SYCL™ Specification

Generic heterogeneous computing for modern C++

Version 2020 pre-provisional

Document Revision: 0

Revision Date: May 7, 2020

Git revision: tags/SYCL-2020/pre-provisional-0-g28e4e4586

Khronos® SYCL™ Working Group

Editors: Ronan Keryell, Maria Rovatsou & Lee Howes

# Contents


**1 Acknowledgements** **21**

**2 Introduction** **23**

**3 SYCL architecture** **25**

3.1 Overview . . . 25
3.2 Anatomy of a SYCL application . . . 26
3.3 Normative references . . . 27
3.4 The SYCL platform model . . . 27
3.5 The SYCL backend model . . . 28
3.5.1 Platform mixed version support . . . 29
3.6 SYCL execution model . . . 29
3.6.1 SYCL application execution model . . . 30
3.6.1.1 SYCL backend resources managed by the SYCL application . . . 30
3.6.1.2 SYCL command groups and execution order . . . 31
3.6.2 SYCL kernel execution model . . . 32
3.7 Memory model . . . 33
3.7.1 SYCL application memory model . . . 33
3.7.2 SYCL device memory model . . . 36
3.7.2.1 Access to memory . . . 37
3.7.2.2 Memory consistency inside SYCL kernels . . . 37
3.7.2.3 Atomic operations . . . 39
3.8 The SYCL programming model . . . 39
3.8.1 Minimum version of C++ . . . 39
3.8.2 Alignment with future versions of C++ . . . 39
3.8.3 Basic data parallel kernels . . . 39
3.8.4 Work-group data parallel kernels . . . 40
3.8.5 Hierarchical data parallel kernels . . . 40
3.8.6 Kernels that are not launched over parallel instances . . . 41
3.8.7 Pre-defined kernels . . . 41
3.8.8 Synchronization . . . 41
3.8.8.1 Synchronization in the SYCL application . . . 41
3.8.8.2 Synchronization in SYCL kernels . . . 42
3.8.9 Error handling . . . 42
3.8.10 Fallback mechanism . . . 42
3.8.11 Scheduling of kernels and data movement . . . 43
3.8.12 Managing object lifetimes . . . 43
3.8.13 Device discovery and selection . . . 44
3.8.14 Interfacing with SYCL backend API . . . 44
3.9 Memory objects . . . 44
3.10 SYCL device compiler . . . 45
3.10.1 Building a SYCL program . . . 46
3.10.2 Naming of kernels . . . 46

3.11 Language restrictions in kernels . . . 47
3.11.1 SYCL linker . . . 47
3.11.2 Functions and data types available in kernels . . . 47
3.12 Execution of kernels on the SYCL host device . . . 47
3.13 Endianness support . . . 47
3.14 Example SYCL application . . . 47

**4 SYCL programming interface** **51**
4.1 Backends . . . 51
4.1.1 Backend macros . . . 51
4.2 Generic vs non-generic SYCL . . . 51
4.3 Header files and namespaces . . . 52
4.4 Class availability . . . 52
4.5 Common interface . . . 53
4.5.1 Param traits class . . . 53
4.5.2 Backend interoperability . . . 53
4.5.2.1 Type traits `backend_traits` . . . 53
4.5.2.2 Template function `get_native` . . . 54
4.5.2.3 Template functions `make_*` . . . 54
4.5.3 Common reference semantics . . . 55
4.5.4 Common by-value semantics . . . 58
4.5.5 Properties . . . 59
4.5.5.1 Properties interface . . . 60
4.6 SYCL runtime classes . . . 62
4.6.1 Device selection . . . 62
4.6.1.1 Device selector . . . 62
4.6.2 Platform class . . . 64
4.6.2.1 Platform interface . . . 65
4.6.2.2 Platform information descriptors . . . 67
4.6.3 Context class . . . 68
4.6.3.1 Context interface . . . 68
4.6.3.2 Context information descriptors . . . 70
4.6.3.3 Context properties . . . 71
4.6.4 Device class . . . 71
4.6.4.1 Device interface . . . 72
4.6.4.2 Device information descriptors . . . 75
4.6.4.3 Device aspects . . . 87
4.6.5 Queue class . . . 90
4.6.5.1 Queue interface . . . 91
4.6.5.2 Queue information descriptors . . . 99
4.6.5.3 Queue properties . . . 99
4.6.5.4 Queue error handling . . . 100
4.6.6 Event class . . . 100
4.6.6.1 Event information and profiling descriptors . . . 104
4.7 Data access and storage in SYCL . . . 104
4.7.1 Host allocation . . . 105
4.7.1.1 Default allocators . . . 105
4.7.2 Buffers . . . 105
4.7.2.1 Buffer interface . . . 106
4.7.2.2 Buffer properties . . . 117
4.7.2.3 Buffer synchronization rules . . . 118

4.7.3 Images . . . 120
4.7.3.1 Unsampled image interface . . . 120
4.7.3.2 Sampled image interface . . . 130
4.7.3.3 Image properties . . . 135
4.7.3.4 Image synchronization rules . . . 136
4.7.4 Sharing host memory with the SYCL data management classes . . . 136
4.7.4.1 Default behavior . . . 137
4.7.4.2 SYCL ownership of the host memory . . . 137
4.7.4.3 Shared SYCL ownership of the host memory . . . 137
4.7.5 Synchronization primitives . . . 138
4.7.6 Accessors . . . 139
4.7.6.1 Access targets . . . 139
4.7.6.2 Access modes . . . 140
4.7.6.3 Access tags . . . 141
4.7.6.4 Device and host accessors . . . 141
4.7.6.5 Placeholder accessor . . . 142
4.7.6.6 Accessor declaration . . . 142
4.7.6.7 Constness of the accessor data type . . . 143
4.7.6.8 Implicit accessor conversions . . . 143
4.7.6.9 Device buffer accessor . . . 144
4.7.6.9.1 Device buffer accessor interface . . . 145
4.7.6.9.2 Device buffer accessor properties . . . 155
4.7.6.10 Host buffer accessor . . . 156
4.7.6.10.1 Host buffer accessor interface . . . 156
4.7.6.10.2 Host buffer accessor properties . . . 164
4.7.6.11 Local accessor . . . 164
4.7.6.11.1 Local accessor interface . . . 165
4.7.6.11.2 Local accessor properties . . . 170
4.7.6.12 Image accessor . . . 170
4.7.6.12.1 Image accessor interface . . . 171
4.7.6.12.2 Image accessor properties . . . 175
4.7.7 Address space classes . . . 175
4.7.7.1 Multi-pointer class . . . 175
4.7.7.2 Explicit pointer aliases . . . 186
4.7.8 Samplers . . . 187
4.8 Unified shared memory . . . 189
4.8.1 USM introduction . . . 189
4.8.2 SYCL memory management . . . 190
4.8.3 Unified addressing . . . 190
4.8.4 Unified shared memory . . . 190
4.8.4.1 Explicit USM . . . 190
4.8.4.2 Restricted USM . . . 190
4.8.4.3 Concurrent USM . . . 190
4.8.4.4 System USM . . . 191
4.8.5 USM allocations . . . 191
4.8.6 C++ allocator interface . . . 191
4.8.7 Utility functions . . . 193
4.8.7.1 Explicit USM . . . 193
4.8.7.1.1 `malloc` . . . 193
4.8.7.1.2 `aligned_alloc` . . . 194
4.8.7.1.3 `memcpy` . . . 195

4.8.7.1.4 memset . . . 195
4.8.7.1.5 fill . . . 196
4.8.7.2 Restricted USM . . . 196
4.8.7.2.1 malloc . . . 196
4.8.7.2.2 aligned_alloc . . . 198
4.8.7.2.3 Performance hints . . . 200
4.8.7.2.3.1 prefetch . . . 200
4.8.7.3 Concurrent USM . . . 200
4.8.7.3.1 Performance hints . . . 200
4.8.7.3.1.1 prefetch . . . 200
4.8.7.3.1.2 mem_advise . . . 200
4.8.7.4 General . . . 201
4.8.7.4.1 malloc . . . 201
4.8.7.4.2 aligned_alloc . . . 202
4.8.7.4.3 free . . . 203
4.8.8 Unified shared memory information . . . 203
4.8.8.1 Pointer queries . . . 203
4.8.8.1.1 get_pointer_type . . . 203
4.8.8.1.2 get_pointer_device . . . 204
4.9 SYCL scheduling . . . 204
4.9.1 DAGs without accessors . . . 204
4.9.2 Coarse grain DAGs with depends_on . . . 204
4.10 Expressing parallelism through kernels . . . 205
4.10.1 Ranges and index space identifiers . . . 205
4.10.1.1 range class . . . 206
4.10.1.2 nd_range class . . . 208
4.10.1.3 id class . . . 209
4.10.1.4 item class . . . 212
4.10.1.5 nd_item class . . . 214
4.10.1.6 h_item class . . . 217
4.10.1.7 group class . . . 220
4.10.1.8 sub_group class . . . 225
4.10.1.9 device_event class . . . 226
4.10.2 Reduction variables . . . 227
4.10.2.1 reduction interface . . . 228
4.10.2.2 reducer class . . . 230
4.10.3 Command group scope . . . 234
4.10.4 Command group handler class . . . 235
4.10.5 Class kernel_handler . . . 238
4.10.5.1 Constructors . . . 238
4.10.5.2 Member functions . . . 238
4.10.6 SYCL functions for adding requirements . . . 239
4.10.7 SYCL functions for invoking kernels . . . 239
4.10.7.1 single_task invoke . . . 244
4.10.7.2 parallel_for invoke . . . 244
4.10.7.3 Parallel for hierarchical invoke . . . 248
4.10.8 SYCL functions for explicit memory operations . . . 251
4.11 Host tasks . . . 253
4.11.1 Overview . . . 253
4.11.2 Class interop_handle . . . 255
4.11.2.1 Constructors . . . 255

4.11.2.2 Template member functions get_native_* . . . 255
4.11.3 Handler class . . . 257
4.11.3.1 Member function host_task . . . 257
4.11.4 Functions for using a module . . . 257
4.11.5 Functions for using specialization constants . . . 258
4.12 Kernel class . . . 258
4.12.1 Kernel information descriptors . . . 260
4.13 Modules . . . 262
4.13.1 Overview . . . 262
4.13.2 Specialization constants . . . 263
4.13.2.1 Synopsis . . . 263
4.13.3 Enum class module_state . . . 265
4.13.4 Class template specialization_id . . . 266
4.13.4.1 Constructors . . . 266
4.13.4.2 Special member functions . . . 266
4.13.5 Class template module . . . 266
4.13.5.1 Constructors . . . 267
4.13.5.2 Member functions . . . 267
4.13.6 Free functions . . . 269
4.13.7 Namespace this_module . . . 271
4.13.7.1 Type traits . . . 271
4.13.7.2 Free functions . . . 271
4.14 Defining kernels . . . 272
4.14.1 Defining kernels as named function objects . . . 272
4.14.2 Defining kernels as lambda functions . . . 273
4.14.3 Defining kernels using modules . . . 274
4.14.4 Rules for parameter passing to kernels . . . 275
4.15 Error handling . . . 276
4.15.1 Error handling rules . . . 276
4.15.1.1 Asynchronous error handler . . . 276
4.15.1.2 Behavior without an async_handler . . . 276
4.15.1.3 Priorities of async_handlers . . . 276
4.15.1.4 Asynchronous errors with a secondary queue . . . 277
4.15.2 Exception class interface . . . 278
4.16 Data types . . . 283
4.16.1 Scalar data types . . . 283
4.16.2 Vector types . . . 284
4.16.2.1 Vec interface . . . 284
4.16.2.2 Aliases . . . 302
4.16.2.3 Swizzles . . . 302
4.16.2.4 Swizzled vec class . . . 303
4.16.2.5 Rounding modes . . . 303
4.16.2.6 Memory layout and alignment . . . 304
4.16.2.7 Considerations for endianness . . . 304
4.16.2.8 Performance note . . . 304
4.16.3 Math array types . . . 304
4.16.3.1 Math array interface . . . 304
4.16.3.2 Aliases . . . 315
4.16.3.3 Memory layout and alignment . . . 315
4.17 Synchronization and atomics . . . 315
4.17.1 Barriers and fences . . . 316

4.17.2 Atomic references . . . 316
4.17.3 Atomic types (deprecated) . . . 326
4.18 Stream class . . . 334
4.18.1 Stream class interface . . . 335
4.18.2 Synchronization . . . 338
4.18.3 Implicit flush . . . 339
4.18.4 Performance note . . . 339
4.19 SYCL built-in functions for SYCL host and device . . . 339
4.19.1 Description of the built-in types available for SYCL host and device . . . 339
4.19.2 Work-item functions . . . 341
4.19.3 Function objects . . . 342
4.19.4 Algorithms library . . . 344
4.19.4.1 `any_of`, `all_of` and `none_of` . . . 344
4.19.4.2 `reduce` . . . 345
4.19.4.3 `exclusive_scan` and `inclusive_scan` . . . 346
4.19.5 Group functions . . . 348
4.19.5.1 `group_broadcast` . . . 349
4.19.5.2 `group_leader` . . . 349
4.19.5.3 `group_barrier` . . . 350
4.19.5.4 `group_any_of`, `group_all_of` and `group_none_of` . . . 350
4.19.5.5 `group_reduce` . . . 351
4.19.5.6 `group_exclusive_scan` and `group_inclusive_scan` . . . 352
4.19.6 Math functions . . . 353
4.19.7 Integer functions . . . 358
4.19.8 Common functions . . . 360
4.19.9 Geometric functions . . . 361
4.19.10 Relational functions . . . 362
4.19.11 Vector data load and store functions . . . 365
4.19.12 Synchronization functions . . . 365
4.19.13 `printf` function . . . 365

**5 SYCL Device Compiler** **367**
5.1 Offline compilation of SYCL source files . . . 367
5.2 Naming of kernels . . . 367
5.3 Language restrictions for kernels . . . 368
5.4 Compilation of functions . . . 369
5.5 Built-in scalar data types . . . 370
5.6 Preprocessor directives and macros . . . 371
5.7 Kernel attributes . . . 371
5.7.1 Core kernel attributes . . . 371
5.7.2 Example attribute syntax . . . 373
5.7.3 Deprecated attribute syntax . . . 373
5.8 Address-space deduction . . . 374
5.8.1 Address space assignement . . . 374
5.8.2 Common address space deduction rules . . . 374
5.8.3 Generic as default address space . . . 375
5.8.4 Inferred address space . . . 375
5.9 SYCL offline linking . . . 376
5.9.1 SYCL functions and methods linkage . . . 376

**6 SYCL Extensions** **377**

6.1 Definition of an extension . . . 377
6.2 Predefined macros . . . 378
6.3 Device aspects and conditional features . . . 379
6.4 Backends . . . 380
6.5 Conditional features and compilation errors . . . 380

**A Information descriptors** **383**
A.1 Platform information descriptors . . . 383
A.2 Context information descriptors . . . 383
A.3 Device information descriptors . . . 383
A.4 Queue information descriptors . . . 386
A.5 Kernel information descriptors . . . 386
A.6 Event information descriptors . . . 387

**B Feature sets** **389**
B.1 Full feature set . . . 389
B.2 Reduced feature set . . . 389
B.3 Compatibility . . . 389
B.4 Conformance . . . 389

**C Host backend specification** **391**
C.1 Mapping of the SYCL programming model on the host . . . 391
C.1.1 SYCL memory model on the host . . . 391
C.2 Interoperability with the host application . . . 392

**D OpenCL backend specification** **393**
D.1 SYCL for OpenCL framework . . . 393
D.2 Mapping of SYCL programming model on top of OpenCL . . . 393
D.2.1 Platform mixed version support . . . 394
D.2.2 OpenCL memory model . . . 394
D.2.3 OpenCL resources managed by SYCL application . . . 394
D.3 Interoperability with the OpenCL API . . . 395
D.4 Programming interface . . . 397
D.4.1 Reference counting . . . 398
D.4.2 Errors and limitations . . . 398
D.4.3 Interoperability with modules . . . 399
D.4.3.1 Free functions . . . 399
D.4.4 Interoperability with kernels . . . 400
D.4.5 OpenCL kernel conventions and SYCL . . . 401
D.4.6 Data types . . . 401
D.5 Preprocessor directives and macros . . . 403
D.5.1 Offline linking with OpenCL C libraries . . . 403
D.6 SYCL support of non-core OpenCL features . . . 404
D.6.1 Half precision floating-point . . . 404
D.6.2 Writing to 3D image memory objects . . . 405
D.6.3 Interoperability with OpenGL . . . 405

**E What changed from previous versions** **407**
E.1 What changed from SYCL 1.2.1 to SYCL 2020 . . . 407

**References** **410**

**Glossary** **413**

# List of Tables


3.1 Combined requirement from two different accessor access modes within the same command group. The rules are commutative and associative . . . 35

4.1 Common special member functions for reference semantics . . . 57
4.2 Common hidden friend functions for reference semantics . . . 57
4.2 Common hidden friend functions for reference semantics . . . 58
4.3 Common special member functions for by-value semantics . . . 59
4.4 Common hidden friend functions for by-value semantics . . . 59
4.5 Traits for properties . . . 61
4.6 Common member functions of the SYCL `property` interface . . . 62
4.7 Constructors of the SYCL `property_list` class . . . 62
4.8 Standard device selectors included with all SYCL implementations . . . 63
4.9 Constructors of the SYCL `platform` class . . . 65
4.9 Constructors of the SYCL `platform` class . . . 66
4.10 Member functions of the SYCL `platform` class . . . 66
4.10 Member functions of the SYCL `platform` class . . . 67
4.11 Static member functions of the SYCL `platform` class . . . 67
4.12 Platform information descriptors . . . 67
4.13 Constructors of the SYCL `context` class . . . 69
4.14 Member functions of the `context` class . . . 69
4.14 Member functions of the `context` class . . . 70
4.15 Context information descriptors . . . 70
4.15 Context information descriptors . . . 71
4.16 Constructors of the SYCL `device` class . . . 73
4.17 Member functions of the SYCL `device` class . . . 73
4.17 Member functions of the SYCL `device` class . . . 74
4.17 Member functions of the SYCL `device` class . . . 75
4.18 Static member functions of the SYCL `device` class . . . 75
4.19 Device information descriptors . . . 75
4.19 Device information descriptors . . . 76
4.19 Device information descriptors . . . 77
4.19 Device information descriptors . . . 78
4.19 Device information descriptors . . . 79
4.19 Device information descriptors . . . 80
4.19 Device information descriptors . . . 81
4.19 Device information descriptors . . . 82
4.19 Device information descriptors . . . 83
4.19 Device information descriptors . . . 84
4.19 Device information descriptors . . . 85
4.19 Device information descriptors . . . 86
4.19 Device information descriptors . . . 87
4.20 Device aspects defined by the core SYCL specification . . . 89
4.20 Device aspects defined by the core SYCL specification . . . 90

4.21 Constructors of the queue class . . . 93
4.21 Constructors of the queue class . . . 94
4.21 Constructors of the queue class . . . 95
4.22 Member functions for queue class . . . 95
4.22 Member functions for queue class . . . 96
4.22 Member functions for queue class . . . 97
4.22 Member functions for queue class . . . 98
4.22 Member functions for queue class . . . 99
4.23 Queue information descriptors . . . 99
4.24 Properties supported by the SYCL queue class . . . 100
4.25 Constructors of the queue property classes . . . 100
4.26 Constructors of the event class . . . 102
4.27 Member functions for the event class . . . 102
4.27 Member functions for the event class . . . 103
4.28 Event class information descriptors . . . 104
4.29 Profiling information descriptors for the SYCL event class . . . 104
4.30 SYCL Default Allocators . . . 105
4.31 Constructors of the buffer class . . . 109
4.31 Constructors of the buffer class . . . 110
4.31 Constructors of the buffer class . . . 111
4.31 Constructors of the buffer class . . . 112
4.31 Constructors of the buffer class . . . 113
4.31 Constructors of the buffer class . . . 114
4.32 Member functions for the buffer class . . . 114
4.32 Member functions for the buffer class . . . 115
4.32 Member functions for the buffer class . . . 116
4.32 Member functions for the buffer class . . . 117
4.33 Properties supported by the SYCL buffer class . . . 117
4.33 Properties supported by the SYCL buffer class . . . 118
4.34 Constructors of the buffer property classes . . . 118
4.35 Member functions of the buffer property classes . . . 118
4.36 Constructors of the unsampled_image class template . . . 123
4.36 Constructors of the unsampled_image class template . . . 124
4.36 Constructors of the unsampled_image class template . . . 125
4.36 Constructors of the unsampled_image class template . . . 126
4.36 Constructors of the unsampled_image class template . . . 127
4.36 Constructors of the unsampled_image class template . . . 128
4.36 Constructors of the unsampled_image class template . . . 129
4.37 Member functions of the unsampled_image class template . . . 129
4.37 Member functions of the unsampled_image class template . . . 130
4.38 Constructors of the sampled_image class template . . . 132
4.38 Constructors of the sampled_image class template . . . 133
4.38 Constructors of the sampled_image class template . . . 134
4.39 Member functions of the sampled_image class template . . . 134
4.39 Member functions of the sampled_image class template . . . 135
4.40 Properties supported by the SYCL image classes . . . 135
4.41 Constructors of the image property classes . . . 135
4.41 Constructors of the image property classes . . . 136
4.42 Member functions of the image property classes . . . 136
4.43 Enumeration of access modes available to accessors . . . 140
4.44 Enumeration of access modes available to accessors . . . 140

4.44 Enumeration of access modes available to accessors . . . 141
4.45 Enumeration of access tags available to accessors . . . 142
4.46 Description of all the device buffer accessor capabilities . . . 145
4.47 Member types of the accessor class template buffer specialization . . . 148
4.47 Member types of the accessor class template buffer specialization . . . 149
4.48 Constructors of the accessor class template buffer specialization . . . 150
4.48 Constructors of the accessor class template buffer specialization . . . 151
4.48 Constructors of the accessor class template buffer specialization . . . 152
4.49 Member functions of the accessor class template buffer specialization . . . 152
4.49 Member functions of the accessor class template buffer specialization . . . 153
4.49 Member functions of the accessor class template buffer specialization . . . 154
4.49 Member functions of the accessor class template buffer specialization . . . 155
4.50 Properties supported by the SYCL accessor class . . . 155
4.51 Constructors of the accessor property classes . . . 155
4.52 Member types of the host_accessor class template . . . 159
4.53 Constructors of the host_accessor class template . . . 160
4.53 Constructors of the host_accessor class template . . . 161
4.53 Constructors of the host_accessor class template . . . 162
4.54 Member functions of the host_accessor class template . . . 162
4.54 Member functions of the host_accessor class template . . . 163
4.54 Member functions of the host_accessor class template . . . 164
4.55 Description of all the local accessor capabilities . . . 165
4.56 Member types of the accessor class template local specialization . . . 167
4.57 Constructors of the accessor class template local specialization . . . 167
4.57 Constructors of the accessor class template local specialization . . . 168
4.58 Member functions of the accessor class template local specialization . . . 168
4.58 Member functions of the accessor class template local specialization . . . 169
4.58 Member functions of the accessor class template local specialization . . . 170
4.59 Description of all the image accessor capabilities . . . 171
4.60 Constructors of the accessor class template image specialization . . . 173
4.61 Member functions of the accessor class template image specialization . . . 173
4.61 Member functions of the accessor class template image specialization . . . 174
4.62 Constructors of the SYCL multi_ptr class template . . . 181
4.62 Constructors of the SYCL multi_ptr class template . . . 182
4.63 Operators of multi_ptr class . . . 182
4.63 Operators of multi_ptr class . . . 183
4.63 Operators of multi_ptr class . . . 184
4.64 Member functions of multi_ptr class . . . 184
4.65 Hidden friend functions of the multi_ptr class . . . 184
4.65 Hidden friend functions of the multi_ptr class . . . 185
4.65 Hidden friend functions of the multi_ptr class . . . 186
4.66 Addressing modes description . . . 188
4.67 Filtering modes description . . . 188
4.68 Coordinate normalization modes description . . . 189
4.69 Constructors the sampler class . . . 189
4.70 Member functions for the sampler class . . . 189
4.71 Summary of types used to identify points in an index space, and ranges over which those points can vary . . . 205
4.72 Constructors of the range class template . . . 207
4.73 Member functions of the range class template . . . 207
4.74 Hidden friend functions of the SYCL range class template . . . 207

4.74 Hidden friend functions of the SYCL range class template . . . 208
4.75 Constructors of the nd_range class . . . 209
4.76 Member functions for the nd_range class . . . 209
4.77 Constructors of the id class template . . . 210
4.77 Constructors of the id class template . . . 211
4.78 Member functions of the id class template . . . 211
4.79 Hidden friend functions of the id class template . . . 211
4.79 Hidden friend functions of the id class template . . . 212
4.80 Member functions for the item class . . . 213
4.80 Member functions for the item class . . . 214
4.81 Member functions for the nd_item class . . . 216
4.81 Member functions for the nd_item class . . . 217
4.82 Member functions for the h_item class . . . 219
4.82 Member functions for the h_item class . . . 220
4.83 Member functions for the group class . . . 222
4.83 Member functions for the group class . . . 223
4.83 Member functions for the group class . . . 224
4.83 Member functions for the group class . . . 225
4.84 Member functions for the sub_group class . . . 226
4.85 Member functions of the SYCL device_event class . . . 227
4.86 Constructors of the device_event class . . . 227
4.87 Overloads of the reduction interface . . . 229
4.87 Overloads of the reduction interface . . . 230
4.88 Constructors of the reducer class . . . 232
4.88 Constructors of the reducer class . . . 233
4.89 Member functions of the reducer class . . . 233
4.89 Member functions of the reducer class . . . 234
4.90 Operators of the reducer class . . . 234
4.91 Constructors of the handler class . . . 238
4.92 Member functions of the handler class . . . 239
4.93 Member functions of the handler class . . . 239
4.93 Member functions of the handler class . . . 240
4.93 Member functions of the handler class . . . 241
4.93 Member functions of the handler class . . . 242
4.93 Member functions of the handler class . . . 243
4.93 Member functions of the handler class . . . 244
4.94 Constructor of the private_memory class . . . 249
4.95 Member functions of the private_memory class . . . 249
4.96 Member functions of the handler class . . . 252
4.96 Member functions of the handler class . . . 253
4.97 Member functions of the kernel class . . . 259
4.97 Member functions of the kernel class . . . 260
4.98 Kernel class information descriptors . . . 260
4.98 Kernel class information descriptors . . . 261
4.99 Device-specific kernel information descriptors . . . 261
4.100 Kernel work-group information descriptors . . . 262
4.101 Member functions of the SYCL exception class . . . 280
4.101 Member functions of the SYCL exception class . . . 281
4.101 Member functions of the SYCL exception class . . . 282
4.102 Member functions of the exception_list . . . 282
4.103 Values of the SYCL errc enum . . . 282

4.103 Values of the SYCL errc enum . . . 283
4.104 SYCL error code helper functions . . . 283
4.105 Additional scalar data types supported by SYCL . . . 283
4.106 Constructors of the SYCL vec class template . . . 288
4.107 Member functions for the SYCL vec class template . . . 288
4.107 Member functions for the SYCL vec class template . . . 289
4.107 Member functions for the SYCL vec class template . . . 290
4.107 Member functions for the SYCL vec class template . . . 291
4.107 Member functions for the SYCL vec class template . . . 292
4.108 Hidden friend functions of the vec class template . . . 293
4.108 Hidden friend functions of the vec class template . . . 294
4.108 Hidden friend functions of the vec class template . . . 295
4.108 Hidden friend functions of the vec class template . . . 296
4.108 Hidden friend functions of the vec class template . . . 297
4.108 Hidden friend functions of the vec class template . . . 298
4.108 Hidden friend functions of the vec class template . . . 299
4.108 Hidden friend functions of the vec class template . . . 300
4.108 Hidden friend functions of the vec class template . . . 301
4.108 Hidden friend functions of the vec class template . . . 302
4.109 Rounding modes for the SYCL vec class template . . . 303
4.110 Constructors of the SYCL marray class template . . . 307
4.111 Member functions for the SYCL marray class template . . . 307
4.111 Member functions for the SYCL marray class template . . . 308
4.112 Non-member functions of the marray class template . . . 308
4.112 Non-member functions of the marray class template . . . 309
4.112 Non-member functions of the marray class template . . . 310
4.112 Non-member functions of the marray class template . . . 311
4.112 Non-member functions of the marray class template . . . 312
4.112 Non-member functions of the marray class template . . . 313
4.112 Non-member functions of the marray class template . . . 314
4.112 Non-member functions of the marray class template . . . 315
4.113 Constructors of the SYCL atomic_ref class template . . . 322
4.114 Member functions available on any object of type atomic_ref<T> . . . 322
4.114 Member functions available on any object of type atomic_ref<T> . . . 323
4.115 Additional member functions available on an object of type atomic_ref<T> for integral T . . . 324
4.115 Additional member functions available on an object of type atomic_ref<T> for integral T . . . 325
4.116 Additional member functions available on an object of type atomic_ref<T> for floating-point T . . . 325
4.116 Additional member functions available on an object of type atomic_ref<T> for floating-point T . . . 326
4.117 Additional member functions available on an object of type atomic_ref<T*> . . . 326
4.118 Constructors of the cl::sycl::atomic class template . . . 329
4.119 Member functions available on an object of type cl::sycl::atomic<T> . . . 329
4.119 Member functions available on an object of type cl::sycl::atomic<T> . . . 330
4.119 Member functions available on an object of type cl::sycl::atomic<T> . . . 331
4.119 Member functions available on an object of type cl::sycl::atomic<T> . . . 332
4.120 Global functions available on atomic types . . . 333
4.120 Global functions available on atomic types . . . 334
4.121 Operand types supported by the stream class . . . 336
4.121 Operand types supported by the stream class . . . 337
4.122 Manipulators supported by the stream class . . . 337
4.123 Constructors of the stream class . . . 338
4.124 Member functions of the stream class . . . 338

4.125 Global functions of the stream class . . . 338
4.126 Generic type name description, which serves as a description for all valid types of parameters to kernel functions [1] . . . 340
4.126 Generic type name description, which serves as a description for all valid types of parameters to kernel functions [1] . . . 341
4.127 Member functions for the plus function object . . . 343
4.128 Member functions for the multiplies function object . . . 343
4.129 Member functions for the bit_and function object . . . 343
4.130 Member functions for the bit_or function object . . . 343
4.131 Member functions for the bit_xor function object . . . 343
4.132 Member functions for the logical_and function object . . . 343
4.133 Member functions for the logical_or function object . . . 344
4.134 Member functions for the minimum function object . . . 344
4.135 Member functions for the maximum function object . . . 344
4.136 Overloads for the any_of function . . . 345
4.137 Overloads for the all_of function . . . 345
4.138 Overloads for the none_of function . . . 345
4.139 Overloads of the reduce function . . . 346
4.140 Overloads of the exclusive_scan function . . . 347
4.141 Overloads of the inclusive_scan function . . . 348
4.142 Overloads of the group_broadcast function . . . 349
4.143 Overloads of the group_leader function . . . 349
4.144 Overloads for the group_barrier function . . . 350
4.145 Overloads for the group_any_of function . . . 350
4.146 Overloads for the group_all_of function . . . 350
4.146 Overloads for the group_all_of function . . . 351
4.147 Overloads for the group_none_of function . . . 351
4.148 Overloads of the group_reduce function . . . 351
4.149 Overloads of the group_exclusive_scan function . . . 352
4.150 Overloads of the group_inclusive_scan function . . . 353
4.151 Math functions which work on SYCL host and device. They correspond to Table 6.8 of the OpenCL 1.2 specification [1] . . . 354
4.151 Math functions which work on SYCL host and device. They correspond to Table 6.8 of the OpenCL 1.2 specification [1] . . . 355
4.151 Math functions which work on SYCL host and device. They correspond to Table 6.8 of the OpenCL 1.2 specification [1] . . . 356
4.151 Math functions which work on SYCL host and device. They correspond to Table 6.8 of the OpenCL 1.2 specification [1] . . . 357
4.152 Native math functions . . . 357
4.152 Native math functions . . . 358
4.153 Half precision math functions . . . 358
4.154 Integer functions which work on SYCL host and device, are available in the sycl namespace . . . 359
4.154 Integer functions which work on SYCL host and device, are available in the sycl namespace . . . 360
4.155 Common functions which work on SYCL host and device, are available in the sycl namespace. They correspond to Table 6.12 of the OpenCL 1.2 specification [1] . . . 360
4.155 Common functions which work on SYCL host and device, are available in the sycl namespace. They correspond to Table 6.12 of the OpenCL 1.2 specification [1] . . . 361
4.156 Geometric functions which work on SYCL host and device, are available in the sycl namespace. They correspond to Table 6.13 of the OpenCL 1.2 specification [1] . . . 361
4.156 Geometric functions which work on SYCL host and device, are available in the sycl namespace. They correspond to Table 6.13 of the OpenCL 1.2 specification [1] . . . 362

4.157 Relational functions for vec template class which work on SYCL host and device, are available in the sycl namespace. They correspond to Table 6.14 of the OpenCL 1.2 specification [1] . . . . . 363
4.157 Relational functions for vec template class which work on SYCL host and device, are available in the sycl namespace. They correspond to Table 6.14 of the OpenCL 1.2 specification [1] . . . . . 364
4.158 Relational functions for scalar data types and marray template class template class which work on SYCL host and device, are available in the sycl namespace. . . . . . . . . . . . . . . . . . . . . . 364
4.158 Relational functions for scalar data types and marray template class template class which work on SYCL host and device, are available in the sycl namespace. . . . . . . . . . . . . . . . . . . . . . 365

5.1 Fundamental data types supported by SYCL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 370
5.2 Attributes supported by the SYCL General programming interface . . . . . . . . . . . . . . . . . . 372
5.2 Attributes supported by the SYCL General programming interface . . . . . . . . . . . . . . . . . . 373

C.1 Mapping of SYCL memory regions into host memory regions . . . . . . . . . . . . . . . . . . . . . 391

D.1 Mapping of SYCL memory regions into OpenCL memory regions . . . . . . . . . . . . . . . . . . 394
D.2 List of native types per SYCL object in the OpenCL backend . . . . . . . . . . . . . . . . . . . . . 396
D.3 List of native types per SYCL object on Kernel code . . . . . . . . . . . . . . . . . . . . . . . . . 396
D.6 Example range mapping from SYCL enqueued three dimensional global range to OpenCL and SYCL queries . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 402
D.7 Scalar data type aliases supported by SYCL OpenCL backend . . . . . . . . . . . . . . . . . . . . 402
D.7 Scalar data type aliases supported by SYCL OpenCL backend . . . . . . . . . . . . . . . . . . . . 403
D.8 SYCL support for OpenCL 1.2 extensions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 404

# List of Figures


3.1 Execution order of three command groups submitted to the same queue . . . . . . . . . . . . . . . 32
3.2 Execution order of three command groups submitted to the different queues . . . . . . . . . . . . 32
3.3 Actions performed when three command groups are submitted to two distinct queues, and potential implementation in an OpenCL SYCL backend by a SYCL runtime. Note that in this example, each SYCL buffer ($b1$, $b2$) is implemented as separate `cl_mem` objects per context . . . . . . . . . 34
3.4 Requirements on overlapping vs non-overlapping *sub-buffer* . . . . . . . . . . . . . . . . . . . . . 36
3.5 Execution of command groups when using host accessors . . . . . . . . . . . . . . . . . . . . . . . 36

# 1. Acknowledgements

**Editors**

- Maria Rovatsou, Codeplay
- Lee Howes, Qualcomm
- Ronan Keryell, Xilinx (current)

**Contributors**

- Eric Berdahl, Adobe
- Shivani Gupta, Adobe
- David Neto, Altera
- Brian Sumner, AMD
- Hal Finkel, Argonne National Laboratory
- Nevin Liber, Argonne National Laboratory
- Anastasia Stulova, ARM
- Balázs Keszthelyi, Broadcom
- Gordon Brown, Codeplay
- Morris Hafner, Codeplay
- Alexander Johnston, Codeplay
- Marios Katsigiannis, Codeplay
- Paul Keir, Codeplay
- Victor Lomüller, Codeplay
- Tomas Matheson, Codeplay
- Duncan McBain, Codeplay
- Ralph Potter, Codeplay
- Ruyman Reyes, Codeplay
- Andrew Richards, Codeplay
- Maria Rovatsou, Codeplay
- Panagiotis Stratis, Codeplay
- Michael Wong, Codeplay
- Peter Žužek, Codeplay
- Matt Newport, EA
- Alexey Bader, Intel
- James Brodman, Intel
- Allen Hux, Intel
- Michael Kinsner, Intel
- John Pennycook, Intel
- Roland Schulz, Intel
- Sergey Semenov, Intel
- Jason Sewall, Intel
- Kathleen Mattson, Miller & Mattson, LLC
- Dave Miller, Miller & Mattson, LLC
- Lee Howes, Qualcomm
- Chu-Cheow Lim, Qualcomm

- Jack Liu, Qualcomm
- Dave Airlie, Red Hat
- Aksel Alpay, Self
- Dániel Berényi, Self
- Máté Nagy-Egri, Stream HPC
- Tom Deakin, University of Bristol
- Paul Preney, University of Windsor
- Andrew Gozillon, Xilinx
- Ronan Keryell, Xilinx
- Lin-Ya Yu, Xilinx

# 2. Introduction

SYCL (pronounced “sickle”) is a royalty-free, cross-platform abstraction C++ programming model for heterogeneous computing. SYCL builds on the underlying concepts, portability and efficiency of parallel API or standards like OpenCL while adding much of the ease of use and flexibility of single-source C++.

Developers using SYCL are able to write standard modern C++ code, with many of the techniques they are accustomed to, such as inheritance and templating. At the same time developers have access to the full range of capabilities of the underlying implementation such as OpenCL both through the features of the SYCL libraries and, where necessary, through interoperation with code written directly using the underneath implementation, such as the OpenCL APIs.

SYCL implements a single-source multiple compiler-passes (SMCP) design which offers the power of source integration while allowing toolchains to remain flexible. The SMCP design supports embedding of code intended to be compiled for a device, for example a GPU, inline with host code. This embedding of code offers three primary benefits:

**Simplicity** For novice programmers using frameworks like OpenCL, the separation of host and device source code in OpenCL can become complicated to deal with, particularly when similar kernel code is used for multiple different operations on different data types. A single compiler flow and integrated tool chain combined with libraries that perform a lot of simple tasks simplifies initial OpenCL programs to a minimum complexity. This reduces the learning curve for programmers new to OpenCL and allows them to concentrate on parallelization techniques rather than syntax.

**Reuse** C++’s type system allows for complex interactions between different code units and supports efficient abstract interface design and reuse of library code. For example, a *transform* or *map* operation applied to an array of data may allow specialization on both the operation applied to each element of the array and on the type of the data. The SMCP design of SYCL enables this interaction to bridge the host code/device code boundary such that the device code to be specialized on both of these factors directly from the host code.

**Efficiency** Tight integration with the type system and reuse of library code enables a compiler to perform inlining of code and to produce efficient specialized device code based on decisions made in the host code without having to generate kernel source strings dynamically.

SYCL is designed to allow a compilation flow where the source file is passed through multiple different compilers, including a standard C++ host compiler of the developer’s choice, and where the resulting application combines the results of these compilation passes. This is distinct from a single-source flow that might use language extensions that preclude the use of a standard host compiler. The SYCL standard does not preclude the use of a single compiler flow, but is designed to not require it.

The advantages of this design are two-fold. First, it offers better integration with existing tool chains. An application that already builds using a chosen compiler can continue to do so when SYCL code is added. Using the SYCL tools on a source file within a project will both compile for an OpenCL device and let the same source file be compiled using the same host compiler that the rest of the project is compiled with. Linking and library relationships are unaffected. This design simplifies porting of pre-existing applications to SYCL. Second, the design allows the optimal compiler to be chosen for each device where different vendors may provide optimized tool-chains.

SYCL is designed to be as close to standard C++ as possible. In practice, this means that as long as no dependence is created on SYCL's integration with the underlying implementation like OpenCL, a standard C++ compiler can compile the SYCL programs and they will run correctly on host CPU. Any use of specialized low-level features can be masked using the C pre-processor in the same way that compiler-specific intrinsics may be hidden to ensure portability between different host compilers.

SYCL retains the execution model, runtime feature set and device capabilities inspired by the OpenCL standard. This standard imposes some limitations on the full range of C++ features that SYCL is able to support. This ensures portability of device code across as wide a range of devices as possible. As a result, while the code can be written in standard C++ syntax with interoperability with standard C++ programs, the entire set of C++ features is not available in SYCL device code. In particular, SYCL device code, as defined by this specification, does not support virtual function calls, function pointers in general, exceptions, runtime type information or the full set of C++ libraries that may depend on these features or on features of a particular host compiler. Nevertheless, these basic restrictions can be relieved by some specific Khronos or vendor extensions.

The use of C++ features such as generic programming, templated code, functional programming and inheritance on top of existing heterogeneous execution model opens a wide scope for innovation in software design for heterogeneous systems. Clean integration of device and host code within a single C++ type system enables the development of modern, templated generic and adaptable libraries that build simple, yet efficient, interfaces to offer more developers access to heterogeneous computing capabilities and devices. SYCL is intended to serve as a foundation for innovation in programming models for heterogeneous systems, that builds on open and widely implemented standard foundation like OpenCL or Vulkan.

To reduce programming effort and increase the flexibility with which developers can write code, SYCL extends the concepts found in standards like OpenCL model in two ways beyond the general use of C++ features:

- The hierarchical parallelism syntax offers a way of expressing the data-parallel similar to the OpenCL device or OpenMP target device execution model in an easy-to-understand modern C++ form. It more cleanly layers parallel loops and synchronization points to avoid fragmentation of code and to more efficiently map to CPU-style architectures.

- Data access in SYCL is separated from data storage. By relying on the C++-style resource acquisition is initialization (RAII) idiom to capture data dependencies between device code blocks, the runtime library can track data movement and provide correct behavior without the complexity of manually managing event dependencies between kernel instances and without the programming having to explicitly move data. This approach enables the data-parallel task-graphs that might be already part of the execution model to be built up easily and safely by SYCL programmers.

To summarize, SYCL enables computational kernels to be written inside C++ source files as normal C++ code, leading to the concept of "single-source" programming. This means that software developers can develop and use generic algorithms and data structures using standard C++ template techniques, while still supporting the multi-platform, multi-device heterogeneous execution of existing API such as OpenCL. The specification has been designed to enable implementation across as wide a variety of platforms as possible as well as ease of integration with other platform-specific technologies, thereby letting both users and implementers build on top of SYCL as an open platform for system-wide heterogeneous processing innovation.

# 3. SYCL architecture

This chapter describes the structure of a SYCL Application, and how the SYCL generic programming model lays out on top of a number of SYCL backends.

## 3.1 Overview

SYCL is an open industry standard for programming a heterogeneous system. The design of SYCL allows standard C++ source code to be written such that it can run on either an heterogeneous device or on the host.

The terminology used for SYCL inherits historically from OpenCL with some SYCL-specific additions. However SYCL is a generic C++ programming model that can be layed out on top of other heterogeneous APIs apart from OpenCL. SYCL implementations can provide SYCL backends for various heterogeneous APIs, implementing the SYCL general specification on top of them. We refer to this heterogeneous API as the SYCL backend API. The SYCL general specification defines the behavior that all SYCL implementations must expose to SYCL users for a SYCL application to behave as expected.

A function object that can execute on a device exposed by a SYCL backend API is called a SYCL kernel function.

To ensure maximum interoperability with different SYCL backend APIs, software developers can access the SYCL backend API alongside the SYCL general API whenever they include the SYCL backend interoperability headers. However, interoperability is a SYCL backend-specific feature. An application that uses interoperability does not conform to the SYCL general application model, since is not portable across backends.

The target users of SYCL are C++ programmers who want all the performance and portability features of OpenCL, but with the flexibility to use higher-level C++ abstractions across the host/device code boundary. Developers can use most of the abstraction features of C++, such as templates, classes and operator overloading.

However, some C++ language features are not permitted inside kernels, due to the limitations imposed by the capabilities of the underlying heterogeneous platforms. These features include virtual functions, virtual inheritance, throwing/catching exceptions, and run-time type-information. These features are available outside kernels as normal. Within these constraints, developers can use abstractions defined by SYCL, or they can develop their own on top. These capabilities make SYCL ideal for library developers, middleware providers and applications developers who want to separate low-level highly-tuned algorithms or data structures that work on heterogeneous systems from higher-level software development. Software developers can produce templated algorithms that are easily usable by developers in other fields.

