceType::MFX_RESOURCE_DX9_SURFACEExStatus::MFX_ERR_NOT_INITIALIZED (*C++ enumerator*), 708
mfx ResourceType::MFX_RESOURCE_SYSTEM_SURFACEExStatus::MFX_ERR_NULL_PTR (*C++ enumerator*), 708

mfxStatus::MFX_ERR_REALLOC_SURFACE (*C++ enumerator*), 709
mfxStatus::MFX_ERR_UNDEFINED_BEHAVIOR (*C++ enumerator*), 709
mfxStatus::MFX_ERR_UNKNOWN (*C++ enumerator*), 708
mfxStatus::MFX_ERR_UNSUPPORTED (*C++ enumerator*), 708
mfxStatus::MFX_TASK_BUSY (*C++ enumerator*), 710
mfxStatus::MFX_TASK_DONE (*C++ enumerator*), 709
mfxStatus::MFX_TASK_WORKING (*C++ enumerator*), 709
mfxStatus::MFX_WRN_DEVICE_BUSY (*C++ enumerator*), 709
mfxStatus::MFX_WRN_FILTER_SKIPPED (*C++ enumerator*), 709
mfxStatus::MFX_WRN_IN_EXECUTION (*C++ enumerator*), 709
mfxStatus::MFX_WRN_INCOMPATIBLE_VIDEO_PARAM (*C++ enumerator*), 709
mfxStatus::MFX_WRN_OUT_OF_RANGE (*C++ enumerator*), 709
mfxStatus::MFX_WRN_PARTIAL_ACCELERATION (*C++ enumerator*), 709
mfxStatus::MFX_WRN_VALUE_NOT_CHANGED (*C++ enumerator*), 709
mfxStatus::MFX_WRN_VIDEO_PARAM_CHANGED (*C++ enumerator*), 709
mfxStructVersion (*C++ union*), 745
mfxStructVersion::Major (*C++ member*), 745
mfxStructVersion::Minor (*C++ member*), 745
mfxStructVersion::Version (*C++ member*), 745
mfxStructVersion::[anonymous] (*C++ member*), 745
mfxSyncPoint (*C++ type*), 698
mfxThreadTask (*C++ type*), 697
mfxU16 (*C++ type*), 697
mfxU32 (*C++ type*), 697
mfxU64 (*C++ type*), 697
mfxU8 (*C++ type*), 697
mfxUL32 (*C++ type*), 697
MFXUnload (*C++ function*), 705
mfxVariant (*C++ struct*), 699
mfxVariant::Data (*C++ member*), 699
mfxVariant::data (*C++ union*), 699
mfxVariant::data::F32 (*C++ member*), 699
mfxVariant::data::F64 (*C++ member*), 699
mfxVariant::data::I16 (*C++ member*), 699
mfxVariant::data::I32 (*C++ member*), 699
mfxVariant::data::I64 (*C++ member*), 699
mfxVariant::data::I8 (*C++ member*), 699
mfxVariant::data::Ptr (*C++ member*), 699
mfxVariant::data::U16 (*C++ member*), 699
mfxVariant::data::U32 (*C++ member*), 699
mfxVariant::data::U64 (*C++ member*), 699
mfxVariant::data::U8 (*C++ member*), 699
mfxVariant::Type (*C++ member*), 699
mfxVariant::Version (*C++ member*), 699
mfxVariantType (*C++ enum*), 698
mfxVariantType::MFX_VARIANT_TYPE_F32 (*C++ enumerator*), 698
mfxVariantType::MFX_VARIANT_TYPE_F64 (*C++ enumerator*), 699
mfxVariantType::MFX_VARIANT_TYPE_I16 (*C++ enumerator*), 698
mfxVariantType::MFX_VARIANT_TYPE_I32 (*C++ enumerator*), 698
mfxVariantType::MFX_VARIANT_TYPE_I64 (*C++ enumerator*), 698
mfxVariantType::MFX_VARIANT_TYPE_I8 (*C++ enumerator*), 698
mfxVariantType::MFX_VARIANT_TYPE_PTR (*C++ enumerator*), 699
mfxVariantType::MFX_VARIANT_TYPE_U16 (*C++ enumerator*), 698
mfxVariantType::MFX_VARIANT_TYPE_U32 (*C++ enumerator*), 698
mfxVariantType::MFX_VARIANT_TYPE_U64 (*C++ enumerator*), 698
mfxVariantType::MFX_VARIANT_TYPE_U8 (*C++ enumerator*), 698
mfxVariantType::MFX_VARIANT_TYPE_UNSET (*C++ enumerator*), 698
mfxVersion (*C++ union*), 745
mfxVersion::Major (*C++ member*), 745
mfxVersion::Minor (*C++ member*), 745
mfxVersion::Version (*C++ member*), 745
mfxVersion::[anonymous] (*C++ member*), 745
MFXVideoCORE_GetHandle (*C++ function*), 826
MFXVideoCORE_QueryPlatform (*C++ function*), 826
MFXVideoCORE_SetFrameAllocator (*C++ function*), 826
MFXVideoCORE_SetHandle (*C++ function*), 826
MFXVideoCORE_SyncOperation (*C++ function*), 827
MFXVideoDECODE_Close (*C++ function*), 834
MFXVideoDECODE_DecodeFrameAsync (*C++ function*), 835
MFXVideoDECODE_DecodeHeader (*C++ function*), 832
MFXVideoDECODE_GetDecodeStat (*C++ function*), 835
MFXVideoDECODE_GetPayload (*C++ function*), 835

MFXVideoDECODE_GetVideoParam (*C++ function*), 834
MFXVideoDECODE_Init (*C++ function*), 833
MFXVideoDECODE_Query (*C++ function*), 832
MFXVideoDECODE_QueryIOSurf (*C++ function*), 833
MFXVideoDECODE_Reset (*C++ function*), 834
MFXVideoDECODE_SetSkipMode (*C++ function*), 835
MFXVideoENCODE_Close (*C++ function*), 830
MFXVideoENCODE_EncodeFrameAsync (*C++ function*), 831
MFXVideoENCODE_GetEncodeStat (*C++ function*), 831
MFXVideoENCODE_GetVideoParam (*C++ function*), 830
MFXVideoENCODE_Init (*C++ function*), 829
MFXVideoENCODE_Query (*C++ function*), 828
MFXVideoENCODE_QueryIOSurf (*C++ function*), 829
MFXVideoENCODE_Reset (*C++ function*), 830
mfxVideoParam (*C++ struct*), 753
mfxVideoParam::AllocId (*C++ member*), 753
mfxVideoParam::AsyncDepth (*C++ member*), 753
mfxVideoParam::ExtParam (*C++ member*), 754
mfxVideoParam::IOPattern (*C++ member*), 753
mfxVideoParam::mfx (*C++ member*), 753
mfxVideoParam::NumExtParam (*C++ member*), 754
mfxVideoParam::Protected (*C++ member*), 753
mfxVideoParam::vpp (*C++ member*), 753
MFXVideoVPP_Close (*C++ function*), 838
MFXVideoVPP_GetVideoParam (*C++ function*), 839
MFXVideoVPP_GetVPPStat (*C++ function*), 839
MFXVideoVPP_Init (*C++ function*), 838
MFXVideoVPP_Query (*C++ function*), 837
MFXVideoVPP_QueryIOSurf (*C++ function*), 837
MFXVideoVPP_Reset (*C++ function*), 838
MFXVideoVPP_RunFrameVPPAsync (*C++ function*), 839
mfxVP9SegmentParam (*C++ struct*), 796
mfxVP9SegmentParam::FeatureEnabled (*C++ member*), 796
mfxVP9SegmentParam::LoopFilterLevelDelta (*C++ member*), 796
mfxVP9SegmentParam::QIndexDelta (*C++ member*), 796
mfxVP9SegmentParam::ReferenceFrame (*C++ member*), 796
mfxVP9TemporalLayer (*C++ struct*), 798
mfxVP9TemporalLayer::FrameRateScale (*C++ member*), 798
mfxVP9TemporalLayer::TargetKbps (*C++ member*), 798
mfxVPPCompInputStream (*C++ struct*), 806
mfxVPPCompInputStream::DstH (*C++ member*), 806
mfxVPPCompInputStream::DstW (*C++ member*), 806
mfxVPPCompInputStream::DstX (*C++ member*), 806
mfxVPPCompInputStream::DstY (*C++ member*), 806
mfxVPPCompInputStream::GlobalAlpha (*C++ member*), 807
mfxVPPCompInputStream::GlobalAlphaEnable (*C++ member*), 807
mfxVPPCompInputStream::LumaKeyEnable (*C++ member*), 806
mfxVPPCompInputStream::LumaKeyMax (*C++ member*), 806
mfxVPPCompInputStream::LumaKeyMin (*C++ member*), 806
mfxVPPCompInputStream::PixelAlphaEnable (*C++ member*), 807
mfxVPPCompInputStream::TileId (*C++ member*), 807
mfxVPPDescription (*C++ struct*), 702
mfxVPPDescription::filter (*C++ struct*), 703
mfxVPPDescription::filter::FilterFourCC (*C++ member*), 703
mfxVPPDescription::filter::MaxDelayInFrames (*C++ member*), 703
mfxVPPDescription::filter::MemDesc (*C++ member*), 703
mfxVPPDescription::filter::memdesc (*C++ struct*), 703
mfxVPPDescription::filter::memdesc::format (*C++ struct*), 703
mfxVPPDescription::filter::memdesc::format::NumOutP (*C++ member*), 704
mfxVPPDescription::filter::memdesc::format::OutForm (*C++ member*), 704
mfxVPPDescription::filter::memdesc::format::reserved (*C++ member*), 704
mfxVPPDescription::filter::memdesc::Formats (*C++ member*), 703
mfxVPPDescription::filter::memdesc::Height (*C++ member*), 703
mfxVPPDescription::filter::memdesc::MemHandleType (*C++ member*), 703
mfxVPPDescription::filter::memdesc::NumInFormats (*C++ member*), 703
mfxVPPDescription::filter::memdesc::reserved (*C++ member*), 703
mfxVPPDescription::filter::memdesc::Width