## 3.2 Anatomy of a SYCL application

Below is an example of a typical SYCL application which schedules a job to run in parallel on any heterogeneous device available.

```
1  #include <SYCL/sycl.hpp>
2  #include <iostream>
3
4  int main() {
5    using namespace sycl;
6
7    int data[1024];  // Allocate data to be worked on
8
9    // By sticking all the SYCL work in a {} block, we ensure
10   // all SYCL tasks must complete before exiting the block,
11   // because the destructor of resultBuf will wait.
12   {
13     // Create a queue to enqueue work to
14     queue myQueue;
15
16     // Wrap our data variable in a buffer
17     buffer<int, 1> resultBuf { data, range<1> { 1024 } };
18
19     // Create a command_group to issue commands to the queue
20     myQueue.submit([&](handler& cgh) {
21       // request access to the buffer
22       accessor writeResult { resultBuf, cgh, write_only, noinit };
23
24       // Enqueue a parallel_for task
25       cgh.parallel_for<class simple_test>(1024, [=](auto idx) {
26         writeResult[idx] = idx;
27       });  // End of the kernel function
28     });    // End of our commands for this queue
29   }        // End of scope, so we wait for work producing resultBuf to complete
30
31   // Print result
32   for (int i = 0; i < 1024; i++)
33     std::cout << "data[" << i << "] = " << data[i] << std::endl;
34
35   return 0;
36 }
```

At line 1, we “`#include`” the SYCL header files, which provide all of the SYCL features that will be used.

A SYCL application runs on a SYCL Platform (see Section 3.4). The application is structured in three scopes which specify the different sections; application scope, command group scope and kernel scope. The kernel scope specifies a single kernel function that will be, or has been, compiled by a device compiler and executed on a device. In this example kernel scope is defined by lines 25 to 27. The command group scope specifies a unit of work which will comprise of a SYCL kernel function and accessors. In this example command group scope is defined by lines 20 to 28. The application scope specifies all other code outside of a command group scope. These three scopes are used to control the application flow and the construction and lifetimes of the various objects used within SYCL, as explained in Section 3.8.12.

A SYCL kernel function is the scoped block of code that will be compiled using a device compiler. This code may be defined by the body of a lambda function, by the `operator()` function of a function object. Each instance of the SYCL kernel function will be executed as a single, though not necessarily entirely independent, flow of execution and has to adhere to restrictions on what operations may be allowed to enable device compilers to safely compile it to a range of underlying devices.

The `parallel_for` function is templated with a class, in this case called `class simple_test`. This class is used to manually name the kernel when desired, such as to avoid a compiler-generated name when debugging a kernel defined through a lambda, to provide a known name with which to apply build options to a kernel, or to ensure compatibility with multiple compiler-pass implementations.

The `parallel_for` method creates an instance of a kernel object. The kernel object is the entity that will be enqueued within a command group. In the case of `parallel_for` the SYCL kernel function will be executed over the given range from 0 to 1023. The different methods to execute kernels can be found in Section 4.10.7.

A SYCL kernel function can only be defined within a command group scope, and a command group scope may include only a single SYCL kernel function. Command group scope is the syntactic scope wrapped by the construction of a command group function object as seen on line 20. The command group function object takes as a parameter a command group `handler` which is a runtime constructed object.

All the requirements for a kernel to execute are defined in this command group scope, as described in Section 3.6.1. In this case the constructor used for `myQueue` on line 14 is the default constructor, which allows the queue to select the best underlying device to execute on, leaving the decision up to the runtime.

In SYCL, data that is required within a SYCL kernel function must be contained within a buffer or image, as described in Section 3.7. We construct a buffer on line 17. Access to the buffer is controlled via an accessor which is constructed on line 22 through the `get_access` method of the buffer. The buffer is used to keep track of access to the data and the accessor is used to request access to the data on a queue, as well as to track the dependencies between SYCL kernel function. In this example the accessor is used to write to the data buffer on line 26. All buffers must be constructed in the application-scope, whereas all accessors must be constructed in the command group scope.

## 3.3 Normative references

The documents in the following list are referred to within this SYCL specification, and their content is a requirement for this document.

1. **C++17:** ISO/IEC 14882:2017 Sections 1-19 [2], referred to in this specification as the C++ core language.
2. **C++2a:** Working Draft, Standard for Programming Language C++ [3], referred to in this specification as the next C++ specification.

## 3.4 The SYCL platform model

The SYCL platform model is based on the OpenCL platform model. The model consists of a host connected to one or more heterogeneous devices, called devices.

A SYCL context is constructed, either directly by the user or implicitly when creating a queue, to hold all the runtime information required by the SYCL runtime and the SYCL backend to operate on a device, or group of devices. When a group of devices can be grouped together on the same context, they have some visibility of each other memory objects. The SYCL runtime can assume that memory is visible across all devices in the same context. Not all devices exposed from the same platform can be grouped together in the same context.

A SYCL application executes on the host as a standard C++ program. Devices are exposed through different SYCL backends to the SYCL Application. The SYCL application submits command group function objects to queues. Each queue enables execution on a given device.

The SYCL runtime then extracts operations from the command group function object, e.g. an explicit copy operation or a SYCL kernel function. When the operation is a SYCL kernel function, the SYCL runtime uses a SYCL backend-specific mechanism to extract the device binary from the SYCL application and pass it to the heterogeneous API for execution on the device.

A SYCL device is divided into one or more compute units (CUs) which are each divided into one or more processing elements (PEs). Computations on a device occur within the processing elements. How computation is mapped to PEs is SYCL backend and device specific. Two devices exposed via two different backends can map computations differently to the same device.

When a SYCL Application contains SYCL kernel function objects, the SYCL implementation must provide an offline compilation mechanism that enables the integration of the device binaries into the SYCL Application. The output of the offline compiler can be an intermediate representation, such as SPIR-V, that will be finalized during execution or a final device ISA.

A device may expose special purpose functionality as a *built-in* function. The SYCL API exposes functions to query and dispatch said *built-in* functions. Some SYCL backends and device may not support programmable kernels, and only support *built-in* functions.

## 3.5 The SYCL backend model

SYCL is a generic programming model for the C++ language that can target multiple heterogeneous APIs, such as OpenCL.

SYCL implementations enable these target APIs by implementing SYCL backends. For a SYCL implementation to be conformant on said SYCL backend, it must execute the SYCL generic programming model on the backend.

A SYCL implementation provides at least one SYCL backend to implement the SYCL API. All SYCL implementations must provide a host SYCL backend, which executes the SYCL generic code on the host platform directly without using an API to execute on an accelerator. SYCL implementations can make multiple SYCL backends available, both at compilation and at runtime.

The present document covers the SYCL generic interface available to all SYCL backends. How the SYCL generic interface maps to a particular SYCL backend is defined either by a separate SYCL backend specification document, provided by the Khronos SYCL group, or by the SYCL implementation documentation. Whenever there is a SYCL backend specification document, this takes precedence over SYCL implementation documentation.

When a SYCL user builds their SYCL application, she decides which of the SYCL backends will be used to build the SYCL application. This is called the set of *active backends*. Implementations must ensure that the active backends selected by the user can be used simultaneously by the SYCL implementation at runtime. If two backends are available at compile time but will produce an invalid SYCL application at runtime, the SYCL implementation must emit a compilation error.

A SYCL application built with a number of active backends does not necessarily guarantee that said backends can be executed at runtime. The subset of active backends available at runtime is called *available backends*. A backend is said to be *available* if the host platform where the SYCL application is executed exposes support for the heterogeneous API required for the SYCL backend.

It is implementation dependent whether certain backends require third-party libraries to be available in the system. Building with only the host as an active backend guarantees the binary will be executed on any platform without requiring third-party libraries. Failure to have all dependencies required for all active backends at runtime will cause the SYCL application to not run.

Once the application is running, users can query what SYCL platforms are available. SYCL implementations will expose the devices provided by each backend grouped into platforms. A backend must expose at least one platform.

Under the SYCL backend model, SYCL objects can contain one or multiple references to a certain SYCL backend native type. Not all SYCL objects will map directly to a SYCL backend native type. The mapping of SYCL objects to SYCL backend native types is defined by the SYCL backend specification document when available, or by the SYCL implementation otherwise.

To guarantee that multiple SYCL backend objects can interoperate with each other, SYCL memory objects are not bound to a particular SYCL backend. SYCL memory objects can be accessed from any device exposed by an *available* backend. SYCL Implementations can potentially map SYCL memory objects to multiple native types in different SYCL backends.

Since SYCL memory objects are independent of any particular SYCL backend, SYCL command groups can request access to memory objects allocated by any SYCL backend, and execute it on the backend associated with the queue. This requires the SYCL implementation to be able to transfer memory objects across SYCL backends

When a SYCL application runs on any number of SYCL backends without relying on any SYCL backend-specific behaviour or interoperability, it is said to be a SYCL General Application, and it is expected to run in any SYCL-conformant implementation that supports the required features for the application.

### 3.5.1 Platform mixed version support

The SYCL generic programming model exposes SYCL backends as a number of platforms that expose a number of devices. The SYCL generic programming model exposes a number of platforms, each of them exposing a number of devices. Each platform is bound to a certain SYCL backend. SYCL devices associated with said platform are associated with that SYCL backend.

Each SYCL backend can expose different versions for the devices and the platforms supported. Refer to the SYCL backend specification document or the SYCL implementation documentation as to how to query for device and platform versions.

However, In SYCL, the source language is compiled offline, so the language version is not available at runtime. Instead, the SYCL language version is available as a compile-time macro: `SYCL_LANGUAGE_VERSION`.

## 3.6 SYCL execution model

The execution of a SYCL program occurs in two parts: SYCL kernel functions and a SYCL application that executes on the host. The SYCL kernels execution is governed by the *SYCL Kernel Execution Model*, whereas the SYCL application that executes on the host is governed by the *SYCL Application Execution Model*.

A SYCL application can be executed on the host directly without a physical accelerator present when using the SYCL host backend.

### 3.6.1 SYCL application execution model

The SYCL application defines the execution order of the kernels by grouping each kernel with its requirements into a command group object. command group objects are submitted to execution via queue objects, which defines the device where the kernel will run. The same command group object can be submitted to different queues. When a command group is submitted to a SYCL queue, the requirements of the kernel execution are captured. The kernels are executed as soon as their requirements have been satisfied.

#### 3.6.1.1 SYCL backend resources managed by the SYCL application

The SYCL runtime integrated with the SYCL application will manage the resources required by the SYCL backend API to manage the heterogeneous devices is providing access to. This includes, but is not limited to, resource handlers, memory pools, dispatch queues and other temporary handler objects.

The SYCL programming interface represents the lifetime of the resources managed by the SYCL application using RAII rules. Construction of a SYCL object will typically entail the creation of multiple SYCL backend objects, which will be properly released on destruction of said SYCL object. The overall rules for construction and destruction are detailed in the SYCL Programming Interface chapter 4. Those SYCL backends with a SYCL backend document will detail how the resource management from SYCL objects maps down to the SYCL backend objects.

In SYCL, the minimum required object for submitting work to devices is the queue, which contains references to a platform, device and a context internally.

The resources managed by SYCL are:

1. Platforms: all features of SYCL backend APIs are implemented by platforms. A platform can be viewed as a given vendor's runtime and the devices accessible through it. Some devices will only be accessible to one vendor's runtime and hence multiple platforms may be present. SYCL manages the different platforms for the user.

2. Contexts: any SYCL backend resource that is acquired by the user is attached to a context. A context contains a collection of devices that the host can use and manages memory objects that can be shared between the devices. Devices belonging to the same context must be able to access each other global memory using some SYCL implementation-specific mechanism. A given context can only wrap devices owned by a single platform.

3. Devices: platforms provide one or more devices for executing SYCL kernels. In SYCL, a device is accessible through a `sycl::device` object.

4. Kernels: the SYCL functions that run on SYCL devices are defined as C++ function objects (a named function object type or a lambda function).

   Note that some SYCL backends may expose non-programable functionality as pre-defined kernels.

5. Modules: The SYCL backend API will expose the device binary(s) in some form of a program-object file or module. This is handled by SYCL using the `module` objects.

6. Queues: SYCL kernels execute in command queues. The user must create a queue, which references an associated context, platform and device. The context, platform and device may be chosen automatically, or specified by the user. SYCL queues execute kernels on a particular device of a particular context, but can have dependencies from any device on any available SYCL backend.

The SYCL implementation guarantees the correct initialization and destruction of any resource handled by the underlying SYCL backend API, except for those the user has obtained manually via the SYCL interoperability API.

#### 3.6.1.2 SYCL command groups and execution order

SYCL offers a higher abstraction in terms of queue ordering synchronization. All SYCL queues execute kernels in out-of-order fashion, regardless of the underlying SYCL backend mechanism used for dispatching. Developers only need to specify what data is required to execute a particular kernel. The SYCL runtime will guarantee that kernels are executed in an order that guarantees correctness. By specifying access modes and types of memory, a directed acyclic dependency graph (DAG) of kernels is built at runtime. This is achieved via the usage of command group objects. A SYCL command group object defines a set of requisites ($R$) and a kernel function ($k$). A command group is *submitted* to a queue when using the `sycl::queue::submit` method.

A **requisite** ($r_i$) is a requirement that must be fulfilled for a kernel-function ($k$) to be executed on a particular device. For example, a requirement may be that certain data is available on a device, or that another command group has finished execution. An implementation may evaluate the requirements of a command group at any point after it has been submitted. The *processing of a command group* is the process by which a SYCL runtime evaluates all the requirements in a given $R$. The SYCL runtime will execute $k$ only when all $r_i$ are satisfied (i.e, when all requirements are satisfied). To simplify the notation, in the specification we refer to the set of requirements of a command group named *foo* as $CG_{foo} = r_1, \ldots, r_n$.

The *evaluation of a requisite* (Satisfied($r_i$)) returns the status of the requisite, which can be *True* or *False*. A *satisfied* requisite implies the requirement is met. Satisfied($r_i$) never alters the requisite, only observes the current status. The implementation may not block to check the requisite, and the same check can be performed multiple times.

An **action** ($a_i$) is a collection of implementation-defined operations that must be performed in order to satisfy a requisite. The set of actions for a given command group $A$ is permitted to be empty if no operation is required to satisfy the requirement. The notation $a_i$ represents the action required to satisfy $r_i$. Actions of different requisites can be satisfied in any order w.r.t each other without side effects (i.e, given two requirements $r_j$ and $r_k$, $(r_j, r_k) \equiv (r_k, r_j)$). The intersection of two actions is not necessarily empty. **Actions** can include (but are not limited to): memory copy operations, mapping operations, host side synchronization, or implementation-specific behavior.

Finally, *Performing an action* (Perform($a_i$)) executes the action operations required to satisfy the requisite $r_j$. Note that, after Perform($a_i$), the evaluation Satisfied($r_j$) will return *True* until the kernel is executed. After the kernel execution, it is not defined whether a different command group with the same requirements needs to perform the action again, where actions of different requisites inside the same command group object can be satisfied in any order w.r.t each other without side effects: Given two requirements $r_j$ and $r_k$, Perform($a_j$) followed by Perform($a_k$) is equivalent to Perform($a_k$) followed by Perform($a_j$).

The requirements of different command groups submitted to the same or different queues are evaluated in the relative order of submission. command group objects whose intersection of requirement sets is not empty are said to depend on each other. They are executed in order of submission to the queue. If command groups are submitted to different queues or by multiple threads, the order of execution is determined by the SYCL runtime. Note that independent command group objects can be submitted simultaneously without affecting dependencies.

Figure 3.1 illustrates the execution order of three command group objects ($CG_a, CG_b, CG_c$) with certain requirements submitted to the same queue. Both $CG_a$ and $CG_b$ only have one requirement, $r_1$ and $r_2$ respectively. $CG_c$ requires both $r_1$ and $r_2$. This enables the SYCL runtime to potentially execute $CG_a$ and $CG_b$ simultaneously, whereas $CG_c$ cannot be executed until both $CG_a$ and $CG_b$ have been completed. The SYCL runtime evaluates the **requisites** and performs the **actions** required (if any) for the $CG_a$ and $CG_b$. When evaluating the **requisites** of

$CG_c$, they will be satisfied once the $CG_a$ and $CG_b$ have finished.

**SYCL Application Enqueue Order** **SYCL Kernel Execution Order**

`sycl::queue syclQueue;`
`syclQueue.submit(`$CG_a(r_1)$`);`
`syclQueue.submit(`$CG_b(r_2)$`);`
`syclQueue.submit(`$CG_c(r_1, r_2)$`);`

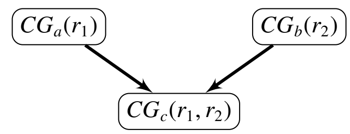



Figure 3.1: Execution order of three command groups submitted to the same queue.

Figure 3.2 uses three separate SYCL queue objects to submit the same command group objects as before. Regardless of using three different queues, the execution order of the different command group objects is the same. When different threads enqueue to different queues, the execution order of the command group will be the order in which the submit methods is executed. In this case, since the different command group objects execute on different devices, the **actions** required to satisfy the **requirements** may be different (e.g, the SYCL runtime may need to copy data to a different device in a separate context).

**SYCL Application Enqueue Order** **SYCL Kernel Execution Order**

`sycl::queue syclQueue1; sycl::queue`
`syclQueue2; sycl::queue syclQueue3;`
`syclQueue1.submit(`$CG_a(r_1)$`);`
`syclQueue2.submit(`$CG_b(r_2)$`);`
`syclQueue3.submit(`$CG_c(r_1, r_2)$`);`

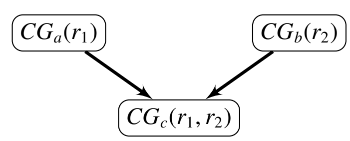



Figure 3.2: Execution order of three command groups submitted to the different queues.

### 3.6.2 SYCL kernel execution model

When a kernel is submitted for execution an index space is defined. An instance of the kernel body executes for each point in this index space. This kernel instance is called a work-item and is identified by its point in the index space, which provides a global id for the work-item. Each work-item executes the same code but the specific execution pathway through the code and the data operated upon can vary by using the work-item global id to specialize the computation.

Work-items are organized into work-groups. The work-groups provide a more coarse-grained decomposition of the index space. Each work-group is assigned a unique work-group id with the same dimensionality as the index space used for the work-items. Work-items are each assigned a local id, unique within the work-group, so that a single work-item can be uniquely identified by its global id or by a combination of its local id and work-group id. The work-items in a given work-group execute concurrently on the processing elements of a single compute unit.

The index space supported in SYCL is called an nd-range. An ND-range is an $N$-dimensional index space, where

$N$ is one, two or three. In SYCL, the ND-range is represented via the `nd_range<N>` class. An `nd_range<N>` is made up of a global range and a local range, each represented via values of type `range<N>` and a global offset, represented via a value of type `id<N>`. The types `nd_range<N>` and `id<N>` are each $N$-element arrays of integers. The iteration space defined via an `range<N>` is an $N$-dimensional index space starting at the ND-range's global offset and being of the size of its global range, split into work-groups of the size of its local range.

Each work-item in the ND-range is identified by a value of type `nd_item<N>`. The type `nd_item<N>` encapsulates a global id, local id and work-group id, all of type `id<N>`, the iteration space offset also of type `id<N>`, as well as global and local ranges and synchronization operations necessary to make work-groups useful. Work-groups are assigned ids using a similar approach to that used for work-item global ids. Work-items are assigned to a work-group and given a local id with components in the range from zero to the size of the work-group in that dimension minus one. Hence, the combination of a work-group id and the local id within a work-group uniquely defines a work-item.

SYCL allows a simplified execution model in which the work-group size is left unspecified. A kernel invoked over a `range<N>`, instead of an `nd_range<N>` is executed within an iteration space of unspecified work-group size. In this case, less information is available to each work-item through the simpler `item<N>` class.

SYCL allows SYCL backends to expose fixed functionality as non-programmable kernels. The behavior of these functions are SYCL backend specific, and do not necessarily follow the SYCL Kernel Execution model.

## 3.7 Memory model

Since SYCL is a single-source programming model, the memory model affects both the Application and the Device Kernel parts of a program. On the SYCL Application, the SYCL Runtime will make sure data is available for execution of the kernels. On the SYCL Device kernel, SYCL backend rules are mapped to SYCL constructs to provide the same capabilities using C++ kernels.

### 3.7.1 SYCL application memory model

The application running on the host uses SYCL buffer objects using instances of the `sycl::buffer` class to allocate memory in the global address space, or can allocate specialized image memory using the `sycl::unsampled_image` and `sycl::sampled_image` classes.

In the SYCL Application, memory objects are bound to all devices in which they are used, regardless of the SYCL context where they reside. SYCL memory objects (namely, buffer and image objects) can encapsulate multiple underlying SYCL backend memory objects together with multiple host memory allocations to enable the same object to be shared between devices in different contexts, platforms or backends.

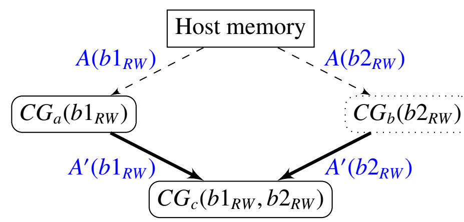
The order of execution of command group objects ensures a sequentially consistent access to the memory from the different devices to the memory objects.

To access a memory object, the user must create an accessor object which parameterizes the type of access to the memory object that a kernel or the host requires. The accessor object defines a requirement to access a memory object, and this requirement is defined by construction of an accessor, regardless of whether there are any uses in a kernel or by the host. The `cl::sycl::accessor` object specifies whether the access is via global memory, constant memory or image samplers and their associated access functions. The accessor also specifies whether the access is read-only (RO), write-only (WO) or read-write (RW). An optional *discard* flag can be added to an accessor to tell the system to discard any previous contents of the data the accessor refers to, e.g. discard write-only (DW). For simplicity, when a **requisite** represents an accessor object in a certain access mode, we represent it as $\text{MemoryObject}_{\text{AccessMode}}$. For example, an accessor that accesses memory object **buf1** in **RW** mode is represented as $buf1_{RW}$. A command group object that uses such an accessor is represented as $CG(buf1_{RW})$. The **action** required to satisfy a requisite and the location of the latest copy of a memory object will vary depending on the implementation.

Figure 3.3 illustrates an example where command group objects are enqueued to two separate SYCL queues executing in devices in different contexts. The **requisites** for the command group execution are the same, but the **actions** to satisfy them are different. For example, if the data is on the host before execution, $A(b1_{RW})$ and $A(b2_{RW})$ can potentially be implemented as copy operations from the host memory to `context1` or `context2` respectively. After $CG_a$ and $CG_b$ are executed, $A'(b1_{RW})$ will likely be an empty operation, since the result of the kernel can stay on the device. On the other hand, the results of $CG_b$ are now on a different context than $CG_c$ is executing, therefore $A'(b2_{RW})$ will need to copy data across two separate OpenCL contexts using an implementation specific mechanism.

**SYCL Application Enqueue Order** **SYCL Kernel Execution Order**

```
sycl::queue q1(context1);
sycl::queue q2(context2);
q1.submit(CG_a(b1_RW));
q2.submit(CG_b(b2_RW));
q1.submit(CG_c(b1_RW, b2_RW));
```




**Possible implementation by a SYCL Runtime**

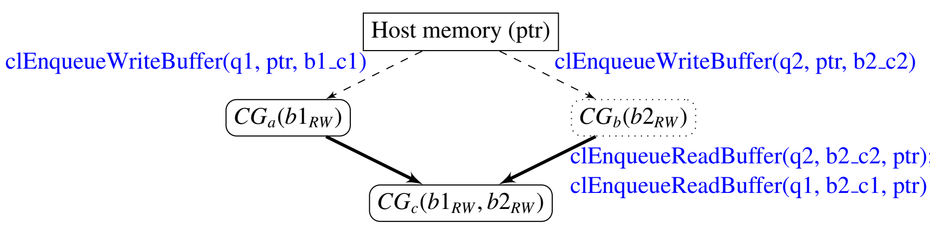



Figure 3.3: Actions performed when three command groups are submitted to two distinct queues, and potential implementation in an OpenCL SYCL backend by a SYCL runtime. Note that in this example, each SYCL buffer $(b1, b2)$ is implemented as separate `cl_mem` objects per context.

Note that the order of the definition of the accessors within the command group is irrelevant to the requirements they define. All accessors always apply to the entire command group object where they are defined.

When multiple accessors in the same command group define different requisites to the same memory object these requisites must be resolved.

Firstly, any requisites with different access modes but the same access target are resolved into a single requisite with the union of the different access modes according to Table 3.1. The atomic access mode acts as if it was read-write (RW) when determining the combined requirement. The rules in Table 3.1 are commutative and associative.

| One access mode | Other access mode | Combined requirement |
|---|---|---|
| read (RO) | write (WO) | read-write (RW) |
| read (RO) | read-write (RW) | read-write (RW) |
| write (WO) | read-write (RW) | read-write (RW) |
| discard-write (DW) | discard-read-write (DRW) | discard-read-write (DRW) |
| discard-write (DW) | write (WO) | write (WO) |
| discard-write (DW) | read (RO) | read-write (RW) |
| discard-write (DW) | read-write (RW) | read-write (RW) |
| discard-read-write (DRW) | write (WO) | read-write (RW) |
| discard-read-write (DRW) | read (RO) | read-write (RW) |
| discard-read-write (DRW) | read-write (RW) | read-write (RW) |

Table 3.1: Combined requirement from two different accessor access modes within the same command group. The rules are commutative and associative.

The result of this should be that there should not be any requisites with the same access target.

Secondly, the remaining requisites the must adhere to the following rule. Only one of the requisites may have write access ($W$ or $RW$), otherwise the SYCL runtime must through an exception. All requisites create a requirement for the data they represent to made available in the specified access target, however only the requisite with write access determines the side effects of the command group, i.e. only the data which that requisite represents will be updated.

For example:

- $CG(b1^G_{RW}, b1^H_R$ is permitted.
- $CG(b1^G_{RW}, b1^H_{RW}$ is **not** be permitted.
- $CG(b1^G_W, b1^C_{RW}$ is **not** be permitted.

Where $G$, $C$ and $H$ corresponding to the `global_buffer`, `constant_buffer` and `host_buffer` respectively.

A buffer created from a range of an existing buffer is called a *sub-buffer*. A buffer may be overlaid with any number of sub-buffers. Accessors can be created to operate on these *sub-buffers*. Refer to 4.7.2 for details on *sub-buffer* creation and restrictions. A requirement to access a sub-buffer is represented by specifying its range, e.g. $CG(b1_{RW,[0,5)})$ represents the requirement of accessing the range $[0,5)$ buffer $b1$ in read write mode.

If two accessors are constructed to access the same buffer, but both are to non-overlapping sub-buffers of the buffer, then the two accessors are said to not *overlap*, otherwise the accessors do overlap. Overlapping is the test that is used to determine the scheduling order of command groups. Command-groups with non-overlapping requirements may execute concurrently.

It is permissible for command groups that only read data to not copy that data back to the host or other devices after reading and for the runtime to maintain multiple read-only copies of the data on multiple devices.

A special case of requirement is the one defined by a **host accessor**. Host accessors are represented with $H(\text{MemoryObject}_{\text{accessMode}})$, e.g, $H(b1_{RW})$ represents a host accessor to $b1$ in read-write mode. Host accessors are a special type of accessor constructed from a memory object outside a command group, and require that the data associated with the given memory object is available on the host in the given pointer. This causes the runtime to block on construction of this object until the requirement has been satisfied. **Host accessor** objects are effec-

**SYCL Application Enqueue Order** **SYCL Kernel Execution Order**

`sycl::queue q1(context1);`
`q1.submit(`$CG_a(b1_{RW,[0,10)})$`);`
`q1.submit(`$CG_b(b1_{RW,[10,20)})$`);`
`q1.submit(`$CG_c(b1_{RW,[5,15)})$`);`

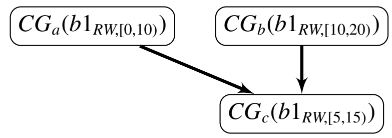



Figure 3.4: Requirements on overlapping vs non-overlapping *sub-buffer*.

tively barriers on all accesses to a certain memory object. Figure 3.5 shows an example of multiple command groups enqueued to the same queue. Once the host accessor $H(b1_{RW})$ is reached, the execution cannot proceed until $CG_a$ is finished. However, $CG_b$ does not have any requirements on $b1$, therefore, it can execute concurrently with the barrier. Finally, $CG_c$ will be enqueued after $H(b1_{RW})$ is finished, but still has to wait for $CG_b$ to conclude for all its requirements to be satisfied. See 3.8.8 for details on synchronization rules.

**SYCL Application Enqueue Order** **SYCL Kernel Execution Order**

`sycl::queue q1; q1.submit(`$CG_a(b1_{RW})$`);`
`q1.submit(`$CG_b(b2_{RW})$`);`
$H(b1_{RW})$`;`
`q1.submit(`$CG_c(b1_{RW}, b2_{RW})$`);`

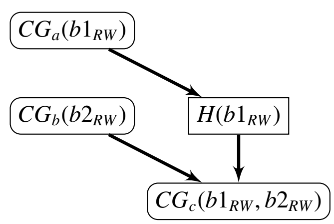



Figure 3.5: Execution of command groups when using host accessors.

### 3.7.2 SYCL device memory model

The memory model for SYCL Devices is based on the OpenCL 1.2 memory model. Work-items executing in a kernel have access to four distinct address spaces (memory regions) and a virtual address space overlapping some concrete address spaces:

- Global memory is accessible to all work-items in all work-groups. Work-items can read from or write to any element of a global memory object. Reads and writes to global memory may be cached depending on the capabilities of the device. Global memory is persistent across kernel invocations, however there is no guarantee that two concurrently executing kernels can simultaneously write to the same memory object and expect correct results.

- Constant memory is a region of global memory that remains constant during the execution of a kernel. The host allocates and initializes memory objects placed into constant memory.

- Local memory is accessible to all work-items in a single work-group and inaccessible to work-items in other work-groups. This memory region can be used to allocate variables that are shared by all work-items in a work-group. Work-group-level visibility allows local memory to be implemented as dedicated regions of the device memory where this is appropriate.
- Private memory is a region of memory private to a work-item. Variables defined in one work-item's private memory are not visible to another work-item.
- Generic memory is a virtual address space which overlaps the global, local and private address spaces.

#### 3.7.2.1 Access to memory

Accessors in the device kernels provide access to the memory objects, acting as pointers to the corresponding address space.

It is not possible to pass a pointer into host memory directly as a kernel parameter because the devices may be unable to support the same address space as the host.

To allocate local memory within a kernel, the user can either pass a `sycl::local_accessor` object to the kernel as a parameter, or can define a variable in work-group scope inside `sycl::parallel_for_work_group`.

Any variable defined inside a `sycl::parallel_for` scope or `sycl::parallel_for_work_item` scope will be allocated in private memory. Any variable defined inside a `sycl::parallel_for_work_group` scope will be allocated in local memory.

Users can create accessors that reference sub-buffers as well as entire buffers.

Within kernels, the underlying C++ pointer types can be obtained from an accessor. The pointer types will contain a compile-time deduced address space. So, for example, if a C++ pointer is obtained from an accessor to global memory, the C++ pointer type will have a global address space attribute attached to it. The address space attribute will be compile-time propagated to other pointer values when one pointer is initialized to another pointer value using a defined mechanism.

When developers need to explicitly state the address space of a pointer value, one of the explicit pointer classes can be used. There is a different explicit pointer class for each address space: `sycl::raw_local_ptr` , `sycl::raw_global_ptr`, `sycl::raw_private_ptr`, `sycl::raw_constant_ptr`, `sycl::raw_generic_ptr`, `sycl::decorated_local_ptr`, `sycl::decorated_global_ptr`, `sycl::decorated_private_ptr`, `sycl::decorated_constant_ptr`, or `sycl::decorated_generic_ptr`. The prefix `raw` and `decorated` determine whether the interface exposes undecorated types or decorated types respectively. Accessors with an access target `global_buffer`, `constant_buffer`, or `local`, can be converted into explicit pointer classes (`multi_ptr`). Explicit pointer class values cannot be passed as parameters to kernels or stored in global memory.

For templates that need to adapt to different address spaces, a `sycl::multi_ptr` class is defined which is templated via a compile-time constant enumerator value to specify the address space.

#### 3.7.2.2 Memory consistency inside SYCL kernels

The SYCL memory consistency model is based upon the memory consistency model of the C++ core language. Where SYCL offers extensions to classes and functions that may affect memory consistency (e.g. `sycl::atomic_ref`), the default behavior when these extensions are not used always matches the behavior of standard C++.

A SYCL implementation must guarantee that the same memory consistency model is used across host and device code. Every device compiler must support the memory model defined by the minimum version of C++ described in Section 3.8.1; SYCL implementations supporting additional versions of C++ must also support the corresponding memory models.

The full C++ memory model is not guaranteed to be supported by every device, nor across all combinations of devices within a context. The memory orderings supported by a specific device and context can be queried using functionalities of the `sycl::device` and `sycl::context` classes, respectively.

Within a SYCL kernel, all memory has load/store consistency within a work-item. Ensuring memory consistency across different work-items requires careful usage of group barrier operations, mem-fence operations and atomic operations. On any SYCL device, local and global memory may be made consistent across work-items in a single group through use of a group barrier operation. On SYCL devices supporting acquire-release or sequentially consistent memory orderings, all memory visible to a set of work-items may be made consistent (with a single *happens-before* relation) across the work-items in that set through the use of mem-fence and atomic operations.

The set of work-items and devices to which the memory ordering constraints of a given atomic operation apply is controlled by a memory scope constraint, which can take one of the following values:

- `memory_scope::work_item` The ordering constraint applies only to the calling work-item. This is only useful for image operations, since all other operations within a work-item are guaranteed to execute in program order.
- `memory_scope::sub_group` The ordering constraint applies only to work-items in the same sub-group as the calling work-item.
- `memory_scope::work_group` The ordering constraint applies only to work-items in the same work-group as the calling work-item. This is the broadest scope that can be applied to atomic operations in work-group local memory. Using any broader scope for atomic operations in work-group local memory is treated as though `memory_scope::work_group` was specified.
- `memory_scope::device` The ordering constraint applies only to work-items executing on the same device as the calling work-item.
- `memory_scope::system` The ordering constraint applies to any work-item or host thread in the system that is currently permitted to access the memory allocation containing the referenced object, as defined by the capabilities of buffers and USM.

**Note for this provisional version:** The addition of memory scopes to the C++ memory model modifies the definition of some concepts from the C++ core language. For example: data races, the synchronizes-with relationship and sequential consistency must be defined in a way that accounts for atomic operations with differing (but compatible) scopes, in a manner similar to the OpenCL 2.0 specification [4]. Modified definitions of these concepts will be included in the final version of this specification.

These memory consistency guarantees are independent of any forward progress guarantees. Communication and synchronization between work-items in different work-groups is unsafe in general, but is supported on devices where all of the following conditions are true:

- Acquire-release or sequentially consistent memory ordering is supported at device scope
- Work-items in different work-groups make independent forward progress

#### 3.7.2.3 Atomic operations

Atomic operations can be performed on memory in buffers and USM. The range of atomic operations available on a specific device is limited by the atomic capabilities of that device. The `sycl::atomic_ref` class must be used to provide safe atomic access to the buffer or USM allocation from device code.

## 3.8 The SYCL programming model

A SYCL program is written in standard C++. Host code and device code is written in the same C++ source file, enabling instantiation of templated kernels from host code and also enabling kernel source code to be shared between host and device. The device kernels are encapsulated C++ function objects (a type callable with `operator()` or a lambda function), which have been designated to be compiled as SYCL kernels.

SYCL programs target heterogeneous systems. The kernels may be compiled and optimized for multiple different processor architectures with very different binary representations.

### 3.8.1 Minimum version of C++

The C++ features used in SYCL are based on a specific version of C++. Implementations of SYCL must support this minimum C++ version, which defines the C++ constructs that can consequently be used by SYCL feature definitions (for example, lambdas).

The minimum C++ version of this SYCL specification is determined by the normative C++ core language defined in Section 3.3. All implementations of this specification must support at least this core language, and features within this specification are defined using features of the core language. Note that not all core language constructs are supported within SYCL kernel functions or code invoked by a SYCL kernel function, as detailed by Section 5.3.

Implementations may support newer C++ versions than the minimum required by SYCL. Code written using newer features than the SYCL requirement, though, may not be portable to other implementations that don't support the same C++ version.

### 3.8.2 Alignment with future versions of C++

Some features of SYCL are aligned with the next C++ specification, as defined in Section 3.3.

The following features are pre-adopted by SYCL 2020 and made available in the `sycl::` namespace: `std::span`, `std::bit_cast`. The implementations of pre-adopted features are compliant with the next C++ specification, and are expected to forward directly to standard C++ features in a future version of SYCL.

The following features of SYCL 2020 use syntax based on the next C++ specification: `sycl::atomic_ref`. These features behave as described in the next C++ specification, barring modifications to ensure compatibility with other SYCL 2020 features and heterogeneous programming. Any such modifications are documented in the corresponding sections of this specification.

### 3.8.3 Basic data parallel kernels

Data-parallel kernels that execute as multiple work-items and where no local synchronization is required are enqueued with the `sycl::parallel_for` function parameterized by a `sycl::range` parameter. These kernels will execute the kernel function body once for each work-item in the range. The range passed to `sycl::parallel_for` represents the global size of a SYCL kernel and will be divided into work-groups whose size is chosen by the SYCL runtime. Barrier synchronization is not valid within these work-groups.

Variables with reduction semantics can be added to basic data parallel kernels using the features described in Section 4.10.2.

### 3.8.4 Work-group data parallel kernels

Data parallel kernels can also execute in a mode where the set of work-items is divided into work-groups of user-defined dimensions. The user specifies the global range and local work-group size as parameters to the `sycl::parallel_for` function with a `sycl::nd_range` parameter. In this mode of execution, kernels execute over the nd_range in work-groups of the specified size. It is possible to share data among work-items within the same work-group in local or global memory and to synchronize between work-items in the same work-group by calling the `nd_item::barrier()` function. All work-groups in a given `parallel_for` will be the same size and the global size defined in the nd_range must be a multiple of the work-group size in each dimension.

Work-groups may be further subdivided into sub-groups. The size and number of sub-groups is implementation-defined and may differ for each kernel, and different devices may make different guarantees with respect to how sub-groups within a work-group are scheduled. The maximum number of work-items in any sub-group in a kernel is based on a combination of the kernel and its dispatch dimensions. The size of any sub-group in the dispatch is between 1 and this maximum sub-group size, and the size of an individual sub-group is invariant for the duration of a kernel's execution.

To maximize portability across devices, developers should not assume that work-items within a sub-group execute in any particular order, that work-groups are subdivided into sub-groups in a specific way, nor that two sub-groups within a work-group will make independent forward progress with respect to one another.

Variables with reduction semantics can be added to work-group data parallel kernels using the features described in Section 4.10.2.

### 3.8.5 Hierarchical data parallel kernels

**Note for this provisional version:** The hierarchical data parallel kernel feature described next is being reworked to better align with the frameworks and patterns prevalent in modern programming, based on developer feedback, and the feature as written is therefore temporarily discouraged from use in development of new code. This "temporary discouragement" will be removed in a future version of the SYCL specification, when full use of the updated feature will be recommended for use in new code.

The SYCL compiler provides a way of specifying data parallel kernels that execute within work-groups via a different syntax which highlights the hierarchical nature of the parallelism. This mode is purely a compiler feature and does not change the execution model of the kernel. Instead of calling `sycl::parallel_for` the user calls `sycl::parallel_for_work_group` with a `sycl::range` value representing the number of work-groups to launch and optionally a second `sycl::range` representing the size of each work-group for performance tuning. All code within the `parallel_for_work_group` scope effectively executes once per work-group. Within the `parallel_for_work_group` scope, it is possible to call `parallel_for_work_item` which creates a new scope in which all work-items within the current work-group execute. This enables a programmer to write code that looks like there is an inner work-item loop inside an outer work-group loop, which closely matches the effect of the execution model. All variables declared inside the `parallel_for_work_group` scope are allocated in work-group local memory, whereas all variables declared inside the `parallel_for_work_item` scope are declared in private memory. All `parallel_for_work_item` calls within a given `parallel_for_work_group` execution must have the same dimensions.

### 3.8.6 Kernels that are not launched over parallel instances

Simple kernels for which only a single instance of the kernel function will be executed are enqueued with the `sycl::single_task` function. The kernel enqueued takes no "work-item id" parameter and will only execute once. The behavior is logically equivalent to executing a kernel on a single compute unit with a single work-group comprising only one work-item. Such kernels may be enqueued on multiple queues and devices and as a result may, like any other OpenCL entity, be executed in task-parallel fashion.

### 3.8.7 Pre-defined kernels

Some SYCL backends may expose pre-defined functionality to users as kernels. These kernels are not programmable, hence they are not bound by the SYCL C++ programming model restrictions and how they are written is implementation-defined.

### 3.8.8 Synchronization

Synchronization of processing elements executing inside a device is handled by the SYCL device kernel following SYCL Kernel Execution Model. The synchronization of the different SYCL device kernels executing with the host memory is handled by the SYCL Application via the SYCL runtime.

#### 3.8.8.1 Synchronization in the SYCL application

Synchronization points between host and device(s) are exposed through the following operations:

- *Buffer destruction*: The destructors for `sycl::buffer`, `sycl::unsampled_image` and `sycl::sampled_image` objects wait for all submitted work on those objects to complete and copies the data back to host memory before returning, if there is anything to copy back to the host or if the objects were constructed with attached host memory.

  More complex forms of synchronization on buffer destruction can be specified by the user by constructing buffers with other kinds of references to memory, such as `shared_ptr` and `unique_ptr`.

- *Host Accessors*: The constructor for a host accessor waits for all kernels that modify the same buffer (or image) in any queues to complete and then copies data back to host memory before the constructor returns. Any command groups with requirements to the same memory object cannot execute until the host accessor is destroyed (see 3.5).

- *Command group enqueue*: The SYCL runtime internally ensures that any command groups added to queues have the correct event dependencies added to those queues to ensure correct operation. Adding command groups to queues never blocks. Instead any required synchronization is added to the queue and events of type `sycl::event` are returned by the queue's submit function that contain event information related to the specific command group.

- *Queue operations*: The user can manually use queue operations, such as **wait** to block execution of the caller thread until all the command groups submitted to the queue have finished execution. Note that this will also affect the dependencies of those command groups in other queues.

- *SYCL event objects*: SYCL provides `sycl::event` objects which can be used for user synchronization. If synchronization is required across SYCL context from different SYCL backends, then the SYCL runtime ensures that any extra host-based synchronization is added to enable the SYCL event objects to operate between contexts correctly.

Note that the destructors of other SYCL objects (`sycl::queue`, `sycl::context`...) do not block. Only a `sycl::buffer` or `sycl::unsampled_image` destructor might block. The rationale is that an object without any side effect on the host does not need to block on destruction as it would impact the performance. So it is up to the programmer to use a method to wait for completion in some cases if this does not fit the goal. See Section 3.8.12 for more information on object life time.

#### 3.8.8.2 Synchronization in SYCL kernels

In SYCL, synchronization can be either global or local within a group of work-items. Synchronization between work-items in a single group is achieved using a group barrier.

All the work-items of a group must execute the barrier before any are allowed to continue execution beyond the barrier. Note that the group barrier must be encountered by all work-items of a group executing the kernel or by none at all. In SYCL, work-group barrier and sub-group barrier functions are exposed through methods of the `group` class and `sub_group` class.

There is no mechanism for synchronization between work-items of different work-groups.

### 3.8.9 Error handling

In SYCL, there are two types of error: synchronous errors that can be detected immediately when an API call is made, and asynchronous errors that can only be detected later after an API call has returned. Synchronous errors, such as failure to construct an object, are reported immediately by the runtime throwing an exception. Asynchronous errors, such as an error occurring during execution of a kernel on a device, are reported via an asynchronous error-handler mechanism.

Asynchronous errors are not reported immediately as they occur. The asynchronous error handler for a context or queue is called with a `sycl::exception_list` object, which contains a list of asynchronously-generated exception objects, on the conditions described by 4.15.1.1 and 4.15.1.2.

Asynchronous errors may be generated regardless of whether the user has specified any asynchronous error handler(s), as described in 4.15.1.2.

Some SYCL backends can report errors that are specific to the platform they are targeting, or that are more concrete than the errors provided by the SYCL API. Any error reported by a SYCL backend must derive from the base `sycl::exception`. When a user wishes to capture specifically an error thrown by a SYCL backend, she must include the SYCL backend-specific headers for said SYCL backend.

### 3.8.10 Fallback mechanism

A command group function object can be submitted either to a single queue to be executed on, or to a secondary queue. If a command group function object fails to be enqueued to the primary queue, then the system will attempt to enqueue it to the secondary queue, if given as a parameter to the submit function. If the command group function object fails to be queued to both of these queues, then a synchronous SYCL exception will be thrown.

It is possible that a command group may be successfully enqueued, but then asynchronously fail to run, for some reason. In this case, it may be possible for the runtime system to execute the command group function object on the secondary queue, instead of the primary queue. The situations where a SYCL runtime may be able to achieve this asynchronous fall-back is implementation-defined.