(C++ member), 703
`mfxVPPDescription::filter::NumMemTypes`
 (C++ member), 703
`mfxVPPDescription::filter::reserved`
 (C++ member), 703
`mfxVPPDescription::Filters` (C++ member),
 703
`mfxVPPDescription::NumFilters` (C++ member),
 703
`mfxVPPDescription::reserved` (C++ member),
 703
`mfxVPPDescription::Version` (C++ member),
 703
`mfxVPPStat` (C++ struct), 768
`mfxVPPStat::NumCachedFrame` (C++ member),
 768
`mfxVPPStat::NumFrame` (C++ member), 768
`mfxY410` (C++ struct), 754
`mfxY410::A` (C++ member), 754
`mfxY410::U` (C++ member), 754
`mfxY410::V` (C++ member), 754
`mfxY410::Y` (C++ member), 754
`minmag` (C++ function), 1350
`Model`, 218
`modf` (C++ function), 1326
`mul` (C++ function), 1191
`mulbyconj` (C++ function), 1193

N

`nearbyint` (C++ function), 1323
`nextafter` (C++ function), 1342
 Nominal feature, 218
`not_complete` (C macro), 373
`not_initialized` (C++ member), 374
`null_mutex` (C++ function), 634
`null_rw_mutex` (C++ function), 635

O

Observation, 218
`observe` (C++ function), 378
`on_scheduler_entry` (C++ function), 378
`on_scheduler_exit` (C++ function), 379
`onedal::data_format` (C++ class), 240
`onedal::data_layout` (C++ class), 240
`onedal::data_type` (C++ class), 241
`onedal::feature_info` (C++ class), 241
`onedal::feature_type` (C++ class), 241
`onedal::homogen_table` (C++ class), 238
`onedal::homogen_table::data_pointer`
 (C++ member), 238
`onedal::homogen_table::data_type` (C++
 member), 238
`onedal::homogen_table::feature_types_equal`
 (C++ member), 238

`onedal::homogen_table::homogen_table`
 (C++ function), 238
`onedal::homogen_table::operator=` (C++
 function), 238
`onedal::infer` (C++ function), 251, 264
`onedal::kmeans::desc` (C++ class), 246
`onedal::kmeans::desc::accuracy_threshold`
 (C++ member), 246
`onedal::kmeans::desc::cluster_count`
 (C++ member), 246
`onedal::kmeans::desc::desc` (C++ function),
 246
`onedal::kmeans::desc::max_iteration_count`
 (C++ member), 246
`onedal::kmeans::infer_input` (C++ class),
 249
`onedal::kmeans::infer_input::data` (C++
 member), 250
`onedal::kmeans::infer_input::infer_input`
 (C++ function), 249, 250
`onedal::kmeans::infer_input::model` (C++
 member), 250
`onedal::kmeans::infer_result` (C++ class),
 250
`onedal::kmeans::infer_result::infer_result`
 (C++ function), 250
`onedal::kmeans::infer_result::labels`
 (C++ member), 250
`onedal::kmeans::infer_result::objective_function_va`
 (C++ member), 250
`onedal::kmeans::method::by_default` (C++
 type), 245
`onedal::kmeans::method::lloyd` (C++
 struct), 245
`onedal::kmeans::model` (C++ class), 247
`onedal::kmeans::model::centroids` (C++
 member), 247
`onedal::kmeans::model::cluster_count`
 (C++ member), 247
`onedal::kmeans::model::model` (C++ func-
 tion), 247
`onedal::kmeans::train_input` (C++ class),
 247
`onedal::kmeans::train_input::data` (C++
 member), 248
`onedal::kmeans::train_input::initial_centroids`
 (C++ member), 248
`onedal::kmeans::train_input::train_input`
 (C++ function), 247, 248
`onedal::kmeans::train_result` (C++ class),
 248
`onedal::kmeans::train_result::iteration_count`
 (C++ member), 248
`onedal::kmeans::train_result::labels`

(C++ member), 248
 onedal::kmeans::train_result::model (C++ member), 248
 onedal::kmeans::train_result::objective_function (C++ member), 249
 onedal::kmeans::train_result::train_result (C++ function), 248
 onedal::knn::descriptor (C++ class), 253
 onedal::knn::descriptor::class_count (C++ member), 253
 onedal::knn::descriptor::descriptor (C++ function), 253
 onedal::knn::descriptor::neighbor_count (C++ member), 253
 onedal::knn::infer (C++ function), 257
 onedal::knn::infer_input (C++ class), 256
 onedal::knn::infer_input::data (C++ member), 256
 onedal::knn::infer_input::infer_input (C++ function), 256
 onedal::knn::infer_input::model (C++ member), 256
 onedal::knn::infer_result (C++ class), 256
 onedal::knn::infer_result::infer_result (C++ function), 256
 onedal::knn::infer_result::labels (C++ member), 256