### 3.8.11 Scheduling of kernels and data movement

A command group function object takes a reference to a command group `handler` as a parameter and anything within that scope is immediately executed and has to get the handler object as a parameter. The intention is that a user will perform calls to SYCL functions, methods, destructors and constructors inside that scope. These calls will be non-blocking on the host, but enqueue operations to the queue that the command group is submitted to. All user functions within the command group scope will be called on the host as the command group function object is executed, but any runtime SYCL operations will be queued.

SYCL queue objects execute all operations inside the command group function object in order, but command group function objects execute out-of-order from each other.

### 3.8.12 Managing object lifetimes

A SYCL application does not initialize any SYCL backend features until a `sycl::context` object is created. A user does not need to explicitly create a `sycl::context` object, but they do need to explicitly create a `sycl::queue` object, for which a `sycl::context` object will be implicitly created if not provided by the user.

All SYCL backend objects encapsulated in SYCL objects are reference-counted and will be destroyed once all references have been released. This means that a user needs only create a SYCL queue (which will automatically create an SYCL context) for the lifetime of their application to initialize and release any SYCL backend objects safely.

There is no global state specified to be required in SYCL implementations. This means, for example, that if the user creates two queues without explicitly constructing a common context, then a SYCL implementation does not have to create a shared context for the two queues. Implementations are free to share or cache state globally for performance, but it is not required.

Memory objects can be constructed with or without attached host memory. If no host memory is attached at the point of construction, then destruction of that memory object is non-blocking. The user may use C++ standard pointer classes for sharing the host data with the user application and for defining blocking, or non-blocking behavior of the buffers and images. If host memory is attached by using a raw pointer, then the default behavior is followed, which is that the destructor will block until any command groups operating on the memory object have completed, then, if the contents of the memory object is modified on a device those contents are copied back to host and only then does the destructor return. Instead of a raw pointer, a `unique_ptr` may be provided, which uses move semantics for initializing and using the associated host memory. In this case, the behavior of the buffer in relation to the user application will be non-blocking on destruction. In the case where host memory is shared between the user application and the SYCL runtime, then the reference counter of the `shared_ptr` determines whether the buffer needs to copy data back on destruction, and in that case the blocking or non-blocking behavior depends on the user application.

As said in Section 3.8.8, the only blocking operations in SYCL (apart from explicit wait operations) are:

- host accessor constructor, which waits for any kernels enqueued before its creation that write to the corresponding object to finish and be copied back to host memory before it starts processing. The host accessor does not necessarily copy back to the same host memory as initially given by the user;
- memory object destruction, in the case where copies back to host memory have to be done or when the host memory is used as a backing-store.

### 3.8.13 Device discovery and selection

A user specifies which queue to submit a command group function object on and each queue is targeted to run on a specific device (and context). A user can specify the actual device on queue creation, or they can specify a device selector which causes the SYCL runtime to choose a device based on the user's provided preferences. Specifying a device selector causes the SYCL runtime to perform device discovery. No device discovery is performed until a SYCL device selector is passed to a queue constructor. Device topology may be cached by the SYCL runtime, but this is not required.

Device discovery will return all devices from all platforms exposed by all the supported SYCL backends, including the host backend.

### 3.8.14 Interfacing with SYCL backend API

There are two styles of developing a SYCL application: (1) Writing a pure SYCL generic application or (2) Writing a SYCL application that relies on some SYCL backend specific behavior.

When users follow (1), there is no assumption about what SYCL backend will be used during compilation or execution of the SYCL Application. Therefore, the SYCL backend API is not assumed to be available to the developer. Only standard C++ types and interfaces are assumed to be available, as described in Section 3.8. Users only need to include the `SYCL/sycl.hpp` header to write a SYCL generic application.

On the other hand, when users follow (2), they must know what SYCL backend APIs are they using. In this case, any header required for the normal programmability of the SYCL backend API is assumed to be available to the user. In addition to the `SYCL/sycl.hpp` header, users must also include the SYCL backend-specific header `SYCL/backend/backend_name/backend_name.hpp`, where backend_name represents the name of the backend in lower case, e.g. `SYCL/backend/opencl/opencl.hpp`. The SYCL backend-specific header provides the interoperability interface for the SYCL API to interact with native backend objects. Any type or header from the underlying SYCL backend API is included by the SYCL backend-specific header, under a namespace named after the backend name, e.g. `namespace opencl` would encapsulate the OpenCL headers. This avoids accidental pollution of user space with SYCL backend-specific types.

The interoperability API is defined in the 4.

## 3.9 Memory objects

SYCL memory objects represent data that is handled by the SYCL runtime and can represent allocations in one or multiple devices at any time. Memory objects, both buffers and images, may have one or more underlying native backend objects to ensure that queues objects can use data in any device. A SYCL implementation may have multiple native backend objects for the same device. The SYCL runtime is responsible of ensuring the different copies are up-to-date whenever necessary, using whatever mechanism is available in the system to update the copies of the underlying native backend objects.

**Implementation Note**: A valid mechanism for this update is to transfer the data from one SYCL backend into the system memory using the SYCL backend-specific mechanism available, and then transfer it to a different device using the mechanism exposed by the new SYCL backend.

Memory objects in SYCL fall into one of two categories: buffer objects and image objects. A buffer object stores a one-, two- or three-dimensional collection of elements that are stored linearly directly back to back in the same way C or C++ stores arrays. An image object is used to store a one-, two- or three-dimensional texture, frame-buffer or image that may be stored in an optimized and device-specific format in memory and must be accessed through specialized operations.

Elements of a buffer object can be a scalar data type (such as an `int`, `float`), vector data type, or a user-defined structure. In SYCL, a buffer object is a templated type (`sycl::buffer`), parameterized by the element type and number of dimensions. An image object is stored in one of a limited number of formats. The elements of an image object are selected from a list of predefined image formats which are provided by an underlying SYCL backend implementation. Images are encapsulated in the `sycl::unsampled_image` or `sycl::sampled_image` types, which are templated by the number of dimensions in the image. The minimum number of elements in a memory object is one.

The fundamental differences between a buffer and an image object are:

- Elements in a buffer are stored in an array of 1, 2 or 3 dimensions and can be accessed using an accessor by a kernel executing on a device. The accessors for kernels provide a method to get C++ pointer types, or the `sycl::global_ptr`, `sycl::constant_ptr` classes. Elements of an image are stored in a format that is opaque to the user and cannot be directly accessed using a pointer. SYCL provides image accessors and samplers to allow a kernel to read from or write to an image.

- For a buffer object the data is accessed within a kernel in the same format as it is stored in memory, but in the case of an image object the data is not necessarily accessed within a kernel in the same format as it is stored in memory.

  Image elements are always a 4-component vector (each component can be a float or signed/ unsigned integer) in a kernel. The SYCL accessor and sampler methods to read from an image convert an image element from the format that it is stored in to a 4-component vector. Similarly, the SYCL accessor methods provided to write to an image convert the image element from a 4-component vector to the appropriate image format specified such as 4 8-bit elements, for example.

Users may want fine-grained control of the synchronization, memory management and storage semantics of SYCL image or buffer objects. For example, a user may wish to specify the host memory for a memory object to use, but may not want the memory object to block on destruction.

Depending on the control and the use cases of the SYCL applications, well established C++ classes and patterns can be used for reference counting and sharing data between user applications and the SYCL runtime. For control over memory allocation on the host and mapping between host and device memory, pre-defined or user-defined C++ `allocator` classes are used. For better control of synchronization between a SYCL and a non SYCL application that share data, `shared_ptr` and `mutex` classes are used.

## 3.10 SYCL device compiler

To enable SYCL to work on a variety of platforms, with different devices, operating systems, build systems and host compilers, SYCL provides a number of options to implementers to enable the compilation of SYCL kernels for devices, while still providing a unified programming model to the user.

### 3.10.1 Building a SYCL program

A SYCL program runs on a *host* and one or more SYCL devices. This requires a compilation model that enables compilation for a variety of targets. There is only ever one host for the SYCL program, so the compilation of the source code for the host must happen once and only once. Both kernel and non-kernel source code is compiled for the host.

The design of SYCL enables a single SYCL source file to be passed to multiple, different compilers, using the SMCP technique. This is an implementation option and is not required. What this option enables is for an implementer to provide a device compiler only and not have to provide a host compiler. A programmer who uses such an implementation will compile the same source file twice: once with the host compiler of their choice and once with a device compiler. This approach allows the advantages of having a single source file for both host code and kernels, while still allowing users an independent choice of host and SYCL device compilers.

Only the kernels are compiled for SYCL devices. Therefore, any compiler that compiles only for one or more devices must not compile non-kernel source code. Kernels are contained within C++ source code and may be dependent on lambda capture and template parameters, so compilation of the non-kernel code must determine lambda captures and template parameters, but not generate device code for non-kernel code.

Compilation of a SYCL program may follow either of the following options. The choice of option is made by the implementer:

1. *Separate compilation*: One or more device compilers compile just the SYCL kernels for one or more devices. The device compilers all produce header files for interfacing between the host compiler and the SYCL runtime,which are integrated together with a tool that produces a single header file. The user compiles the source file with a normal C++ host compiler for their platform. The user must ensure that the host compiler is given the correct command-line arguments to ensure that the device compiler output header file is `#included` from inside the SYCL header files.

2. *Single-source compiler*: In this approach, a single compiler may compile an entire source file for both host and one or more devices. It is the responsibility of the single-source compiler to enable kernels to be compiled correctly for devices and enqueued from the host.

An implementer of SYCL may choose an implementation approach from the options above.

### 3.10.2 Naming of kernels

SYCL kernels are extracted from C++ source files and stored in an implementation-defined format. When the SYCL runtime needs to enqueue a SYCL kernel, it is necessary for the SYCL runtime to load the kernel and pass it to a SYCL backend API. This requires the kernel to have a name that is unique at enclosing namespace scope, to enable an association between the kernel invocation and the kernel itself. The association is achieved using a kernel name, which is a C++ type name.

For a named function object, the kernel name can be the same type as the function object itself, as long as the function object type is unique across the enclosing namespace scopes. For a lambda function the user may optionally provide a name for debugging or other reasons. In SYCL, this optional name is provided as a template parameter to the kernel invocation, e.g. `parallel_for<class kernelName>`, and this name may optionally be forward declared at namespace scope (but must always avoid conflict with another name at enclosing namespace scope).

A device compiler should detect the kernel invocations (e.g. `parallel_for`) in the source code and compile the enclosed kernels, storing them with their associated type name. For details please refer to 5.2.

## 3.11 Language restrictions in kernels

The SYCL kernels are executed on SYCL devices and all of the functions called from a SYCL kernel are going to be compiled for the device by a SYCL device compiler. Due to restrictions of the heterogeneous devices where the SYCL kernel will execute, there are certain restrictions on the base C++ language features that can be used inside kernel code. For details on language restrictions please refer to 5.3.

SYCL kernels use parameters that are captured by value in the command group scope or are passed from the host to the device using the data management runtime classes of `sycl::accessors`. Sharing data structures between host and device code imposes certain restrictions, such as use of only user defined classes that are *C++ trivially copyable* classes for the data structures, and in general, no pointers initialized for the host can be used on the device. The only way of passing pointers to a kernel is through the `sycl::accessor` class, which supports the `sycl::buffer`, `sycl::unsampled_image` and `sycl::sampled_image` classes. No hierarchical structures of these classes are supported and any other data containers need to be converted to the SYCL data management classes using the SYCL interface. For more details on the rules for kernel parameter passing, please refer to 4.14.4.

### 3.11.1 SYCL linker

In SYCL only offline linking is supported for SYCL programs and libraries, however the mechanism is optional. In the case of linking C++ functions to a SYCL application, where the definitions are not available in the same translation unit of the compiler, then the macro `SYCL_EXTERNAL` has to be provided.

### 3.11.2 Functions and data types available in kernels

Inside kernels, the functions and data types available are restricted by the underlying capabilities of SYCL backend devices.

## 3.12 Execution of kernels on the SYCL host device

## 3.13 Endianness support

SYCL supports both big-endian and little-endian systems as long as it is supported by the used SYCL backends. However SYCL does not support mix-endian systems and does not support specifying the endianness of data within a SYCL kernel function.

Users must be aware of the endianness of the host and the SYCL backend devices they are targeting to ensure kernel arguments are processed correctly when applicable.

## 3.14 Example SYCL application

Below is a more complex example application, combining some of the features described above.

```
#include <sycl.hpp>
#include <iostream>

using namespace sycl;

// Size of the matrices
```

```
const size_t N = 2000;
const size_t M = 3000;

int main() {
  // Create a queue to work on
  queue myQueue;

  // Create some 2D buffers of float for our matrices
  buffer<float, 2> a { range<2>{N, M} };
  buffer<float, 2> b { range<2>{N, M} };
  buffer<float, 2> c { range<2>{N, M} };

  // Launch a first asynchronous kernel to initialize a
  myQueue.submit( [&](handler& cgh) {
    // The kernel write a, so get a write accessor on it
    accessor A { a, cgh, write_only };

    // Enqueue a parallel kernel iterating on a N*M 2D iteration space
    cgh.parallel_for<class init_a>(range<2> {N, M}, [=](id<2> index) {
      A[index] = index[0] * 2 + index[1]; });
  });

  // Launch an asynchronous kernel to initialize b
  myQueue.submit( [&](handler& cgh) {
    // The kernel write b, so get a write accessor on it
    accessor B { b, cgh, write_only };
    /* From the access pattern above, the SYCL runtime detect this
       command_group is independent from the first one and can be
       scheduled independently */

    // Enqueue a parallel kernel iterating on a N*M 2D iteration space
    cgh.parallel_for<class init_b>(range<2> {N, M}, [=](id<2> index) {
      B[index] = index[0] * 2014 + index[1] * 42;
    });
  });

  // Launch an asynchronous kernel to compute matrix addition c = a + b
  myQueue.submit( [&](handler& cgh) {
    // In the kernel a and b are read, but c is written
    accessor A { a, cgh, read_only };
    accessor B { b, cgh, read_only };
    accessor C { c, cgh, write_only };
    // From these accessors, the SYCL runtime will ensure that when
    // this kernel is run, the kernels computing a and b completed

    // Enqueue a parallel kernel iterating on a N*M 2D iteration space
    cgh.parallel_for<class matrix_add>(range<2> {N, M}, [=](id<2> index) {
        C[index] = A[index] + B[index]; });
    });

  /* Ask an access to read c from the host-side.
     This form implies access::target::host_buffer. The SYCL runtime
     ensures that c is ready when the accessor is returned */
  host_accessor C { c, read_only };
  std::cout << std::endl << "Result:" << std::endl;
```

```
  for (size_t i = 0; i < N; i++) {
    for (size_t j = 0; j < M; j++) {
      // Compare the result to the analytic value
      if (C[i][j] != i * (2 + 2014) + j * (1 + 42)) {
        std::cout << "Wrong value " << C[i][j] << " on element " << i << " "
                  << j << std::endl;
        exit(-1);
      }
    }
  }

  std::cout << "Good computation!" << std::endl;
  return 0;
}
```

# 4. SYCL programming interface

The SYCL programming interface provides a common abstracted feature set to one or more SYCL backend APIs. This section describes the C++ library interface to the SYCL runtime which executes across those SYCL backends.

The entirety of the SYCL interface defined in this section is required to be available for any SYCL backends, with the exception of the interoperability interface, which is described in general terms in this document, not pertaining to any particular SYCL backend.

SYCL guarantees that all the member functions and special member functions of the SYCL classes described are thread safe.

## 4.1 Backends

The SYCL backends that are available to a SYCL implementation can be identified using the `enum class backend`
.

```
namespace sycl {
enum class backend {
  <see-below>
};
}  // namespace sycl
```

The `enum class backend` is implementation defined and must be populated with a unique identifier for each SYCL backend that the SYCL implementation supports, containing at least one SYCL backend that is a SYCL host backend.

Each named SYCL backend enumerated in the `enum class backend` must be associated with a SYCL backend specification. Many sections of this specification will refer to the associated SYCL backend specification.

### 4.1.1 Backend macros

As the identifiers defined in `enum class backend` are implementation defined a SYCL implementation must also define a pre-processor macro for each of these identifiers. If the SYCL backend is defined by the Khronos SYCL group, the name of the macro has the form SYCL_BACKEND_<backend_name>, where *backend_name* is the associated identifier from `backend` in all upper-case. See Chapter 6 for the name of the macro if the vendor defines the SYCL backend outside of the Khronos SYCL group.

## 4.2 Generic vs non-generic SYCL

The SYCL programming API is split into two categories; generic SYCL and non-generic SYCL. Almost everything in the SYCL programming API is considered generic SYCL. However any usage of the the `enum class` backend is considered non-generic SYCL and should only be used for SYCL backend specialized code paths, as the identifiers defined in backend are implementation defined.

In any non-generic SYCL application code where the backend enum class is used, the expression must be guarded with a pre-process #ifdef guard using the associated pre-process macro to ensure that the SYCL application will compile even if the SYCL implementation does not support that SYCL backend being specialized for.

## 4.3 Header files and namespaces

SYCL provides one standard header file: "SYCL/sycl.hpp", which needs to be included in every translation unit which uses the SYCL programming API.

All SYCL classes, constants, types and functions defined by this specification should exist within the ::sycl namespace.

For compatibility with SYCL 1.2.1, SYCL provides another standard header file: "CL/sycl.hpp", which can be included in place of "SYCL/sycl.hpp".

In that case, all SYCL classes, constants, types and functions defined by this specification should exist within the ::cl::sycl C++ namespace.

For consistency the programming API will only refer to the "sycl.hpp" header and the ::sycl but this should be considered synonymous with the SYCL 1.2.1 header and namespace.

The sycl::detail namespace is reserved for implementation details.

When a SYCL backend is defined by the Khronos SYCL group, functionality for that SYCL backend is available via the header "SYCL/backend/<backend_name>.hpp", and all SYCL backend-specific functionality is made available in the namespace sycl::<backend_name> where *backend_name* is the name of the SYCL backend as defined in the SYCL backend specification.

Chapter 6 defines the allowable header files and namespaces for any extensions that a vendor may provide, including any SYCL backend that the vendor may define outside of the Khronos SYCL group.

## 4.4 Class availability

In SYCL some SYCL runtime classes are available to the SYCL application, some are available within a SYCL kernel function and some are available on both and can be passed as parameters to a SYCL kernel function.

Each of the following SYCL runtime classes: buffer, buffer_allocator, context, device, event, exception, handler, id, image_allocator, kernel, marray, module, nd_range, platform, queue, range, sampled_image, sampler, stream, unsampled_image and vec must be available to the host application.

Each of the following SYCL runtime classes: accessor, atomic_ref, device_event, group, h_item, id, item, marray, multi_ptr, nd_item, range, reducer, sampler, stream, sub_group and vec must be available within a SYCL kernel function.

Each of the following SYCL runtime classes: accessor, id, marray, range, reducer, sampler, stream and vec are permitted as parameters to a SYCL kernel function.

# 4.5 Common interface

When a dimension template parameter is used in SYCL classes, it is defaulted as 1 in most cases.

## 4.5.1 Param traits class

The class `param_traits` is a C++ type trait for providing an alias to the return type associated with each info parameter. An implementation must provide a specialization of the `param_traits` class for every info parameter with the associated return type as defined in the info parameter tables.

```
namespace sycl {
namespace info {
template <typename T, T param>
class param_traits {
 public:

   using return_type = __return_type__<T, param>;

};
}  // namespace info
}  // namespace sycl
```

## 4.5.2 Backend interoperability

Many of the SYCL runtime classes may be implemented such that they encapsulate an object unique to the SYCL backend that underpins the functionality of that class. Where appropriate, these classes may provide an interface for interoperating between the SYCL runtime object and the native backend object in order to support interoperability within an applications between SYCL and the associated SYCL backend API.

There are two forms of interoperability with SYCL runtime classes; interoperability on the SYCL application with the SYCL backend API and interoperability within a SYCL kernel function with the equivalent kernel language types of the SYCL backend. SYCL application interoperability and SYCL kernel function interoperability are provided via different interfaces and may have different native backend object types.

SYCL application interoperability may be provided for `buffer`, `context`, `device`, `event`, `kernel`, `module`, `platform`, `queue`, `sampled_image`, `sampler`, `stream` and `unsampled_image`.

SYCL kernel function interoperability may be provided for `accessor`, `stream` and `device_event` inside the SYCL kernel function scope only. SYCL kernel function interoperability is not available inside command group scope.

Support for SYCL backend interoperability is optional and therefore not required to be provided by a SYCL implementation. A SYCL application using SYCL backend interoperability is considered to be non-generic SYCL.

Details on the interoperability for a given SYCL backend are available on the SYCL backend specification document for that SYCL backend.

#### 4.5.2.1 Type traits `backend_traits`

```
namespace sycl {

```

```
template <backend Backend>
class backend_traits {
 public:

  template <class T>
  using native_type = see-below;

  using errc = see-below;

};

}  // namespace sycl
```

A series of type traits are provided for SYCL backend interoperability, defined in the `backend_traits` class.

A specialization of `backend_traits` must be provided for each named SYCL backend enumerated in the enum class `backend`.

- For each SYCL runtime class `T` which supports SYCL application interoperability with the SYCL backend, a specialisation of `native_type` must be defined as the type of SYCL application interoperability native backend object associated with `T` for the SYCL backend, specified in the SYCL backend specification.
- For each SYCL runtime class `T` which supports kernel function interoperability with the SYCL backend, a specialisation of `native_type` within `backend_traits` must be defined as the type of the kernel function interoperability native backend object associated with `T` for the SYCL backend, specified in the backend specification.
- A specialization of `errc` must be defined as the SYCL backend error code type.

#### 4.5.2.2 Template function `get_native`

```
namespace sycl {

template<backend Backend, class T>
backend_traits<Backend>::native_type<T> get_native(const T &syclObject);

}  // namespace sycl
```

For each SYCL runtime class `T` which supports SYCL application interoperability, a specialisation of `get_native` must be defined, which takes an instance of `T` and returns a SYCL application interoperability native backend object associated with `syclObject` which can be used for SYCL application interoperability. The lifetime of the object returned are backend-defined and specified in the backend specification.

For each SYCL runtime class `T` which supports kernel function interoperability, a specialisation of `get_native` must be defined, which takes an instance of `T` and returns the kernel function interoperability native backend object associated with `syclObject` which can be used for kernel function interoperability. The lifetime of the object returned are backend-defined and specified in the backend specification.

#### 4.5.2.3 Template functions `make_*`

```
namespace sycl {

```

```
template<backend Backend>
platform make_platform(const backend_traits<Backend>::native_type<platform> &backendObject);

template<backend Backend>
device make_device(const backend_traits<Backend>::native_type<device> &backendObject);

template<backend Backend>
context make_context(const backend_traits<Backend>::native_type<context> &backendObject);

template<backend Backend>
queue make_queue(const backend_traits<Backend>::native_type<queue> &backendObject,
                 const context &targetContext);

template<backend Backend>
queue make_event(const backend_traits<Backend>::native_type<event> &backendObject,
                 const context &targetContext);

template<backend Backend>
buffer make_buffer(const backend_traits<Backend>::native_type<buffer> &backendObject,
                   const context &targetContext);

template<backend Backend>
image make_image(const backend_traits<Backend>::native_type<image> &backendObject,
                 const context &targetContext);

template<backend Backend>
sampler make_sampler(const backend_traits<Backend>::native_type<sampler> &backendObject,
                     const context &targetContext);

template<backend Backend>
stream make_stream(const backend_traits<Backend>::native_type<stream> &backendObject,
                   const context &targetContext);

template<backend Backend>
kernel make_kernel(const backend_traits<Backend>::native_type<event> &backendObject,
                   const context &targetContext);

}  // namespace sycl
```

For each SYCL runtime class `T` which supports SYCL application interoperability, a specialisation of the appropriate template function `make_{sycl_class}` where `{sycl_class}` is the class name of `T`, must be defined, which takes a SYCL application interoperability native backend object and constructs and returns an instance of `T`. The lifetime of the object returned is backend-defined and specified in the backend specification.

### 4.5.3 Common reference semantics

Each of the following SYCL runtime classes: `accessor`, `buffer`, `context`, `device`, `event`, `kernel`, `module`, `queue`, `sampled_image`, `sampler` and `unsampled_image`, must obey the following statements, where `T` is the runtime class type:

- `T` must be copy constructible and copy assignable on the host application and within SYCL kernel functions in the case that `T` is a valid kernel argument. Any instance of `T` that is constructed as a copy of another instance, via either the copy constructor or copy assignment operator, must behave as-if it were the original instance and as-if any action performed on it were also performed on the original instance and if said instance is not a host object must represent and continue to represent the same underlying native backend object as the original instance where applicable.

- `T` must be destructible on the host application and within SYCL kernel functions in the case that `T` is a valid kernel argument. When any instance of `T` is destroyed, including as a result of the copy assignment operator, any behavior specific to `T` that is specified as performed on destruction is only performed if this instance is the last remaining host copy, in accordance with the above definition of a copy and the destructor requirements described in 4.5.2 where applicable.

- `T` must be move constructible and move assignable on the host application and within SYCL kernel functions in the case that T is a valid kernel argument. Any instance of T that is constructed as a move of another instance, via either the move constructor or move assignment operator, must replace the original instance rendering said instance invalid and if said instance is not a host object must represent and continue to represent the same underlying native backend object as the original instance where applicable.

- `T` must be equality comparable on the host application. Equality between two instances of `T` (i.e. `a == b`) must be true if one instance is a copy of the other and non-equality between two instances of `T` (i.e. `a != b`) must be true if neither instance is a copy of the other, in accordance with the above definition of a copy, unless either instance has become invalidated by a move operation. By extension of the requirements above, equality on `T` must guarantee to be reflexive (i.e. `a == a`), symmetric (i.e. `a == b` implies `b == a` and `a != b` implies `b != a`) and transitive (i.e. `a == b && b == c` implies `c == a`).

- A specialization of `std::hash` for `T` must exist on the host application that returns a unique value such that if two instances of T are equal, in accordance with the above definition, then their resulting hash values are also equal and subsequently if two hash values are not equal, then their corresponding instances are also not equal, in accordance with the above definition.

Some SYCL runtime classes will have additional behavior associated with copy, movement, assignment or destruction semantics. If these are specified they are in addition to those specified above unless stated otherwise.

Each of the runtime classes mentioned above must provide a common interface of special member functions in order to fulfil the copy, move, destruction requirements and hidden friend functions in order to fulfil the equality requirements.

A hidden friend function is a function first declared via a `friend` declaration with no additional out of class or namespace scope declarations. Hidden friend functions are only visible to ADL (Argument Dependent Lookup) and are hidden from qualified and unqualified lookup. Hidden friend functions have the benefits of avoiding accidental implicit conversions and faster compilation.

These common special member functions and hidden friend functions are described in Tables 4.1 and 4.2 respectively.

```
namespace sycl {

class T {
  ...

 public:
  T(const T &rhs);

  T(T &&rhs);

  T &operator=(const T &rhs);
```

```

  T &operator=(T &&rhs);

  ~T();

  ...

  friend bool operator==(const T &lhs, const T &rhs) { /* ... */ }

  friend bool operator!=(const T &lhs, const T &rhs) { /* ... */ }

  ...
};
} // namespace sycl
```

| Special member function | Description |
|---|---|
| `T(const T &rhs)` | Constructs a `T` instance as a copy of the RHS SYCL `T` in accordance with the requirements set out above. |
| `T(T &&rhs)` | Constructs a SYCL `T` instance as a move of the RHS SYCL `T` in accordance with the requirements set out above. |
| `T &operator=(const T &rhs)` | Assigns this SYCL `T` instance with a copy of the RHS SYCL `T` in accordance with the requirements set out above. |
| `T &operator=(T &&rhs)` | Assigns this SYCL `T` instance with a move of the RHS SYCL `T` in accordance with the requirements set out above. |
| `~T()` | Destroys this SYCL `T` instance in accordance with the requirements set out in 4.5.3. On destruction of the last copy, may perform additional lifetime related operations required for the underlying native backend object specified in the SYCL backend specification, if this SYCL `T` instance was originally constructed using one of the backend interoperability `make_*` functions specified in 4.5.2.3. |
| | End of table |

Table 4.1: Common special member functions for reference semantics.

| Hidden friend function | Description |
|---|---|
| `bool operator==(const T &lhs, const T &rhs)` | Returns true if this LHS SYCL `T` is equal to the RHS SYCL `T` in accordance with the requirements set out above, otherwise returns false. |
| | Continued on next page |

Table 4.2: Common hidden friend functions for reference semantics.

| Hidden friend function | Description |
|---|---|
| `bool operator!=(const T &lhs, const T &rhs)` | Returns true if this LHS SYCL `T` is not equal to the RHS SYCL `T` in accordance with the requirements set out above, otherwise returns false. |
| | End of table |

Table 4.2: Common hidden friend functions for reference semantics.

### 4.5.4 Common by-value semantics

Each of the following SYCL runtime classes: `id`, `range`, `item`, `nd_item`, `h_item`, `group`, `sub_group` and `nd_range` must follow the following statements, where `T` is the runtime class type:

- `T` must be default copy constructible and copy assignable on the host application and within SYCL kernel functions.
- `T` must be default destructible on the host application and within SYCL kernel functions.
- `T` must be default move constructible and default move assignable on the host application and within SYCL kernel functions.
- `T` must be equality comparable on the host application and within SYCL kernel functions. Equality between two instances of `T` (i.e. `a == b`) must be true if the value of all members are equal and non-equality between two instances of `T` (i.e. `a != b`) must be true if the value of any members are not equal, unless either instance has become invalidated by a move operation. By extension of the requirements above, equality on `T` must guarantee to be reflexive (i.e. `a == a`), symmetric (i.e. `a == b` implies `b == a` and `a != b` implies `b != a`) and transitive (i.e. `a == b && b == c` implies `c == a`).

Some SYCL runtime classes will have additional behavior associated with copy, movement, assignment or destruction semantics. If these are specified they are in addition to those specified above unless stated otherwise.

Each of the runtime classes mentioned above must provide a common interface of special member functions and member functions in order to fulfil the copy, move, destruction and equality requirements, following the rule of five and the rule of zero.

These common special member functions and hidden friend functions are described in Tables 4.3 and 4.4 respectively.

```
namespace sycl {

class T {
  ...

 public:
  // If any of the following five special member functions are not
  // public, inline or defaulted, then all five of them should be
  // explicitly declared (see rule of five).
  // Otherwise, none of them should be explicitly declared
  // (see rule of zero).

  // T(const T &rhs);
```

```

  // T(T &&rhs);

  // T &operator=(const T &rhs);

  // T &operator=(T &&rhs);

  // ~T();

  ...

  friend bool operator==(const T &lhs, const T &rhs) { /* ... */ }

  friend bool operator!=(const T &lhs, const T &rhs) { /* ... */ }

  ...
};
} // namespace sycl
```

| **Special member function** *(see rule of five & rule of zero)* | **Description** |
|---|---|
| `T(const T &rhs);` | Copy constructor. |
| `T(T &&rhs);` | Move constructor. |
| `T &operator=(const T &rhs);` | Copy assignment operator. |
| `T &operator=(T &&rhs);` | Move assignment operator. |
| `~T();` | Destructor. |
| | End of table |

Table 4.3: Common special member functions for by-value semantics.

| **Hidden friend function** | **Description** |
|---|---|
| `bool operator==(const T &lhs, const T &rhs)` | Returns true if this LHS SYCL `T` is equal to the RHS SYCL `T` in accordance with the requirements set out above, otherwise returns false. |
| `bool operator!=(const T &lhs, const T &rhs)` | Returns true if this LHS SYCL `T` is not equal to the RHS SYCL `T` in accordance with the requirements set out above, otherwise returns false. |
| | End of table |

Table 4.4: Common hidden friend functions for by-value semantics.

### 4.5.5 Properties

Each of the following SYCL runtime classes: `context`, `queue`, `buffer`, `unsampled_image`, `sampled_image`, `accessor`, and `program` provide an optional parameter in each of their constructors to provide a `property_list` which contains zero or more properties. Each of those properties augments the semantics of the class with a particular feature. Each of those classes must also provide `has_property` and `get_property` member functions for querying for a particular property.

The listing below illustrates the usage of various buffer properties, described in 4.7.2.2.

The example illustrates how using properties does not affect the type of the object, thus, does not prevent the usage of SYCL objects in containers.

```
{
  context myContext;

  std::vector<buffer<int, 1>> bufferList {
    buffer<int, 1>{ptr, rng},
    buffer<int, 1>{ptr, rng, property::use_host_ptr{}},
    buffer<int, 1>{ptr, rng, property::context_bound{myContext}}
  };

  for(auto& buf : bufferList) {
    if (buf.has_property<property::context_bound>()) {
      auto prop = buf.get_property<property::context_bound>();
      assert(myContext == prop.get_context());
    }
  }
}
```

Each property is represented by a unique class and an instance of a property is an instance of that type. Some properties can be default constructed while other will require an argument on construction. A property may be applicable to more than one class, however some properties may not be compatible with each other. See the requirements for the properties of the SYCL `buffer` class, SYCL `unsampled_image` class and SYCL `sampled_image` class in Table 4.33 and Table 4.40 respectively.

Any property that is provided to a SYCL runtime class via an instance of the SYCL `property_list` class must become encapsulated by that class and therefore shared between copies of that class. As a result properties must inherit the copy and move semantics of that class as described in 4.5.3.

A SYCL implementation or a SYCL backend may provide additional properties other than those defined here, provided they are defined in accordance with the requirements described in 4.3.

#### 4.5.5.1 Properties interface

Each of the runtime classes mentioned above must provide a common interface of member functions in order to fulfil the property interface requirements.

A synopsis of the common properties interface, the SYCL `property_list` class and the SYCL property classes is provided below. The member functions of the common properties interface are listed in Table 4.6. The constructors of the SYCL `property_list` class are listed in Table 4.7.

```
namespace sycl {

template <typename propertyT>
struct is_property;

template <typename propertyT, typename syclObjectT>
struct is_property_of;

class T {
```

```
  ...

  template <typename propertyT>
  bool has_property() const;

  template <typename propertyT>
  propertyT get_property() const;

  ...
};

class property_list {
 public:
   template <typename... propertyTN>
   property_list(propertyTN... props);
};
}  // namespace sycl
```

| Traits | Description |
|---|---|
| `template <typename propertyT>`<br>`struct is_property` | An explicit specialization of `is_property` that inherits from `std::true_type` must be provided for each property, where `propertyT` is the class defining the property. This includes both standard properties described in this specification and any additional non-standard properties defined by an implementation. All other specializations of `is_property` must inherit from `std::false_type`. |
| `template <typename propertyT, syclObjectT>`<br>`struct is_property_of` | An explicit specialization of `is_property_of` that inherits from `std::true_type` must be provided for each property that can be used in constructing a given SYCL class, where `propertyT` is the class defining the property and `syclObjectT` is the SYCL class. This includes both standard properties described in this specification and any additional non-standard properties defined by an implementation. All other specializations of `is_property_of` must inherit from `std::false_type`. |
| | End of table |

Table 4.5: Traits for properties.

| Member function | Description |
|---|---|
| `template <typename propertyT>` `bool has_property()const` | Returns true if `T` was constructed with the property specified by `propertyT`. Returns false if it was not. |
| `template <typename propertyT>` `propertyT get_property()const` | Returns a copy of the property of type `propertyT` that `T` was constructed with. Must throw an `exception` with the `errc::invalid_object_error` error code if `T` was not constructed with the `propertyT` property. |
| | End of table |

Table 4.6: Common member functions of the SYCL `property` interface.

| Constructor | Description |
|---|---|
| `template <typename... propertyTN>` `property_list(propertyTN... props)` | Available only when: `is_property<property>::value` evaluates to `true` where `property` is each property in `propertyTN`. Construct a SYCL `property_list` with zero or more properties. |
| | End of table |

Table 4.7: Constructors of the SYCL `property_list` class.

# 4.6 SYCL runtime classes

## 4.6.1 Device selection

Since a system can have several SYCL-compatible devices attached, it is useful to have a way to select a specific device or a set of devices to construct a specific object such as a `device` (see Section 4.6.4) or a `queue` (see Section 4.6.5), or perform some operations on a device subset.

Device selection is done either by having already a specific instance of a `device` (see Section 4.6.4) or by providing a device selector which is a ranking function that will give an integer ranking value to all the devices on the system.

#### 4.6.1.1 Device selector

The actual interface for a device selector is a callable taking a `const device` reference and returning a value implicitly convertible to a `int`.

At any point where the SYCL runtime needs to select a SYCL `device` using a device selector, the system will query all available SYCL `device`s from all SYCL backends in the system including the SYCL host backend, will call the device selector on each device and select the one which returns the highest score. If the highest value is negative no device is selected.

In places where only one device has to be picked and high score is obtained by more than one device, then one of the tied devices will be returned, but which one is not defined and may depend on enumeration order, for example, outside the control of the SYCL runtime.

Some predefined device selectors are provided by the system as described on Table 4.8 in a header file with some definition similar to the following:

| SYCL device selectors | Description |
|---|---|
| default_selector_v | Select a SYCL device from any supported SYCL backend based on an implementation-defined heuristic. Must select the host device if no other suitable device can be found. |
| gpu_selector_v | Select a SYCL device from any supported SYCL backend for which the device type is info::device::device_type::gpu. The SYCL class constructor using it must throw an exception with the errc::runtime_error error code if no device matching this requirement can be found. |
| accelerator_selector_v | Select a SYCL device from any supported SYCL backend for which the device type is info::device::device_type::accelerator. The SYCL class constructor using it must throw an exception with the errc::runtime_error error code if no device matching this requirement can be found. |
| cpu_selector_v | Select a SYCL device from any supported SYCL backend for which the device type is info::device::device_type::cpu. The SYCL class constructor using it must throw an exception with the errc::runtime_error error code if no device matching this requirement can be found. |
| host_selector_v | Select the SYCL host device from the SYCL host backend. This must always return a valid SYCL device. |
| | End of table |

Table 4.8: Standard device selectors included with all SYCL implementations.

```
namespace sycl {

// Predefined device selectors
__unspecified__ default_selector_v;
__unspecified__ host_selector_v;
__unspecified__ cpu_selector_v;
__unspecified__ gpu_selector_v;
__unspecified__ accelerator_selector_v;

// Predefined types for compatibility with old SYCL 1.2.1 device selectors
using default_selector = __unspecified__;
using host_selector = __unspecified__;
```

```
using cpu_selector = __unspecified__;
using gpu_selector = __unspecified__;
using accelerator_selector = __unspecified__;

} // namespace sycl
```

Typical examples of default and user-provided device selectors could be:

```
sycl::device my_gpu { sycl::gpu_selector_v };

sycl::queue my_accelerator { sycl::accelerator_selector_v };

int prefer_my_vendor(const sycl::device & d) {
  // Return 1 if the vendor name in "MyVendor" or 0 else.
  // 0 does not prevent another device to be picked as a second choice
  return d.get_info<info::device::vendor>() == "MyVendor";
}

// Get the preferred device or another one if not available
sycl::device preferred_device { prefer_my_vendor };

// This throws if there is no such device in the system
sycl::queue half_precision_controller {
  // Can use a lambda as a device ranking function.
  // Returns a negative number to fail in the case there is no such device
  [] (auto &d) { return d.has(aspect::fp16) ? 1 : -1; }
};

// To ease porting SYCL 1.2.1 code, there are style some types whose
// construction leads to the equivalent predefined device selector
sycl::queue my_old_style_gpu { sycl::gpu_selector {} };
```

Note: in SYCL 1.2.1 the predefined device selectors were actually types that had to be instantiated to be used. Now they are just instances. To simplify porting code using the old type instantiations, an old-looking API is still provided, such as `sycl::default_selector`. The new predefined device selectors have their new names appended with _v to avoid conflicts, thus following the naming style used by traits in the C++ standard library. There is no requirement for the implementation to have for example `sycl::gpu_selector_v` being an instance of `sycl::gpu_selector`.

Implementation note: the SYCL API might rely on SFINAE or C++20 concepts to resolve some ambiguity in constructors with default parameters.

### 4.6.2 Platform class

The SYCL `platform` class encapsulates a single SYCL platform on which SYCL kernel functions may be executed. A SYCL platform must be associated with a single SYCL backend and may encapsulate a native backend object.

A SYCL `platform` is also associated with one or more SYCL `device`s associated with the same SYCL backend.

All member functions of the `platform` class are synchronous and errors are handled by throwing synchronous SYCL exceptions.

The default constructor of the SYCL `platform` class will construct a platform associated with the SYCL host backend.

The explicit constructor of the SYCL `platform` class which takes a device selector will construct a platform that is associated with the SYCL backend that is associated with the `device` that the device selector would construct, according to Sections 4.6.1.1 and 4.6.4.

The SYCL `platform` class provides the common reference semantics (see Section 4.5.3).

#### 4.6.2.1 Platform interface

A synopsis of the SYCL `platform` class is provided below. The constructors, member functions and static member functions of the SYCL `platform` class are listed in Tables 4.9, 4.10 and 4.11 respectively. The additional common special member functions and common member functions are listed in 4.5.3 in Tables 4.1 and 4.2 respectively.

```
namespace sycl {
class platform {
 public:
  platform();

  template <typename DeviceSelector>
  explicit platform(const DeviceSelector &deviceSelector);

  /* -- common interface members -- */

  backend get_backend() const;

  std::vector<device> get_devices(
    info::device_type = info::device_type::all) const;

  template <info::platform param>
  typename info::param_traits<info::platform, param>::return_type get_info() const;

  template <typename BackendEnum, BackendEnum param>
  typename info::param_traits<BackendEnum, param>::return_type
  get_backend_info() const;

  bool has(aspect asp) const;

  bool has_extension(const std::string &extension) const; // Deprecated

  bool is_host() const;

  static std::vector<platform> get_platforms();
};
}  // namespace sycl
```

| Constructor | Description |
|---|---|
| `platform()` | Constructs a SYCL `platform` instance as a host platform. |
| | Continued on next page |

Table 4.9: Constructors of the SYCL `platform` class.

| Constructor | Description |
|---|---|
| `template <typename DeviceSelector>`<br>`explicit device(const DeviceSelector &)` | Constructs a SYCL `platform` instance using the device selector parameter. One of the SYCL `device`s that is associated with the constructed SYCL `platform` instance must be the SYCL `device` that is produced from the provided device ranking function. |
| | End of table |

Table 4.9: Constructors of the SYCL `platform` class.

| Member function | Description |
|---|---|
| `backend get_backend()const` | Returns the a `backend` identifying the SYCL backend associated with this `platform`. |
| `template <info::platform param>`<br>`typename info::param_traits<info::platform, param>::return_type`<br>`get_info()const` | Queries this SYCL `platform` for information requested by the template parameter `param`. Specializations of `info::param_traits` must be defined in accordance with the info parameters in Table 4.19 to facilitate returning the type associated with the `param` parameter. |
| `template <typename BackendEnum, BackendEnum param>`<br>`typename info::param_traits<BackendEnum, param>::return_type`<br>`get_backend_info()const` | Queries this SYCL `platform` for SYCL backend-specific information requested by the template parameter `param`. `BackendEnum` can be any enum class type specified by the SYCL backend specification of a supported SYCL backend named according to the convention `info::<backend_name>::platform` and `param` must be a valid enumeration of that enum class. Specializations of `info::param_traits` must be defined for `BackendEnum` in accordance with the SYCL backend specification. Must throw an `exception` with the `errc::invalid_object_error` if the SYCL backend that corresponds with `BackendEnum` is different from the SYCL backend that is associated with this `platform`. |
| `bool has(aspect asp)const` | Returns true if all of the SYCL `device`s associated with this SYCL `platform` have the given aspect. |
| `bool has_extension(const std::string & extension) const` | Deprecated, use `has()` instead.<br>Returns true if this SYCL `platform` supports the extension queried by the `extension` parameter. A SYCL `platform` can only support an extension if all associated SYCL `device`s support that extension. |
| | Continued on next page |

Table 4.10: Member functions of the SYCL `platform` class.

| Member function | Description |
|---|---|
| `bool is_host()const` | Returns true if the backend associated with this SYCL `platform` is a SYCL host backend. |
| `std::vector<device> get_devices( info::device_type = info::device_type::all)const` | Returns a `std::vector` containing all SYCL `device`s associated with this SYCL `platform`. The returned `std::vector` must contain only a single SYCL `device` that is a host device if this SYCL `platform` is a host platform. Must return an empty `std::vector` instance if there are no devices that match the given `info::device_type`. |
| | End of table |

Table 4.10: Member functions of the SYCL `platform` class.

| Static member function | Description |
|---|---|
| `static std::vector<platform> get_platforms()` | Returns a `std::vector` containing all SYCL `platform`s from all SYCL backends available in the system. The `std::vector` returned must contain at least one SYCL `platform` that is from a SYCL host backend. |
| | End of table |

Table 4.11: Static member functions of the SYCL `platform` class.