 onedal::knn::method::bruteforce (C++ struct), 253
 onedal::knn::method::by_default (C++ type), 254
 onedal::knn::method::kd_tree (C++ struct), 253
 onedal::knn::model (C++ class), 254
 onedal::knn::model::model (C++ function), 254
 onedal::knn::train (C++ function), 255
 onedal::knn::train_input (C++ class), 254
 onedal::knn::train_input::data (C++ member), 254
 onedal::knn::train_input::labels (C++ member), 254
 onedal::knn::train_input::train_input (C++ function), 254
 onedal::knn::train_result (C++ class), 255
 onedal::knn::train_result::model (C++ member), 255
 onedal::knn::train_result::train_result (C++ function), 255
 onedal::pca::desc (C++ class), 259
 onedal::pca::desc::component_count (C++ member), 259
 onedal::pca::desc::desc (C++ function), 259
 onedal::pca::desc::set_deterministic (C++ member), 260
 onedal::pca::infer_input (C++ class), 263
 onedal::pca::infer_input::data (C++ member), 264
 onedal::pca::infer_input::infer_input (C++ function), 263
 onedal::pca::infer_input::model (C++ member), 263
 onedal::pca::infer_result (C++ class), 263
 onedal::pca::infer_result::infer_result (C++ function), 263
 onedal::pca::infer_result::transformed_data (C++ member), 263
 onedal::pca::method::by_default (C++ type), 259
 onedal::pca::method::cov (C++ struct), 259
 onedal::pca::method::svd (C++ struct), 259
 onedal::pca::model (C++ class), 260
 onedal::pca::model::component_count (C++ member), 260
 onedal::pca::model::eigenvectors (C++ member), 260
 onedal::pca::model::model (C++ function), 260
 onedal::pca::train_input (C++ class), 261
 onedal::pca::train_input::data (C++ member), 261
 onedal::pca::train_input::train_input (C++ function), 261
 onedal::pca::train_result (C++ class), 261
 onedal::pca::train_result::eigenvalues (C++ member), 262
 onedal::pca::train_result::means (C++ member), 261
 onedal::pca::train_result::model (C++ member), 261
 onedal::pca::train_result::train_result (C++ function), 261
 onedal::pca::train_result::variances (C++ member), 262
 onedal::table (C++ class), 236
 onedal::table::feature_count (C++ member), 237
 onedal::table::is_empty (C++ member), 237
 onedal::table::metadata (C++ member), 237
 onedal::table::observation_count (C++ member), 237
 onedal::table::operator= (C++ function), 236
 onedal::table::table (C++ function), 236
 onedal::table_meta (C++ class), 239
 onedal::table_meta::feature (C++ member), 239
 onedal::table_meta::feature_count (C++ member), 239

onedal::table_meta::format (*C++ member*), 240
 onedal::table_meta::is_contiguous (*C++ member*), 240
 onedal::table_meta::is_homogeneous (*C++ function*), 240
 onedal::table_meta::layout (*C++ member*), 240
 onedal::train (*C++ function*), 249, 262
 onemkl::blas::asum (*C++ function*), 854, 855
 onemkl::blas::axpy (*C++ function*), 856
 onemkl::blas::axpy_batch (*C++ function*), 961–963
 onemkl::blas::copy (*C++ function*), 858
 onemkl::blas::dot (*C++ function*), 859, 860
 onemkl::blas::dotc (*C++ function*), 863
 onemkl::blas::dotu (*C++ function*), 865
 onemkl::blas::gbmv (*C++ function*), 885, 886
 onemkl::blas::gemm (*C++ function*), 937, 938
 onemkl::blas::gemm_batch (*C++ function*), 964, 966, 968
 onemkl::blas::gemm_bias (*C++ function*), 978, 979
 onemkl::blas::gemmt (*C++ function*), 975, 976
 onemkl::blas::gemv (*C++ function*), 887, 888
 onemkl::blas::ger (*C++ function*), 890
 onemkl::blas::gerc (*C++ function*), 892
 onemkl::blas::geru (*C++ function*), 894
 onemkl::blas::hbmv (*C++ function*), 896
 onemkl::blas::hemm (*C++ function*), 940, 941
 onemkl::blas::hemv (*C++ function*), 898, 899
 onemkl::blas::her (*C++ function*), 900, 901
 onemkl::blas::her2 (*C++ function*), 902, 903
 onemkl::blas::her2k (*C++ function*), 945, 946
 onemkl::blas::herk (*C++ function*), 942, 943
 onemkl::blas::hpmv (*C++ function*), 904, 905
 onemkl::blas::hpr (*C++ function*), 906, 907
 onemkl::blas::hpr2 (*C++ function*), 908, 909
 onemkl::blas::iamax (*C++ function*), 881
 onemkl::blas::iamin (*C++ function*), 883
 onemkl::blas::nrm2 (*C++ function*), 866, 867
 onemkl::blas::rot (*C++ function*), 868
 onemkl::blas::rotg (*C++ function*), 870
 onemkl::blas::rotm (*C++ function*), 871, 873
 onemkl::blas::rotmg (*C++ function*), 875, 876
 onemkl::blas::sbmv (*C++ function*), 910, 911
 onemkl::blas::scal (*C++ function*), 878
 onemkl::blas::sdssdot (*C++ function*), 861, 862
 onemkl::blas::spmv (*C++ function*), 913
 onemkl::blas::spr (*C++ function*), 915
 onemkl::blas::spr2 (*C++ function*), 916, 917
 onemkl::blas::swap (*C++ function*), 879, 880
 onemkl::blas::symm (*C++ function*), 947, 948
 onemkl::blas::symv (*C++ function*), 918, 919
 onemkl::blas::syr (*C++ function*), 920, 921
 onemkl::blas::syr2 (*C++ function*), 922, 923
 onemkl::blas::syr2k (*C++ function*), 952, 953
 onemkl::blas::syrk (*C++ function*), 950, 951
 onemkl::blas::tbbmv (*C++ function*), 924, 925
 onemkl::blas::tbsv (*C++ function*), 926, 927
 onemkl::blas::tpmv (*C++ function*), 928, 929
 onemkl::blas::tpsv (*C++ function*), 930, 931
 onemkl::blas::trmm (*C++ function*), 955, 956
 onemkl::blas::trmv (*C++ function*), 932, 933
 onemkl::blas::trsm (*C++ function*), 958, 959
 onemkl::blas::trsm_batch (*C++ function*), 970, 972, 973
 onemkl::blas::trsv (*C++ function*), 934, 935
 onemkl::lapack::gebrd (*C++ function*), 1031, 1032
 onemkl::lapack::gebrd_scratchpad_size (*C++ function*), 1034
 onemkl::lapack::geqrf (*C++ function*), 982, 983
 onemkl::lapack::geqrf_batch (*C++ function*), 1091
 onemkl::lapack::geqrf_scratchpad_size (*C++ function*), 984
 onemkl::lapack::gesvd (*C++ function*), 1035, 1037
 onemkl::lapack::gesvd_scratchpad_size (*C++ function*), 1039
 onemkl::lapack::getrf (*C++ function*), 985, 986
 onemkl::lapack::getrf_batch (*C++ function*), 1092
 onemkl::lapack::getrf_scratchpad_size (*C++ function*), 988
 onemkl::lapack::getri (*C++ function*), 989
 onemkl::lapack::getri_batch (*C++ function*), 1093
 onemkl::lapack::getri_scratchpad_size (*C++ function*), 991
 onemkl::lapack::getrs (*C++ function*), 992, 993
 onemkl::lapack::getrs_batch (*C++ function*), 1094
 onemkl::lapack::getrs_scratchpad_size (*C++ function*), 994
 onemkl::lapack::heevd (*C++ function*), 1041, 1042
 onemkl::lapack::heevd_scratchpad_size (*C++ function*), 1043
 onemkl::lapack::hegvd (*C++ function*), 1044, 1046
 onemkl::lapack::hegvd_scratchpad_size (*C++ function*), 1048
 onemkl::lapack::hetrd (*C++ function*), 1049, 1050
 onemkl::lapack::hetrd_scratchpad_size (*C++ function*), 1052