#### 4.6.2.2 Platform information descriptors

A platform can be queried for information using the `get_info` member function of the `platform` class, specifying one of the info parameters enumerated in `info::platform`. Every platform (including a host platform) must produce a valid value for each info parameter. The possible values for each info parameter and any restriction are defined in the specification of the SYCL backend associated with the platform. All info parameters in `info::platform` are specified in Table 4.12 and the synopsis for `info::platform` is described in appendix A.1.

| Platform descriptors | Return type | Description |
|---|---|---|
| `info::platform::version` | `std::string` | Returns the software driver version of the device. |
| `info::platform::name` | `std::string` | Returns the name of the platform. |
| `info::platform::vendor` | `std::string` | Returns the name of the vendor providing the platform. |
| `info::platform::extensions` | `std::vector<std::string>` | Deprecated, use `device::get_info()` with `info::device::aspects` instead.<br>Returns the extensions supported by the platform. |
| | | End of table |

Table 4.12: Platform information descriptors.

### 4.6.3 Context class

The context class represents a SYCL context. A context represents the runtime data structures and state required by a SYCL backend API to interact with a group of devices associated with a platform.

The SYCL `context` class provides the common reference semantics (see Section 4.5.3).

#### 4.6.3.1 Context interface

The constructors and member functions of the SYCL `context` class are listed in Tables 4.13 and 4.14, respectively. The additional common special member functions and common member functions are listed in 4.5.3 in Tables 4.1 and 4.2, respectively.

All member functions of the context class are synchronous and errors are handled by throwing synchronous SYCL exceptions.

All constructors of the SYCL context class will construct an instance associated with a particular SYCL backend, determined by the constructor parameters or, in the case of the default constructor, the SYCL `device` produced by the `default_selector_v`.

A SYCL `context` can optionally be constructed with an `async_handler` parameter. In this case the `async_handler` is used to report asynchronous SYCL exceptions, as described in 4.15.

Information about a SYCL context may be queried through the `get_info()` member function.

```
namespace sycl {
class context {
 public:
  explicit context(const property_list &propList = {});

  explicit context(async_handler asyncHandler,
                   const property_list &propList = {});

  explicit context(const device &dev, const property_list &propList = {});

  explicit context(const device &dev, async_handler asyncHandler,
                   const property_list &propList = {});

  explicit context(const std::vector<device> &deviceList,
                   const property_list &propList = {});

  explicit context(const std::vector<device> &deviceList,
                   async_handler asyncHandler,
                   const property_list &propList = {});

  /* -- property interface members -- */

  /* -- common interface members -- */

  backend get_backend() const;

  bool is_host() const;

  platform get_platform() const;
```

```

  std::vector<device> get_devices() const;

  template <info::context param>
  typename info::param_traits<info::context, param>::return_type get_info() const;

  template <typename BackendEnum, BackendEnum param>
  typename info::param_traits<BackendEnum, param>::return_type
  get_backend_info() const;
};
}  // namespace sycl
```

| Constructor | Description |
|---|---|
| `explicit context(async_handler asyncHandler = {})` | Constructs a SYCL context instance using an instance of default_selector_v to select the associated SYCL platform and device (s). The devices that are associated with the constructed context are implementation defined but must contain the device chosen by the device selector. The constructed SYCL context will use the asyncHandler parameter to handle exceptions. |
| `explicit context(const device &dev, async_handler asyncHandler = {})` | Constructs a SYCL context instance using the dev parameter as the associated SYCL device and the SYCL platform associated with the dev parameter as the associated SYCL platform. The constructed SYCL context will use the asyncHandler parameter to handle exceptions. |
| `explicit context(const std::vector<device> & deviceList, async_handler asyncHandler = {})` | Constructs a SYCL context instance using the SYCL device(s) in the deviceList parameter as the associated SYCL device (s) and the SYCL platform associated with each SYCL device in the deviceList parameter as the associated SYCL platform . This requires that all SYCL devices in the deviceList parameter have the same associated SYCL platform. The constructed SYCL context will use the asyncHandler parameter to handle exceptions. |
| | End of table |

Table 4.13: Constructors of the SYCL context class.

| Member function | Description |
|---|---|
| `backend get_backend()const` | Returns the a backend identifying the SYCL backend associated with this context. |
| `bool is_host()const` | Returns true if the backend associated with this SYCL context is a SYCL host backend. |
| | Continued on next page |

Table 4.14: Member functions of the context class.

| Member function | Description |
|---|---|
| `template <info::context param> typename info::param_traits<info::context, param>::return_type get_info()const` | Queries this SYCL `context` for information requested by the template parameter `param` using the `param_traits` class template to facilitate returning the appropriate type associated with the `param` parameter. |
| `template <typename BackendEnum, BackendEnum param> typename info::param_traits<BackendEnum, param>::return_type get_backend_info()const` | Queries this SYCL `context` for SYCL backend-specific information requested by the template parameter `param`. `BackendEnum` can be any enum class type specified by the SYCL backend specification of a supported SYCL backend named according to the convention `info::<backend_name>::context` and `param` must be a valid enumeration of that enum class. Specializations of `info::param_traits` must be defined for `BackendEnum` in accordance with the SYCL backend specification. Must throw an `exception` with the `errc::invalid_object_error` error code if the SYCL backend that corresponds with `BackendEnum` is different from the SYCL backend that is associated with this `context`. |
| `platform get_platform()const` | Returns the SYCL `platform` that is associated with this SYCL `context`. The value returned must be equal to that returned by `get_info<info::context::platform>()`. |
| `std::vector<device> get_devices()const` | Returns a `std::vector` containing all SYCL `device`s that are associated with this SYCL `context`. The value returned must be equal to that returned by `get_info<info::context::devices>()`. |
| | End of table |

Table 4.14: Member functions of the `context` class.

#### 4.6.3.2 Context information descriptors

A context can be queried for information using the `get_info` member function of the `context` class, specifying one of the info parameters enumerated in `info::context`. Every context (including a host context) must produce a valid value for each info parameter. The possible values for each info parameter and any restriction are defined in the specification of the SYCL backend associated with the context. All info parameters in `info::context` are specified in Table 4.15 and the synopsis for `info::context` is described in appendix A.2.

| Context Descriptors | Return type | Description |
|---|---|---|
| `info::context::platform` | `platform` | Returns the platform associated with the context. |
| `info::context::devices` | `std::vector<device>` | Returns all of the devices associated with the context. |
| | | Continued on next page |

Table 4.15: Context information descriptors.

| Context Descriptors | Return type | Description |
|---|---|---|
| info::context:: atomic_memory_order_capabilities | std::vector<memory_order> | Returns the set of memory orderings supported by atomic operations on all devices in the context, which is guaranteed to include relaxed.<br>The memory ordering of the context determines the behavior of atomic operations applied to any memory that can be concurrently accessed by multiple devices in the context. |
| info::context:: atomic_fence_order_capabilities | std::vector<memory_order> | Returns the set of memory orderings supported by atomic_fence on all devices in the context, which is guaranteed to include relaxed.<br>The memory ordering of the context determines the behavior of fence operations applied to any memory that can be concurrently accessed by multiple devices in the context. |
| info::context:: atomic_memory_scope_capabilities | std::vector<memory_scope> | Returns the set of memory scopes supported by atomic operations on all devices in the context, which is guaranteed to include work_group. |
| info::context:: atomic_fence_scope_capabilities | std::vector<memory_scope> | Returns the set of memory orderings supported by atomic_fence on all devices in the context, which is guaranteed to include work_group. |
| | | End of table |

Table 4.15: Context information descriptors.

#### 4.6.3.3 Context properties

The `property_list` constructor parameters are present for extensibility.

### 4.6.4 Device class

The SYCL `device` class encapsulates a single SYCL device on which kernels can be executed. A SYCL device object can map to a native backend object.

All member functions of the `device` class are synchronous and errors are handled by throwing synchronous SYCL exceptions.

The default constructor of the SYCL `device` class will construct a host device from the host SYCL backend.

The explicit constructor of the SYCL `device` class which takes a device selector will construct a device selected by the device selector according to Section 4.6.1.1.

A SYCL `device` can be partitioned into multiple SYCL devices, by calling the `create_sub_devices()` member function template. The resulting SYCL `device`s are considered sub devices, and it is valid to partition these sub devices further. The range of support for this feature is SYCL backend and device specific and can be queried for through `get_info()`.

The SYCL `device` class provides the common reference semantics (see Section 4.5.3).

#### 4.6.4.1 Device interface

A synopsis of the SYCL `device` class is provided below. The constructors, member functions and static member functions of the SYCL `device` class are listed in Tables 4.16, 4.17 and 4.18 respectively. The additional common special member functions and common member functions are listed in 4.5.3 in Tables 4.1 and 4.2, respectively.

```
namespace sycl {

class device {
 public:
  device();

  template <typename DeviceSelector>
  explicit device(const DeviceSelector &deviceSelector);

  /* -- common interface members -- */

  backend get_backend() const;

  bool is_host() const;

  bool is_cpu() const;

  bool is_gpu() const;

  bool is_accelerator() const;

  platform get_platform() const;

  template <info::device param>
  typename info::param_traits<info::device, param>::return_type
  get_info() const;

  template <typename BackendEnum, BackendEnum param>
  typename info::param_traits<BackendEnum, param>::return_type
  get_backend_info() const;

  bool has(aspect asp) const;

  bool has_extension(const std::string &extension) const; // Deprecated

  // Available only when prop == info::partition_property::partition_equally
  template <info::partition_property prop>
  std::vector<device> create_sub_devices(size_t nbSubDev) const;

  // Available only when prop == info::partition_property::partition_by_counts
  template <info::partition_property prop>
  std::vector<device> create_sub_devices(const std::vector<size_t> &counts) const;

  // Available only when prop == info::partition_property::partition_by_affinity_domain
```

```
  template <info::partition_property prop>
  std::vector<device> create_sub_devices(info::affinity_domain affinityDomain) const;

  static std::vector<device> get_devices(
      info::device_type deviceType = info::device_type::all);
};
} // namespace sycl
```

| Constructor | Description |
|---|---|
| `device()` | Constructs a SYCL device instance as a host device. |
| `template <typename DeviceSelector> explicit device(const DeviceSelector & deviceSelector)` | Constructs a SYCL device instance using the device selected by the device selector provided. |
| | End of table |

Table 4.16: Constructors of the SYCL device class.

| Member function | Description |
|---|---|
| `backend get_backend()const` | Returns the a `backend` identifying the SYCL backend associated with this `device`. |
| `platform get_platform()const` | Returns the associated SYCL `platform`. The value returned must be equal to that returned by `get_info<info::device::platform>()`. |
| `bool is_host()const` | Returns the same value as `has(aspect::host)`. See Table 4.20. |
| `bool is_cpu()const` | Returns the same value as `has(aspect::cpu)`. See Table 4.20. |
| `bool is_gpu()const` | Returns the same value as `has(aspect::gpu)`. See Table 4.20. |
| `bool is_accelerator()const` | Returns the same value as `has(aspect::accelerator)`. See Table 4.20. |
| `template <info::device param> typename info::param_traits<info::device, param>::return_type get_info()const` | Queries this SYCL `device` for information requested by the template parameter `param`. Specializations of `info::param_traits` must be defined in accordance with the info parameters in Table 4.19 to facilitate returning the type associated with the `param` parameter. |
| | Continued on next page |

Table 4.17: Member functions of the SYCL device class.

| Member function | Description |
|---|---|
| `template <typename BackendEnum, BackendEnum param> typename info::param_traits<BackendEnum, param>::return_type get_backend_info()const` | Queries this SYCL `device` for SYCL backend-specific information requested by the template parameter `param`. `BackendEnum` can be any enum class type specified by the SYCL backend specification of a supported SYCL backend named according to the convention `info::<backend_name>::device` and `param` must be a valid enumeration of that enum class. Specializations of `info::param_traits` must be defined for `BackendEnum` in accordance with the SYCL backend specification. Must throw an `exception` with the `errc::invalid_object_error` error code if the SYCL backend that corresponds with `BackendEnum` is different from the SYCL backend that is associated with this `device`. |
| `bool has(aspect asp)const` | Returns true if this SYCL `device` has the given aspect. SYCL applications can use this member function to determine which optional features this device supports (if any). |
| `bool has_extension (const std::string &extension) const` | Deprecated, use `has()` instead.<br>Returns true if this SYCL `device` supports the extension queried by the `extension` parameter. |
| `template <info::partition_property prop> std::vector<device> create_sub_devices( size_t nbSubDev)const` | Available only when `prop` is `info::partition_property::partition_equally`. Returns a `std::vector` of sub devices partitioned from this SYCL `device` equally based on the `nbSubDev` parameter. If this SYCL `device` does not support `info::partition_property::partition_equally` an `exception` with the `errc::feature_not_supported` error code must be thrown. |
| `template <info::partition_property prop> std::vector<device> create_sub_devices( const std::vector<size_t> &counts)const` | Available only when `prop` is `info::partition_property::partition_by_count`. Returns a `std::vector` of sub devices partitioned from this SYCL `device` by count sizes based on the `counts` parameter. If the SYCL `device` does not support `info::partition_property::partition_by_count` an `exception` with the `errc::feature_not_supported` error code must be thrown. |
| | Continued on next page |

Table 4.17: Member functions of the SYCL `device` class.

| Member function | Description |
|---|---|
| `template <info::partition_property prop> std::vector<device> create_sub_devices( info::affinity_domain affinityDomain)const` | Available only when prop is `info::partition_property::partition_by_affinity_domain`. Returns a `std::vector` of sub devices partitioned from this SYCL `device` by affinity domain based on the `affinityDomain` parameter. Partitions the device into sub devices based upon the affinity domain. If the SYCL `device` does not support `info::partition_property::partition_by_affinity_domain` or the SYCL `device` does not support `info::affinity_domain` provided an `exception` with the `errc::feature_not_supported` error code must be thrown. |
| | End of table |

Table 4.17: Member functions of the SYCL `device` class.

| Static member function | Description |
|---|---|
| `static std::vector<device> get_devices( info::device_type deviceType = info::device_type::all)` | Returns a `std::vector` containing all SYCL `device`s from all SYCL backends available in the system of the device type specified by the parameter `deviceType`. Note that when the `device_type` is `info::device_type::all` or `info::device_type::host`, the `std::vector` returned must contain at least one host device from the SYCL host backend. |
| | End of table |

Table 4.18: Static member functions of the SYCL `device` class.

#### 4.6.4.2 Device information descriptors

A device can be queried for information using the `get_info` member function of the `device` class, specifying one of the info parameters enumerated in `info::device`. Every device (including a host device) must produce a valid value for each info parameter. The possible values for each info parameter and any restriction are defined in the specification of the SYCL backend associated with the device. All info parameters in `info::device` are specified in Table 4.19 and the synopsis for `info::device` is described in appendix A.3.

| Device descriptors | Return type | Description |
|---|---|---|
| `info::device::device_type` | `info::device_type` | Returns the device type associated with the device. May not return `info::device_type::all`. |
| `info::device::vendor_id` | `uint32_t` | Returns a unique vendor device identifier. |
| | | Continued on next page |

Table 4.19: Device information descriptors.

| Device descriptors | Return type | Description |
|---|---|---|
| info::device::max_compute_units | uint32_t | Returns the number of parallel compute units available to the device. The minimum value is 1. |
| info::device::max_work_item_dimensions | uint32_t | Returns the maximum dimensions that specify the global and local work-item IDs used by the data parallel execution model. The minimum value is 3 if this SYCL device is not of device type info::device_type::custom. |
| info::device::max_work_item_sizes | id<3> | Returns the maximum number of work-items that are permitted in each dimension of the work-group of the nd_range. The minimum value is (1, 1, 1) for devices that are not of device type info::device_type::custom. |
| info::device::max_work_group_size | size_t | Returns the maximum number of work-items that are permitted in a work-group executing a kernel on a single compute unit. The minimum value is 1. |
| info::device::max_num_sub_groups | uint32_t | Returns the maximum number of sub-groups in a work-group for any kernel executed on the device. The minimum value is 1. |
| info::device::sub_group_independent_forward_progress | bool | Returns true if the device supports independent forward progress of sub-groups with respect to other sub-groups in the same work-group. |
| info::device::sub_group_sizes | std::vector<size_t> | Returns a std::vector of size_t containing the set of sub-group sizes supported by the device. |
| info::device::preferred_vector_width_char info::device::preferred_vector_width_short info::device::preferred_vector_width_int info::device::preferred_vector_width_long info::device::preferred_vector_width_float info::device::preferred_vector_width_double info::device::preferred_vector_width_half | uint32_t | Returns the preferred native vector width size for built-in scalar types that can be put into vectors. The vector width is defined as the number of scalar elements that can be stored in the vector. Must return 0 for info::device::preferred_vector_width_double if the device does not have aspect::fp64 and must return 0 for info::device::preferred_vector_width_half if the device does not have aspect::fp16. |

Continued on next page

Table 4.19: Device information descriptors.

| Device descriptors | Return type | Description |
|---|---|---|
| info::device::native_vector_width_char<br>info::device::native_vector_width_short<br>info::device::native_vector_width_int<br>info::device::native_vector_width_long<br>info::device::native_vector_width_float<br>info::device::native_vector_width_double<br>info::device::native_vector_width_half | uint32_t | Returns the native ISA vector width. The vector width is defined as the number of scalar elements that can be stored in the vector. Must return 0 for info::device::preferred_vector_width_double if the device does not have aspect::fp64 and must return 0 for info::device::preferred_vector_width_half if the device does not have aspect::fp16. |
| info::device::max_clock_frequency | uint32_t | Returns the maximum configured clock frequency of this SYCL device in MHz. |
| info::device::address_bits | uint32_t | Returns the default compute device address space size specified as an unsigned integer value in bits. Must return either 32 or 64. |
| info::device::max_mem_alloc_size | uint64_t | Returns the maximum size of memory object allocation in bytes. The minimum value is max (1/4th of info::device::global_mem_size ,128*1024*1024) if this SYCL device is not of device type info::device_type::custom. |
| info::device::image_support | bool | Deprecated.<br>Returns the same value as device::has( aspect::image). |
| info::device::max_read_image_args | uint32_t | Returns the maximum number of simultaneous image objects that can be read from by a kernel. The minimum value is 128 if the SYCL device has aspect::image. |
| info::device::max_write_image_args | uint32_t | Returns the maximum number of simultaneous image objects that can be written to by a kernel. The minimum value is 8 if the SYCL device has aspect::image. |
| info::device::image2d_max_width | size_t | Returns the maximum width of a 2D image or 1D image in pixels. The minimum value is 8192 if the SYCL device has aspect::image. |
| info::device::image2d_max_height | size_t | Returns the maximum height of a 2D image in pixels. The minimum value is 8192 if the SYCL device has aspect::image. |
| info::device::image3d_max_width | size_t | Returns the maximum width of a 3D image in pixels. The minimum value is 2048 if the SYCL device has aspect::image. |
| info::device::image3d_max_height | size_t | Returns the maximum height of a 3D image in pixels. The minimum value is 2048 if the SYCL device has aspect::image. |

Continued on next page

Table 4.19: Device information descriptors.

| Device descriptors | Return type | Description |
|---|---|---|
| info::device::image3d_max_depth | size_t | Returns the maximum depth of a 3D image in pixels. The minimum value is 2048 if the SYCL device has aspect::image. |
| info::device::image_max_buffer_size | size_t | Returns the number of pixels for a 1D image created from a buffer object. The minimum value is 65536 if the SYCL device has aspect::image. Note that this information is intended for OpenCL interoperability only as this feature is not supported in SYCL. |
| info::device::image_max_array_size | size_t | Returns the maximum number of images in a 1D or 2D image array. The minimum value is 2048 if the SYCL device has aspect::image. |
| info::device::max_samplers | uint32_t | Returns the maximum number of samplers that can be used in a kernel. The minimum value is 16 if the SYCL device has aspect::image. |
| info::device::max_parameter_size | size_t | Returns the maximum size in bytes of the arguments that can be passed to a kernel. The minimum value is 1024 if this SYCL device is not of device type info::device_type::custom. For this minimum value, only a maximum of 128 arguments can be passed to a kernel. |
| info::device::mem_base_addr_align | uint32_t | Returns the minimum value in bits of the largest supported SYCL built-in data type if this SYCL device is not of device type info::device_type::custom. |

Continued on next page

Table 4.19: Device information descriptors.

| Device descriptors | Return type | Description |
|---|---|---|
| info::device::half_fp_config | std::vector< info::fp_config > | Returns a std::vector of info::fp_config describing the half precision floating-point capability of this SYCL device. The std::vector may contain zero or more of the following values:<br>• info::fp_config::denorm: denorms are supported.<br>• info::fp_config::inf_nan: INF and quiet NaNs are supported.<br>• info::fp_config::round_to_nearest: round to nearest even rounding mode is supported.<br>• info::fp_config::round_to_zero : round to zero rounding mode is supported.<br>• info::fp_config::round_to_inf: round to positive and negative infinity rounding modes are supported.<br>• info::fp_config::fma: IEEE754-2008 fused multiply add is supported.<br>• info::fp_config::correctly_rounded_divide_sqrt: divide and sqrt are correctly rounded as defined by the IEEE754 specification.<br>• info::fp_config::soft_float: basic floating-point operations (such as addition, subtraction, multiplication) are implemented in software.<br>If half precision is supported by this SYCL device (i.e. the device has aspect::fp16) there is no minimum floating-point capability. If half support is not supported the returned std::vector must be empty. |

Continued on next page

Table 4.19: Device information descriptors.

| Device descriptors | Return type | Description |
| --- | --- | --- |
| info::device::single_fp_config | std::vector< info::fp_config > | Returns a std::vector of info::fp_config describing the single precision floating-point capability of this SYCL device. The std::vector must contain one or more of the following values:<br>• info::fp_config::denorm: denorms are supported.<br>• info::fp_config::inf_nan: INF and quiet NaNs are supported.<br>• info::fp_config::round_to_nearest: round to nearest even rounding mode is supported.<br>• info::fp_config::round_to_zero : round to zero rounding mode is supported.<br>• info::fp_config::round_to_inf: round to positive and negative infinity rounding modes are supported.<br>• info::fp_config::fma: IEEE754-2008 fused multiply add is supported.<br>• info::fp_config::correctly_rounded_divide_sqrt: divide and sqrt are correctly rounded as defined by the IEEE754 specification.<br>• info::fp_config::soft_float: basic floating-point operations (such as addition, subtraction, multiplication) are implemented in software.<br>If this SYCL device is not of type info::device_type::custom then the minimum floating-point capability must be: info::fp_config::round_to_nearest and info::fp_config::inf_nan. |

Continued on next page

Table 4.19: Device information descriptors.

| Device descriptors | Return type | Description |
|---|---|---|
| info::device::double_fp_config | std::vector<info::fp_config> | Returns a std::vector of info::fp_config describing the double precision floating-point capability of this SYCL device. The std::vector may contain zero or more of the following values:<br>• info::fp_config::denorm: denorms are supported.<br>• info::fp_config::inf_nan: INF and NaNs are supported.<br>• info::fp_config::round_to_nearest: round to nearest even rounding mode is supported.<br>• info::fp_config::round_to_zero: round to zero rounding mode is supported.<br>• info::fp_config::round_to_inf: round to positive and negative infinity rounding modes are supported.<br>• info::fp_config::fma: IEEE754-2008 fused multiply-add is supported.<br>• info::fp_config::soft_float: basic floating-point operations (such as addition, subtraction, multiplication) are implemented in software.<br>If double precision is supported by this SYCL device (i.e. the device has aspect::fp64) and this SYCL device is not of type info::device_type::custom then the minimum floating-point capability must be: info::fp_config::fma, info::fp_config::round_to_nearest, info::fp_config::round_to_zero, info::fp_config::round_to_inf, info::fp_config::inf_nan and info::fp_config::denorm. If double support is not supported the returned std::vector must be empty. |
| info::device::global_mem_cache_type | info::global_mem_cache_type | Returns the type of global memory cache supported. |
| info::device::global_mem_cache_line_size | uint32_t | Returns the size of global memory cache line in bytes. |
| info::device::global_mem_cache_size | uint64_t | Returns the size of global memory cache in bytes. |
| info::device::global_mem_size | uint64_t | Returns the size of global device memory in bytes. |

Continued on next page

Table 4.19: Device information descriptors.

| Device descriptors | Return type | Description |
|---|---|---|
| info::device:: max_constant_buffer_size | uint64_t | Returns the maximum size in bytes of a constant buffer allocation. The minimum value is 64 KB if this SYCL device is not of type info ::device_type::custom. |
| info::device:: max_constant_args | uint32_t | Returns the maximum number of constant arguments that can be declared in a kernel. The minimum value is 8 if this SYCL device is not of type info::device_type::custom. |
| info::device::local_mem_type | info:: local_mem_type | Returns the type of local memory supported. This can be info::local_mem_type::local implying dedicated local memory storage such as SRAM, or info::local_mem_type:: global. If this SYCL device is of type info::device_type::custom this can also be info::local_mem_type::none, indicating local memory is not supported. |
| info::device::local_mem_size | uint64_t | Returns the size of local memory arena in bytes. The minimum value is 32 KB if this SYCL device is not of type info:: device_type::custom. |
| info::device:: error_correction_support | bool | Returns true if the device implements error correction for all accesses to compute device memory (global and constant). Returns false if the device does not implement such error correction. |
| info::device:: host_unified_memory | bool | Deprecated, use device::has() with one of the aspect::usm_* aspects instead.<br>Returns true if the device and the host have a unified memory subsystem and returns false otherwise. |
| info::device:: atomic_memory_order_capabilities | std::vector< memory_order> | Returns the set of memory orderings supported by atomic operations on the device, which is guaranteed to include relaxed.<br>If this device is a host device, the set must include all values of the memory_order enum class: relaxed, acquire, release, acq_rel and seq_cst. |
| info::device:: atomic_fence_order_capabilities | std::vector< memory_order> | Returns the set of memory orderings supported by atomic_fence on the device, which is guaranteed to include relaxed.<br>If this device is a host device, the set must include all values of the memory_order enum class: relaxed, acquire, release, acq_rel and seq_cst. |

Continued on next page

Table 4.19: Device information descriptors.

| Device descriptors | Return type | Description |
|---|---|---|
| info::device::atomic_memory_scope_capabilities | std::vector< memory_scope> | Returns the set of memory scopes supported by atomic operations on the device, which is guaranteed to include work_group.<br>If this device is a host device, the set must include all values of the memory_scope enum class: work_item, sub_group, work_group, device and system. |
| info::device::atomic_fence_scope_capabilities | std::vector< memory_scope> | Returns the set of memory scopes supported by atomic_fence on the device, which is guaranteed to include work_group.<br>If this device is a host device, the set must include all values of the memory_scope enum class: work_item, sub_group, work_group, device and system. |
| info::device::profiling_timer_resolution | size_t | Returns the resolution of device timer in nanoseconds. |
| info::device::is_endian_little | bool | Returns true if this SYCL device is a little endian device and returns false otherwise. |
| info::device::is_available | bool | Returns true if the SYCL device is available and returns false if the device is not available. |
| info::device::is_compiler_available | bool | Deprecated.<br>Returns the same value as device::has( aspect::online_compiler). |
| info::device::is_linker_available | bool | Deprecated.<br>Returns the same value as device::has( aspect::online_linker). |
| info::device::execution_capabilities | std::vector <info::execution_capability> | Returns a std::vector of the info::execution_capability describing the supported execution capabilities. Note that this information is intended for OpenCL interoperability only as SYCL only supports info::execution_capability::exec_kernel. |
| info::device::queue_profiling | bool | Deprecated.<br>Returns the same value as device::has( aspect::queue_profiling). |
| info::device::built_in_kernels | std::vector<std::string> | Returns a std::vector of built-in OpenCL kernels supported by this SYCL device. |
| info::device::platform | platform | Returns the SYCL platform associated with this SYCL device. |
| info::device::name | std::string | Returns the device name of this SYCL device. |
| info::device::vendor | std::string | Returns the vendor of this SYCL device. |
| info::device::driver_version | std::string | Returns the OpenCL software driver version as a std::string in the form: major_number.minor_number, if this SYCL device is an OpenCL device. Must return a std::string with the value "1.2" if this SYCL device is a host device. |

Continued on next page

Table 4.19: Device information descriptors.

| Device descriptors | Return type | Description |
|---|---|---|
| info::device::profile | std::string | Returns the OpenCL profile as a std::string , if this SYCL device is an OpenCL device. The value returned can be one of the following strings:<br>• FULL_PROFILE - if the device supports the OpenCL specification (functionality defined as part of the core specification and does not require any extensions to be supported).<br>• EMBEDDED_PROFILE - if the device supports the OpenCL embedded profile.<br>Must return a std::string with the value "FULL PROFILE" if this is a host device. |
| info::device::version | std::string | Returns the SYCL version as a std::string in the form: <major_version>.<minor_version>. If this SYCL device is a host device, the ¡major_version¿.¡minor_version¿ value returned must be "1.2". |
| info::device::backend_version | std::string | Returns a string describing the version of the SYCL backend associated with the device. The possible values are specified in the SYCL backend specification of the SYCL backend associated with the device. |
| info::device::aspects | std::vector<aspect> | Returns a std::vector of aspect values supported by this SYCL device. |
| | | Continued on next page |

Table 4.19: Device information descriptors.

| Device descriptors | Return type | Description |
|---|---|---|
| info::device::extensions | std::vector<std::string> | Deprecated, use info::device::aspects instead.<br>Returns a std::vector of extension names (the extension names do not contain any spaces) supported by this SYCL device. The extension names returned can be vendor supported extension names and one or more of the following Khronos approved extension names:<br>• cl_khr_int64_base_atomics<br>• cl_khr_int64_extended_atomics<br>• cl_khr_3d_image_writes<br>• cl_khr_fp16<br>• cl_khr_gl_sharing<br>• cl_khr_gl_event<br>• cl_khr_d3d10_sharing<br>• cl_khr_dx9_media_sharing<br>• cl_khr_d3d11_sharing<br>• cl_khr_depth_images<br>• cl_khr_gl_depth_images<br>• cl_khr_gl_msaa_sharing<br>• cl_khr_image2d_from_buffer<br>• cl_khr_initialize_memory<br>• cl_khr_context_abort<br>• cl_khr_spir<br>If this SYCL device is an OpenCL device then following approved Khronos extension names must be returned by all device that support OpenCL C 1.2:<br>• cl_khr_global_int32_base_atomics<br>• cl_khr_global_int32_extended_atomics<br>• cl_khr_local_int32_base_atomics<br>• cl_khr_local_int32_extended_atomics<br>• cl_khr_byte_addressable_store<br>• cl_khr_fp64 (for backward compatibility if double precision is supported)<br>Please refer to the OpenCL 1.2 Extension Specification for a detailed description of these extensions. |
| info::device::printf_buffer_size | size_t | Returns the maximum size of the internal buffer that holds the output of printf calls from a kernel. The minimum value is 1 MB if info::device::profile returns true for this SYCL device. |

Continued on next page

Table 4.19: Device information descriptors.

| Device descriptors | Return type | Description |
|---|---|---|
| info::device::preferred_interop_user_sync | bool | Returns true if the preference for this SYCL device is for the user to be responsible for synchronization, when sharing memory objects between OpenCL and other APIs such as DirectX, false if the device/implementation has a performant path for performing synchronization of memory object shared between OpenCL and other APIs such as DirectX. |
| info::device::parent_device | device | Returns the parent SYCL device to which this sub-device is a child if this is a sub-device. Must throw an exception with the errc::invalid_object_error error code if this SYCL device is not a sub device. |
| info::device::partition_max_sub_devices | uint32_t | Returns the maximum number of sub-devices that can be created when this SYCL device is partitioned. The value returned cannot exceed the value returned by info::device::device_max_compute_units. |
| info::device::partition_properties | std::vector <info::partition_property> | Returns the partition properties supported by this SYCL device; a vector of info::partition_property. If this SYCL device cannot be partitioned into at least two sub devices then the returned vector must be empty. |
| info::device::partition_affinity_domains | std::vector <info::parition_affinity_domain> | Returns a std::vector of the partition affinity domains supported by this SYCL device when partitioning with info::partition_property::parition_by_affinity_domain. |
| info::device::partition_type_property | info::partition_property | Returns the partition property of this SYCL device. If this SYCL device is not a sub device then the the return value must be info::partition_property::no_partition, otherwise it must be one of the following values:<br>• info::partition_property::partition_equally<br>• info::partition_property::partition_by_counts<br>• info::partition_property::partition_by_affinity_domain |

Continued on next page

Table 4.19: Device information descriptors.

| Device descriptors | Return type | Description |
| --- | --- | --- |
| `info::device::partition_type_affinity_domain` | `info::partition_affinity_domain` | Returns the partition affinity domain of this SYCL `device`. If this SYCL `device` is not a sub device or the sub device was not partitioned with `info::partition_type::partition_by_affinity_domain` then the the return value must be `info::partition_affinity_domain::not_applicable`, otherwise it must be one of the following values:<br>• info::partition_affinity_domain::numa<br>• info::partition_affinity_domain::L4_cache<br>• info::partition_affinity_domain::L3_cache<br>• info::partition_affinity_domain::L2_cache<br>• info::partition_affinity_domain::L1_cache<br>• info::partition_affinity_domain::next_partitionable |
| `info::device::reference_count` | `uint32_t` | Returns the device reference count. If the device is not a sub-device the value returned must be 1. |
| | | End of table |

Table 4.19: Device information descriptors.

#### 4.6.4.3 Device aspects

Every SYCL device has an associated set of "aspects" which identify characteristics of the `device`. Aspects are defined via the `enum class aspect` enumeration:

```
namespace sycl {

enum class aspect {
  host,
  cpu,
  gpu,
  accelerator,
  custom,
  fp16,
  fp64,
  int64_base_atomics,
  int64_extended_atomics,
  image,
  online_compiler,
  online_linker,
  queue_profiling,
  usm_device_allocations,
  usm_host_allocations,
```

```
  usm_shared_allocations,
  usm_restricted_shared_allocations,
  usm_system_allocator
};

} // namespace sycl
```

Table 4.20 lists the aspects that are defined in the core SYCL specification. However, a SYCL backend or extension may provide additional aspects. If so, the SYCL backend specification document or the extension document describes them. If a SYCL backend defined by the Khronos SYCL group provides aspects, their enumerated values are defined in the backend's namespace. For example, an `aspect` specific to the OpenCL backend could be defined like this:

```
namespace sycl {
namespace opencl {
namespace aspect {

static constexpr auto bar = static_cast<sycl::aspect>(-1);

} // namespace aspect
} // namespace opencl
} // namespace sycl
```

Aspects provided by an extension or a vendor's SYCL backend are defined as described in Chapter 6.

SYCL applications can query the aspects for a `device` via `device::has()` in order to determine whether the `device` supports any optional features. Table 4.20 tells which optional features are enabled by aspects in the core SYCL specification, but backends and extensions may provide optional features also. If so, the SYCL backend specification document or the extension document describes which features are enabled by each `aspect`.

A SYCL application can also use the `is_aspect_active<aspect>::value` trait to test whether an aspect is "active" at compile time. An aspect is active if the compilation environment supports any device which has that aspect. For example, if the implementation supports no devices with `aspect::custom`, the trait `is_aspect_active<aspect::custom>::value` will be false. The set of active aspects could also be affected by command line options passed to the compiler. For example, if an implementation provides a command line option that disables `aspect::accelerator` devices, that trait will be false when the option is passed to the compiler.

*[Note: Like any type trait, the value of `is_aspect_active<aspect>::value` has a uniform value across all parts of a SYCL application. If an implementation uses SMCP, all compiler passes define a particular aspect's `is_aspect_active` type trait with the same value, regardless of whether that compiler pass's device supports the aspect. Thus, `is_aspect_active` cannot be used to determine whether any particular device supports an aspect. Instead, applications must use `device::has()` or `platform::has()` for this. – end note]*

The trait `sycl::is_aspect_active<aspect>::value` must be defined as either true or false for all the core SYCL aspects listed in Table 4.20, all aspects from Khronos ratified extensions, and all of a vendor's own extension aspects.

| Aspect | Description |
|---|---|
| `aspect::host` | A device that runs on the host CPU and exposes the host SYCL backend. Devices with this `aspect` have device type `info::device_type::host`. |
| `aspect::cpu` | A device that runs on a CPU, but doesn't use the host SYCL backend. Devices with this `aspect` have device type `info::device_type::cpu`. |
| `aspect::gpu` | A device that can also be used to accelerate a 3D graphics API. Devices with this `aspect` have device type `info::device_type::gpu`. |
| `aspect::accelerator` | A dedicated accelerator device, usually using a peripheral interconnect for communication. Devices with this `aspect` have device type `info::device_type::accelerator`. |
| `aspect::custom` | A dedicated accelerator that can use the SYCL API, but programmable kernels cannot be dispatched to the device, only fixed functionality is available. See Section 3.8.7. Devices with this `aspect` have device type `info::device_type::custom`. |
| `aspect::fp16` | Indicates that the device supports half precision floating point operations. |
| `aspect::fp64` | Indicates that the device supports 64-bit precision floating point operations. |
| `aspect::int64_base_atomics` | Indicates that the device supports the following atomic operations on 64-bit values: `atomic::load`, `atomic::store`, `atomic::fetch_add`, `atomic::fetch_sub` , `atomic::exchange`, and `atomic::compare_exchange_strong`. |
| `aspect::int64_extended_atomics` | Indicates that the device supports the following atomic operations on 64-bit values: `atomic::fetch_min`, `atomic::fetch_max`, `atomic::fetch_and`, `atomic::fetch_or`, and `atomic::fetch_xor`. |
| `aspect::image` | Indicates that the device supports images (Section 4.7.3). Devices of type `info::device_type::host` always have this support. |
| `aspect::online_compiler` | Indicates that the device supports online compilation of device code. Devices that have this aspect support the `build()` and `compile()` functions defined in Section 4.13.6. |
| | Continued on next page |

Table 4.20: Device aspects defined by the core SYCL specification.

| Aspect | Description |
|---|---|
| aspect::online_linker | Indicates that the device supports online linking of device code. Devices that have this aspect support the link() functions defined in Section 4.13.6. All devices that have this aspect also have aspect::online_compiler. |
| aspect::queue_profiling | Indicates that the device supports queue profiling via property::queue::enable_profiling. |
| aspect::usm_device_allocations | Indicates that the device supports explicit USM allocations as described in Section 4.8. |
| aspect::usm_host_allocations | Indicates that the device can access USM memory allocated via usm::alloc::host. (See Section 4.8.) |
| aspect::usm_shared_allocations | Indicates that the device supports USM memory allocated via usm::alloc::shared as restricted USM, concurrent USM, or both. (See Section 4.8.) |
| aspect::usm_restricted_shared_allocations | Indicates that the device supports USM memory allocated via usm::alloc::shared as restricted USM. Any device with this aspect will also have aspect::usm_shared_allocations. (See Section 4.8.) |
| aspect::usm_system_allocator | Indicates that the system allocator may be used instead of SYCL USM allocation mechanisms for usm::alloc::shared allocations on this device. (See Section 4.8.) |
| | End of table |

Table 4.20: Device aspects defined by the core SYCL specification.

### 4.6.5 Queue class

The SYCL queue class encapsulates a single SYCL queue which schedules kernels on a SYCL device. The SYCL queue can encapsulate to one or multiple native backend objects.

A SYCL queue can be used to submit command groups to be executed by the SYCL runtime using the submit member function.

All member functions of the queue class are synchronous and errors are handled by throwing synchronous SYCL exceptions. The submit member function schedules command groups asynchronously, so any errors in the submission of a command group are handled by throwing synchronous SYCL exceptions. Any exceptions from the command group after it has been submitted are handled by passing asynchronous errors at specific times to an async_handler, as described in 4.15.

A SYCL queue can wait for all command groups that it has submitted by calling wait or wait_and_throw.

The default constructor of the SYCL queue class will construct a queue based on the SYCL device returned from the default_selector_v (see Section 4.6.1.1).

All other constructors construct a host or device queue, determined by the parameters provided. All constructors will implicitly construct a SYCL `platform`, `device` and `context` in order to facilitate the construction of the queue.

Each constructor takes as the last parameter an optional SYCL `property_list` to provide properties to the SYCL `queue`.

The SYCL `queue` class provides the common reference semantics (see Section 4.5.3).