onemkl::lapack::orgbr (*C++ function*), 1054, 1055
 onemkl::lapack::orgbr_scratchpad_size (*C++ function*), 1056
 onemkl::lapack::orgqr (*C++ function*), 996, 997
 onemkl::lapack::orgqr_batch (*C++ function*), 1096
 onemkl::lapack::orgqr_scratchpad_size (*C++ function*), 998
 onemkl::lapack::orgtr (*C++ function*), 1058
 onemkl::lapack::orgtr_scratchpad_size (*C++ function*), 1060
 onemkl::lapack::ormqr (*C++ function*), 999, 1000
 onemkl::lapack::ormqr_scratchpad_size (*C++ function*), 1002
 onemkl::lapack::ormtr (*C++ function*), 1061, 1062
 onemkl::lapack::ormtr_scratchpad_size (*C++ function*), 1064
 onemkl::lapack::potrf (*C++ function*), 1003, 1004
 onemkl::lapack::potrf_batch (*C++ function*), 1097
 onemkl::lapack::potrf_scratchpad_size (*C++ function*), 1006
 onemkl::lapack::potri (*C++ function*), 1007, 1008
 onemkl::lapack::potri_scratchpad_size (*C++ function*), 1009
 onemkl::lapack::potrs (*C++ function*), 1010, 1011
 onemkl::lapack::potrs_batch (*C++ function*), 1098
 onemkl::lapack::potrs_scratchpad_size (*C++ function*), 1013
 onemkl::lapack::syevd (*C++ function*), 1065, 1066
 onemkl::lapack::syevd_scratchpad_size (*C++ function*), 1068
 onemkl::lapack::sygvd (*C++ function*), 1069, 1071
 onemkl::lapack::sygvd_scratchpad_size (*C++ function*), 1073
 onemkl::lapack::sytrd (*C++ function*), 1074, 1075
 onemkl::lapack::sytrd_scratchpad_size (*C++ function*), 1077
 onemkl::lapack::sytrf (*C++ function*), 1014, 1016
 onemkl::lapack::sytrf_scratchpad_size (*C++ function*), 1017
 onemkl::lapack::trtrs (*C++ function*), 1019, 1020
 onemkl::lapack::trtrs_scratchpad_size (*C++ function*), 1021
 onemkl::lapack::ungbr (*C++ function*), 1078, 1080
 onemkl::lapack::ungbr_scratchpad_size (*C++ function*), 1081
 onemkl::lapack::ungqr (*C++ function*), 1023, 1024
 onemkl::lapack::ungqr_scratchpad_size (*C++ function*), 1025
 onemkl::lapack::ungtr (*C++ function*), 1083
 onemkl::lapack::ungtr_scratchpad_size (*C++ function*), 1085
 onemkl::lapack::unmqr (*C++ function*), 1026, 1027
 onemkl::lapack::unmqr_scratchpad_size (*C++ function*), 1029
 onemkl::lapack::unmtr (*C++ function*), 1086, 1087
 onemkl::lapack::unmtr_scratchpad_size (*C++ function*), 1089
 onemkl::sparse::gemm (*C++ function*), 1102
 onemkl::sparse::gemv (*C++ function*), 1104
 onemkl::sparse::gemvdot (*C++ function*), 1106
 onemkl::sparse::gemvOptimize (*C++ function*), 1107
 onemkl::sparse::matrixInit (*C++ function*), 1100
 onemkl::sparse::setCSRstructure (*C++ function*), 1101
 onemkl::sparse::symv (*C++ function*), 1108
 onemkl::sparse::trmv (*C++ function*), 1110
 onemkl::sparse::trmvOptimize (*C++ function*), 1112
 onemkl::sparse::trsv (*C++ function*), 1113
 onemkl::sparse::trsvOptimize (*C++ function*), 1115
 Online mode, 220
 operator bool (*C++ function*), 294
 operator split (*C++ function*), 315
 operator != (*C++ function*), 619, 621, 622
 operator+ (*C++ function*), 273
 operator= (*C++ function*), 273, 280–283, 609
 operator== (*C++ function*), 619, 621, 622
 operator& (*C++ function*), 302
 operator- (*C++ function*), 273
 operator< (*C++ function*), 268, 273
 Ordinal feature, 219
 Outlier, 219
 output_ports (*C++ function*), 355, 358
 overwrite_node (*C++ function*), 338

P

parameter::max_allowed_parallelism (*C++*

enum), 370
parameter::*thread_stack_size* (*C++ enum*), 370
pow (*C++ function*), 1219
pow2o3 (*C++ function*), 1216
pow3o2 (*C++ function*), 1217
powr (*C++ function*), 1224
powx (*C++ function*), 1222
priority_queue_node (*C++ function*), 344
proportional_split (*C++ function*), 315

Q

queue_node (*C++ function*), 343
queuing_mutex (*C++ function*), 632
queuing_rw_mutex (*C++ function*), 633

R

R::~*R* (*C++ function*), 268
R::*empty* (*C++ function*), 268
R::*is_divisible* (*C++ function*), 269
R::*R* (*C++ function*), 268, 269
range (*C++ function*), 615
Ratio feature, 219
Reduction::*operator()* (*C++ function*), 271
Reference-counted object, 220
Regression, 219
remainder (*C++ function*), 1203
reset (*C++ function*), 317, 369
Response, 219
right (*C++ function*), 315
rint (*C++ function*), 1325
round (*C++ function*), 1322
run (*C++ function*), 372
run_and_wait (*C++ function*), 372
RWM::~*scoped_lock* (*C++ function*), 277
RWM::*scoped_lock* (*C++ function*), 277
RWM::*scoped_lock* (*C++ type*), 277
RWM::*scoped_lock*::*acquire* (*C++ function*), 277
RWM::*scoped_lock*:: *downgrade_to_reader* (*C++ function*), 277
RWM::*scoped_lock*::*release* (*C++ function*), 277
RWM::*scoped_lock*::*try_acquire* (*C++ function*), 277
RWM::*scoped_lock*::*upgrade_to_writer* (*C++ function*), 277

S

S::~*S* (*C++ function*), 283
S::*operator()* (*C++ function*), 283
S::*S* (*C++ function*), 283
scalable_allocation_command (*C function*), 626
scalable_allocation_mode (*C++ function*), 625
scalable_msize (*C++ function*), 625
Scan::*operator()* (*C++ function*), 273
scoped_lock (*C++ class*), 628–633, 635
sequencer_node (*C++ function*), 346
set_external_ports (*C++ function*), 358
set_mode (*C++ function*), 1330
set_status (*C++ function*), 1332
Setter, 220
setValue (*C++ function*), 1119
sin (*C++ function*), 1247
sincos (*C++ function*), 1249
sind (*C++ function*), 1276
sinh (*C++ function*), 1282
sinpi (*C++ function*), 1263
size (*C++ function*), 306, 614, 617
size_type (*C++ type*), 305
skip_ahead (*C++ function*), 1145
speculative_spin_mutex (*C++ function*), 630
speculative_spin_rw_mutex (*C++ function*), 631
spin_mutex (*C++ function*), 628
spin_rw_mutex (*C++ function*), 629
SPIR-V, 220
split_node (*C++ function*), 354
sqr (*C++ function*), 1189
sqrt (*C++ function*), 1209
stop (*C++ function*), 303
sub (*C++ function*), 1187
Supervised learning, 219
swap (*C++ function*), 268
SYCL, 220

T

T::*begin* (*C++ function*), 272
T::*end* (*C++ function*), 272
T::*release_wait* (*C++ function*), 281
T::*reserve_wait* (*C++ function*), 281
T::*try_put* (*C++ function*), 281
Table, 220
tag (*C++ function*), 362
tagged_msg (*C++ function*), 362
tan (*C++ function*), 1252
tand (*C++ function*), 1278
tanh (*C++ function*), 1285
tanpi (*C++ function*), 1265
task_arena (*C++ function*), 375
task_group (*C++ function*), 372
task_group_context (*C++ function*), 369
task_scheduler_observer (*C++ function*), 378
TBBMALLOC_CLEAN_ALL_BUFFERS (*C macro*), 626
TBBMALLOC_CLEAN_THREAD_BUFFERS (*C macro*), 626

TBBMALLOC_SET_HUGE_SIZE_THRESHOLD (*C macro*), 626
TBBMALLOC_SET_SOFT_HEAP_LIMIT (*C macro*), 626
TBBMALLOC_USE_HUGE_PAGES (*C macro*), 625
terminate (*C++ function*), 375
tgamma (*C++ function*), 1312
Training, 219
Training set, 219
traits (*C++ function*), 369
traits_type::fp_settings (*C++ enum*), 368
trunc (*C++ function*), 1320
try_get (*C++ function*), 327, 338, 340, 342, 343, 346, 348, 349, 356
try_lock (*C++ function*), 628, 629, 634, 635
try_lock_shared (*C++ function*), 629, 635
try_put (*C++ function*), 338, 340, 342–346, 348, 349, 354

U

unlock (*C++ function*), 628, 629, 634, 635
unlock_shared (*C++ function*), 629, 635
Unsupervised learning, 219
upstream_resource (*C++ function*), 623

V

Value::~Value (*C++ function*), 273
Value::Value (*C++ function*), 273

W

wait (*C++ function*), 372
wait_for_all (*C++ function*), 317
Workload, 220
write_once_node (*C++ function*), 340

X

X::X (*C++ function*), 269