#### 4.6.5.1 Queue interface

A synopsis of the SYCL `queue` class is provided below. The constructors and member functions of the SYCL `queue` class are listed in Tables 4.21 and 4.22 respectively. The additional common special member functions and common member functions are listed in 4.5.3 in Tables 4.1 and 4.2, respectively.

```
namespace sycl {
class queue {
 public:
  explicit queue(const property_list &propList = {});

  explicit queue(const async_handler &asyncHandler,
                 const property_list &propList = {});

  template <typename DeviceSelector>
  explicit queue(const DeviceSelector &deviceSelector,
                 const property_list &propList = {});

  template <typename DeviceSelector>
  explicit queue(const DeviceSelector &deviceSelector,
                 const async_handler &asyncHandler,
                 const property_list &propList = {});

  explicit queue(const device &syclDevice, const property_list &propList = {});

  explicit queue(const device &syclDevice, const async_handler &asyncHandler,
                 const property_list &propList = {});

  template <typename DeviceSelector>
  explicit queue(const context &syclContext,
                 const DeviceSelector &deviceSelector,
                 const property_list &propList = {});

  template <typename DeviceSelector>
  explicit queue(const context &syclContext,
                 const DeviceSelector &deviceSelector,
                 const async_handler &asyncHandler,
                 const property_list &propList = {});

  explicit queue(const context &syclContext, const device &syclDevice,
                 const property_list &propList = {});

  explicit queue(const context &syclContext, const device &syclDevice,
                 const async_handler &asyncHandler,
                 const property_list &propList = {});

```

```
explicit queue(cl_command_queue clQueue, const context& syclContext,
               const async_handler &asyncHandler = {});

/* -- common interface members -- */

/* -- property interface members -- */

backend get_backend() const;

context get_context() const;

device get_device() const;

bool is_host() const;

bool is_in_order() const;

template <info::queue param>
typename info::param_traits<info::queue, param>::return_type get_info() const;

template <typename BackendEnum, BackendEnum param>
typename info::param_traits<BackendEnum, param>::return_type
get_backend_info() const;

template <typename T>
event submit(T cgf);

template <typename T>
event submit(T cgf, const queue &secondaryQueue);

void wait();

void wait_and_throw();

void throw_asynchronous();

/* -- convenience shortcuts -- */

template <typename KernelName, typename KernelType>
event single_task(const KernelType &KernelFunc);

template <typename KernelName, typename KernelType>
event single_task(event DepEvent, const KernelType &KernelFunc);

template <typename KernelName, typename KernelType>
event single_task(const std::vector<event> &DepEvents,
                  const KernelType &KernelFunc);

template <typename KernelName, typename KernelType, int Dims>
event parallel_for(range<Dims> NumWorkItems, const KernelType &KernelFunc);

template <typename KernelName, typename KernelType, int Dims>
event parallel_for(range<Dims> NumWorkItems, event DepEvent,
                   const KernelType &KernelFunc);
```

```
  template <typename KernelName, typename KernelType, int Dims>
  event parallel_for(range<Dims> NumWorkItems,
                     const std::vector<event> &DepEvents,
                     const KernelType &KernelFunc);

  template <typename KernelName, typename KernelType, int Dims>
  event parallel_for(range<Dims> NumWorkItems, id<Dims> WorkItemOffset,
                     const KernelType &KernelFunc);

  template <typename KernelName, typename KernelType, int Dims>
  event parallel_for(range<Dims> NumWorkItems, id<Dims> WorkItemOffset,
                     event DepEvent, const KernelType &KernelFunc);

  template <typename KernelName, typename KernelType, int Dims>
  event parallel_for(range<Dims> NumWorkItems, id<Dims> WorkItemOffset,
                     const std::vector<event> &DepEvents,
                     const KernelType &KernelFunc);

  template <typename KernelName, typename KernelType, int Dims>
  event parallel_for(nd_range<Dims> ExecutionRange, const KernelType &KernelFunc);

  template <typename KernelName, typename KernelType, int Dims>
  event parallel_for(nd_range<Dims> ExecutionRange, event DepEvent,
                     const KernelType &KernelFunc);

  template <typename KernelName, typename KernelType, int Dims>
  event parallel_for(nd_range<Dims> ExecutionRange,
                     const std::vector<event> &DepEvents,
                     const KernelType &KernelFunc);
};
}  // namespace sycl
```

| Constructor | Description |
|---|---|
| `explicit queue(const property_list &propList = {})` | Constructs a SYCL `queue` instance using the device constructed from the `default_selector_v`. Zero or more properties can be provided to the constructed SYCL `queue` via an instance of `property_list`. |
| `explicit queue(const async_handler &asyncHandler, const property_list &propList = {})` | Constructs a SYCL `queue` instance with an `async_handler` using the device constructed from the `default_selector_v`. Zero or more properties can be provided to the constructed SYCL `queue` via an instance of `property_list`. |
| `template <typename DeviceSelector> explicit queue(const DeviceSelector & deviceSelector, const property_list &propList = {})` | Constructs a SYCL `queue` instance using the device returned by the device selector provided. Zero or more properties can be provided to the constructed SYCL `queue` via an instance of `property_list`. |

Continued on next page

Table 4.21: Constructors of the queue class.

| Constructor | Description |
|---|---|
| `template <typename DeviceSelector>`<br>`explicit queue(const DeviceSelector &deviceSelector,`<br>`const async_handler &asyncHandler,`<br>`const property_list &propList = {})` | Constructs a SYCL `queue` instance with an `async_handler` using the device returned by the device selector provided. Zero or more properties can be provided to the constructed SYCL `queue` via an instance of `property_list`. |
| `explicit queue(const device &syclDevice,`<br>`const property_list &propList = {})` | Constructs a SYCL `queue` instance using the `syclDevice` provided. Zero or more properties can be provided to the constructed SYCL `queue` via an instance of `property_list`. |
| `explicit queue(const device &syclDevice,`<br>`const async_handler &asyncHandler,`<br>`const property_list &propList = {})` | Constructs a SYCL `queue` instance with an `async_handler` using the `syclDevice` provided. Zero or more properties can be provided to the constructed SYCL `queue` via an instance of `property_list`. |
| `template <typename DeviceSelector>`<br>`explicit queue(const context &syclContext,`<br>`const DeviceSelector &deviceSelector,`<br>`const property_list &propList = {})` | Constructs a SYCL `queue` instance that is associated with the `syclContext` provided, using the device returned by the device selector provided. Must throw an `exception` with the `errc::invalid_object_error` error code if `syclContext` does not encapsulate the SYCL `device` returned by `deviceSelector`. Zero or more properties can be provided to the constructed SYCL `queue` via an instance of `property_list`. |
| `template <typename DeviceSelector>`<br>`explicit queue(const context &syclContext,`<br>`const DeviceSelector &deviceSelector,`<br>`const async_handler &asyncHandler,`<br>`const property_list &propList = {})` | Constructs a SYCL `queue` instance with an `async_handler` that is associated with the `syclContext` provided, using the device returned by the device selector provided. Must throw an `exception` with the `errc::invalid_object_error` error code if `syclContext` does not encapsulate the SYCL `device` returned by `deviceSelector` . Zero or more properties can be provided to the constructed SYCL `queue` via an instance of `property_list`. |
| `explicit queue(const context &syclContext,`<br>`const device &syclDevice,`<br>`const property_list &propList = {})` | Constructs a SYCL `queue` instance using the `syclDevice` provided, and associated with the `syclContext` provided. Must throw an `exception` with the `errc ::invalid_object_error` error code if `syclContext` does not encapsulate the SYCL `device` `syclDevice`. Zero or more properties can be provided to the constructed SYCL `queue` via an instance of `property_list`. |

Continued on next page

Table 4.21: Constructors of the `queue` class.

| Constructor | Description |
|---|---|
| `explicit queue(const context &syclContext, const device &syclDevice, const async_handler &asyncHandler, const property_list &propList = {})` | Constructs a SYCL `queue` instance with an `async_handler` using the `syclDevice` provided, and associated with the `syclContext` provided. Must throw an `exception` with the `errc::invalid_object_error` error code if `syclContext` does not encapsulate the SYCL `device` `syclDevice`. Zero or more properties can be provided to the constructed SYCL `queue` via an instance of `property_list`. |
| | End of table |

Table 4.21: Constructors of the `queue` class.

| Member function | Description |
|---|---|
| `backend get_backend()const` | Returns the a `backend` identifying the SYCL backend associated with this `queue`. |
| `context get_context ()const` | Returns the SYCL queue's context. Reports errors using SYCL exception classes. The value returned must be equal to that returned by `get_info<info::queue::context>()`. |
| `device get_device ()const` | Returns the SYCL device the queue is associated with. Reports errors using SYCL exception classes. The value returned must be equal to that returned by `get_info<info::queue::devices>()`. |
| `bool is_host()const` | Returns true if the backend associated with this SYCL `queue` is a SYCL host backend. |
| `bool is_in_order()const` | Returns true if the SYCL `queue` was created with the `in_order` property. Equivalent to `has_property<property::queue::in_order>()`. |
| `void wait()` | Performs a blocking wait for the completion of all enqueued tasks in the queue. Synchronous errors will be reported through SYCL exceptions. |
| | Continued on next page |

Table 4.22: Member functions for `queue` class.

| Member function | Description |
|---|---|
| `void wait_and_throw ()` | Performs a blocking wait for the completion of all enqueued tasks in the queue. Synchronous errors will be reported through SYCL exceptions. Any unconsumed asynchronous errors will be passed to the `async_handler` associated with the queue or enclosing context. If no user defined `async_handler` is associated with the queue or enclosing context, then an implementation defined default `async_handler` is called to handle any errors, as described in 4.15.1.2. |
| `void throw_asynchronous ()` | Checks to see if any unconsumed asynchronous errors have been produced by the queue and if so reports them by passing them to the `async_handler` associated with the queue or enclosing context. If no user defined `async_handler` is associated with the queue or enclosing context, then an implementation defined default `async_handler` is called to handle any errors, as described in 4.15.1.2. |
| `template <info::queue param> typename info::param_traits <info::queue, param>::return_type get_info ()const` | Queries the platform for `cl_command_queue_info` |
| `template <typename T> event submit(T cgf)` | Submit a command group function object to the queue, in order to be scheduled for execution on the device. |
| `template <typename T> event submit(T cgf, queue & secondaryQueue)` | Submit a command group function object to the queue, in order to be scheduled for execution on the device. On a kernel error, this command group function object, is then scheduled for execution on the secondary queue. Returns an event, which corresponds to the queue the command group function object is being enqueued on. |

Continued on next page

Table 4.22: Member functions for queue class.

| Member function | Description |
|---|---|
| `template <typename BackendEnum, BackendEnum param> typename info::param_traits<BackendEnum, param>::return_type get_backend_info()const` | Queries this SYCL queue for SYCL backend-specific information requested by the template parameter param. BackendEnum can be any enum class type specified by the SYCL backend specification of a supported SYCL backend named according to the convention info::<backend_name>::queue and param must be a valid enumeration of that enum class. Specializations of info::param_traits must be defined for BackendEnum in accordance with the SYCL backend specification. Must throw an exception with the errc::invalid_object_error error code if the SYCL backend that corresponds with BackendEnum is different from the SYCL backend that is associated with this queue. |
| `template <typename KernelName, typename KernelType> void single_task(const KernelType &kernelFunc)` | Defines and invokes a SYCL kernel function as a lambda function or a named function object type. The kernel function is submitted to the queue, in order to be scheduled for execution on the device. |
| `template <typename KernelName, typename KernelType> void single_task( event DepEvent, const KernelType &kernelFunc)` | Defines and invokes a SYCL kernel function as a lambda function or a named function object type. The kernel function is submitted to the queue, in order to be scheduled for execution on the device once the event specified by DepEvent has completed. |
| `template <typename KernelName, typename KernelType> void single_task( const std::vector<event> &DepEvents, const KernelType &kernelFunc)` | Defines and invokes a SYCL kernel function as a lambda function or a named function object type. The kernel function is submitted to the queue, in order to be scheduled for execution on the device once every event specified by DepEvents has completed. |
| `template <typename KernelName, typename KernelType, int dimensions> void parallel_for( range<dimensions> numWorkItems, const KernelType &kernelFunc)` | Defines and invokes a SYCL kernel function as a lambda function or a named function object type, for the specified range and given an id or item for indexing in the indexing space defined by range. The kernel function is submitted to the queue, in order to be scheduled for execution on the device. |

Continued on next page

Table 4.22: Member functions for queue class.

| Member function | Description |
|---|---|
| `template <typename KernelName, typename KernelType, int dimensions>`<br>`void parallel_for(`<br>`range<dimensions> numWorkItems, event DepEvent,`<br>`const KernelType &kernelFunc)` | Defines and invokes a SYCL kernel function as a lambda function or a named function object type, for the specified range and given an id or item for indexing in the indexing space defined by range. The kernel function is submitted to the queue, in order to be scheduled for execution on the device once the `event` specified by `DepEvent` has completed. |
| `template <typename KernelName, typename KernelType, int dimensions>`<br>`void parallel_for(`<br>`range<dimensions> numWorkItems,`<br>`const std::vector<event> &DepEvents,`<br>`const KernelType &kernelFunc)` | Defines and invokes a SYCL kernel function as a lambda function or a named function object type, for the specified range and given an id or item for indexing in the indexing space defined by range. The kernel function is submitted to the queue, in order to be scheduled for execution on the device once every `event` specified by `DepEvents` has completed. |
| `template <typename KernelName, typename KernelType, int dimensions>`<br>`void parallel_for(`<br>`range<dimensions> numWorkItems,`<br>`id<dimensions> workItemOffset,`<br>`const KernelType &kernelFunc)` | Defines and invokes a SYCL kernel function as a lambda function or a named function object type, for the specified range and given an id or item for indexing in the indexing space defined by range. The kernel function is submitted to the queue, in order to be scheduled for execution. |
| `template <typename KernelName, typename KernelType, int dimensions>`<br>`void parallel_for(`<br>`range<dimensions> numWorkItems,`<br>`id<dimensions> workItemOffset,`<br>`event DepEvent,`<br>`const KernelType &kernelFunc)` | Defines and invokes a SYCL kernel function as a lambda function or a named function object type, for the specified range and given an id or item for indexing in the indexing space defined by range. The kernel function is submitted to the queue, in order to be scheduled for execution on the device once the `event` specified by `DepEvent` has completed. |
| `template <typename KernelName, typename KernelType, int dimensions>`<br>`void parallel_for(`<br>`range<dimensions> numWorkItems,`<br>`id<dimensions> workItemOffset,`<br>`const std::vector<event> &DepEvents,`<br>`const KernelType &kernelFunc)` | Defines and invokes a SYCL kernel function as a lambda function or a named function object type, for the specified range and given an id or item for indexing in the indexing space defined by range. The kernel function is submitted to the queue, in order to be scheduled for execution on the device once every `event` specified by `DepEvents` has completed. |
| | Continued on next page |

Table 4.22: Member functions for queue class.

| Member function | Description |
|---|---|
| `template <typename KernelName, typename KernelType, int dimensions> void parallel_for( nd_range<dimensions> executionRange, const KernelType &kernelFunc)` | Defines and invokes a SYCL kernel function as a lambda function or a named function object type, for the specified nd-range and given an nd-item for indexing in the indexing space defined by the nd-range. The kernel function is submitted to the queue, in order to be scheduled for execution. |
| `template <typename KernelName, typename KernelType, int dimensions> void parallel_for( nd_range<dimensions> executionRange, event DepEvent, const &KernelType kernelFunc)` | Defines and invokes a SYCL kernel function as a lambda function or a named function object type, for the specified nd-range and given an nd-item for indexing in the indexing space defined by the nd-range. The kernel function is submitted to the queue, in order to be scheduled for execution on the device once the `event` specified by `DepEvent` has completed. |
| `template <typename KernelName, typename KernelType, int dimensions> void parallel_for( nd_range<dimensions> executionRange, const std::vector<event> &DepEvents, const KernelType &kernelFunc)` | Defines and invokes a SYCL kernel function as a lambda function or a named function object type, for the specified nd-range and given an nd-item for indexing in the indexing space defined by the nd-range. The kernel function is submitted to the queue, in order to be scheduled for execution on the device once every `event` specified by `DepEvents` has completed. |
| | End of table |

Table 4.22: Member functions for `queue` class.

#### 4.6.5.2 Queue information descriptors

A queue can be queried for information using the `get_info` member function of the `queue` class, specifying one of the info parameters enumerated in `info::queue`. Every queue (including a host queue) must produce a valid value for each info parameter. The possible values for each info parameter and any restriction are defined in the specification of the SYCL backend associated with the queue. All info parameters in `info::queue` are specified in Table 4.23 and the synopsis for `info::queue` is described in appendix A.4.

| Queue Descriptors | Return type | Description |
|---|---|---|
| `info::queue::context` | `context` | Returns the SYCL `context` associated with this SYCL `queue`. |
| `info::queue::device` | `device` | Returns the SYCL device associated with this SYCL `queue`. |
| | | End of table |

Table 4.23: Queue information descriptors.

#### 4.6.5.3 Queue properties

The properties that can be provided when constructing the SYCL `queue` class are describe in Table 4.24.

| Property | Description |
|---|---|
| property::queue::enable_profiling | The enable_profiling property adds the requirement that the SYCL runtime must capture profiling information for the command groups that are submitted from this SYCL queue and provide said information via the SYCL event class get_profiling_info member function, if the associated SYCL device has aspect::queue_profiling. |
| property::queue::in_order | The in_order property adds the requirement that the SYCL queue provides in-order semantics where tasks are executed in the order in which they are submitted. Tasks submitted in this fashion can be viewed as having an implicit dependence on the previously submitted operation. |
| | End of table |

Table 4.24: Properties supported by the SYCL queue class.

The constructors of the queue property classes are listed in Table 4.25.

| Constructor | Description |
|---|---|
| property::queue::enable_profiling::enable_profiling () | Constructs a SYCL enable_profiling property instance. |
| | End of table |

Table 4.25: Constructors of the queue property classes.

#### 4.6.5.4 Queue error handling

Queue errors come in two forms:

- **Synchronous Errors** are those that we would expect to be reported directly at the point of waiting on an event, and hence waiting for a queue to complete, as well as any immediate errors reported by enqueuing work onto a queue. Such errors are reported through C++ exceptions.

- Asynchronous errors are those that are produced or detected after associated host API calls have returned (so can't be thrown as exceptions by the API call), and that are handled by an async_handler through which the errors are reported. Handling of asynchronous errors from a queue occurs at specific times, as described by 4.15.

Note that if there are asynchronous errors to be processed when a queue is destructed, the handler is called and this might delay or block the destruction, according to the behavior of the handler.

### 4.6.6 Event class

An event in SYCL is an object that represents the status of an operation that is being executed by the SYCL runtime.

Typically in SYCL, data dependency and execution order is handled implicitly by the SYCL runtime. However, in some circumstances developers want fine grain control the execution, or want to retrieve properties of a command that is running.

A SYCL event maps to a single SYCL backend object when available. Note that, although an event represents the status of a particular operation, the dependencies of a certain event can be used to keep track of multiple steps required to synchronize said operation.

A SYCL event is returned by the submission of a command group. The dependencies of the event returned via the submission of the command group are the implementation-defined commands associated with the command group execution.

The SYCL `event` class provides the common reference semantics (see Section 4.5.3).

The constructors and member functions of the SYCL `event` class are listed in Tables 4.26 and 4.27, respectively. The additional common special member functions and common member functions are listed in Tables 4.1 and 4.2, respectively.

```
namespace sycl {

class event {
 public:
  event();

  /* -- common interface members -- */

  backend get_backend() const;

  bool is_host() const;

  std::vector<event> get_wait_list();

  void wait();

  static void wait(const std::vector<event> &eventList);

  void wait_and_throw();

  static void wait_and_throw(const std::vector<event> &eventList);

  template <info::event param>
  typename info::param_traits<info::event, param>::return_type
  get_info() const;

  template <typename BackendEnum, BackendEnum param>
  typename info::param_traits<BackendEnum, param>::return_type
  get_backend_info() const;

  template <info::event_profiling param>
  typename info::param_traits<info::event_profiling, param>::return_type
  get_profiling_info() const;
};

}  // namespace sycl
```

| Constructor | Description |
| --- | --- |
| `event ()` | Constructs a ready SYCL `event`. If the constructed SYCL `event` is waited on it will complete immediately. |
| | End of table |

Table 4.26: Constructors of the `event` class.

| Member function | Description |
| --- | --- |
| `backend get_backend()const` | Returns the a `backend` identifying the SYCL backend associated with this `event`. |
| `bool is_host()const` | Returns true if this SYCL `event` is a host event. |
| `std::vector<event> get_wait_list()` | Return the list of events that this event waits for in the dependence graph. Only direct dependencies are returned, and not transitive dependencies that direct dependencies wait on. Whether already completed events are included in the returned list is implementation defined. |
| `void wait()` | Wait for the event and the command associated with it to complete. |
| `void wait_and_throw()` | Wait for the event and the command associated with it to complete.<br>Any unconsumed asynchronous errors from any context that the event was waiting on executions from will be passed to the `async_handler` associated with the context. If no user defined `async_handler` is associated with the context, then an implementation defined default `async_handler` is called to handle any errors, as described in 4.15.1.2. |
| `static void wait(const std::vector<event> &eventList)` | Synchronously wait on a list of events. |
| `static void wait_and_throw(const std::vector<event> &eventList)` | Synchronously wait on a list of events.<br>Any unconsumed asynchronous errors from any context that the event was waiting on executions from will be passed to the `async_handler` associated with the context. If no user defined `async_handler` is associated with the context, then an implementation defined default `async_handler` is called to handle any errors, as described in 4.15.1.2. |
| | Continued on next page |

Table 4.27: Member functions for the `event` class.

| Member function | Description |
|---|---|
| `template <info::event param> typename info::param_traits <info::event, param>::return_type get_info()const` | Queries this SYCL `event` for information requested by the template parameter `param` . Specializations of `info::param_traits` must be defined in accordance with the info parameters in Table 4.28 to facilitate returning the type associated with the `param` parameter. |
| `template <typename BackendEnum, BackendEnum param> typename info::param_traits<BackendEnum, param>:: return_type get_backend_info()const` | Queries this SYCL `event` for SYCL backend-specific information requested by the template parameter `param`. `BackendEnum` can be any enum class type specified by the SYCL backend specification of a supported SYCL backend named according to the convention `info::<backend_name>::event` and `param` must be a valid enumeration of that enum class. Specializations of `info::param_traits` must be defined for `BackendEnum` in accordance with the SYCL backend specification. Must throw an `exception` with the `errc::invalid_object_error` error code if the SYCL backend that corresponds with `BackendEnum` is different from the SYCL backend that is associated with this `event`. |
| `template <info::event_profiling param> typename info::param_traits <info::event_profiling, param>::return_type get_profiling_info ()const` | Queries this SYCL `event` for profiling information requested by the parameter `param` . If the requested profiling information is unavailable when `get_profiling_info` is called due to incompletion of command groups associated with the `event`, then the call to `get_profiling_info` will block until the requested profiling information is available. An example is asking for `info::event_profiling::command_end` when the associated command group has yet to finish execution. Calls to `get_profiling_info` must throw an `exception` with the `errc::invalid_object_error` error code if the SYCL `queue` which submitted the command group this SYCL `event` is associated with was not constructed with the `property::queue::enable_profiling` property. Specializations of `info::param_traits` must be defined in accordance with the info parameters in Table 4.29 to facilitate returning the type associated with the `param` parameter. |
| | End of table |

Table 4.27: Member functions for the `event` class.

#### 4.6.6.1 Event information and profiling descriptors

An event can be queried for information using the `get_info` member function of the `event` class, specifying one of the info parameters enumerated in `info::event`. Every event (including a host event) must produce a valid value for each info parameter. The possible values for each info parameter and any restriction are defined in the specification of the SYCL backend associated with the event. All info parameters in `info::event` are specified in Table 4.28 and the synopsis for `info::event` is described in appendix A.6.

| Event Descriptors | Return type | Description |
| --- | --- | --- |
| `info::event::command_execution_status` | `info::event_command_status` | Returns the event status of the command group associated with this SYCL `event`. |
| | | End of table |

Table 4.28: Event class information descriptors.

An event can be queried for profiling information use the `get_profiling_info` member function of the `event` class, specifying one of the profiling info parameters enumerated in `info::event_profiling`. Every event (including a host event) must produce a valid value for each info parameter. The possible values for each info parameter and any restriction are defined in the specification of the SYCL backend associated with the event. All info parameters in `info::event_profiling` are specified in Table 4.29 and the synopsis for `info::event_profiling` is described in appendix A.6.

| Event information profiling descriptor | Return type | Description |
| --- | --- | --- |
| `info::event_profiling::command_submit` | `uint64_t` | Returns an implementation defined 64-bit value describing the time in nanoseconds when the associated command group was submitted. |
| `info::event_profiling::command_start` | `uint64_t` | Returns an implementation defined 64-bit value describing the time in nanoseconds when the associated command group started executing. |
| `info::event_profiling::command_end` | `uint64_t` | Returns an implementation defined 64-bit value describing the time in nanoseconds when the associated command group finished executing. |
| | | End of table |

Table 4.29: Profiling information descriptors for the SYCL `event` class.

## 4.7 Data access and storage in SYCL

In SYCL, data storage and access are handled by separate classes. Buffers and images handle storage and ownership of the data, whereas accessors handle access to the data. Buffers and images in SYCL can be bound to more than one device or context, including across different SYCL backends. They also handle ownership of the data, while allowing exception handling for blocking and non-blocking data transfers. Accessors manage data transfers between the host and all of the devices in the system, as well as tracking of data dependencies.

### 4.7.1 Host allocation

A SYCL runtime may need to allocate temporary objects on the host to handle some operations (such as copying data from one context to another). Allocation on the host is managed using an allocator object, following the standard C++ allocator class definition. The default allocator for memory objects is implementation defined, but the user can supply their own allocator class.

```
{
  buffer<int, 1, UserDefinedAllocator<int> > b(d);
}
```

When an allocator returns a `nullptr`, the runtime cannot allocate data on the host. Note that in this case the runtime will raise an error if it requires host memory but it is not available (e.g when moving data across SYCL backend contexts).

The definition of allocators extends the current functionality of SYCL, ensuring that users can define allocator functions for specific hardware or certain complex shared memory mechanisms (e.g. NUMA), and improves interoperability with STL-based libraries (e.g, Intel's TBB provides an allocator).

#### 4.7.1.1 Default allocators

A default allocator is always defined by the implementation, and it is guaranteed to return non-`nullptr` and new memory positions every call. The default allocator for const buffers will remove the const-ness of the type (therefore, the default allocator for a buffer of type "const int" will be an `Allocator<int>`). This implies that host accessors will not synchronize with the pointer given by the user in the buffer/image constructor, but will use the memory returned by the `Allocator` itself for that purpose. The user can implement an allocator that returns the same address as the one passed in the buffer constructor, but it is the responsibility of the user to handle the potential race conditions.

| Allocators | Description |
|---|---|
| `buffer_allocator` | It is the default buffer allocator used by the runtime, when no allocator is defined by the user. |
| `image_allocator` | It is the default allocator used by the runtime for the SYCL `unsampled_image` and `sampled_image` classes when no allocator is provided by the user. The `image_allocator` is required allocate in elements of `byte`. |
| | End of table |

Table 4.30: SYCL Default Allocators.

See Section 4.7.5 for details on manual host-device synchronization.

### 4.7.2 Buffers

The `buffer` class defines a shared array of one, two or three dimensions that can be used by the SYCL kernel and has to be accessed using accessor classes. Buffers are templated on both the type of their data, and the number of dimensions that the data is stored and accessed through.

A buffer does not map to only one underlying backend object, and all SYCL backend memory objects may be temporary for use within a command group on a specific device. Note that if no source data is provided for a buffer, the buffer uses uninitialized memory for performance reasons. So it is up to the programmer to explicitly construct the objects in this case if required.

More generally, since the value type of a buffer is required to be trivially copyable, there is no constructor or destructor called in any case.

A SYCL buffer can construct an instance of a SYCL buffer that reinterprets the original SYCL buffer with a different type, dimensionality and range using the member function reinterpret. The reinterpreted SYCL buffer that is constructed must behave as though it were a copy of the SYCL buffer that constructed it (see sec 4.5.3) with the exception that the type, dimensionality and range of the reinterpreted SYCL buffer must reflect the type, dimensionality and range specified when calling the reinterpret member function. By extension of this the class member types value_type, reference and const_reference, and the member functions get_range and get_count of the reinterpreted SYCL buffer must reflect the new type, dimensionality and range. The data that the original SYCL buffer and the reinterpreted SYCL buffer manage remains unaffected, though the representation of the data when accessed through the reinterpreted SYCL buffer may alter to reflect the new type, dimensionality and range. It is important to note that a reinterpreted SYCL buffer is a copy of the original SYCL buffer only, and not a new SYCL buffer. Constructing more than one SYCL buffer managing the same host pointer is still undefined behavior.

The SYCL buffer class template provides the common reference semantics (see Section 4.5.3).

#### 4.7.2.1 Buffer interface

The constructors and member functions of the SYCL buffer class template are listed in Tables 4.31 and 4.32, respectively. The additional common special member functions and common member functions are listed in Tables 4.1 and 4.2, respectively.

Each constructor takes as the last parameter an optional SYCL property_list to provide properties to the SYCL buffer.

The SYCL buffer class template takes a template parameter AllocatorT for specifying an allocator which is used by the SYCL runtime when allocating temporary memory on the host. If no template argument is provided then the default allocator for the SYCL buffer class buffer_allocator will be used (see 4.7.1.1).

```
namespace sycl {
namespace property {
namespace buffer {
class use_host_ptr {
  public:
    use_host_ptr() = default;
};

class use_mutex {
  public:
    use_mutex(std::mutex &mutexRef);

    std::mutex *get_mutex_ptr() const;
};

class context_bound {
  public:
```

```
    context_bound(context boundContext);

    context get_context() const;
};
}  // namespace buffer
}  // namespace property

template <typename T, int dimensions = 1,
          typename AllocatorT = sycl::buffer_allocator>
class buffer {
 public:
  using value_type = T;
  using reference = value_type &;
  using const_reference = const value_type &;
  using allocator_type = AllocatorT;

  buffer(const range<dimensions> &bufferRange,
         const property_list &propList = {});

  buffer(const range<dimensions> &bufferRange, AllocatorT allocator,
         const property_list &propList = {});

  buffer(T *hostData, const range<dimensions> &bufferRange,
         const property_list &propList = {});

  buffer(T *hostData, const range<dimensions> &bufferRange,
         AllocatorT allocator, const property_list &propList = {});

  buffer(const T *hostData, const range<dimensions> &bufferRange,
         const property_list &propList = {});

  buffer(const T *hostData, const range<dimensions> &bufferRange,
         AllocatorT allocator, const property_list &propList = {});

  buffer(const std::shared_ptr<T> &hostData,
         const range<dimensions> &bufferRange, AllocatorT allocator,
         const property_list &propList = {});

  buffer(const std::shared_ptr<T> &hostData,
         const range<dimensions> &bufferRange,
         const property_list &propList = {});

  template <class InputIterator>
  buffer<T, 1>(InputIterator first, InputIterator last, AllocatorT allocator,
               const property_list &propList = {});

  template <class InputIterator>
  buffer<T, 1>(InputIterator first, InputIterator last,
               const property_list &propList = {});

  buffer(buffer<T, dimensions, AllocatorT> b, const id<dimensions> &baseIndex,
         const range<dimensions> &subRange);

  /* -- common interface members -- */

```

```
  /* -- property interface members -- */

  range<dimensions> get_range() const;

  size_t get_count() const;

  size_t get_size() const;

  AllocatorT get_allocator() const;

  template <access::mode mode, access::target target = access::target::global_buffer>
  accessor<T, dimensions, mode, target> get_access(
      handler &commandGroupHandler);

  template <access::mode mode>
  accessor<T, dimensions, mode, access::target::host_buffer> get_access();

  template <access::mode mode, access::target target = access::target::global_buffer>
  accessor<T, dimensions, mode, target> get_access(
      handler &commandGroupHandler, range<dimensions> accessRange,
      id<dimensions> accessOffset = {});

  template <access::mode mode>
  accessor<T, dimensions, mode, access::target::host_buffer> get_access(
    range<dimensions> accessRange, id<dimensions> accessOffset = {});

  template<typename... Ts>
  auto get_access(Ts...);

  template<typename... Ts>
  auto get_host_access(Ts...);

  template <typename Destination = std::nullptr_t>
  void set_final_data(Destination finalData = nullptr);

  void set_write_back(bool flag = true);

  bool is_sub_buffer() const;

  template <typename ReinterpretT, int ReinterpretDim>
  buffer<ReinterpretT, ReinterpretDim, AllocatorT>
  reinterpret(range<ReinterpretDim> reinterpretRange) const;

  // Only available when ReinterpretDim == 1
  // or when (ReinterpretDim == dimensions) &&
  //         (sizeof(ReinterpretT) == sizeof(T))
  template <typename ReinterpretT, int ReinterpretDim = dimensions>
  buffer<ReinterpretT, ReinterpretDim, AllocatorT>
  reinterpret() const;
};

// Deduction guides
template <class InputIterator, class AllocatorT>
buffer(InputIterator, InputIterator, AllocatorT, const property_list & = {})
    -> buffer<typename std::iterator_traits<InputIterator>::value_type, 1,
```

```
             AllocatorT>;
template <class InputIterator>
buffer(InputIterator, InputIterator, const property_list & = {})
    -> buffer<typename std::iterator_traits<InputIterator>::value_type, 1>;
template <class T, int dimensions, class AllocatorT>
buffer(const T *, const range<dimensions> &, AllocatorT,
       const property_list & = {})
    -> buffer<T, dimensions, AllocatorT>;
template <class T, int dimensions>
buffer(const T *, const range<dimensions> &, const property_list & = {})
    -> buffer<T, dimensions>;

}  // namespace sycl
```

| Constructor | Description |
|---|---|
| `buffer(const range<dimensions> & bufferRange, const property_list &propList = {})` | Construct a SYCL `buffer` instance with uninitialized memory. The constructed SYCL `buffer` will use a default constructed `AllocatorT` when allocating memory on the host. The range of the constructed SYCL `buffer` is specified by the `bufferRange` parameter provided. Data is not written back to the host on destruction of the `buffer` unless the `buffer` has a valid non-null pointer specified via the member function `set_final_data()`. Zero or more properties can be provided to the constructed SYCL `buffer` via an instance of `property_list`. |
| `buffer(const range<dimensions> & bufferRange, AllocatorT allocator, const property_list &propList = {})` | Construct a SYCL `buffer` instance with uninitialized memory. The constructed SYCL `buffer` will use the `allocator` parameter provided when allocating memory on the host. The range of the constructed SYCL `buffer` is specified by the `bufferRange` parameter provided. Data is not written back to the host on destruction of the `buffer` unless the `buffer` has a valid non-null pointer specified via the member function `set_final_data()`. Zero or more properties can be provided to the constructed SYCL `buffer` via an instance of `property_list`. |

Continued on next page

Table 4.31: Constructors of the `buffer` class.

| Constructor | Description |
|---|---|
| `buffer(T* hostData, const range<dimensions> & bufferRange, const property_list &propList = {})` | Construct a SYCL `buffer` instance with the `hostData` parameter provided. The ownership of this memory is given to the constructed SYCL `buffer` for the duration of its lifetime. The constructed SYCL `buffer` will use a default constructed `AllocatorT` when allocating memory on the host. The range of the constructed SYCL `buffer` is specified by the `bufferRange` parameter provided. Zero or more properties can be provided to the constructed SYCL `buffer` via an instance of `property_list`. |
| `buffer(T* hostData, const range<dimensions> & bufferRange, AllocatorT allocator, const property_list &propList = {})` | Construct a SYCL `buffer` instance with the `hostData` parameter provided. The ownership of this memory is given to the constructed SYCL `buffer` for the duration of its lifetime. The constructed SYCL `buffer` will use the `allocator` parameter provided when allocating memory on the host. The range of the constructed SYCL `buffer` is specified by the `bufferRange` parameter provided. Zero or more properties can be provided to the constructed SYCL `buffer` via an instance of `property_list`. |
| | Continued on next page |

Table 4.31: Constructors of the `buffer` class.

| Constructor | Description |
| --- | --- |
| `buffer(const T* hostData, const range<dimensions> & bufferRange, const property_list &propList = {})` | Construct a SYCL `buffer` instance with the `hostData` parameter provided. The ownership of this memory is given to the constructed SYCL `buffer` for the duration of its lifetime.<br>The constructed SYCL `buffer` will use a default constructed `AllocatorT` when allocating memory on the host.<br>The host address is `const T`, so the host accesses can be read-only. However, the `typename T` is not const so the device accesses can be both read and write accesses. Since the `hostData` is const, this buffer is only initialized with this memory and there is no write back after its destruction, unless the `buffer` has another valid non-null final data address specified via the member function `set_final_data()` after construction of the `buffer`.<br>The range of the constructed SYCL `buffer` is specified by the `bufferRange` parameter provided.<br>Zero or more properties can be provided to the constructed SYCL `buffer` via an instance of `property_list`. |

Continued on next page

Table 4.31: Constructors of the `buffer` class.

| Constructor | Description |
|---|---|
| `buffer(const T* hostData, const range<dimensions> & bufferRange, AllocatorT allocator, const property_list &propList = {})` | Construct a SYCL `buffer` instance with the `hostData` parameter provided. The ownership of this memory is given to the constructed SYCL `buffer` for the duration of its lifetime.<br>The constructed SYCL `buffer` will use the `allocator` parameter provided when allocating memory on the host.<br>The host address is `const T`, so the host accesses can be read-only. However, the `typename T` is not const so the device accesses can be both read and write accesses. Since, the `hostData` is const, this buffer is only initialized with this memory and there is no write back after its destruction, unless the `buffer` has another valid non-null final data address specified via the member function `set_final_data()` after construction of the `buffer`.<br>The range of the constructed SYCL `buffer` is specified by the `bufferRange` parameter provided.<br>Zero or more properties can be provided to the constructed SYCL `buffer` via an instance of `property_list`. |
| `buffer(const std::shared_ptr<T> &hostData, const range<dimensions> & bufferRange, const property_list &propList = {})` | Construct a SYCL `buffer` instance with the `hostData` parameter provided. The ownership of this memory is given to the constructed SYCL `buffer` for the duration of its lifetime. The constructed SYCL `buffer` will use a default constructed `AllocatorT` when allocating memory on the host. The range of the constructed SYCL `buffer` is specified by the `bufferRange` parameter provided. Zero or more properties can be provided to the constructed SYCL `buffer` via an instance of `property_list`. |

Continued on next page

Table 4.31: Constructors of the `buffer` class.

| Constructor | Description |
| --- | --- |
| `buffer(const std::shared_ptr<void> &hostData, const range<dimensions> & bufferRange, AllocatorT allocator, const property_list &propList = {})` | Construct a SYCL `buffer` instance with the `hostData` parameter provided. The ownership of this memory is given to the constructed SYCL `buffer` for the duration of its lifetime. The constructed SYCL `buffer` will use the `allocator` parameter provided when allocating memory on the host. The range of the constructed SYCL `buffer` is specified by the `bufferRange` parameter provided. Zero or more properties can be provided to the constructed SYCL `buffer` via an instance of `property_list`. |
| `template <typename InputIterator> buffer(InputIterator first, InputIterator last, const property_list &propList = {})` | Create a new allocated 1D buffer initialized from the given elements ranging from `first` up to one before `last`. The data is copied to an intermediate memory position by the runtime. Data is not written back to the same iterator set provided. However, if the `buffer` has a valid non-const iterator specified via the member function `set_final_data()`, data will be copied back to that iterator. The constructed SYCL `buffer` will use a default constructed `AllocatorT` when allocating memory on the host. Zero or more properties can be provided to the constructed SYCL `buffer` via an instance of `property_list`. |
| `template <typename InputIterator> buffer(InputIterator first, InputIterator last, AllocatorT allocator = {}, const property_list &propList = {})` | Create a new allocated 1D buffer initialized from the given elements ranging from `first` up to one before `last`. The data is copied to an intermediate memory position by the runtime. Data is not written back to the same iterator set provided. However, if the `buffer` has a valid non-const iterator specified via the member function `set_final_data()`, data will be copied back to that iterator. The constructed SYCL `buffer` will use the `allocator` parameter provided when allocating memory on the host. Zero or more properties can be provided to the constructed SYCL `buffer` via an instance of `property_list`. |

Continued on next page

Table 4.31: Constructors of the `buffer` class.

| Constructor | Description |
|---|---|
| `buffer(buffer<T, dimensions, AllocatorT> &b, const id<dimensions> & baseIndex, const range<dimensions> & subRange)` | Create a new sub-buffer without allocation to have separate accessors later. `b` is the buffer with the real data, which must not be a sub-buffer. `baseIndex` specifies the origin of the sub-buffer inside the buffer `b` . `subRange` specifies the size of the sub-buffer. The sum of `baseIndex` and `subRange` in any dimension must not exceed the parent buffer (`b`) size (`bufferRange`) in that dimension, and an `exception` with the `errc::invalid_object_error` error code must be thrown if violated.<br>The offset and range specified by `baseIndex` and `subRange` together must represent a contiguous region of the original SYCL `buffer`. If a non-contiguous region of a buffer is requested when constructing ga sub-buffer, then an `exception` with the `errc::invalid_object_error` error code must be thrown.<br>The origin (based on `baseIndex`) of the sub-buffer being constructed must be a multiple of the memory base address alignment of each SYCL `device` that is executed on, otherwise the SYCL runtime must throw an asynchronous `exception` with the `errc::invalid_object_error` error code.<br>This value is retrievable via the SYCL `device` class info query `info::device::mem_base_addr_align`. Must throw `exception` with the `errc::invalid_object_error` error code if `b` is already a sub-buffer. |
| | End of table |

Table 4.31: Constructors of the `buffer` class.

| Member function | Description |
|---|---|
| `range<dimensions> get_range()const` | Return a range object representing the size of the buffer in terms of number of elements in each dimension as passed to the constructor. |
| `size_t get_count()const` | Returns the total number of elements in the buffer. Equal to `get_range()[0] * ... * get_range()[dimensions-1]`. |
| `size_t get_size()const` | Returns the size of the buffer storage in bytes. Equal to `get_count()*sizeof(T)`. |
| | Continued on next page |

Table 4.32: Member functions for the `buffer` class.

| Member function | Description |
| --- | --- |
| `AllocatorT get_allocator()const` | Returns the allocator provided to the buffer. |
| `template<access::mode mode, access::target target = access::target::global_buffer> accessor<T, dimensions, mode, target> get_access(handler &commandGroupHandler)` | Returns a valid `accessor` to the buffer with the specified access mode and target in the command group buffer. The value of target can be `access::target::global_buffer` or `access::constant_buffer`. |
| `template<access::mode mode> accessor<T, dimensions, mode, access::target::host_buffer> get_access()` | Returns a valid `host_accessor` to the buffer with the specified access mode and target. |
| `template<access::mode mode, access::target target= access::target::global_buffer> accessor<T, dimensions, mode, target> get_access(handler &commandGroupHandler, range<dimensions> accessRange, id<dimensions> accessOffset = {})` | Returns a valid `accessor` to the buffer with the specified access mode and target in the command group buffer. Only the values starting from the given offset and up to the given range are guaranteed to be updated. The value of target can be `access::target::global_buffer` or `access::constant_buffer`. |
| `template<access::mode mode> accessor<T, dimensions, mode, access::target::host_buffer> get_access(range<dimensions> accessRange, id<dimensions> accessOffset = {})` | Returns a valid `host_accessor` to the buffer with the specified access mode and target. Only the values starting from the given offset and up to the given range are guaranteed to be updated. The value of target can only be `access::target::host_buffer`. |
| `template<typename... Ts> auto get_access(Ts... args)` | Returns a valid `accessor` as if constructed via passing the buffer and all provided arguments to the SYCL `accessor`.<br><br>Possible implementation:<br>`return accessor { *this, args... };` |
| `template<typename... Ts> auto get_host_access(Ts... args)` | Returns a valid `host_accessor` as if constructed via passing the buffer and all provided arguments to the SYCL `host_accessor`.<br><br>Possible implementation:<br>`return host_accessor { *this, args... };` |

Continued on next page

Table 4.32: Member functions for the `buffer` class.

| Member function | Description |
|---|---|
| `template <typename Destination = std::nullptr_t> void set_final_data(Destination finalData = nullptr)` | The `finalData` points to where the outcome of all the buffer processing is going to be copied to at destruction time, if the buffer was involved with a write accessor.<br>Destination can be either an output iterator or a `std::weak_ptr<T>`.<br>Note that a raw pointer is a special case of output iterator and thus defines the host memory to which the result is to be copied.<br>In the case of a weak pointer, the output is not updated if the weak pointer has expired.<br>If `Destination` is `std::nullptr_t`, then the copy back will not happen. |
| `void set_write_back(bool flag = true)` | This method allows dynamically forcing or canceling the write-back of the data of a buffer on destruction according to the value of `flag`.<br>Forcing the write-back is similar to what happens during a normal write-back as described in § 4.7.2.3 and 4.7.4.<br>If there is nowhere to write-back, using this function does not have any effect. |
| `bool is_sub_buffer()const` | Returns true if this SYCL `buffer` is a sub-buffer, otherwise returns false. |
| `template <typename ReinterpretT, int ReinterpretDim> buffer<ReinterpretT, ReinterpretDim, AllocatorT> reinterpret(range<ReinterpretDim> reinterpretRange)const` | Creates and returns a reinterpreted SYCL `buffer` with the type specified by `ReinterpretT`, dimensions specified by `ReinterpretDim` and range specified by `reinterpretRange`. The buffer object being reinterpreted can be a SYCL sub-buffer that was created from a SYCL `buffer`. Must throw `exception` with the `errc::invalid_object_error` error code if the total size in bytes represented by the type and range of the reinterpreted SYCL `buffer` (or sub-buffer) does not equal the total size in bytes represented by the type and range of this SYCL `buffer` (or sub-buffer). Reinterpreting a sub-buffer provides a reinterpreted view of the sub-buffer only, and does not change the offset or size of the sub-buffer view (in bytes) relative to the parent `buffer`. |

Continued on next page

Table 4.32: Member functions for the `buffer` class.

| Member function | Description |
|---|---|
| `template <typename ReinterpretT, int ReinterpretDim = dimensions> buffer<ReinterpretT, ReinterpretDim, AllocatorT> reinterpret()const` | Creates and returns a reinterpreted SYCL `buffer` with the type specified by `ReinterpretT` and dimensions specified by `ReinterpretDim`. Only valid when `(ReinterpretDim == 1)` or when `((ReinterpretDim == dimensions)&& (sizeof(ReinterpretT)== sizeof(T)))`. The buffer object being reinterpreted can be a SYCL sub-buffer that was created from a SYCL `buffer`. Must throw an `exception` with the `errc::invalid_object_error` error code if the total size in bytes represented by the type and range of the reinterpreted SYCL `buffer` (or sub-buffer) does not equal the total size in bytes represented by the type and range of this SYCL `buffer` (or sub-buffer). Reinterpreting a sub-buffer provides a reinterpreted view of the sub-buffer only, and does not change the offset or size of the sub-buffer view (in bytes) relative to the parent `buffer`. |
| | End of table |

Table 4.32: Member functions for the `buffer` class.

#### 4.7.2.2 Buffer properties

The properties that can be provided when constructing the SYCL `buffer` class are describe in Table 4.33.

| Property | Description |
|---|---|
| `property::buffer::use_host_ptr` | The `use_host_ptr` property adds the requirement that the SYCL runtime must not allocate any memory for the SYCL `buffer` and instead uses the provided host pointer directly. This prevents the SYCL runtime from allocating additional temporary storage on the host. |
| `property::buffer::use_mutex` | The `use_mutex` property is valid for the SYCL `buffer`, `unsampled_image` and `sampled_image` classes. The property adds the requirement that the memory which is owned by the SYCL `buffer` can be shared with the application via a `std::mutex` provided to the property. The mutex `m` is locked by the runtime whenever the data is in use and unlocked otherwise. Data is synchronized with `hostData`, when the mutex is unlocked by the runtime. |
| | Continued on next page |

Table 4.33: Properties supported by the SYCL `buffer` class.

| Property | Description |
|---|---|
| property::buffer::context_bound | The context_bound property adds the requirement that the SYCL buffer can only be associated with a single SYCL context that is provided to the property. |
| | End of table |

Table 4.33: Properties supported by the SYCL buffer class.

The constructors and special member functions of the buffer property classes are listed in Tables 4.34 and 4.35 respectively.

| Constructor | Description |
|---|---|
| property::buffer::use_host_ptr::use_host_ptr() | Constructs a SYCL use_host_ptr property instance. |
| property::buffer::use_mutex::use_mutex(std::mutex & mutexRef) | Constructs a SYCL use_mutex property instance with a reference to mutexRef parameter provided. |
| property::buffer::context_bound::context_bound( context boundContext) | Constructs a SYCL context_bound property instance with a copy of a SYCL context. |
| | End of table |

Table 4.34: Constructors of the buffer property classes.

| Member function | Description |
|---|---|
| std::mutex *property::buffer::use_mutex:: get_mutex_ptr()const | Returns the std::mutex which was specified when constructing this SYCL use_mutex property. |
| context property::buffer::context_bound::get_context ()const | Returns the context which was specified when constructing this SYCL context_bound property. |
| | End of table |

Table 4.35: Member functions of the buffer property classes.

#### 4.7.2.3 Buffer synchronization rules

Buffers are reference-counted. When a buffer value is constructed from another buffer, the two values reference the same buffer and a reference count is incremented. When a buffer value is destroyed, the reference count is decremented. Only when there are no more buffer values that reference a specific buffer is the actual buffer destroyed and the buffer destruction behavior defined below is followed.

If any error occurs on buffer destruction, it is reported via the associated queue's asynchronous error handling mechanism.

The basic rule for the blocking behavior of a buffer destructor is that it blocks if there is some data to write back because a write-accessor on it has been created, or if the buffer was constructed with attached host memory and is still in use.

More precisely:

1. A buffer can be constructed with just a size and using the default buffer allocator. The memory management for this type of buffer is entirely handled by the SYCL system. The destructor for this type of buffer does not need to block, even if work on the buffer has not completed. Instead, the SYCL system frees any storage required for the buffer asynchronously when it is no longer in use in queues. The initial contents of the buffer are unspecified.

2. A buffer can be constructed with associated host memory and a default buffer allocator. The buffer will use this host memory for its full lifetime, but the contents of this host memory are unspecified for the lifetime of the buffer. If the host memory is modified by the host, or mapped to another buffer or image during the lifetime of this buffer, then the results are undefined. The initial contents of the buffer will be the contents of the host memory at the time of construction.

   When the buffer is destroyed, the destructor will block until all work in queues on the buffer have completed, then copy the contents of the buffer back to the host memory (if required) and then return.

   (a) If the type of the host data is `const`, then the buffer is read-only; only read accessors are allowed on the buffer and no-copy-back to host memory is performed (although the host memory must still be kept available for use by SYCL). When using the default buffer allocator, the const-ness of the type will be removed in order to allow host allocation of memory, which will allow temporary host copies of the data by the SYCL runtime, for example for speeding up host accesses.

   When the buffer is destroyed, the destructor will block until all work in queues on the buffer have completed and then return, as there is no copy of data back to host.

   (b) If the type of the host data is not `const` but the pointer to host data is `const`, then the read-only restriction applies only on host and not on device accesses.

   When the buffer is destroyed, the destructor will block until all work in queues on the buffer have completed.

3. A buffer can be constructed using a `shared_ptr` to host data. This pointer is shared between the SYCL application and the runtime. In order to allow synchronization between the application and the runtime a `mutex` is used which will be locked by the runtime whenever the data is in use, and unlocked when it is no longer needed.

   The `shared_ptr` reference counting is used in order to prevent destroying the buffer host data prematurely. If the `shared_ptr` is deleted from the user application before buffer destruction, the buffer can continue securely because the pointer hasn't been destroyed yet. It will not copy data back to the host before destruction, however, as the application side has already deleted its copy.

   Note that since there is an implicit conversion of a `std::unique_ptr` to a `std::shared_ptr`, a `std::unique_ptr` can also be used to pass the ownership to the SYCL runtime.

4. A buffer can be constructed from a pair of iterator values. In this case, the buffer construction will copy the data from the data range defined by the iterator pair. The destructor will not copy back any data and does not need to block.

If `set_final_data()` is used to change where to write the data back to, then the destructor of the buffer will block if a write-accessor on it has been created.

A sub-buffer object can be created which is a sub-range reference to a base buffer. This sub-buffer can be used to create accessors to the base buffer, which have access to the range specified at time of construction of the sub-buffer. Sub-buffers cannot be created from sub-buffers, but only from a base buffer which is not already a sub-buffer.

Sub-buffers must be constructed from a contiguous region of memory in a buffer. This requirement is potentially non-intuitive when working with buffers that have dimensionality larger than one, but maps to one-dimensional SYCL backend native allocations without performance cost from index mapping math. For example:

```
buffer<int,2> parent_buffer { range<2>{ 8,8 } };  // Create 2-d buffer with 8x8 ints

// OK: Contiguous region from middle of buffer
buffer<int,2> sub_buf1 { parent_buffer, /*offset*/ range<2>{ 2,0 }, /*size*/ range<2>{ 2,8 } };

// invalid_object_error exception: Non-contiguous regions of 2-d buffer
buffer<int,2> sub_buf2 { parent_buffer, /*offset*/ range<2>{ 2,0 }, /*size*/ range<2>{ 2,2 } };
buffer<int,2> sub_buf3 { parent_buffer, /*offset*/ range<2>{ 2,2 }, /*size*/ range<2>{ 2,6 } };

// invalid_object_error exception: Out-of-bounds size
buffer<int,2> sub_buf4 { parent_buffer, /*offset*/ range<2>{ 2,2 }, /*size*/ range<2>{ 2,8 } };
```

### 4.7.3 Images

The classes `unsampled_image`<(>Table 4.36) and `sampled_image`<(>Table 4.38) define shared image data of one, two or three dimensions, that can be used by kernels in queues and have to be accessed using accessor classes with image accessor modes.

The constructors and member functions of the SYCL `unsampled_image` and `sampled_image` class templates are listed in Tables 4.36, 4.39, 4.36 and 4.39, respectively. The additional common special member functions and common member functions are listed in Tables 4.1 and 4.2, respectively.

Where relevant, it is the responsibility of the user to ensure that the format of the data matches the format described by `image_format`.

The allocator template parameter of the SYCL `unsampled_image` and `sampled_image` classes can be any allocator type including a custom allocator, however it must allocate in units of `byte`.

For any image that is constructed with the range $(r1, r2, r3)$ with an element type size in bytes of $s$, the image row pitch and image slice pitch should be calculated as follows:

$$r1 \cdot s \tag{4.1}$$

$$r1 \cdot r2 \cdot s \tag{4.2}$$

The SYCL `unsampled_image` and `sampled_image` class templates provide the common reference semantics (see Section 4.5.3).

#### 4.7.3.1 Unsampled image interface

Each constructor of the `unsampled_image` an `image_format` to describe the data layout of the image data.

Each constructor additionally takes as the last parameter an optional SYCL property_list to provide properties to the SYCL unsampled_image.

The SYCL unsampled_image class template takes a template parameter AllocatorT for specifying an allocator which is used by the SYCL runtime when allocating temporary memory on the host. If no template argument is provided the default allocator for the SYCL unsampled_image class image_allocator will be used 4.7.1.1.

```
namespace sycl {
namespace property {
namespace image {
class use_host_ptr {
  public:
    use_host_ptr() = default;
};

class use_mutex {
  public:
    use_mutex(std::mutex &mutexRef);

    std::mutex *get_mutex_ptr() const;
};

class context_bound {
  public:
    context_bound(context boundContext);

    context get_context() const;
};
}  // namespace image
}  // namespace property

enum class image_format : unsigned int {
  r8g8b8a8_unorm,
  r16g16b16a16_unorm,
  r8g8b8a8_sint,
  r16g16b16a16_sint,
  r32b32g32a32_sint,
  r8g8b8a8_uint,
  r16g16b16a16_uint,
  r32b32g32a32_uint,
  r16b16g16a16_sfloat,
  r32g32b32a32_sfloat,
  b8g8r8a8_unorm,
};

using byte = unsigned char;

template <int dimensions = 1, typename AllocatorT = sycl::image_allocator>
class unsampled_image {
 public:
  unsampled_image(image_format format, const range<dimensions> &rangeRef,
                  const property_list &propList = {});

  unsampled_image(image_format format, const range<dimensions> &rangeRef,
                  AllocatorT allocator, const property_list &propList = {});
```

```

/* Available only when: dimensions > 1 */
unsampled_image(image_format format, const range<dimensions> &rangeRef,
                const range<dimensions -1> &pitch,
                const property_list &propList = {});

/* Available only when: dimensions > 1 */
unsampled_image(image_format format, const range<dimensions> &rangeRef,
                const range<dimensions -1> &pitch, AllocatorT allocator,
                const property_list &propList = {});

unsampled_image(void *hostPointer, image_format format,
                const range<dimensions> &rangeRef,
                const property_list &propList = {});

unsampled_image(void *hostPointer, image_format format,
                const range<dimensions> &rangeRef, AllocatorT allocator,
                const property_list &propList = {});

/* Available only when: dimensions > 1 */
unsampled_image(void *hostPointer, image_format format,
                const range<dimensions> &rangeRef,
                const range<dimensions -1> &pitch,
                const property_list &propList = {});

/* Available only when: dimensions > 1 */
unsampled_image(void *hostPointer, image_format format,
                const range<dimensions> &rangeRef,
                const range<dimensions -1> &pitch, AllocatorT allocator,
                const property_list &propList = {});

unsampled_image(std::shared_ptr<void> &hostPointer, image_format format,
                const range<dimensions> &rangeRef,
                const property_list &propList = {});

unsampled_image(std::shared_ptr<void> &hostPointer, image_format format,
                const range<dimensions> &rangeRef, AllocatorT allocator,
                const property_list &propList = {});

/* Available only when: dimensions > 1 */
unsampled_image(std::shared_ptr<void> &hostPointer, image_format format,
                const range<dimensions> &rangeRef,
                const range<dimensions -1> &pitch,
                const property_list &propList = {});

/* Available only when: dimensions > 1 */
unsampled_image(std::shared_ptr<void> &hostPointer, image_format format,
                const range<dimensions> &rangeRef,
                const range<dimensions -1> &pitch, AllocatorT allocator,
                const property_list &propList = {});

/* -- common interface members -- */

/* -- property interface members -- */

```

```
  range<dimensions> get_range() const;

  /* Available only when: dimensions > 1 */
  range<dimensions - 1> get_pitch() const;

  size_t get_count() const;

  size_t get_size() const;

  AllocatorT get_allocator() const;

  template <typename dataT, access::mode accessMode>
  accessor<dataT, dimensions, accessMode, access::target::unsampled_image>
  get_access(handler & commandGroupHandler);

  template <typename dataT, access::mode accessMode>
  accessor<dataT, dimensions, accessMode, access::target::host_unsampled_image>
  get_access();

  template <typename Destination = std::nullptr_t>
  void set_final_data(Destination finalData = std::nullptr);

  void set_write_back(bool flag = true);
};
}  // namespace sycl
```

| Constructor | Description |
|---|---|
| `unsampled_image(image_format format, const range<dimensions> &rangeRef, const property_list &propList = {})` | Construct a SYCL `unsampled_image` instance with uninitialized memory. The constructed SYCL `unsampled_image` will use a default constructed `AllocatorT` when allocating memory on the host. The element size of the constructed SYCL `unsampled_image` will be derived from the `format` parameter. The range of the constructed SYCL `unsampled_image` is specified by the `rangeRef` parameter provided. The pitch of the constructed SYCL `unsampled_image` will be the default size determined by the SYCL runtime. Unless the member function `set_final_data()` is called with a valid non-null pointer there will be no write back on destruction. Zero or more properties can be provided to the constructed SYCL `unsampled_image` via an instance of `property_list`. |

Continued on next page

Table 4.36: Constructors of the `unsampled_image` class template.

| Constructor | Description |
| --- | --- |
| `unsampled_image(image_format format, const range<dimensions> &rangeRef, AllocatorT allocator, const property_list &propList = {})` | Construct a SYCL `unsampled_image` instance with uninitialized memory. The constructed SYCL `unsampled_image` will use the `allocator` parameter provided when allocating memory on the host. The element size of the constructed SYCL `unsampled_image` will be derived from the `format` parameter. The range of the constructed SYCL `unsampled_image` is specified by the `rangeRef` parameter provided. The pitch of the constructed SYCL `unsampled_image` will be the default size determined by the SYCL runtime. Unless the member function `set_final_data()` is called with a valid non-null pointer there will be no write back on destruction. Zero or more properties can be provided to the constructed SYCL `unsampled_image` via an instance of `property_list`. |
| `unsampled_image(image_format format, const range<dimensions> &rangeRef, const range<dimensions-1> &pitch, const property_list &propList = {})` | Available only when: `dimensions > 1`. Construct a SYCL `unsampled_image` instance with ininitialized memory. The constructed SYCL `unsampled_image` will use a default constructed `AllocatorT` when allocating memory on the host. The element size of the constructed SYCL `unsampled_image` will be derived from the `format` parameter. The range of the constructed SYCL `unsampled_image` is specified by the `rangeRef` parameter provided. The pitch of the constructed SYCL `unsampled_image` will be the `pitch` parameter provided. Unless the member function `set_final_data()` is called with a valid non-null pointer there will be no write back on destruction. Zero or more properties can be provided to the constructed SYCL `unsampled_image` via an instance of `property_list`. |

Continued on next page

Table 4.36: Constructors of the `unsampled_image` class template.

| Constructor | Description |
| --- | --- |
| `unsampled_image(image_format format, const range<dimensions> &rangeRef, const range<dimensions-1> &pitch, AllocatorT allocator, const property_list &propList = {})` | Available only when: `dimensions > 1`. Construct a SYCL `unsampled_image` instance with uninitialized memory. The constructed SYCL `unsampled_image` will use the `allocator` parameter provided when allocating memory on the host. The element size of the constructed SYCL `unsampled_image` will be derived from the `format` parameter. The range of the constructed SYCL `unsampled_image` is specified by the `rangeRef` parameter provided. The pitch of the constructed SYCL `unsampled_image` will be the `pitch` parameter provided. Unless the member function `set_final_data()` is called with a valid non-null pointer there will be no write back on destruction. Zero or more properties can be provided to the constructed SYCL `unsampled_image` via an instance of `property_list`. |
| `unsampled_image(void *hostPointer, image_format format, const range<dimensions> &rangeRef, const property_list &propList = {})` | Construct a SYCL `unsampled_image` instance with the `hostPointer` parameter provided. The ownership of this memory is given to the constructed SYCL `unsampled_image` for the duration of its lifetime. The constructed SYCL `unsampled_image` will use a default constructed `AllocatorT` when allocating memory on the host. The element size of the constructed SYCL `unsampled_image` will be derived from the `format` parameter. The range of the constructed SYCL `unsampled_image` is specified by the `rangeRef` parameter provided. The pitch of the constructed SYCL `unsampled_image` will be the default size determined by the SYCL runtime. Unless the member function `set_final_data()` is called with a valid non-null pointer any memory allocated by the SYCL runtime is written back to `hostPointer`. Zero or more properties can be provided to the constructed SYCL `unsampled_image` via an instance of `property_list`. |

Continued on next page

Table 4.36: Constructors of the `unsampled_image` class template.

| Constructor | Description |
|---|---|
| `unsampled_image(void *hostPointer, image_format format, const range<dimensions> &rangeRef, AllocatorT allocator, const property_list &propList = {})` | Construct a SYCL `unsampled_image` instance with the `hostPointer` parameter provided. The ownership of this memory is given to the constructed SYCL `unsampled_image` for the duration of its lifetime. The constructed SYCL `unsampled_image` will use the `allocator` parameter provided when allocating memory on the host. The element size of the constructed SYCL `unsampled_image` will be derived from the `format` parameter. The range of the constructed SYCL `unsampled_image` is specified by the `rangeRef` parameter provided. The pitch of the constructed SYCL `unsampled_image` will be the default size determined by the SYCL runtime. Unless the member function `set_final_data()` is called with a valid non-null pointer any memory allocated by the SYCL runtime is written back to `hostPointer`. Zero or more properties can be provided to the constructed SYCL `unsampled_image` via an instance of `property_list`. |
| `unsampled_image(void *hostPointer, image_format format, const range<dimensions> &rangeRef, const range<dimensions-1> &pitch, const property_list &propList = {})` | Available only when: `dimensions > 1`. Construct a SYCL `unsampled_image` instance with the `hostPointer` parameter provided. The ownership of this memory is given to the constructed SYCL `unsampled_image` for the duration of its lifetime. The constructed SYCL `unsampled_image` will use a default constructed `AllocatorT` when allocating memory on the host. The element size of the constructed SYCL `unsampled_image` will be derived from the `format` parameter. The range of the constructed SYCL `unsampled_image` is specified by the `rangeRef` parameter provided. The pitch of the constructed SYCL `unsampled_image` will be the `pitch` parameter provided. Unless the member function `set_final_data()` is called with a valid non-null pointer any memory allocated by the SYCL runtime is written back to `hostPointer`. Zero or more properties can be provided to the constructed SYCL `unsampled_image` via an instance of `property_list`. |
| Continued on next page | |

Table 4.36: Constructors of the `unsampled_image` class template.

| Constructor | Description |
| --- | --- |
| `unsampled_image(void *hostPointer, image_format format, const range<dimensions> &rangeRef, const range<dimensions-1> &pitch, AllocatorT allocator, const property_list &propList = {})` | Available only when: `dimensions > 1`. Construct a SYCL `unsampled_image` instance with the `hostPointer` parameter provided. The ownership of this memory is given to the constructed SYCL `unsampled_image` for the duration of its lifetime. The constructed SYCL `unsampled_image` will use the `allocator` parameter provided when allocating memory on the host. The element size of the constructed SYCL `unsampled_image` will be derived from the `format` parameter. The range of the constructed SYCL `unsampled_image` is specified by the `rangeRef` parameter provided. The pitch of the constructed SYCL `unsampled_image` will be the `pitch` parameter provided. Unless the member function `set_final_data()` is called with a valid non-null pointer any memory allocated by the SYCL runtime is written back to `hostPointer`. Zero or more properties can be provided to the constructed SYCL `unsampled_image` via an instance of `property_list`. |
| `unsampled_image(std::shared_ptr<void>& hostPointer, image_format format, const range<dimensions> &rangeRef, const property_list &propList = {})` | Construct a SYCL `unsampled_image` instance with the `hostPointer` parameter provided. The ownership of this memory is given to the constructed SYCL `unsampled_image` for the duration of its lifetime. The constructed SYCL `unsampled_image` will use a default constructed `AllocatorT` when allocating memory on the host. The element size of the constructed SYCL `unsampled_image` will be derived from the `format` parameter. The range of the constructed SYCL `unsampled_image` is specified by the `rangeRef` parameter provided. The pitch of the constructed SYCL `unsampled_image` will be the default size determined by the SYCL runtime. Unless the member function `set_final_data()` is called with a valid non-null pointer any memory allocated by the SYCL runtime is written back to `hostPointer`. Zero or more properties can be provided to the constructed SYCL `unsampled_image` via an instance of `property_list`. |

Continued on next page

Table 4.36: Constructors of the `unsampled_image` class template.

| Constructor | Description |
| --- | --- |
| `unsampled_image(std::shared_ptr<void>& hostPointer, image_format format, const range<dimensions> &rangeRef, AllocatorT allocator, const property_list &propList = {})` | Construct a SYCL `unsampled_image` instance with the `hostPointer` parameter provided. The ownership of this memory is given to the constructed SYCL `unsampled_image` for the duration of its lifetime. The constructed SYCL `unsampled_image` will use the `allocator` parameter provided when allocating memory on the host. The element size of the constructed SYCL `unsampled_image` will be derived from the `format` parameter. The range of the constructed SYCL `unsamlped_image` is specified by the `rangeRef` parameter provided. The pitch of the constructed SYCL `unsampled_image` will be the default size determined by the SYCL runtime. Unless the member function `set_final_data()` is called with a valid non-null pointer any memory allocated by the SYCL runtime is written back to `hostPointer`. Zero or more properties can be provided to the constructed SYCL `unsampled_image` via an instance of `property_list`. |
| `unsampled_image(std::shared_ptr<void>& hostPointer, image_format format, const range<dimensions> &rangeRef, const range<dimensions-1> & pitch, const property_list &propList = {})` | Construct a SYCL `unsampled_image` instance with the `hostPointer` parameter provided. The ownership of this memory is given to the constructed SYCL `unsampled_image` for the duration of its lifetime. The constructed SYCL `unsampled_image` will use a default constructed `AllocatorT` when allocating memory on the host. The element size of the constructed SYCL `unsampled_image` will be derived from the `format` parameter. The range of the constructed SYCL `unsampled_image` is specified by the `rangeRef` parameter provided. The pitch of the constructed SYCL `unsampled_image` will be the `pitch` parameter provided. Unless the member function `set_final_data()` is called with a valid non-null pointer any memory allocated by the SYCL runtime is written back to `hostPointer`. Zero or more properties can be provided to the constructed SYCL `unsampled_image` via an instance of `property_list`. |

Continued on next page

Table 4.36: Constructors of the `unsampled_image` class template.

| Constructor | Description |
| --- | --- |
| `unsampled_image(std::shared_ptr<void>& hostPointer, image_format format, const range<dimensions> &rangeRef, const range<dimensions-1> & pitch, AllocatorT allocator, const property_list &propList = {})` | Construct a SYCL `unsampled_image` instance with the `hostPointer` parameter provided. The ownership of this memory is given to the constructed SYCL `unsampled_image` for the duration of its lifetime. The constructed SYCL `unsampled_image` will use the `allocator` parameter provided when allocating memory on the host. The element size of the constructed SYCL `unsampled_image` will be derived from the `format` parameter. The range of the constructed SYCL `unsampled_image` is specified by the `rangeRef` parameter provided. The pitch of the constructed SYCL `unsampled_image` will be the `pitch` parameter provided. Unless the member function `set_final_data()` is called with a valid non-null pointer any memory allocated by the SYCL runtime is written back to `hostPointer`. Zero or more properties can be provided to the constructed SYCL `unsampled_image` via an instance of `property_list`. |
| | End of table |

Table 4.36: Constructors of the `unsampled_image` class template.

| Member function | Description |
| --- | --- |
| `range<dimensions> get_range()const` | Return a range object representing the size of the image in terms of the number of elements in each dimension as passed to the constructor. |
| `range<dimensions-1> get_pitch()const` | Available only when: `dimensions > 1`. Return a range object representing the pitch of the image in bytes. |
| `size_t get_count()const` | Returns the total number of elements in the image. Equal to `get_range()[0] * ... * get_range()[dimensions-1]`. |
| `size_t get_size()const` | Returns the size of the image storage in bytes. The number of bytes may be greater than `get_count()*element size` due to padding of elements, rows and slices of the image for efficient access. |
| `AllocatorT get_allocator()const` | Returns the allocator provided to the image. |
| | Continued on next page |

Table 4.37: Member functions of the `unsampled_image` class template.

| Member function | Description |
|---|---|
| `template<typename dataT, access::mode accessMode> accessor<dataT, dimensions, accessMode, access::target::unsampled_image> get_access(handler & commandGroupHandler)` | Returns a valid accessor to the image with the specified access mode and target. The only valid types for `dataT` are `int4`, `uint4`, `float4` and `half4`. |
| `template<typename dataT, access::mode accessMode> accessor<dataT, dimensions, accessMode, access::target::host_unsampled_image> get_access()` | Returns a valid accessor to the image with the specified access mode and target. The only valid types for `dataT` are `int4`, `uint4`, `float4` and `half4`. |
| `template <typename Destination = std::nullptr_t> void set_final_data(Destination finalData = nullptr)` | The `finalData` point to where the output of all the image processing is going to be copied to at destruction time, if the image was involved with a write accessor.<br>Destination can be either an output iterator, a `std::weak_ptr<T>`.<br>Note that a raw pointer is a special case of output iterator and thus defines the host memory to which the result is to be copied.<br>In the case of a weak pointer, the output is not copied if the weak pointer has expired.<br>If `Destination` is `std::nullptr_t`, then the copy back will not happen. |
| `void set_write_back(bool flag = true)` | This method allows dynamically forcing or canceling the write-back of the data of an image on destruction according to the value of `flag`.<br>Forcing the write-back is similar to what happens during a normal write-back as described in § 4.7.3.4 and 4.7.4.<br>If there is nowhere to write-back, using this function does not have any effect. |
| | End of table |

Table 4.37: Member functions of the `unsampled_image` class template.

#### 4.7.3.2 Sampled image interface

4.7.8

Each constructor of the `sampled_image` class requires a pointer to the host data the image will sample, an `image_format` to describe the data layout and an `image_sampler` to describe how to sample the image data.

Each constructor addtionally takes as the last parameter an optional SYCL `property_list` to provide properties to the SYCL `sampled_image`.

```
namespace sycl {
namespace property {
namespace image {
class use_host_ptr {
  public:
```

```
    use_host_ptr() = default;
};

class use_mutex {
  public:
    use_mutex(std::mutex &mutexRef);

    std::mutex *get_mutex_ptr() const;
};

class context_bound {
  public:
    context_bound(context boundContext);

    context get_context() const;
};
}  // namespace image
}  // namespace property

enum class image_format : unsigned int {
  r8g8b8a8_unorm,
  r16g16b16a16_unorm,
  r8g8b8a8_sint,
  r16g16b16a16_sint,
  r32b32g32a32_sint,
  r8g8b8a8_uint,
  r16g16b16a16_uint,
  r32b32g32a32_uint,
  r16b16g16a16_sfloat,
  r32g32b32a32_sfloat,
  b8g8r8a8_unorm,
};

using byte = unsigned char;

template <int dimensions = 1, typename AllocatorT = sycl::image_allocator>
class sampled_image {
 public:
  sampled_image(const void *hostPointer, image_format format,
                image_sampler sampler, const range<dimensions> &rangeRef,
                const property_list &propList = {});

  /* Available only when: dimensions > 1 */
  sampled_image(const void *hostPointer, image_format format,
                image_sampler sampler, const range<dimensions> &rangeRef,
                const range<dimensions -1> &pitch,
                const property_list &propList = {});

  sampled_image(std::shared_ptr<const void> &hostPointer, image_format format,
                image_sampler sampler, const range<dimensions> &rangeRef,
                const property_list &propList = {});

  /* Available only when: dimensions > 1 */
  sampled_image(std::shared_ptr<const void> &hostPointer, image_format format,
                image_sampler sampler, const range<dimensions> &rangeRef,
```

```
                const range<dimensions -1> &pitch,
                const property_list &propList = {});

  /* -- common interface members -- */

  /* -- property interface members -- */

  range<dimensions> get_range() const;

  /* Available only when: dimensions > 1 */
  range<dimensions - 1> get_pitch() const;

  size_t get_count() const;

  size_t get_size() const;

  template <typename dataT, access::mode accessMode>
  accessor<dataT, dimensions, accessMode, access::target::sampled_image>
  get_access(handler & commandGroupHandler);

  template <typename dataT, access::mode accessMode>
  accessor<dataT, dimensions, accessMode, access::target::host_sampled_image>
  get_access();
};
} // namespace sycl
```

| Constructor | Description |
| --- | --- |
| `sampled_image(const void *hostPointer, image_format format, image_sampler sampler, const range<dimensions> &rangeRef, const property_list &propList = {})` | Construct a SYCL `sampled_image` instance with the `hostPointer` parameter provided. The ownership of this memory is given to the constructed SYCL `sampled_image` for the duration of its lifetime. The host address is `const T`, so the host accesses must be read-only. Since, the `hostPointer` is const, this image is only initialized with this memory and there is no write after its destruction. The element size of the constructed SYCL `sampled_image` will be derived from the `format` parameter. The sampling method of the constructed SYCL `sampled_image` will be derived from the `sampler` parameter. The range of the constructed SYCL `sampled_image` is specified by the `rangeRef` parameter provided. The pitch of the constructed SYCL `sampled_image` will be the default size determined by the SYCL runtime. Zero or more properties can be provided to the constructed SYCL `sampled_image` via an instance of `property_list`. |

Continued on next page

Table 4.38: Constructors of the `sampled_image` class template.

| Constructor | Description |
|---|---|
| `sampled_image(const void *hostPointer, image_format format, image_sampler sampler, const range<dimensions> &rangeRef, const range<dimensions-1> &pitch, const property_list &propList = {})` | Available only when: `dimensions > 1`. Construct a SYCL `sampled_image` instance with the `hostPointer` parameter provided. The ownership of this memory is given to the constructed SYCL `sampled_image` for the duration of its lifetime. The host address is `const T`, so the host accesses must be read-only. Since, the `hostPointer` is `const`, this image is only initialized with this memory and there is no write after destruction. The element size of the constructed SYCL `sampled_image` will be derived from the `format` parameter. The sampling method of the constructed SYCL `sampled_image` will be derived from the `sampler` parameter. The range of the constructed SYCL `sampled_image` is specified by the `rangeRef` parameter provided. The pitch of the constructed SYCL `sampled_image` will be the `pitch` parameter provided. Zero or more properties can be provided to the constructed SYCL `sampled_image` via an instance of `property_list`. |
| `sampled_image(std::shared_ptr<const void>& hostPointer, image_format format, image_sampler sampler, const range<dimensions> &rangeRef, const property_list &propList = {})` | Construct a SYCL `sampled_image` instance with the `hostPointer` parameter provided. The ownership of this memory is given to the constructed SYCL `sampled_image` for the duration of its lifetime. The host address is `const T`, so the host accesses must be read-only. Since, the `hostPointer` is const, this image is only initialized with this memory and there is no write after its destruction. The element size of the constructed SYCL `sampled_image` will be derived from the `format` parameter. The sampling method of the constructed SYCL `sampled_image` will be derived from the `sampler` parameter. The range of the constructed SYCL `sampled_image` is specified by the `rangeRef` parameter provided. The pitch of the constructed SYCL `sampled_image` will be the default size determined by the SYCL runtime. Zero or more properties can be provided to the constructed SYCL `sampled_image` via an instance of `property_list`. |

Continued on next page

Table 4.38: Constructors of the `sampled_image` class template.

| Constructor | Description |
|---|---|
| `sampled_image(std::shared_ptr<const void>& hostPointer, image_format format, image_sampler sampler, const range<dimensions> &rangeRef, const range<dimensions-1> & pitch, const property_list &propList = {})` | Construct a SYCL sampled_image instance with the hostPointer parameter provided. The ownership of this memory is given to the constructed SYCL sampled_image for the duration of its lifetime. The host address is const T, so the host accesses can be read-only. Since, the hostPointer is const, this image is only initialized with this memory and there is no write after its destruction. The element size of the constructed SYCL sampled_image will be derived from the format parameter. The sampling method of the constructed SYCL sampled_image will be derived from the sampler parameter. The range of the constructed SYCL sampled_image is specified by the rangeRef parameter provided. The pitch of the constructed SYCL sampled_image will be the pitch parameter provided. Zero or more properties can be provided to the constructed SYCL sampled_image via an instance of property_list. |
| | End of table |

Table 4.38: Constructors of the sampled_image class template.

| Member function | Description |
|---|---|
| `range<dimensions> get_range()const` | Return a range object representing the size of the image in terms of the number of elements in each dimension as passed to the constructor. |
| `range<dimensions-1> get_pitch()const` | Available only when: `dimensions > 1`. Return a range object representing the pitch of the image in bytes. |
| `size_t get_count()const` | Returns the total number of elements in the image. Equal to `get_range()[0] * ... * get_range()[dimensions-1]`. |
| `size_t get_size()const` | Returns the size of the image storage in bytes. The number of bytes may be greater than `get_count()*element size` due to padding of elements, rows and slices of the image for efficient access. |
| `template<typename dataT, access::mode accessMode> accessor<dataT, dimensions, accessMode, access::target::sampled_image> get_access(handler & commandGroupHandler)` | Returns a valid accessor to the image with the specified access mode and target. The only valid types for dataT are int4, uint4, float4 and half4. |
| | Continued on next page |

Table 4.39: Member functions of the sampled_image class template.

| Member function | Description |
|---|---|
| `template<typename dataT, access::mode accessMode> accessor<dataT, dimensions, accessMode, access::target::host_sampled_image> get_access()` | Returns a valid accessor to the image with the specified access mode and target. The only valid types for `dataT` are `int4`, `uint4`, `float4` and `half4`. |
| | End of table |

Table 4.39: Member functions of the `sampled_image` class template.

#### 4.7.3.3 Image properties

The properties that can be provided when constructing the SYCL `unsampled_image` and `sampled_image` classes are describe in Table 4.40.

| Property | Description |
|---|---|
| `property::image::use_host_ptr` | The `use_host_ptr` property adds the requirement that the SYCL runtime must not allocate any memory for the `image` and instead uses the provided host pointer directly. This prevents the SYCL runtime from allocating additional temporary storage on the host. |
| `property::image::use_mutex` | The `use_mutex` property is valid for the SYCL `unsampled_image` and `sampled_image` classes. The property adds the requirement that the memory which is owned by the SYCL `image` can be shared with the application via a `std::mutex` provided to the property. The `std::mutex` `m` is locked by the runtime whenever the data is in use and unlocked otherwise. Data is synchronized with `hostData`, when the `std::mutex` is unlocked by the runtime. |
| `property::image::context_bound` | The `context_bound` property adds the requirement that the SYCL `image` can only be associated with a single SYCL `context` that is provided to the property. |
| | End of table |

Table 4.40: Properties supported by the SYCL image classes.

The constructors and member functions of the image `property` classes are listed in Tables 4.41 and 4.42

| Constructor | Description |
|---|---|
| `property::image::use_host_ptr::use_host_ptr()` | Constructs a SYCL `use_host_ptr` property instance. |
| | Continued on next page |

Table 4.41: Constructors of the image `property` classes.

| Constructor | Description |
|---|---|
| `property::image::use_mutex::use_mutex(std::mutex & mutexRef)` | Constructs a SYCL `use_mutex` property instance with a reference to `mutexRef` parameter provided. |
| `property::image::context_bound::context_bound( context boundContext)` | Constructs a SYCL `context_bound` property instance with a copy of a SYCL `context`. |
| | End of table |

Table 4.41: Constructors of the image `property` classes.

| Member function | Description |
|---|---|
| `std::mutex *property::image::use_mutex:: get_mutex_ptr()const` | Returns the `std::mutex` which was specified when constructing this SYCL `use_mutex` property. |
| `context property::image::context_bound::get_context ()const` | Returns the `context` which was specified when constructing this SYCL `context_bound` property. |
| | End of table |

Table 4.42: Member functions of the image `property` classes.

#### 4.7.3.4 Image synchronization rules

The rules are similar to those described in § 4.7.2.3.

For the lifetime of the image object, the associated host memory must be left available to the SYCL runtime and the contents of the associated host memory is unspecified until the image object is destroyed. If an image object value is copied, then only a reference to the underlying image object is copied. The underlying image object is reference-counted. Only after all image value references to the underlying image object have been destroyed is the actual image object itself destroyed.

If an image object is constructed with associated host memory, then its destructor blocks until all operations in all SYCL queues on that image object have completed. Any modifications to the image data will be copied back, if necessary, to the associated host memory. Any errors occurring during destruction are reported to any associated context's asynchronous error handler. If an image object is constructed with a storage object, then the storage object defines what synchronization or copying behavior occurs on image object destruction.

### 4.7.4 Sharing host memory with the SYCL data management classes

In order to allow the SYCL runtime to do memory management and allow for data dependencies, there are two classes defined, buffer and image. The default behavior for them is that a “raw” pointer is given during the construction of the data management class, with full ownership to use it until the destruction of the SYCL object.

In this section we go in greater detail on sharing or explicitly not sharing host memory with the SYCL data classes, and we will use the buffer class as an example. The same rules will apply to images as well.

#### 4.7.4.1 Default behavior

When using a SYCL buffer, the ownership of the pointer passed to the constructor of the class is, by default, passed to SYCL runtime, and that pointer cannot be used on the host side until the buffer or image is destroyed. A SYCL application can use memory managed by a SYCL buffer within the buffer scope by using a `host accessor` as defined in 4.7.6. However, there is no guarantee that the host accessor synchronizes with the original host address used in its constructor.

The pointer passed in is the one used to copy data back to the host, if needed, before buffer destruction. The memory pointed by host pointer will not be de-allocated by the runtime, and the data is copied back from the device if there is a need for it.

#### 4.7.4.2 SYCL ownership of the host memory

In the case where there is host memory to be used for initialization of data but there is no intention of using that host memory after the buffer is destroyed, then the buffer can take full ownership of that host memory.

When a buffer owns the host pointer there is no copy back, by default. In this situation the SYCL application may pass a unique pointer to the host data, which will be then used by the runtime internally to initialize the data in the device.

For example, the following could be used:

```
{
  auto ptr = std::make_unique<int>(-1234);
  buffer<int, 1> b { std::move(ptr), range { 1 } };
  // ptr is not valid anymore.
  // There is nowhere to copy data back
}
```

However, optionally the `buffer::set_final_data()` can be set to a `std::weak_ptr` to enable copying data back, to another host memory address that is going to be valid after buffer construction.

```
{
  auto ptr = std::make_unique<int>(-42);
  buffer<int, 1> b { std::move(ptr), range { 1 } };
  // ptr is not valid anymore.
  // There is nowhere to copy data back.
  // To get copy back, a location can be specified:
  b.set_final_data(std::weak_ptr<int> { .... })
}
```

#### 4.7.4.3 Shared SYCL ownership of the host memory

When an instance of `std::shared_ptr` is passed to the buffer constructor, then the buffer object and the developer's application share the memory region. If the shared pointer is still used on the application's side then the data will be copied back from the buffer or image and will be available to the application after the buffer or image is destroyed.

If the memory pointed to by the shared object is initialized to some data, then that data is used to initialize the buffer. If the shared pointer is null, the pointer is initialized by the runtime internally (and, therefore, the user can use it afterwards in the host).

When the buffer is destroyed and the data have potentially been updated, if the number of copies of the shared pointer outside the runtime is 0, there is no user-side shared pointer to read the data. Therefore the data is not copied out, and the buffer destructor does not need to wait for the data processes to be finished from OpenCL, as the outcome is not needed on the application's side.

This behavior can be overridden using the `set_final_data()` method of the buffer class, which will by any means force the buffer destructor to wait until the data is copied to wherever the `set_final_data()` method has put the data (or not wait nor copy if set final data is `nullptr`).

```
{
  std::shared_ptr<int> ptr { data };
  {
    buffer<int, 1> b { ptr, range<2>{ 10, 10 } };
    // update the data
    [...]
  } // Data is copied back because there is an user side shared_ptr
}
```

```
{
  std::shared_ptr<int> ptr { data };
  {
    buffer<int, 1> b { ptr, range<2>{ 10, 10 } };
    // update the data
    [...]
    ptr.reset();
  } // Data is not copied back, there is no user side shared_ptr.
}
```

### 4.7.5 Synchronization primitives

When the user wants to use the buffer simultaneously in the SYCL runtime and their own code (e.g. a multi-threaded mechanism) and want to use manual synchronization without host accessors, a pointer to a `std::mutex` can be passed to the buffer constructor.

The runtime promises to lock the mutex whenever the data is in use and unlock it when it no longer needs it.

```
{
  std::mutex m;
  auto shD = std::make_shared<int> { 42 }
  {
    buffer<int, 1> b { shD, m };

    std::lock_guard<std::mutex> lck { m };
    // User accesses the data
    do_something(shD);
    /* m is unlock when lck goes out of scope, by normal end of this
       block but also if an exception is thrown for example */
  }
}
```

When the runtime releases the mutex the user is guaranteed that the data was copied back on the shared pointer

— unless the final data destination has been changed using the member function `set_final_data()`.

## 4.7.6 Accessors

Accessors provide access to the data managed by a buffer or image, or to shared local memory allocated by the runtime. Accessors allow users to define **requirements** to memory objects (see Section 3.7.1). Note that construction of an accessor is what defines a memory object requirement, and these requirements are independent of whether there are any uses of an accessor.

The SYCL `accessor` class template takes the following template parameters:

- A typename specifying the data type that the `accessor` is providing access to.
- An integer specifying the dimensionality of the accessor.
- A value of `access_mode` specifying the mode of access the `accessor` is providing.
- A value of `target` specifying the target of access the `accessor` is providing.

The SYCL `host_accessor` class template takes the following template parameters:

- A typename specifying the data type that the `host_accessor` is providing access to.
- An integer specifying the dimensionality of the accessor.
- A value of `access_mode` specifying the mode of access the `host_accessor` is providing.

The parameters described above determine the data an accessor provides access to and the way in which that access is provided. This separation allows a SYCL runtime implementation to choose an efficient way to provide access to the data within an execution schedule.

Because of this the interfaces of the `accessor` and the `host_accessor` will be different depending on the possible combinations of those parameters.

There are four categories of accessor; buffer device accesors (see Section 4.7.6.9), buffer host accesors (see Section 4.7.6.10), local accessors (see Section 4.7.6.11) and image accessors (see Section 4.7.6.12).

### 4.7.6.1 Access targets

The access target of an `accessor` specifies what the accessor is providing access to.

The `target` enumeration, shown in Table 4.43, describes the potential targets of an `accessor`.

```
namespace sycl {
enum class target {
  global_buffer = 2014,
  constant_buffer,
  local,
  unsampled_image,
  sampled_image,
  host_buffer,
  host_unsampled_image,
  host_sampled_image
};
```

```

namespace access {
  using sycl::target;
} // namespace access
} // namespace sycl
```

| target | Description |
|---|---|
| target::global_buffer | Access buffer via global memory. |
| target::constant_buffer | Access buffer via constant memory. |
| target::local | Access work-group local memory. |
| target::unsampled_image | Access an unsampled_image. |
| target::sampled_image | Access a sampled_image. |
| target::host_buffer | Access a buffer immediately in host code. |
| target::host_unsampled_image | Access a host_unsampled_image immediately in host code. |
| target::host_sampled_image | Access a host_sampled_image immediately in host code. |
| | End of table |

Table 4.43: Enumeration of access modes available to accessors.

#### 4.7.6.2 Access modes

The access mode of an accessor specifies the kind of access that is being provided. This information is used by the runtime to ensure that any data dependencies are resolved by enqueuing any data transfers before or after the execution of a kernel. If a user wants to modify only certain parts of a buffer, preserving other parts of the buffer, then the user should specify the exact sub-range of modification of the buffer.

The access_mode enumeration, shown in Table 4.44, describes the potential modes of an accessor.

```
namespace sycl {
enum class access_mode {
  read = 1024,
  write,
  read_write,
  discard_write,      // Deprecated in SYCL 2020
  discard_read_write, // Deprecated in SYCL 2020
  atomic              // Deprecated in SYCL 2020
};

namespace access {
  using sycl::access_mode;
}
} // namespace sycl
```

| access_mode | Description |
|---|---|
| access_mode::read | Read-only access. |
| access_mode::write | Write-only access. Previous contents not discarded. |
| | Continued on next page |

Table 4.44: Enumeration of access modes available to accessors.

| access_mode | Description |
|---|---|
| access_mode::read_write | Read and write access. |
| access_mode::discard_write | Deprecated in SYCL 2020. Write-only access. Previous contents discarded. |
| access_mode::discard_read_write | Deprecated in SYCL 2020. Read and write access. Previous contents discarded. |
| access_mode::atomic | Read and write atomic access. Deprecated in SYCL 2020. |
| | End of table |

Table 4.44: Enumeration of access modes available to accessors.

#### 4.7.6.3 Access tags

The access mode and access target can be specified via passing tag types to an accessor or host_accessor constructor. This type is used to deduce template arguments of a class.

The Table 4.45, describes the potential tag types and values modes.

SYCL implementations shall provide sufficient list of deduction guides for all template arguments to be deduced for all accessor constructors except for default one and dimension = 0. If accessor template arguments are specified by user, but constructor is called with non-matching tag, the SYCL implementation must emit a compilation error. If a const data type is specified as an accessor template argument and constructor is called with write_only tag, the SYCL implementation must emit a compilation error.

```
namespace sycl {
template <access_mode>
struct mode_tag_t {
  explicit mode_tag_t() = default;
};

inline constexpr mode_tag_t<access_mode::read>       read_only{};
inline constexpr mode_tag_t<access_mode::read_write> read_write{};
inline constexpr mode_tag_t<access_mode::write>      write_only{};

template <access_mode, target>
struct mode_target_tag_t {
  explicit mode_target_tag_t() = default;
};

inline constexpr mode_target_tag_t<access_mode::read, target::constant_buffer> read_constant{};
}  // namespace sycl
```

#### 4.7.6.4 Device and host accessors

An accessor can be a device accessor in which case it provides access to data within a SYCL kernel function, or a host_accessor in which case it provides immediate access on the host.

If an accessor has the access target target::global_buffer, target::constant_buffer, target::local, target::unsampled_image or target::sampled_image then it is considered a device accessor, and therefore can only be used within a SYCL kernel function and must be associated with a command group. Creating a device

| Tag type | Tag value | Access modes | Accessor target |
|---|---|---|---|
| mode_tag_t | read_write | access_mode::read_write | default |
| mode_tag_t | read_only | access_mode::read | default |
| mode_tag_t | write_only | access_mode::write | default |
| mode_target_tag_t | read_constant | access_mode::read | target::constant_buffer |

Table 4.45: Enumeration of access tags available to accessors.

accessor is a non-blocking operation which defines a requirement on the device and adds the requirement to the queue.

A host_accessor or an accessor with the access target target::host_unsampled_image or target::host_sampled_image is considered a host accessor and can only be used on the host. Creating a host accessor is a blocking operation, if created without providing a handler for a command group. Blocking operation defines a requirement on the host and blocks the caller until the requirement is satisfied. Creating a host accessor with a handler makes it a non-blocking operation.

A blocking accessor provides immediate access and continues to provide access until it is destroyed.

#### 4.7.6.5 Placeholder accessor

Certain accessor types are allowed to be *placeholder* accessors. A *placeholder* accessor defines an accessor instance that is not bound to a specific command group. The accessor defines only the type of the accessor (target memory, access mode, base type, ...). When associated with a command group using the appropriate handler interface, it defines a **requirement** for the command group. The same placeholder accessor can be required by multiple command groups.

Only the following access targets are allowed in placeholder accessors:

- target::global_buffer
- target::constant_buffer

4.7.6.5 specifies enum class placeholder, which can be used as an accessor template parameter isPlaceholder. This accessor template parameter has been deprecated in SYCL 2020 and placeholder semantics apply regardless of the value isPlaceholder.

```
namespace sycl {
enum class placeholder { // Deprecated in SYCL 2020
  false_t,
  true_t,
};
}  // namespace sycl
```

#### 4.7.6.6 Accessor declaration

The declaration for the accessor and the host_accessor classes is provided in 4.7.6.6.

```
namespace sycl {
template <typename dataT,
```

```
          int dimensions = 1,
          access_mode accessmode =
            (std::is_const_v<dataT> ? access_mode::read
                                    : access_mode::read_write),
          target accessTarget = target::global_buffer,
          access::placeholder isPlaceholder = access::placeholder::false_t // Deprecated in SYCL
              2020
>
class accessor;

template <typename dataT,
          int dimensions = 1,
          access_mode accessmode =
            (std::is_const_v<dataT> ? access_mode::read
                                    : access_mode::read_write)
>
class host_accessor;
}  // namespace sycl
```

#### 4.7.6.7 Constness of the accessor data type

An `accessor` or `host_accessor` can be constructed with the underlying data type being `const`, resulting in an accessor of `const dataT`. Having an accessor of `const dataT` is semantically equivalent to having an accessor of `access_mode::read`: they are both read-only accessors.

Some access modes contradict the constness of the data. An underlying data type `const dataT` is only valid with the following access modes:

- `access_mode::read`
- `access_mode::read_write`

Even when the access mode for an accessor is `access_mode::read_write`, adding `const` to `dataT` makes it a read-only accessor. This ensures, among other things, that the read-only accessor will not trigger a copy-back that `access_mode::read_write` normally would require.

Using `const dataT` makes the accessor read-only by default, as shown in the following example:

```
accessor<const int> acc;
static_assert(std::is_same_v<
  decltype(acc),
  accessor<const int, 1, access_mode::read, target::global_buffer>
>);
```

#### 4.7.6.8 Implicit accessor conversions

It is valid to implicitly convert a writable accessor to a read-only one by doing at least one of the following:

- Converting data type from non-const `dataT` to `const dataT`
- Converting access mode from `access_mode::read_write` to `access_mode::read`

Because of the semantic equivalence defined in 4.7.6.7, the following `accessor` types can be implicitly converted to one another:

- accessor<dataT, dimensions, access_mode::read, accessTarget>
- accessor<const dataT, dimensions, access_mode::read, accessTarget>
- accessor<const dataT, dimensions, access_mode::read_write, accessTarget>

And the following host_accessor types can be implicitly converted to one another:

- host_accessor<dataT, dimensions, access_mode::read, accessTarget>
- host_accessor<const dataT, dimensions, access_mode::read, accessTarget>
- host_accessor<const dataT, dimensions, access_mode::read_write, accessTarget>

#### 4.7.6.9 Device buffer accessor

A device buffer accessor provides access to a SYCL buffer instance on a device. A SYCL accessor is considered a device buffer accessor if it has the access target target::global_buffer, or target::constant_buffer.

A device buffer accessor can provide access to memory managed by a SYCL buffer class via either global memory or constant memory, corresponding to the access targets target::global_buffer and target::constant_buffer respectively. A device buffer accessor accessing a SYCL buffer via constant memory is restricted by the available constant memory available on the SYCL device being executed on.

The data type of an accessor must match that of the SYCL buffer which it is accessing.

The dimensionality of a buffer accessor must match that of the SYCL buffer which it is accessing, with the exception of 0 in which case the dimensionality of the SYCL buffer must be 1.

There are three ways a SYCL accessor can provide access to the elements of a SYCL buffer. Firstly by passing a SYCL id instance of the same dimensionality as the SYCL accessor subscript operator. Secondly by passing a single size_t value to multiple consecutive subscript operators (one for each dimension of the SYCL accessor, for example acc[id0][id1][id2]). Finally, in the case of the SYCL accessor being 0 dimensions, by triggering the implicit conversion operator. Whenever a multi-dimensional index is passed to a SYCL accessor the linear index is calculated based on the index {id0, id1, id2} provided and the range of the SYCL accessor {r0, r1, r2} according to row-major ordering as follows:

$$id2 + (id1 \cdot r2) + (id0 \cdot r2 \cdot r1) \tag{4.3}$$

An accessor can optionally provide access to a sub range of a SYCL buffer by providing a range and offset on construction. In this case the SYCL runtime will only guarantee the latest copy of the data is available in that given range and any modifications outside that range are considered undefined behavior. This allows the SYCL runtime to perform optimizations such as reducing copies between devices. The indexing performed when a SYCL accessor provides access to the elements of a SYCL buffer is unaffected, i.e, the accessor will continue to index from {0,0,0}. This allows the offset to be provided either manually or via the parallel_for as in 4.7.6.9.

```
myQueue.submit([&](handler &cgh) {
  auto singleRange = range<3>(8, 16, 16);
  auto offset = id<3>(8, 0, 0);
  // We define the subset of the accessor we require for the kernel
  accessor ptr(syclBuffer, cgh, singleRange, offset);
  // We offset the kernel by the same value to match indexes
```

```
    cgh.parallel_for<kernel>(singleRange, offset, [=](item<3> itemID) {
      ptr[itemID.get_linear_id()] = 2;
    });
  });
```

An accessor with access target target::global_buffer can optionally provide atomic access to a SYCL buffer, using the access mode access_mode::atomic, in which case all operators which return an element of the SYCL buffer return an instance of the deprecated cl::sycl::atomic class. This functionality is provided for backwards compatibility and will be removed in a future version of SYCL.

A device buffer accessor meets the C++ requirement of ContiguousContainer and ReversibleContainer. The exception to this is that the device buffer accessor destructor doesn't destroy any elements or free the memory, because an accessor doesn't own the underlying data. The iterator for the container interface is the same pointer type as obtained by calling get_multi_ptr<access::decorated::no>(). For multidimensional accessors the iterator linearizes the data according to 4.3.

The full list of capabilities that device buffer accessors can support is described in 4.46.

| Access target | Access modes | Data types | Dimensionalities |
|---|---|---|---|
| global_buffer | read<br>write<br>read_write<br>atomic | The data type of the SYCL buffer being accessed. | Between 0 and 3 (inclusive). |
| constant_buffer | read | The data type of the SYCL buffer being accessed. | Between 0 and 3 (inclusive). |

Table 4.46: Description of all the device buffer accessor capabilities.

#### 4.7.6.9.1 Device buffer accessor interface

A synopsis of the SYCL accessor class template buffer specialization is provided below. The member types for this accessor specialization are listed in Tables 4.47. The constructors for this accessor specialization are listed in Tables 4.48. The member functions for this accessor specialization are listed in Tables 4.49. The additional common special member functions and common member functions are listed in 4.5.3 in Tables 4.1 and 4.2, respectively. For valid implicit conversions between accessor types please refer to 4.7.6.8. Additionally, accessors of the same type must be equality comparable not only on the host application, but also within SYCL kernel functions.

```
namespace sycl {
template <typename dataT,
          int dimensions,
          access::mode accessmode,
          access::target accessTarget,
          access::placeholder isPlaceholder>
class accessor {
 public:
  template <access::decorated IsDecorated>
  using accessor_ptr =   // Corresponds to the target address space,
      __pointer_class__; // is pointer-to-const
                         // when (accessmode == access::mode::read);
  using value_type =  // const dataT when (accessmode == access::mode::read),
      __value_type__; // dataT otherwise
  using reference =       // const dataT& when (accessmode == access::mode::read),
```

```
    __reference_type__; // dataT& otherwise
using const_reference = const dataT &;
using iterator =      // Corresponds to the target address space,
    __pointer_type__; // is pointer-to-const
                      // when (accessmode == access::mode::read)
using const_iterator =
    __pointer_to_const_type__; // Corresponds to the target address space
using reverse_iterator = std::reverse_iterator<iterator>;
using const_reverse_iterator = std::reverse_iterator<const_iterator>;
using difference_type =
    typename std::iterator_traits<iterator>::difference_type;
using size_type = size_t;

accessor();

/* Available only when: (dimensions == 0) */
template <typename AllocatorT>
accessor(buffer<dataT, 1, AllocatorT> &bufferRef,
         const property_list &propList = {});

/* Available only when: (dimensions == 0) */
template <typename AllocatorT>
accessor(buffer<dataT, 1, AllocatorT> &bufferRef,
         handler &commandGroupHandlerRef, const property_list &propList = {});

/* Available only when: (dimensions > 0) */
template <typename AllocatorT>
accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
         const property_list &propList = {});

/* Available only when: (dimensions > 0) */
template <typename AllocatorT, typename TagT>
accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef, TagT tag,
         const property_list &propList = {});

/* Available only when: (dimensions > 0) */
template <typename AllocatorT>
accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
         handler &commandGroupHandlerRef, const property_list &propList = {});

/* Available only when: (dimensions > 0) */
template <typename AllocatorT, typename TagT>
accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
         handler &commandGroupHandlerRef, TagT tag,
         const property_list &propList = {});

/* Available only when: (dimensions > 0) */
template <typename AllocatorT>
accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
         range<dimensions> accessRange, const property_list &propList = {});

/* Available only when: (dimensions > 0) */
template <typename AllocatorT, typename TagT>
accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
         range<dimensions> accessRange, TagT tag,
```

```
          const property_list &propList = {});

/* Available only when: (dimensions > 0) */
template <typename AllocatorT>
accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
         range<dimensions> accessRange, id<dimensions> accessOffset,
         const property_list &propList = {});

/* Available only when: (dimensions > 0) */
template <typename AllocatorT, typename TagT>
accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
         range<dimensions> accessRange, id<dimensions> accessOffset,
         TagT tag, const property_list &propList = {});

/* Available only when: (dimensions > 0) */
template <typename AllocatorT>
accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
         handler &commandGroupHandlerRef, range<dimensions> accessRange,
         const property_list &propList = {});

/* Available only when: (dimensions > 0) */
template <typename AllocatorT, typename TagT>
accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
         handler &commandGroupHandlerRef, range<dimensions> accessRange,
         TagT tag, const property_list &propList = {});

/* Available only when: (dimensions > 0) */
template <typename AllocatorT>
accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
         handler &commandGroupHandlerRef, range<dimensions> accessRange,
         id<dimensions> accessOffset, const property_list &propList = {});

/* Available only when: (dimensions > 0) */
template <typename AllocatorT, typename TagT>
accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
         handler &commandGroupHandlerRef, range<dimensions> accessRange,
         id<dimensions> accessOffset, TagT tag,
         const property_list &propList = {});

/* -- common interface members -- */

void swap(accessor &other);

bool is_placeholder() const;

size_type byte_size() const noexcept;

size_type size() const noexcept;

size_type max_size() const noexcept;

size_type get_count() const noexcept;

bool empty() const noexcept;

```

```
  /* Available only when: dimensions > 0 */
  range<dimensions> get_range() const;

  /* Available only when: dimensions > 0 */
  id<dimensions> get_offset() const;

  /* Available only when: (dimensions == 0) */
  operator reference() const;

  /* Available only when: (dimensions > 0) */
  reference operator[](id<dimensions> index) const;

  /* Deprecated in SYCL 2020
  Available only when: accessMode == access::mode::atomic && dimensions ==  0 */
  operator cl::sycl::atomic<dataT, access::address_space::global_space> () const;

  /* Deprecated in SYCL 2020
  Available only when: accessMode == access::mode::atomic && dimensions > 0 */
  cl::sycl::atomic<dataT, access::address_space::global_space> operator[](
    id<dimensions> index) const;

  /* Available only when: dimensions > 1 */
  __unspecified__ &operator[](size_t index) const;

  std::add_pointer_t<value_type> get_pointer() const noexcept;

  template <access::decorated IsDecorated>
  accessor_ptr<IsDecorated> get_multi_ptr() const noexcept;

  iterator data() const noexcept;

  iterator begin() const noexcept;

  iterator end() const noexcept;

  const_iterator cbegin() const noexcept;

  const_iterator cend() const noexcept;
};

} // namespace sycl
```

Listing 4.1: Device accessor class for buffers.

| **Member types** | **Description** |
|---|---|
| `value_type` | If `(accessmode == access_mode::read)`, equal to `const dataT`. In other cases equal to `dataT`. |

Continued on next page

Table 4.47: Member types of the `accessor` class template `buffer` specialization.

| Member types | Description |
|---|---|
| template <access::decorated IsDecorated> accessor_ptr | If (accessTarget == access::target::global_buffer): multi_ptr<value_type, access::address_space::global_space, IsDecorated>. If (accessTarget == access::target::constant_buffer): multi_ptr<value_type, access::address_space::constant_space, IsDecorated>. |
| reference | If (accessmode == access_mode::read), equal to const dataT&. In other cases equal to dataT&. |
| const_reference | const dataT& |
| iterator | If (accessTarget == access::target::global_buffer): raw_global_ptr<value_type>. If (accessTarget == access::target::constant_buffer): raw_constant_ptr<value_type>. |
| const_iterator | If (accessTarget == access::target::global_buffer): raw_global_ptr<const value_type>. If (accessTarget == access::target::constant_buffer): raw_constant_ptr<const value_type>. |
| reverse_iterator | Iterator adaptor that reverses the direction of iterator. |
| const_reverse_iterator | Iterator adaptor that reverses the direction of const_iterator. |
| difference_type | typename std::iterator_traits<iterator>::difference_type |
| size_type | size_t |
| | End of table |

Table 4.47: Member types of the accessor class template buffer specialization.

| Constructor | Description |
|---|---|
| `accessor()` | Constructs an empty accessor. Fulfills the following post-conditions:<br>• `(empty()== true)`<br>• All size queries return `0`.<br>• The only iterator that can be obtained is `nullptr`.<br>• Trying to access the underlying memory is undefined behavior.<br>A default constructed placeholder accessor can be passed to a SYCL kernel, but it is not valid to register it with the command group handler. |
| `accessor(buffer<dataT, 1, AllocatorT> &bufferRef, const property_list &propList = {})` | Available only when: `(dimensions == 0)`. Constructs a placeholder `accessor` instance for accessing a single element of a SYCL `buffer`. The optional `property_list` provides properties for the constructed SYCL `accessor` object. |
| `accessor(buffer<dataT, 1, AllocatorT> &bufferRef, handler &commandGroupHandlerRef, const property_list &propList = {})` | Available only when: `(dimensions == 0)`. Constructs a SYCL `accessor` instance for accessing the first element of a SYCL `buffer` within a SYCL kernel function on the SYCL `queue` associated with `commandGroupHandlerRef`. The optional `property_list` provides properties for the constructed SYCL `accessor` object. |
| `accessor(buffer<dataT, dimensions, AllocatorT> & bufferRef, const property_list &propList = {})` | Available only when: `(dimensions > 0)`. Constructs a placeholder `accessor` for accessing a SYCL `buffer`. The optional `property_list` provides properties for the constructed SYCL `accessor` object. |
| `accessor(buffer<dataT, dimensions, AllocatorT> & bufferRef, TagT tag, const property_list &propList = {})` | Available only when: `(dimensions > 0)`. Constructs a placeholder `accessor` for accessing a SYCL `buffer`. `tag` is used to deduce template arguments of an `accessor`. The optional `property_list` provides properties for the constructed SYCL `accessor` object. |
| `accessor(buffer<dataT, dimensions, AllocatorT> & bufferRef, handler &commandGroupHandlerRef, const property_list &propList = {})` | Available only when: `(dimensions > 0)`. Constructs a SYCL `accessor` instance for accessing a SYCL `buffer` within a SYCL kernel function on the SYCL `queue` associated with `commandGroupHandlerRef`. The optional `property_list` provides properties for the constructed SYCL `accessor` object. |
| | Continued on next page |

Table 4.48: Constructors of the `accessor` class template `buffer` specialization.

| Constructor | Description |
|---|---|
| `accessor(buffer<dataT, dimensions, AllocatorT> & bufferRef, handler &commandGroupHandlerRef, TagT tag, const property_list &propList = {})` | Available only when: `(dimensions > 0)`. Constructs a SYCL `accessor` instance for accessing a SYCL `buffer` within a SYCL kernel function on the SYCL `queue` associated with `commandGroupHandlerRef`. `tag` is used to deduce template arguments of an `accessor`. The optional `property_list` provides properties for the constructed SYCL `accessor` object. |
| `accessor(buffer<dataT, dimensions, AllocatorT> & bufferRef, range<dimensions> accessRange, const property_list &propList = {})` | Available only when: `(dimensions > 0)`. Constructs a placeholder `accessor` for accessing a range of a SYCL `buffer`. The optional `property_list` provides properties for the constructed SYCL `accessor` object. |
| `accessor(buffer<dataT, dimensions, AllocatorT> & bufferRef, range<dimensions> accessRange, TagT tag, const property_list &propList = {})` | Available only when: `(dimensions > 0)`. Constructs a placeholder `accessor` for accessing a range of a SYCL `buffer`. `tag` is used to deduce template arguments of an `accessor`. The optional `property_list` provides properties for the constructed SYCL `accessor` object. |
| `accessor(buffer<dataT, dimensions, AllocatorT> & bufferRef, range<dimensions> accessRange, id<dimensions> accessOffset, const property_list &propList = {})` | Available only when: `(dimensions > 0)`. Constructs a placeholder `accessor` for accessing a range of a SYCL `buffer`. The optional `property_list` provides properties for the constructed SYCL `accessor` object. |
| `accessor(buffer<dataT, dimensions, AllocatorT> & bufferRef, range<dimensions> accessRange, id<dimensions> accessOffset, TagT tag, const property_list &propList = {})` | Available only when: `(dimensions > 0)`. Constructs a placeholder `accessor` for accessing a range of a SYCL `buffer`. `tag` is used to deduce template arguments of an `accessor`. The optional `property_list` provides properties for the constructed SYCL `accessor` object. |
| `accessor(buffer<dataT, dimensions, AllocatorT> & bufferRef, handler &commandGroupHandlerRef, range<dimensions> accessRange, const property_list &propList = {})` | Available only when: `(dimensions > 0)`. Constructs a SYCL `accessor` instance for accessing a range of SYCL `buffer` within a SYCL kernel function on the SYCL `queue` associated with `commandGroupHandlerRef`, specified by `accessRange`. The optional `property_list` provides properties for the constructed SYCL `accessor` object. |

Continued on next page

Table 4.48: Constructors of the `accessor` class template `buffer` specialization.

| Constructor | Description |
|---|---|
| `accessor(buffer<dataT, dimensions, AllocatorT> & bufferRef, handler &commandGroupHandlerRef, range<dimensions> accessRange, TagT tag, const property_list &propList = {})` | Available only when: `(dimensions > 0)`. Constructs a SYCL `accessor` instance for accessing a range of SYCL `buffer` within a SYCL kernel function on the SYCL `queue` associated with `commandGroupHandlerRef`, specified by `accessRange`. `tag` is used to deduce template arguments of an `accessor`. The optional `property_list` provides properties for the constructed SYCL `accessor` object. |
| `accessor(buffer<dataT, dimensions, AllocatorT> & bufferRef, handler &commandGroupHandlerRef, range<dimensions> accessRange, id<dimensions> accessOffset, const property_list &propList = {})` | Available only when: `(dimensions > 0)`. Constructs a SYCL `accessor` instance for accessing a range of SYCL `buffer` within a SYCL kernel function on the SYCL `queue` associated with `commandGroupHandlerRef`, specified by `accessRange` and `accessOffset` . The optional `property_list` provides properties for the constructed SYCL `accessor` object. |
| `accessor(buffer<dataT, dimensions, AllocatorT> & bufferRef, handler &commandGroupHandlerRef, range<dimensions> accessRange, id<dimensions> accessOffset, TagT tag, const property_list &propList = {})` | Available only when: `(dimensions > 0)`. Constructs a SYCL `accessor` instance for accessing a range of SYCL `buffer` within a SYCL kernel function on the SYCL `queue` associated with `commandGroupHandlerRef`, specified by `accessRange` and `accessOffset` . `tag` is used to deduce template arguments of an `accessor`. The optional `property_list` provides properties for the constructed SYCL `accessor` object. |
| | End of table |

Table 4.48: Constructors of the `accessor` class template `buffer` specialization.

| Member function | Description |
|---|---|
| `void swap(accessor &other);` | Swaps the contents of the current accessor with the contents of `other`. |
| `bool is_placeholder()const` | Returns `true` if `(accessTarget != target::host_buffer)` and the accessor has been constructed without a handler. Otherwise returns `false`. |
| `size_type byte_size()const noexcept` | Returns the size in bytes of the region of a SYCL `buffer` that this SYCL `accessor` is accessing. |
| `size_type size()const noexcept` | Returns the number of elements in the region of a SYCL `buffer` that this SYCL `accessor` is accessing. |
| | Continued on next page |

Table 4.49: Member functions of the `accessor` class template buffer specialization.

| Member function | Description |
| --- | --- |
| `size_type max_size()const noexcept` | Returns the maximum number of elements any accessor of this type would be able to access. |
| `size_type get_count()const noexcept` | Returns the same as `size()`. |
| `bool empty()const noexcept` | Returns `true` iff `(size()== 0)`. |
| `range<dimensions> get_range()const` | Available only when: `dimensions > 0`.<br>Returns a `range` object which represents the number of elements of `dataT` per dimension that this `accessor` may access.<br>The `range` object returned must equal to the `range` of the `buffer` this `accessor` is associated with, unless a range was explicitly specified when this `accessor` was constructed. |
| `id<dimensions> get_offset()const` | Available only when: `dimensions > 0`.<br>Returns an `id` object which represents the starting point in number of elements of `dataT` for the `range` that this `accessor` may access.<br>The `id` object returned must equal to `id{0, 0, 0}`, unless an offset was explicitly specified when this `accessor` was constructed. |
| `operator reference()const` | Available only when: `(dimensions == 0)`.<br>Returns a reference to the element stored within the SYCL `buffer` this SYCL `accessor` is accessing. |
| `reference operator[](id<dimensions> index)const` | Available only when: `(dimensions > 0)`.<br>Returns a reference to the element stored within the SYCL `buffer` this SYCL `accessor` is accessing at the index specified by `index`. |
| `operator const dataT &()const` | Available only when: `accessMode == access_mode::read && dimensions == 0)` .<br>Returns a const reference to the element stored within the SYCL `buffer` this SYCL `accessor` is accessing. |
| `const_reference operator[](id<dimensions> index) const` | Available only when: `accessMode == access::_mode::read && dimensions > 0)`.<br>Returns a const reference to the element stored within the SYCL `buffer` this SYCL `accessor` is accessing at the index specified by `index`. |

Continued on next page

Table 4.49: Member functions of the `accessor` class template buffer specialization.

| Member function | Description |
|---|---|
| `operator cl::sycl::atomic<dataT, access::address_space::global_space> ()const` | Deprecated in SYCL 2020. Available only when: `accessMode == access_mode::atomic && dimensions == 0)`. Returns an instance of `cl::sycl::atomic` of type `dataT` providing atomic access to the element stored within the SYCL `buffer` this SYCL `accessor` is accessing. |
| `cl::sycl::atomic<dataT, access::address_space::global_space> operator[](id<dimensions> index)const` | Deprecated in SYCL 2020. Available only when: `accessMode == access_mode::atomic && dimensions > 0)`. Returns an instance of `cl::sycl::atomic` of type `dataT` providing atomic access to the element stored within the SYCL `buffer` this SYCL `accessor` is accessing at the index specified by `index`. |
| `__unspecified__ &operator[](size_t index)const` | Available only when: `dimensions > 1`. Returns an instance of an undefined intermediate type representing a a SYCL `accessor` of the same type as this SYCL `accessor`, with the dimensionality `dimensions-1` and containing an implicit SYCL `id` with index `dimensions` set to `index`. The intermediate type returned must provide all available subscript operators which take a `size_t` parameter defined by the SYCL `accessor` class that are appropriate for the type it represents (including this subscript operator). |
| `std::add_pointer_t<value_type> get_pointer()const noexcept` | Returns a pointer to the memory this SYCL `accessor` memory is accessing. |
| `template <access::decorated IsDecorated> accessor_ptr<IsDecorated> get_multi_ptr()const noexcept` | Returns a `multi_ptr` to the memory this SYCL `accessor` memory is accessing. |
| `iterator data()const noexcept` | Returns a pointer to the memory this SYCL `accessor` memory is accessing. |
| `iterator begin()const noexcept` | Returns an iterator to the first element of the memory within the access range. |
| `iterator end()const noexcept` | Returns an iterator that points past the last element of the memory within the access range. |
| `const_iterator cbegin()const noexcept` | Returns a `const` iterator to the first element of the memory within the access range. |
| `const_iterator cend()const noexcept` | Returns a `const` iterator that points past the last element of the memory within the access range. |

Continued on next page

Table 4.49: Member functions of the `accessor` class template buffer specialization.

| Member function | Description |
|---|---|
| `reverse_iterator rbegin()const noexcept` | Returns an iterator adaptor to the last element of the memory within the access range. |
| `reverse_iterator rend()const noexcept` | Returns an iterator adaptor that points before the first element of the memory within the access range. |
| `const_reverse_iterator crbegin()const noexcept` | Returns a `const` iterator adaptor to the last element of the memory within the access range. |
| `const_reverse_iterator crend()const noexcept` | Returns a `const` iterator adaptor that points before the first element of the memory within the access range. |
| | End of table |

Table 4.49: Member functions of the `accessor` class template buffer specialization.

#### 4.7.6.9.2 Device buffer accessor properties

The properties that can be provided when constructing the SYCL `accessor` class are describe in Table 4.50.

```
namespace sycl {
namespace property {
  struct noinit {};
}  // namespace property

inline constexpr property::noinit noinit;
}  // namespace sycl
```

| Property | Description |
|---|---|
| `property::noinit` | The `noinit` property notifies the SYCL runtime that previous contents of a buffer can be discarded.<br>Replaces deprecated `discard_write` and `discard_read_write` access modes. |
| | End of table |

Table 4.50: Properties supported by the SYCL `accessor` class.

The constructors of the `accessor` property classes are listed in Table 4.51.

| Constructor | Description |
|---|---|
| `property::noinit::noinit()` | Constructs a SYCL `noinit` property instance. |
| | End of table |

Table 4.51: Constructors of the `accessor property` classes.

#### 4.7.6.10 Host buffer accessor

A SYCL host_accessor is a host buffer accessor, which provides access to a SYCL buffer instance on a host.

A host buffer accessor can provide access to memory managed by a SYCL buffer immediately on the host, if created without providing a handler for a command group. Creating a host accessor with a handler makes it non-blocking operation.

If the SYCL buffer this SYCL host_accessor is accessing was constructed with the property property::buffer ::use_host_ptr the address of the memory accessed on the host must be the address the SYCL buffer was constructed with, otherwise the SYCL runtime is free to allocate temporary memory to provide access on the host.

The data type of a host buffer accessor must match that of the SYCL buffer which it is accessing.

The dimensionality of a buffer accessor must match that of the SYCL buffer which it is accessing, with the exception of 0 in which case the dimensionality of the SYCL buffer must be 1.

There are three ways a SYCL host_accessor can provide access to the elements of a SYCL buffer. Firstly by passing a SYCL id instance of the same dimensionality as the SYCL host_accessor subscript operator. Secondly by passing a single size_t value to multiple consecutive subscript operators (one for each dimension of the SYCL host_accessor, for example acc[id0][id1][id2]). Finally, in the case of the SYCL host_accessor being 0 dimensions, by triggering the implicit conversion operator. Whenever a multi-dimensional index is passed to a SYCL host_accessor the linear index is calculated based on the index {id0, id1, id2} provided and the range of the SYCL host_accessor {r0, r1, r2} according to row-major ordering as follows:

$$id2 + (id1 \cdot r2) + (id0 \cdot r2 \cdot r1) \tag{4.4}$$

A local accessor can optionally provide atomic access to allocated memory, using the access mode access::mode ::atomic, in which case all operators which return an element of the allocated memory return an instance of the the deprecated cl::sycl::atomic class. This functionality is provided for backwards compatibility and will be removed in an future version of SYCL.

Local accessors are not valid in the single_task or basic parallel_for SYCL kernel function invocations, due the fact that local work-groups are implicitly created, and the implementation is free to choose any size.

A host buffer accessor can optionally provide access to a sub range of a SYCL buffer by providing a range and offset on construction. In this case the SYCL runtime will only guarantee the latest copy of the data is available in that given range and any modifications outside that range are considered undefined behavior. The indexing performed when a SYCL host_accessor provides access to the elements of a SYCL buffer is unaffected, i.e, the accessor will continue to index from {0,0,0}.

A host buffer accessor meets the C++ requirement of ContiguousContainer and ReversibleContainer. The exception to this is that the device buffer accessor destructor doesn't destroy any elements or free the memory, because a host buffer accessor doesn't own the underlying data. For multidimensional accessors the iterator linearizes the data according to 4.3.

##### 4.7.6.10.1 Host buffer accessor interface

A synopsis of the SYCL host_accessor class template buffer specialization is provided below. The member types for this host_accessor specialization are listed in Tables 4.52. The constructors for this host_accessor specialization are listed in Tables 4.53. The member functions for this host_accessor specialization are listed in

Tables 4.54. The additional common special member functions and common member functions are listed in 4.5.3 in Tables 4.1 and 4.2, respectively. For valid implicit conversions between accessor types please refer to 4.7.6.8. Additionally, accessors of the same type must be equality comparable not only on the host application, but also within SYCL kernel functions.

```
namespace sycl {
template <typename dataT,
          int dimensions,
          access::mode accessmode>
class host_accessor {
 public:
  using value_type =  // const dataT when (accessmode == access::mode::read),
      __value_type__; // dataT otherwise
  using reference =       // const dataT& when (accessmode == access::mode::read),
      __reference_type__; // dataT& otherwise
  using const_reference = const dataT &;
  using iterator =      // const dataT* when (accessmode == access::mode::read),
      __pointer_type__; // dataT* otherwise
  using const_iterator = const dataT *;
  using difference_type =
      typename std::iterator_traits<iterator>::difference_type;
  using size_type = size_t;

  host_accessor();

  /* Available only when: (dimensions == 0) */
  template <typename AllocatorT>
  host_accessor(buffer<dataT, 1, AllocatorT> &bufferRef,
                const property_list &propList = {});

  /* Available only when: (dimensions == 0) */
  template <typename AllocatorT>
  host_accessor(buffer<dataT, 1, AllocatorT> &bufferRef,
                handler &commandGroupHandlerRef, const property_list &propList = {});

  /* Available only when: (dimensions > 0) */
  template <typename AllocatorT>
  host_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
                const property_list &propList = {});

  /* Available only when: (dimensions > 0) */
  template <typename AllocatorT, typename TagT>
  host_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef, TagT tag,
                const property_list &propList = {});

  /* Available only when: (dimensions > 0) */
  template <typename AllocatorT>
  host_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
                handler &commandGroupHandlerRef, const property_list &propList = {});

  /* Available only when: (dimensions > 0) */
  template <typename AllocatorT, typename TagT>
  host_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
                handler &commandGroupHandlerRef, TagT tag,
                const property_list &propList = {});
```

```
  /* Available only when: (dimensions > 0) */
  template <typename AllocatorT>
  host_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
                range<dimensions> accessRange, const property_list &propList = {});

  /* Available only when: (dimensions > 0) */
  template <typename AllocatorT, typename TagT>
  host_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
                range<dimensions> accessRange, TagT tag,
                const property_list &propList = {});

  /* Available only when: (dimensions > 0) */
  template <typename AllocatorT>
  host_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
                range<dimensions> accessRange, id<dimensions> accessOffset,
                const property_list &propList = {});

  /* Available only when: (dimensions > 0) */
  template <typename AllocatorT, typename TagT>
  host_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
                range<dimensions> accessRange, id<dimensions> accessOffset,
                TagT tag, const property_list &propList = {});

  /* Available only when: (dimensions > 0) */
  template <typename AllocatorT>
  host_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
                handler &commandGroupHandlerRef, range<dimensions> accessRange,
                const property_list &propList = {});

  /* Available only when: (dimensions > 0) */
  template <typename AllocatorT, typename TagT>
  host_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
                handler &commandGroupHandlerRef, range<dimensions> accessRange,
                TagT tag, const property_list &propList = {});

  /* Available only when: (dimensions > 0) */
  template <typename AllocatorT>
  host_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
                handler &commandGroupHandlerRef, range<dimensions> accessRange,
                id<dimensions> accessOffset, const property_list &propList = {});

  /* Available only when: (dimensions > 0) */
  template <typename AllocatorT, typename TagT>
  host_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef,
                handler &commandGroupHandlerRef, range<dimensions> accessRange,
                id<dimensions> accessOffset, TagT tag,
                const property_list &propList = {});

  /* -- common interface members -- */

  void swap(host_accessor &other);

  size_type byte_size() const noexcept;

```

```
  size_type size() const noexcept;

  size_type max_size() const noexcept;

  bool empty() const noexcept;

  /* Available only when: dimensions > 0 */
  range<dimensions> get_range() const;

  /* Available only when: dimensions > 0 */
  id<dimensions> get_offset() const;

  /* Available only when: (dimensions == 0) */
  operator reference() const;

  /* Available only when: (dimensions > 0) */
  reference operator[](id<dimensions> index) const;

  /* Available only when: dimensions > 1 */
  __unspecified__ &operator[](size_t index) const;

  iterator data() const noexcept;

  iterator begin() const noexcept;

  iterator end() const noexcept;

  const_iterator cbegin() const noexcept;

  const_iterator cend() const noexcept;
};
}  // namespace sycl
```

Listing 4.2: Host accessor class for buffers.

| Member types | Description |
|---|---|
| value_type | If (accessmode == access_mode::read), equal to const dataT. In other cases equal to dataT. |
| reference | If (accessmode == access_mode::read), equal to const dataT&. In other cases equal to dataT&. |
| const_reference | const dataT& |
| iterator | If (accessmode == access_mode::read), equal to const dataT*. In other cases equal to dataT*. |
| const_iterator | const dataT* |
| difference_type | typename std::iterator_traits< iterator>::difference_type |
| size_type | size_t |
| | End of table |

Table 4.52: Member types of the host_accessor class template .

| Constructor | Description |
|---|---|
| `host_accessor()` | Constructs an empty accessor. Fulfills the following post-conditions:<br>• `(empty()== true)`<br>• All size queries return `0`.<br>• The only iterator that can be obtained is `nullptr`.<br>• Trying to access the underlying memory is undefined behavior. |
| `host_accessor(buffer<dataT, 1, AllocatorT> & bufferRef, const property_list &propList = {})` | Available only when: `(dimensions == 0)`. Constructs a `host_accessor` instance for accessing a single element of a SYCL `buffer` immediately on the host. The optional `property_list` provides properties for the constructed SYCL `host_accessor` object. |
| `host_accessor(buffer<dataT, 1, AllocatorT> & bufferRef, handler &commandGroupHandlerRef, const property_list &propList = {})` | Available only when: `(dimensions == 0)`. Constructs a non-blocking SYCL `host_accessor` instance for accessing a single element of a SYCL `buffer`. The optional `property_list` provides properties for the constructed SYCL `host_accessor` object. |
| `host_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef, const property_list &propList = {})` | Available only when: `(dimensions > 0)`. Constructs a `host_accessor` for accessing a SYCL `buffer` immediately on the host. The optional `property_list` provides properties for the constructed SYCL `host_accessor` object. |
| `host_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef, TagT tag, const property_list &propList = {})` | Available only when: `(dimensions > 0)`. Constructs a `host_accessor` for accessing a SYCL `buffer` immediately on the host. `tag` is used to deduce template arguments of an `host_accessor`. The optional `property_list` provides properties for the constructed SYCL `host_accessor` object. |
| `host_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef, handler &commandGroupHandlerRef, const property_list &propList = {})` | Available only when: `(dimensions > 0)`. Constructs a non-blocking SYCL `host_accessor` instance for accessing a SYCL `buffer`. The optional `property_list` provides properties for the constructed SYCL `host_accessor` object. |

Continued on next page

Table 4.53: Constructors of the `host_accessor` class template.

| Constructor | Description |
| --- | --- |
| `host_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef, handler &commandGroupHandlerRef, TagT tag, const property_list &propList = {})` | Available only when: `(dimensions > 0)`. Constructs a non-blocking SYCL `host_accessor` instance for accessing a SYCL `buffer`. `tag` is used to deduce template arguments of an `host_accessor`. The optional `property_list` provides properties for the constructed SYCL `host_accessor` object. |
| `host_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef, range<dimensions> accessRange, const property_list &propList = {})` | Available only when: `(dimensions > 0)`. Constructs a `host_accessor` for accessing a range of a SYCL `buffer` immediately on the host. The optional `property_list` provides properties for the constructed SYCL `host_accessor` object. |
| `host_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef, range<dimensions> accessRange, TagT tag, const property_list &propList = {})` | Available only when: `(dimensions > 0)`. Constructs a `host_accessor` for accessing a range of a SYCL `buffer` immediately on the host. `tag` is used to deduce template arguments of an `host_accessor`. The optional `property_list` provides properties for the constructed SYCL `host_accessor` object. |
| `host_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef, range<dimensions> accessRange, id<dimensions> accessOffset, const property_list &propList = {})` | Available only when: `(dimensions > 0)`. Constructs a `host_accessor` for accessing a range of a SYCL `buffer` immediately on the host. The optional `property_list` provides properties for the constructed SYCL `host_accessor` object. |
| `host_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef, range<dimensions> accessRange, id<dimensions> accessOffset, TagT tag, const property_list &propList = {})` | Available only when: `(dimensions > 0)`. Constructs a `host_accessor` for accessing a range of a SYCL `buffer` immediately on the host. `tag` is used to deduce template arguments of an `host_accessor`. The optional `property_list` provides properties for the constructed SYCL `host_accessor` object. |
| `host_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef, handler &commandGroupHandlerRef, range<dimensions> accessRange, const property_list &propList = {})` | Available only when: `(dimensions > 0)`. Constructs a non-blocking SYCL `host_accessor` instance for accessing a range of SYCL `buffer`, specified by `accessRange`. The optional `property_list` provides properties for the constructed SYCL `host_accessor` object. |

Continued on next page

Table 4.53: Constructors of the `host_accessor` class template.

| Constructor | Description |
|---|---|
| `host_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef, handler &commandGroupHandlerRef, range<dimensions> accessRange, TagT tag, const property_list &propList = {})` | Available only when: `(dimensions > 0)`. Constructs a non-blocking SYCL `host_accessor` instance for accessing a range of SYCL `buffer`, specified by `accessRange`. `tag` is used to deduce template arguments of an `host_accessor`. The optional `property_list` provides properties for the constructed SYCL `host_accessor` object. |
| `host_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef, handler &commandGroupHandlerRef, range<dimensions> accessRange, id<dimensions> accessOffset, const property_list &propList = {})` | Available only when: `(dimensions > 0)`. Constructs a non-blocking SYCL `host_accessor` instance for accessing a range of SYCL `buffer`, specified by `accessRange` and `accessOffset`. The optional `property_list` provides properties for the constructed SYCL `host_accessor` object. |
| `host_accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef, handler &commandGroupHandlerRef, range<dimensions> accessRange, id<dimensions> accessOffset, TagT tag, const property_list &propList = {})` | Available only when: `(dimensions > 0)`. Constructs a non-blocking SYCL `host_accessor` instance for accessing a range of SYCL `buffer`, specified by `accessRange` and `accessOffset`. `tag` is used to deduce template arguments of an `host_accessor`. The optional `property_list` provides properties for the constructed SYCL `host_accessor` object. |
| | End of table |

Table 4.53: Constructors of the `host_accessor` class template.

| Member function | Description |
|---|---|
| `void swap(host_accessor &other);` | Swaps the contents of the current accessor with the contents of `other`. |
| `size_type byte_size()const noexcept` | Returns the size in bytes of the region of a SYCL `buffer` that this SYCL `host_accessor` is accessing. |
| `size_type size()const noexcept` | Returns the number of elements in the region of a SYCL `buffer` that this SYCL `host_accessor` is accessing. |
| `size_type max_size()const noexcept` | Returns the maximum number of elements any accessor of this type would be able to access. |
| `bool empty()const noexcept` | Returns `true` iff `(size()== 0)`. |
| | Continued on next page |

Table 4.54: Member functions of the `host_accessor` class template.

| Member function | Description |
|---|---|
| `range<dimensions> get_range()const` | Available only when: `dimensions > 0`. Returns a `range` object which represents the number of elements of `dataT` per dimension that this `host_accessor` may access. The `range` object returned must equal to the `range` of the `buffer` this `host_accessor` is associated with, unless a range was explicitly specified when this `host_accessor` was constructed. |
| `id<dimensions> get_offset()const` | Available only when: `dimensions > 0`. Returns an `id` object which represents the starting point in number of elements of `dataT` for the `range` that this `host_accessor` may access. The `id` object returned must equal to `id {0, 0, 0}`, unless an offset was explicitly specified when this `host_accessor` was constructed. |
| `operator reference()const` | Available only when: `(dimensions == 0)`. Returns a reference to the element stored within the SYCL `buffer` this SYCL `host_accessor` is accessing. |
| `reference operator[](id<dimensions> index)const` | Available only when: `(dimensions > 0)`. Returns a reference to the element stored within the SYCL `buffer` this SYCL `host_accessor` is accessing at the index specified by `index`. |
| `__unspecified__ &operator[](size_t index)const` | Available only when: `dimensions > 1`. Returns an instance of an undefined intermediate type representing a a SYCL `host_accessor` of the same type as this SYCL `host_accessor`, with the dimensionality `dimensions-1` and containing an implicit SYCL `id` with index `dimensions` set to `index`. The intermediate type returned must provide all available subscript operators which take a `size_t` parameter defined by the SYCL `host_accessor` class that are appropriate for the type it represents (including this subscript operator). |
| `iterator data()const noexcept` | Returns a pointer to the memory this SYCL `host_accessor` memory is accessing. |
| `iterator begin()const noexcept` | Returns an iterator to the first element of the memory within the access range. |
| `iterator end()const noexcept` | Returns an iterator that points past the last element of the memory within the access range. |

Continued on next page

Table 4.54: Member functions of the `host_accessor` class template.

| Member function | Description |
|---|---|
| `const_iterator cbegin()const noexcept` | Returns a `const` iterator to the first element of the memory within the access range. |
| `const_iterator cend()const noexcept` | Returns a `const` iterator that points past the last element of the memory within the access range. |
| | End of table |

Table 4.54: Member functions of the `host_accessor` class template.

#### 4.7.6.10.2 Host buffer accessor properties

The `host_accessor` supports the same list of properties as a device buffer accessor listed in 4.7.6.9.2.

### 4.7.6.11 Local accessor

A local accessor provides access to SYCL runtime allocated shared memory via local memory. A SYCL `accessor` is considered a local accessor if it has the access target `target::local`. The memory allocated by a local accessor is non-initialised so it is the user's responsibility to construct and destroy objects explicitly if required. The local memory that is allocated is shared between all work-items of a work-group.

A local accessor does not provide access on the host and the memory can not be copied back to the host.

The data type of a local accessor can be any valid SYCL kernel argument (see Section 4.14.4.

The size of memory allocated by the SYCL runtime is specified by a SYCL `range` provided on construction. The dimensionality of the SYCL `range` provided must match the SYCL `accessor`, with the exception of `0` in which case the dimensionality of the SYCL `range` must be `0`.

There are three ways that a SYCL `accessor` can provide access to the elements of the allocated memory. Firstly by passing a SYCL `id` instance of the same dimensionality as the SYCL `accessor` subscript operator. Secondly by passing a single `size_t` value to multiple consecutive subscript operators (one for each dimension of the SYCL `accessor`, for example `acc[z][y][x]`). Finally, in the case of the SYCL `accessor` having `0` dimensions, by triggering the implicit conversion operator. Whenever a multi-dimensional index is passed to a SYCL `accessor`, the linear index is calculated based on the index `{id0, id1, id2}` provided and the range of the SYCL `accessor` `{r0, r1, r2}` according to row-major ordering as follows:

$$id2 + (id1 \cdot r2) + (id0 \cdot r2 \cdot r1) \tag{4.5}$$

A local accessor can optionally provide atomic access to allocated memory, using the access mode `access_mode::atomic`, in which case all operators which return an element of the allocated memory return an instance of the SYCL `atomic` class.

Local accessors are not valid in the `single_task` or basic `parallel_for` SYCL kernel function invocations, due the fact that local work-groups are implicitly created, and the implementation is free to choose any size.

A local accessor meets the C++ requirement of `ContiguousContainer`. The iterator for this container is `multi_ptr<dataT, access::address_space::local_space, access::decorated::no>`. For multidimensional accessors the iterator linearizes the data according to 4.3.

The full list of capabilities that local accessors can support is described in 4.55.

| Access target | Accessor type | Access modes | Data types | Dimensionalities | Placeholder |
|---|---|---|---|---|---|
| local | device | read_write<br>atomic | All available data types supported in a SYCL kernel function. | Between 0 and 3 (inclusive). | No |

Table 4.55: Description of all the local accessor capabilities.

#### 4.7.6.11.1 Local accessor interface

A synopsis of the SYCL accessor class template local specialization is provided below. The member types for this accessor specialization are listed in Tables 4.56. The constructors for this accessor specialization are listed in Tables 4.57. The member functions for this accessor specialization are listed in Tables 4.58. The additional common special member functions and common member functions are listed in 4.5.3 in Tables 4.1 and 4.2, respectively. For valid implicit conversions between accessor types please refer to 4.7.6.8. Additionally, accessors of the same type must be equality comparable not only on the host application, but also within SYCL kernel functions.

```
namespace sycl {
template <typename dataT,
          int dimensions,
          access::mode accessmode,
          access::target accessTarget,
          access::placeholder isPlaceholder>
class accessor {
 public:
  template <typename value_type, access::decorated IsDecorated>
  using accessor_ptr = multi_ptr<value_type, access::address_space::local_space, IsDecorated>;
  using value_type = dataT;
  using reference = dataT &;
  using const_reference = const dataT &;
  using iterator = accessor_ptr<dataT, access::decorated::no>;
  using const_iterator = accessor_ptr<const dataT, access::decorated::no>;
  using reverse_iterator = std::reverse_iterator<iterator>;
  using const_reverse_iterator = std::reverse_iterator<const_iterator>;
  using difference_type =
      typename std::iterator_traits<iterator>::difference_type;
  using size_type = size_t;

  accessor();

  /* Available only when: dimensions == 0 */
  accessor(handler &commandGroupHandlerRef,
           const property_list &propList = {});

  /* Available only when: dimensions > 0 */
  accessor(range<dimensions> allocationSize, handler &commandGroupHandlerRef,
           const property_list &propList = {});

  /* -- common interface members -- */

  void swap(accessor &other);

  size_type byte_size() const noexcept;
```

```

    size_type size() const noexcept;

    size_type max_size() const noexcept;

    size_type get_count() const noexcept;

    bool empty() const noexcept;

    range<dimensions> get_range() const;

    /* Available only when: (dimensions == 0) */
    operator reference() const;

    /* Available only when: (dimensions > 0) */
    reference operator[](id<dimensions> index) const;

    /* Deprecated in SYCL 2020
    Available only when: accessMode == access::mode::atomic && dimensions ==  0 */
    operator cl::sycl::atomic<dataT,access::address_space::local_space> () const;

    /* Deprecated in SYCL 2020
    Available only when: accessMode == access::mode::atomic && dimensions > 0 */
    cl::sycl::atomic<dataT, access::address_space::local_space> operator[](
      id<dimensions> index) const;

    /* Available only when: dimensions > 1 */
    __unspecified__ &operator[](size_t index) const;

    std::add_pointer_t<value_type> get_pointer() const noexcept;

    template <access::decorated IsDecorated>
    accessor_ptr<value_type, IsDecorated> get_multi_ptr() const noexcept;

    iterator data() const noexcept;

    iterator begin() const noexcept;

    iterator end() const noexcept;

    const_iterator cbegin() const noexcept;

    const_iterator cend() const noexcept;

    reverse_iterator rbegin() const noexcept;

    reverse_iterator rend() const noexcept;

    const_reverse_iterator crbegin() const noexcept;

    const_reverse_iterator crend() const noexcept;
  };
  }  // namespace sycl
```

Listing 4.3: Accessor class for locals.

| Member types | Description |
|---|---|
| `template <access::decorated IsDecorated> accessor_ptr` | `multi_ptr<value_type, access::address_space::local_space, IsDecorated>`. |
| `value_type` | `dataT` |
| `reference` | `dataT&` |
| `const_reference` | `const dataT&` |
| `iterator` | `raw_local_ptr<value_type>` |
| `const_iterator` | `raw_local_ptr<const value_type>` |
| `reverse_iterator` | Iterator adaptor that reverses the direction of `iterator`. |
| `const_reverse_iterator` | Iterator adaptor that reverses the direction of `const_iterator`. |
| `difference_type` | `typename std::iterator_traits<iterator>::difference_type` |
| `size_type` | `size_t` |
| | End of table |

Table 4.56: Member types of the `accessor` class template local specialization .

| Constructor | Description |
|---|---|
| `accessor()` | Constructs an empty accessor. Fulfills the following post-conditions:<br>• `(empty()== true)`<br>• All size queries return `0`.<br>• The only iterator that can be obtained is `nullptr`.<br>• Trying to access the underlying memory is undefined behavior. |
| `accessor(handler &commandGroupHandlerRef, const property_list &propList = {})` | Available only when: `dimensions == 0`. Constructs a SYCL `accessor` instance for accessing runtime allocated shared local memory of a single element (of type `dataT`) within a SYCL kernel function on the SYCL `queue` associated with `commandGroupHandlerRef`. The allocation is per work-group. The optional `property_list` provides properties for the constructed SYCL `accessor` object. |
| | Continued on next page |

Table 4.57: Constructors of the `accessor` class template local specialization.

| Constructor | Description |
|---|---|
| `accessor(range<dimensions> allocationSize, handler &commandGroupHandlerRef, const property_list &propList = {})` | Available only when: `dimensions > 0`. Constructs a SYCL `accessor` instance for accessing runtime allocated shared local memory of size specified by `allocationSize` within a SYCL kernel function on the SYCL `queue` associated with `commandGroupHandlerRef`. `allocationSize` defines the number of elements of type `dataT` to be allocated. The allocation is per work-group, and if multiple work-groups execute simultaneously in an implementation, each work-group will receive its own functionally independent allocation of size `allocationSize` elements of type `dataT`. The optional `property_list` provides properties for the constructed SYCL `accessor` object. |
| | End of table |

Table 4.57: Constructors of the `accessor` class template local specialization.

| Member function | Description |
|---|---|
| `void swap(accessor &other);` | Swaps the contents of the current accessor with the contents of `other`. |
| `size_type byte_size()const noexcept` | Returns the size in bytes of the local memory allocation, per work-group, that this SYCL `accessor` is accessing. |
| `size_type size()const noexcept` | Returns the number of `dataT` elements in the local memory allocation, per work-group, that this SYCL `accessor` is accessing. |
| `size_type max_size()const noexcept` | Returns the maximum number of elements any accessor of this type would be able to access. |
| `size_type get_count()const noexcept` | Returns the same as `size()`. |
| `bool empty()const noexcept` | Returns `true` iff `(size()== 0)`. |
| `range<dimensions> get_range()const` | Available only when: `dimensions > 0`. Returns a `range` object which represents the number of elements of `dataT` per dimension that this `accessor` may access, per work-group. |
| `operator reference()const` | Available only when: `(dimensions == 0)`. Returns a reference to the single element stored within the work-group's local memory allocation that this `accessor` is accessing. |
| | Continued on next page |

Table 4.58: Member functions of the `accessor` class template local specialization.

| Member function | Description |
| --- | --- |
| `reference operator[](id<dimensions> index)const` | Available only when: `(dimensions > 0)`. Returns a reference to the element stored within the work-group's local memory allocation that this SYCL `accessor` is accessing, at the index specified by `index`. |
| `operator cl::sycl::atomic<dataT, access::address_space::local_space> &()const` | Deprecated in SYCL 2020. Available only when: `accessMode == access_mode::atomic && dimensions == 0)`. Returns a reference to an instance of `cl::sycl::atomic` of type `dataT` providing atomic access to the element stored within the work-group's local memory allocation that this SYCL `accessor` is accessing. |
| `cl::sycl::atomic<dataT, access::address_space::local_space> & operator[](id<dimensions> index)const` | Deprecated in SYCL 2020. Available only when: `accessMode == access_mode::atomic && dimensions > 0)`. Returns a reference to an instance of `cl::sycl::atomic` of type `dataT` providing atomic access to the element stored within the work-group's local memory allocation that this SYCL `accessor` is accessing, at the index specified by `index`. |
| `__unspecified__ &operator[](size_t index)const` | Available only when: `dimensions > 1`. Returns an instance of an undefined intermediate type representing a a SYCL `accessor` of the same type as this SYCL `accessor`, with the dimensionality `dimensions-1` and containing an implicit SYCL `id` with index `dimensions` set to `index`. The intermediate type returned must provide all available subscript operators which take a `size_t` parameter defined by the SYCL `accessor` class that are appropriate for the type it represents (including this subscript operator). |
| `std::add_pointer_t<value_type> get_pointer()const noexcept` | Returns a pointer to the work-group's local memory allocation that this SYCL `accessor` is accessing. |
| `template <access::decorated IsDecorated> accessor_ptr<IsDecorated> get_multi_ptr()const noexcept` | Returns a `multi_ptr` to the work-group's local memory allocation that this SYCL `accessor` is accessing. |
| `iterator data()const noexcept` | Returns a pointer to the work-group's local memory allocation that this SYCL `accessor` is accessing. |
| `iterator begin()const noexcept` | Returns an iterator to the first element of allocated local memory. |
| | Continued on next page |

Table 4.58: Member functions of the `accessor` class template local specialization.

| Member function | Description |
|---|---|
| iterator end()const noexcept | Returns an iterator that points past the last element of allocated local memory. |
| const_iterator cbegin()const noexcept | Returns a const iterator to the first element of allocated local memory. |
| const_iterator cend()const noexcept | Returns a const iterator that points past the last element of allocated local memory. |
| reverse_iterator rbegin()const noexcept | Returns an iterator adaptor to the last element of the memory of allocated local memory. |
| reverse_iterator rend()const noexcept | Returns an iterator adaptor that points before the first element of the memory of allocated local memory. |
| const_reverse_iterator crbegin()const noexcept | Returns a const iterator adaptor to the last element of the memory of allocated local memory. |
| const_reverse_iterator crend()const noexcept | Returns a const iterator adaptor that points before the first element of the memory of allocated local memory. |
| | End of table |

Table 4.58: Member functions of the accessor class template local specialization.

#### 4.7.6.11.2 Local accessor properties

The property_list constructor parameters are present for extensibility.

### 4.7.6.12 Image accessor

An image accessor provides access to either an instance of a SYCL unsampled_image or sampled_image. A SYCL accessor is considered an image accessor if it has the access target target::unsampled_image, target::sampled_image, target::host_unsampled_image or target::host_sampled_image.

An image accessor can provide access to memory managed by a SYCL unsampled_image or sampled_image class, using the access target target::unsampled_image or target::sampled_image.

Alternatively an image accessor can provide access to memory managed by a SYCL unsampled_image or sampled_image immediately on the host, using the access target target::host_unsampled_image or target::host_sampled_image, respectively. If the SYCL image this SYCL accessor is accessing was constructed with the property property::image::use_host_ptr the address of the memory accessed on the host must be the address the SYCL image was constructed with, otherwise the SYCL runtime is free to allocate temporary memory to provide access on the host.

The data type of an image accessor must be either int4, uint4, float4 or half4.

The dimensionality of an image accessor must match that of the SYCL image which it is providing access to, with the exception of when the access target is target::image_array, in which case the dimensionality of the SYCL accessor must be 1 less.

An image accessor with the access target target::image or target::host_image can provide access to the elements of a SYCL image by passing a SYCL int4 or float4 instance to the read or write member functions. The read member function optionally takes a SYCL sampler instance to perform a sampled read of the image. For example acc.read(coords, sampler).

An image accessor with the access target target::image_array can provide access to a slice of an image array by passing a size_t value to the subscript operator. This returns an instance of __image_array_slice__, an unspecified type providing the interface of accessor<dataT, dimensions, mode, target::image> which will provide access to a slice of the image array specified by index. The __image_array_slice__ returned can then provide access via the read or write member functions as described above. For example acc[arrayIndex].read (coords, sampler).

The full list of capabilities that image accessors can support is described in 4.59.

| Access target | Accessor type | Access modes | Data types | Dimensionalities | Placeholder |
|---|---|---|---|---|---|
| unsampled_image | device | read<br>write<br>discard_write | int4<br>uint4<br>float4<br>half4 | Between 1 and 3 (inclusive). | No |
| sampled_image | device | read | int4<br>uint4<br>float4<br>half4 | Between 1 and 3 (inclusive). | No |
| host_-unsampled_image | host | read<br>write<br>discard_write | int4<br>uint4<br>float4<br>half4 | Between 1 and 3 (inclusive). | No |
| host_sampled_-image | host | read | int4<br>uint4<br>float4<br>half4 | Between 1 and 3 (inclusive). | No |

Table 4.59: Description of all the image accessor capabilities.

#### 4.7.6.12.1 Image accessor interface

A synopsis of the SYCL accessor class template image specialization is provided below. The constructors and member functions of the SYCL accessor class template image specialization are listed in Tables 4.60 and 4.61 respectively. The additional common special member functions and common member functions are listed in 4.5.3 in Tables 4.1 and 4.2, respectively. For valid implicit conversions between accessor types please refer to 4.7.6.8.

Listing 4.4: Accessor interface for images.

```
namespace sycl {

template <typename dataT,
          int dimensions,
          access::mode accessMode,
          access::target accessTarget>
class accessor {
 public:
  using value_type = dataT;
  using reference = dataT &;
  using const_reference = const dataT &;
```

```

  /* Available only when: accessTarget == access::target::host_unsampled_image */
  template <typename AllocatorT>
  accessor(unsampled_image<dimensions, AllocatorT> &imageRef);

  /* Available only when: accessTarget == access::target::host_sampled_image */
  template <typename AllocatorT>
  accessor(sampled_image<dimensions, AllocatorT> &imageRef);

  /* Available only when: accessTarget == access::target::unsampled_image */
  template <typename AllocatorT>
  accessor(unsampled_image<dimensions, AllocatorT> &imageRef,
    handler &commandGroupHandlerRef);

  /* Available only when: accessTarget == access::target::sampled_image */
  template <typename AllocatorT>
  accessor(sampled_image<dimensions, AllocatorT> &imageRef,
    handler &commandGroupHandlerRef);

  /* -- common interface members -- */

  /* -- property interface members -- */

  size_t get_count() const;

  /* Available only when: (accessTarget == access::target::unsampled_image &&
  accessMode == access::mode::read) || (accessTarget ==
  access::target::host_unsampled_image && accessMode == access::mode::read)
  if dimensions == 1, coordT = int
  if dimensions == 2, coordT = int2
  if dimensinos == 4, coordT = int4 */
  template <typename coordT>
  dataT read(const coordT &coords) const noexcept;

  /* Available only when: (accessTarget == access::target::sampled_image &&
  accessMode == access::mode::read) || (accessTarget ==
  access::target::host_sampled_image && accessMode == access::mode::read)
  if dimensions == 1, coordT = float
  if dimensions == 2, coordT = float2
  if dimensions == 3, coordT = float4 */
  template <typename coordT>
  dataT read(const coordT &coords) const noexcept;

  /* Available only when: (accessTarget == access::target::unsampled_image &&
  (accessMode == access::mode::write || accessMode == access::mode::discard_write)) ||
  (accessTarget == access::target::host_unsampled_image && (accessMode == access::mode::write ||
  accessMode == access::mode::discard_write))
  if dimensions == 1, coordT = int
  if dimensions == 2, coordT = int2
  if dimensions == 3, coordT = int4 */
  template <typename coordT>
  void write(const coordT &coords, const dataT &color) const;
};
}  // namespace sycl
```

| Constructor | Description |
| --- | --- |
| `template <typename AllocatorT> accessor(unsampled_image<dimensions, AllocatorT>, &imageRef, const property_list &propList = {})` | Available only when: `accessTarget == target::host_unsampled_image`. Constructs a SYCL `accessor` instance for accessing a SYCL `unsampled_image` immediately on the host. The optional `property_list` provides properties for the constructed SYCL `accessor` object. |
| `template <typename AllocatorT> accessor(sampled_image<dimensions, AllocatorT>, &imageRef, const property_list &propList = {})` | Available only when: `accessTarget == target::host_sampled_image`. Constructs a SYCL `accessor` instance for accessing a SYCL `sampled_image` immediately on the host. The optional `property_list` provides properties for the constructed SYCL `accessor` object. |
| `template <typename AllocatorT> accessor(unsampled_image<dimensions, AllocatorT>, &imageRef, handler &commandGroupHandlerRef, const property_list &propList = {})` | Available only when: `accessTarget == target::unsampled_image`. Constructs a SYCL `accessor` instance for accessing a SYCL `unsampled_image` within a SYCL kernel function on the SYCL `queue` associated with `commandGroupHandlerRef`. The optional `property_list` provides properties for the constructed SYCL `accessor` object. |
| `template <typename AllocatorT> accessor(sampled_image<dimensions, AllocatorT>, &imageRef, handler &commandGroupHandlerRef, const property_list &propList = {})` | Available only when: `accessTarget == target::sampled_image`. Constructs a SYCL `accessor` instance for accessing a SYCL `sampled_image` within a SYCL kernel function on the SYCL `queue` associated with `commandGroupHandlerRef`. The optional `property_list` provides properties for the constructed SYCL `accessor` object. |
| | End of table |

Table 4.60: Constructors of the `accessor` class template image specialization.

| Member function | Description |
| --- | --- |
| `size_t get_size()const` | Returns the size in bytes of the SYCL `unsampled_image` or `sampled_image` this SYCL `accessor` is accessing. |
| `size_t get_count()const` | Returns the number of elements of the SYCL `unsampled_image` or `sampled_image` this SYCL `accessor` is accessing. |
| | Continued on next page |

Table 4.61: Member functions of the `accessor` class template image specialization.

| Member function | Description |
|---|---|
| `template <typename coordT>` `dataT read(const coordT &coords)const` | Available only when: `(accessTarget == target::unsampled_image && accessMode == access_mode::read )|| (accessTarget == target:: host_unsampled_image && accessMode == access_mode::read)`. Reads and returns an element of the `unsampled_image` at the coordinates specified by `coords`. Permitted types for `coordT` are `int32_t` when `dimensions == 1`, `int2` when `dimensions == 2` and `int4` when `dimensions == 3`. |
| `template <typename coordT>` `dataT read(const coordT &coords)const` | Available only when: `(accessTarget == target::sampled_image && accessMode == access_mode::read)|| (accessTarget == target::host_sampled_image && accessMode == access_mode::read)`. Reads and returns a sampled element of the `sampled_image` at the coordinates specified by `coords`. Permitted types for `coordT` are `float` when `dimensions == 1`, `float2` when `dimensions == 2` and `float4` when `dimensions == 3`. |
| `template <typename coordT>` `void write(const coordT &coords, const dataT &color) const` | Available only when: `(accessTarget == target::unsampled_image && ( accessMode == access_mode::write || accessMode == access_mode:: discard_write))|| (accessTarget == target::host_unsampled_image && (accessMode == access_mode::write || accessMode == access_mode:: discard_write))`. Writes the value specified by `color` to the element of the image at the coordinates specified by `coords`. Permitted type for `coordT` are `int` when `dimensions == 1`, `int2` when `dimensions == 2` and `int4` when `dimensions == 3`. |
| `__image_array_slice__` `operator[](size_t index)const` | Available only when: `accessTarget == target::image_array && dimensions < 3` . Returns an instance of `__image_array_slice__`, an unspecified type which provides the interface of `accessor< dataT, dimensions, mode, target::image>` which will provide access to a slice of the image array specified by `index`. |
| | End of table |

Table 4.61: Member functions of the `accessor` class template image specialization.

#### 4.7.6.12.2 Image accessor properties

The `property_list` constructor parameters are present for extensibility.

## 4.7.7 Address space classes

In SYCL, there are five different address spaces: global, local, constant, private and generic. In a SYCL generic implementation, types are not affected by the address spaces. However, there are situations where users needs to explicitly carry address spaces in the type. For example:

- For performance tuning and genericity. Even if the platform support the representation of the generic address space, this may comes at some performance sacrifices. In order to help the target compiler, it can be useful to track specifically which address space a pointer is addressing.

- When linking SYCL kernels with SYCL backend-specific functions. In this case, it might be necessary to specify the address space for any pointer parameters.

Direct declaration of pointers with address spaces is discouraged as the definition is implementation defined. Users must rely on the `multi_ptr` class to handle address space boundaries and interoperability.

### 4.7.7.1 Multi-pointer class

The multi-pointer class is the common interface for the explicit pointer classes, defined in 4.7.7.2.

There are situations where a user may want to make their type address space dependent. This allow to perform generic programming that depends on the address space associated with their data. An example might be wrapping a pointer inside a class, where a user may need to template the class according to the address space of the pointer the class is initialized with. In this case, the `multi_ptr` class enables users to do this in a portable and stable way.

The `multi_ptr` class exposes 2 flavors of the same interface. If the value of `access::decorated` is `access::decorated::no`, the interface exposes pointers and references type that are not decorated by an address space. If the value of `access::decorated` is `access::decorated::yes`, the interface exposes pointers and references type that are decorated by an address space. The decoration is implementation defines and relies on device compiler extensions. The decorated type may be distinct from the non decorated one. For interoperability with SYCL/SYCL backend, users should rely on types exposed by the decorated version.

The template traits `remove_decoration` and type alias `remove_decoration_t` retrieve the non decorated pointer or reference from a decorated one. Using this template trait with a non decorated type is safe and returns the same type.

It is possible to use the `void` type for the `multi_ptr` class, but in that case some functionality is disabled. `multi_ptr<void>` does not provide the `reference` or `const_reference` types, the access operators (`operator*()`, `operator->()`), the arithmetic operators or `prefetch` member function. Conversions from `multi_ptr` to `multi_ptr<void>` of the same address space are allowed, and will occur implicitly. Conversions from `multi_ptr<void>` to any other `multi_ptr` type of the same address space are allowed, but must be explicit. The same rules apply to `multi_ptr<const void>`.

An overview of the interface provided for the `multi_ptr` class follows.

```
namespace sycl {
namespace access {
enum class address_space : int {
```

```
  global_space,
  local_space,
  constant_space,
  private_space,
  generic_space,
};

enum class decorated : int {
  no,
  yes,
};

}  // namespace access

template<typename T> struct remove_decoration {
  using type = /* ... */;
};

template<typename T>
using remove_decoration_t = remove_decoration::type;

template <typename ElementType, access::address_space Space, access::decorated DecorateAddress>
class multi_ptr {
 public:
  static constexpr bool is_decorated = DecorateAddress == access::decorated::yes;
  static constexpr access::address_space address_space = Space;

  using value_type = ElementType;
  using pointer = std::conditional<is_decorated, __unspecified__ *,
                                   std::add_pointer_t<value_type>>;
  using reference = std::conditional<is_decorated, __unspecified__ &,
                                     std::add_lvalue_reference_t<value_type>>;
  using iterator_category = std::random_access_iterator_tag;
  using difference_type = std::ptrdiff_t;

  static_assert(std::is_same_v<remove_decoration_t<pointer>, std::add_pointer_t<value_type>>);
  static_assert(std::is_same_v<remove_decoration_t<reference>, std::add_lvalue_reference_t<
      value_type>>);

  // Constructors
  multi_ptr();
  multi_ptr(const multi_ptr&);
  multi_ptr(multi_ptr&&);
  // Only if DecorateAddress == access::decorated::yes
  explicit multi_ptr(pointer);
  multi_ptr(std::nullptr_t);

  // Only if Space == global_space or generic_space
  template <int dimensions, access::mode Mode, access::placeholder isPlaceholder>
  multi_ptr(accessor<value_type, dimensions, Mode, access::target::global_buffer, isPlaceholder>);

  // Only if Space == local_space or generic_space
  template <int dimensions, access::mode Mode, access::placeholder isPlaceholder>
  multi_ptr(accessor<value_type, dimensions, Mode, access::target::local, isPlaceholder>);

```

```
// Only if Space == constant_space
template <int dimensions, access::mode Mode, access::placeholder isPlaceholder>
multi_ptr(accessor<value_type, dimensions, Mode, access::target::constant_buffer, isPlaceholder
    >);

// Assignment and access operators
multi_ptr &operator=(const multi_ptr&);
multi_ptr &operator=(multi_ptr&&);
multi_ptr &operator=(std::nullptr_t);

// Only if Space == address_space::generic_space
// and ASP != access::address_space::constant_space
template<access::address_space ASP, access::decorated IsDecorated>
multi_ptr &operator=(const multi_ptr<value_type, ASP, IsDecorated>&);
// Only if Space == address_space::generic_space
// and ASP != access::address_space::constant_space
template<access::address_space ASP, access::decorated IsDecorated>
multi_ptr &operator=(multi_ptr<value_type, ASP, IsDecorated>&&);

reference operator*() const;
pointer operator->() const;

pointer get() const;
std::add_pointer_t<value_type> get_raw() const;
__unspecified__ * get_decorated() const;

// Conversion to the underlying pointer type
// Deprecated, get() should be used instead.
operator pointer() const;

// Only if Space == address_space::generic_space
// Cast to private_ptr
explicit operator multi_ptr<value_type, access::address_space::private_space,
                            DecorateAddress>();
// Only if Space == address_space::generic_space
// Cast to private_ptr
explicit
operator multi_ptr<const value_type, access::address_space::private_space,
                   DecorateAddress>() const;
// Only if Space == address_space::generic_space
// Cast to global_ptr
explicit operator multi_ptr<value_type, access::address_space::global_space,
                            DecorateAddress>();
// Only if Space == address_space::generic_space
// Cast to global_ptr
explicit
operator multi_ptr<const value_type, access::address_space::global_space,
                   DecorateAddress>() const;
// Only if Space == address_space::generic_space
// Cast to local_ptr
explicit operator multi_ptr<value_type, access::address_space::local_space,
                            DecorateAddress>();
// Only if Space == address_space::generic_space
// Cast to global_ptr
explicit
```

```
  operator multi_ptr<const value_type, access::address_space::local_space,
                     DecorateAddress>() const;

  // Implicit conversion to a multi_ptr<void>.
  // Only available when value_type is not const-qualified.
  template<access::decorated DecorateAddress>
  operator multi_ptr<void, Space, DecorateAddress>() const;

  // Implicit conversion to a multi_ptr<const void>.
  // Only available when value_type is const-qualified.
  template<access::decorated DecorateAddress>
  operator multi_ptr<const void, Space, DecorateAddress>() const;

  // Implicit conversion to multi_ptr<const value_type, Space>.
  template<access::decorated DecorateAddress>
  operator multi_ptr<const value_type, Space, DecorateAddress>() const;

  // Implicit conversion to the non decorated version of multi_ptr.
  // Only available when is_decorated is true.
  operator multi_ptr<value_type, Space, access::decorated::no>() const;

  // Implicit conversion to the decorated version of multi_ptr.
  // Only available when is_decorated is false.
  operator multi_ptr<value_type, Space, access::decorated::yes>() const;

  void prefetch(size_t numElements) const;

  // Arithmetic operators
  friend multi_ptr& operator++(multi_ptr& mp) { /* ... */ }
  friend multi_ptr operator++(multi_ptr& mp, int) { /* ... */ }
  friend multi_ptr& operator--(multi_ptr& mp) { /* ... */ }
  friend multi_ptr operator--(multi_ptr& mp, int) { /* ... */ }
  friend multi_ptr& operator+=(multi_ptr& lhs, difference_type r) { /* ... */ }
  friend multi_ptr& operator-=(multi_ptr& lhs, difference_type r) { /* ... */ }
  friend multi_ptr operator+(const multi_ptr& lhs, difference_type r) { /* ... */ }
  friend multi_ptr operator-(const multi_ptr& lhs, difference_type r) { /* ... */ }
  friend reference operator*(const multi_ptr& lhs) { /* ... */ }

  friend bool operator==(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */ }
  friend bool operator!=(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */ }
  friend bool operator<(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */ }
  friend bool operator>(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */ }
  friend bool operator<=(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */ }
  friend bool operator>=(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */ }

  friend bool operator==(const multi_ptr& lhs, std::nullptr_t) { /* ... */ }
  friend bool operator!=(const multi_ptr& lhs, std::nullptr_t) { /* ... */ }
  friend bool operator<(const multi_ptr& lhs, std::nullptr_t) { /* ... */ }
  friend bool operator>(const multi_ptr& lhs, std::nullptr_t) { /* ... */ }
  friend bool operator<=(const multi_ptr& lhs, std::nullptr_t) { /* ... */ }
  friend bool operator>=(const multi_ptr& lhs, std::nullptr_t) { /* ... */ }

  friend bool operator==(std::nullptr_t, const multi_ptr& rhs) { /* ... */ }
  friend bool operator!=(std::nullptr_t, const multi_ptr& rhs) { /* ... */ }
  friend bool operator<(std::nullptr_t, const multi_ptr& rhs) { /* ... */ }
```

```
  friend bool operator>(std::nullptr_t, const multi_ptr& rhs) { /* ... */ }
  friend bool operator<=(std::nullptr_t, const multi_ptr& rhs) { /* ... */ }
  friend bool operator>=(std::nullptr_t, const multi_ptr& rhs) { /* ... */ }

};

// Specialization of multi_ptr for void and const void
// VoidType can be either void or const void
template <access::address_space Space, access::decorated DecorateAddress>
class multi_ptr<VoidType, Space, DecorateAddress> {
 public:
  static constexpr bool is_decorated = DecorateAddress == access::decorated::yes;
  static constexpr access::address_space address_space = Space;

  using value_type = VoidType;
  using pointer = std::conditional<is_decorated, __unspecified__ value_type *,
                                   std::add_pointer_t<value_type>>;
  using difference_type = std::ptrdiff_t;

  // Constructors
  multi_ptr();
  multi_ptr(const multi_ptr&);
  multi_ptr(multi_ptr&&);
  explicit multi_ptr(pointer);
  multi_ptr(std::nullptr_t);

  // Only if Space == global_space
  template <typename ElementType, int dimensions, access::mode Mode,
            access::placeholder isPlaceholder>
  multi_ptr(accessor<ElementType, dimensions, Mode,
                     access::target::global_buffer, isPlaceholder>);

  // Only if Space == local_space
  template <typename ElementType, int dimensions, access::mode Mode,
            access::placeholder isPlaceholder>
  multi_ptr(accessor<ElementType, dimensions, Mode, access::target::local,
                     isPlaceholder>);

  // Only if Space == constant_space
  template <typename ElementType, int dimensions, access::mode Mode,
            access::placeholder isPlaceholder>
  multi_ptr(accessor<ElementType, dimensions, Mode,
                     access::target::constant_buffer, isPlaceholder>);

  // Assignment operators
  multi_ptr &operator=(const multi_ptr&);
  multi_ptr &operator=(multi_ptr&&);
  multi_ptr &operator=(std::nullptr_t);

  pointer get() const;

  // Conversion to the underlying pointer type
  explicit operator pointer() const;

  // Explicit conversion to a multi_ptr<ElementType>
```

```
  // If VoidType is const, ElementType must be as well
  template <typename ElementPointer>
  explicit operator multi_ptr<ElementType, Space, DecorateAddress>() const;

  // Implicit conversion to the non decorated version of multi_ptr.
  // Only available when is_decorated is true.
  operator multi_ptr<value_type, Space, access::decorated::no>() const;

  // Implicit conversion to the decorated version of multi_ptr.
  // Only available when is_decorated is false.
  operator multi_ptr<value_type, Space, access::decorated::yes>() const;

  // Implicit conversion to multi_ptr<const void, Space>
  operator multi_ptr<const void, Space, DecorateAddress>() const;

  friend bool operator==(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */ }
  friend bool operator!=(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */ }
  friend bool operator<(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */ }
  friend bool operator>(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */ }
  friend bool operator<=(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */ }
  friend bool operator>=(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */ }

  friend bool operator==(const multi_ptr& lhs, std::nullptr_t) { /* ... */ }
  friend bool operator!=(const multi_ptr& lhs, std::nullptr_t) { /* ... */ }
  friend bool operator<(const multi_ptr& lhs, std::nullptr_t) { /* ... */ }
  friend bool operator>(const multi_ptr& lhs, std::nullptr_t) { /* ... */ }
  friend bool operator<=(const multi_ptr& lhs, std::nullptr_t) { /* ... */ }
  friend bool operator>=(const multi_ptr& lhs, std::nullptr_t) { /* ... */ }

  friend bool operator==(std::nullptr_t, const multi_ptr& rhs) { /* ... */ }
  friend bool operator!=(std::nullptr_t, const multi_ptr& rhs) { /* ... */ }
  friend bool operator<(std::nullptr_t, const multi_ptr& rhs) { /* ... */ }
  friend bool operator>(std::nullptr_t, const multi_ptr& rhs) { /* ... */ }
  friend bool operator<=(std::nullptr_t, const multi_ptr& rhs) { /* ... */ }
  friend bool operator>=(std::nullptr_t, const multi_ptr& rhs) { /* ... */ }

};

// Deprecated, address_space_cast should be used instead.
template <typename ElementType, access::address_space Space, access::decorated DecorateAddress>
multi_ptr<ElementType, Space, DecorateAddress> make_ptr(ElementType *);

template <access::address_space Space, access::decorated DecorateAddress,
          typename ElementType>
multi_ptr<ElementType, Space, DecorateAddress> address_space_cast(ElementType *);

// Deduction guides
template <int dimensions, access::mode Mode, access::placeholder isPlaceholder,
          class T>
multi_ptr(
    accessor<T, dimensions, Mode, access::target::global_buffer, isPlaceholder>)
    -> multi_ptr<T, access::address_space::global_space>;
template <int dimensions, access::mode Mode, access::placeholder isPlaceholder,
          class T>
multi_ptr(accessor<T, dimensions, Mode, access::target::constant_buffer,
```

```
                 isPlaceholder>)
    -> multi_ptr<T, access::address_space::constant_space>;
template <int dimensions, access::mode Mode, access::placeholder isPlaceholder,
          class T>
multi_ptr(accessor<T, dimensions, Mode, access::target::local, isPlaceholder>)
    -> multi_ptr<T, access::address_space::local_space>;

}  // namespace sycl
```

| Constructor | Description |
|---|---|
| `template <typename ElementType, access::address_space Space, access::decorated DecorateAddress>`<br>`multi_ptr()` | Default constructor. |
| `template <typename ElementType, access::address_space Space, access::decorated DecorateAddress>`<br>`multi_ptr(const multi_ptr &)` | Copy constructor. |
| `template <typename ElementType, access::address_space Space, access::decorated DecorateAddress>`<br>`multi_ptr(multi_ptr&&)` | Move constructor. |
| `template <typename ElementType, access::address_space Space, access::decorated DecorateAddress>`<br>`multi_ptr(pointer)` | Available only when: `is_decorated == true`.<br>Constructor that takes as an argument a pointer of type `pointer`. |
| `template <typename ElementType, access::address_space Space, access::decorated DecorateAddress>`<br>`multi_ptr(std::nullptr_t)` | Constructor from a `nullptr`. |
| `template <typename ElementType, access::address_space Space = access::address_space::global_space>`<br>`template <int dimensions, access::mode Mode>`<br>`multi_ptr(`<br>`accessor<ElementType, dimensions, Mode, access::target::global_buffer>)` | Available only when: `Space == access::address_space::global_space`.<br>Constructs a `multi_ptr<ElementType, access::address_space::global_space>` from an accessor of `access::target::global_buffer`. |
| `template <typename ElementType, access::address_space Space = access::address_space::local_space>`<br>`template <int dimensions, access::mode Mode>`<br>`multi_ptr(`<br>`accessor<ElementType, dimensions, Mode, access::target::local>)` | Available only when: `Space == access::address_space::local_space`.<br>Constructs a `multi_ptr<ElementType, access::address_space::local_space>` from an accessor of `access::target::local`. |

Continued on next page

Table 4.62: Constructors of the SYCL `multi_ptr` class template.

| Constructor | Description |
|---|---|
| `template <typename ElementType, access::address_space Space = access::address_space::constant_space> template <int dimensions, access::mode Mode> multi_ptr( accessor<ElementType, dimensions, Mode, access::target::constant_buffer>)` | Available only when: `Space == access::address_space::constant_space`. Constructs a `multi_ptr<ElementType, access::address_space::constant_space>` from an accessor of `access::target::constant_buffer`. |
| `template <typename ElementType, access::address_space Space, access::decorated DecorateAddress> multi_ptr<ElementType, Space, DecorateAddress> make_ptr(ElementType*)` | Global function to create a `multi_ptr` instance depending on the address space of the `pointer` type. An implementation must reject an argument if the deduced address space is not compatible with `Space`. |
| | End of table |

Table 4.62: Constructors of the SYCL `multi_ptr` class template.

| Operators | Description |
|---|---|
| `template <typename value_type, access::address_space Space, access::decorated DecorateAddress> multi_ptr &operator=(const multi_ptr&)` | Copy assignment operator. |
| `template <typename value_type, access::address_space Space, access::decorated DecorateAddress> multi_ptr &operator=(multi_ptr&&)` | Move assignment operator. |
| `template <typename value_type, access::address_space Space, access::decorated DecorateAddress> multi_ptr &operator=(std::nullptr_t)` | Assigns `nullptr` to the `multi_ptr`. |
| `template<access::address_space ASP, access::decorated IsDecorated> multi_ptr &operator=(const multi_ptr<value_type, ASP, IsDecorated>&)` | Available only when: `Space == access::address_space::generic_space && ASP != access::address_space::constant_space`. Assigns the value of the left hand side `multi_ptr` into the `generic_ptr`. |
| `template<access::address_space ASP, access::decorated IsDecorated> multi_ptr &operator=(multi_ptr<value_type, ASP, IsDecorated>&&` | Available only when: `Space == access::address_space::generic_space && ASP != access::address_space::constant_space`. Move the value of the left hand side `multi_ptr` into the `generic_ptr`. |
| `template <typename value_type, access::address_space Space, access::decorated DecorateAddress> pointer operator->()const` | Available only when: `!std::is_void<value_type>::value`. Returns the underlying pointer. |
| `template <typename value_type, access::address_space Space, access::decorated DecorateAddress> reference operator*()const` | Available only when: `!std::is_void<value_type>::value`. Returns a reference to the pointed value. |
| `template <typename value_type, access::address_space Space, access::decorated DecorateAddress> operator pointer()const` | Implicit conversion to the underlying pointer type. **Deprecated:** The member function `get` should be used instead |
| | Continued on next page |

Table 4.63: Operators of `multi_ptr` class.

| Operators | Description |
| --- | --- |
| `template <access::decorated IsDecorated> operator multi_ptr<value_type, access::address_space::private_space, IsDecorated>()const` | Available only when: `Space == access::address_space::generic_space`. Conversion from `generic_ptr` to `private_ptr`. If result is undefined if the pointer does not address the private address space. |
| `template <access::decorated IsDecorated> operator multi_ptr<const value_type, access::address_space::private_space, IsDecorated>()const` | Available only when: `Space == access::address_space::generic_space`. Conversion from `generic_ptr` to `private_ptr`. If result is undefined if the pointer does not address the private address space. |
| `template <access::decorated IsDecorated> operator multi_ptr<value_type, access::address_space::global_space, IsDecorated>()const` | Available only when: `Space == access::address_space::generic_space`. Conversion from `generic_ptr` to `global_ptr`. If result is undefined if the pointer does not address the global address space. |
| `template <access::decorated IsDecorated> operator multi_ptr<const value_type, access::address_space::global_space, IsDecorated>()const` | Available only when: `Space == access::address_space::generic_space`. Conversion from `generic_ptr` to `global_ptr`. If result is undefined if the pointer does not address the global address space. |
| `template <access::decorated IsDecorated> operator multi_ptr<value_type, access::address_space::local_space, IsDecorated>()const` | Available only when: `Space == access::address_space::generic_space`. Conversion from `generic_ptr` to `local_ptr` . If result is undefined if the pointer does not address the local address space. |
| `template <access::decorated IsDecorated> operator multi_ptr<const value_type, access::address_space::local_space, IsDecorated>()const` | Available only when: `Space == access::address_space::generic_space`. Conversion from `generic_ptr` to `local_ptr` . If result is undefined if the pointer does not address the local address space. |
| `template <access::decorated IsDecorated> operator multi_ptr<void, Space, IsDecorated>() const` | Available only when: `!std::is_void<value_type>::value && !std::is_const<value_type>::value`. Implicit conversion to a `multi_ptr` of type `void`. |
| `template <access::decorated IsDecorated> operator multi_ptr<const void, Space, IsDecorated>()const` | Available only when: `!std::is_void<value_type>::value && std::is_const<value_type>::value`. Implicit conversion to a `multi_ptr` of type `const void`. |
| `template <access::decorated IsDecorated> operator multi_ptr<const value_type, Space, IsDecorated>()const` | Implicit conversion to a `multi_ptr` of type `const value_type`. |

Continued on next page

Table 4.63: Operators of `multi_ptr` class.

| Operators | Description |
|---|---|
| `operator multi_ptr<const value_type, Space, access::decorated::no>()const` | Available only when: `is_decorated == true`.<br>Implicit conversion to the equivalent `multi_ptr` object that does not expose decorated pointers or references. |
| `operator multi_ptr<const value_type, Space, access::decorated::yes>()const` | Available only when: `is_decorated == false`.<br>Implicit conversion to the equivalent `multi_ptr` object that exposes decorated pointers and references. |
| | End of table |

Table 4.63: Operators of `multi_ptr` class.

| Member function | Description |
|---|---|
| `pointer get()const` | Returns the underlying pointer. The whether the pointer is decorated depends on the value of `DecorateAddress`. |
| `__unspecified__ * get_decorated()const` | Returns the underlying pointer decorated by the address space that it addressed. Note that the support involves implementation defined device compiler extensions. |
| `std::add_pointer_t<value_type> get_raw()const` | Returns the underlying pointer, always undecorated. |
| `void prefetch(size_t numElements)const` | Available only when: `Space == access::address_space::global_space`.<br>Prefetches a number of elements specified by `numElements` into the global memory cache. This operation is an implementation defined optimization and does not effect the functional behavior of the SYCL kernel function. |
| | End of table |

Table 4.64: Member functions of `multi_ptr` class.

| Hidden friend function | Description |
|---|---|
| `reference operator*(const multi_ptr& mp)` | Available only when: `!std::is_void<ElementType>::value`.<br>Operator that returns a reference to the `value_type` of `mp`. |
| `multi_ptr& operator++(multi_ptr& mp)` | Available only when: `!std::is_void<ElementType>::value`.<br>Increments `mp` by 1 and returns `mp`. |
| | Continued on next page |

Table 4.65: Hidden friend functions of the `multi_ptr` class.

| Hidden friend function | Description |
|---|---|
| `multi_ptr operator++(multi_ptr& mp, int)` | Available only when: `!std::is_void<ElementType>::value`. Increments `mp` by `1` and returns a new `multi_ptr` with the value of the original `mp`. |
| `multi_ptr& operator--(multi_ptr& mp)` | Available only when: `!std::is_void<ElementType>::value`. Decrements `mp` by `1` and returns `mp`. |
| `multi_ptr operator--(multi_ptr& mp, int)` | Available only when: `!std::is_void<ElementType>::value`. Decrements `mp` by `1` and returns a new `multi_ptr` with the value of the original `mp`. |
| `multi_ptr& operator+=(multi_ptr& lhs, difference_type r)` | Available only when: `!std::is_void<ElementType>::value`. Moves `mp` forward by `r` and returns `lhs`. |
| `multi_ptr& operator-=(multi_ptr& lhs, difference_type r)` | Available only when: `!std::is_void<ElementType>::value`. Moves `mp` backward by `r` and returns `lhs`. |
| `multi_ptr operator+(const multi_ptr& lhs, difference_type r)` | Available only when: `!std::is_void<ElementType>::value`. Creates a new `multi_ptr` that points `r` forward compared to `lhs`. |
| `multi_ptr operator-(const multi_ptr& lhs, difference_type r)` | Available only when: `!std::is_void<ElementType>::value`. Creates a new `multi_ptr` that points `r` backward compared to `lhs`. |
| `bool operator==(const multi_ptr& lhs, const multi_ptr& rhs)` | Comparison operator `==` for `multi_ptr` class. |
| `bool operator!=(const multi_ptr& lhs, const multi_ptr& rhs)` | Comparison operator `!=` for `multi_ptr` class. |
| `bool operator<(const multi_ptr& lhs, const multi_ptr & rhs)` | Comparison operator < for `multi_ptr` class. |
| `bool operator>(const multi_ptr& lhs, const multi_ptr & rhs)` | Comparison operator > for `multi_ptr` class. |
| `bool operator<=(const multi_ptr& lhs, const multi_ptr& rhs)` | Comparison operator `<=` for `multi_ptr` class. |
| `bool operator>-=(const multi_ptr& lhs, const multi_ptr& rhs)` | Comparison operator `>=` for `multi_ptr` class. |
| `bool operator==(const multi_ptr& lhs, std::nullptr_t )` | Comparison operator == for `multi_ptr` class with a `std::nullptr_t`. |
| `bool operator!=(const multi_ptr& lhs, std::nullptr_t )` | Comparison operator != for `multi_ptr` class with a `std::nullptr_t`. |
| `bool operator<(const multi_ptr& lhs, std::nullptr_t)` | Comparison operator < for `multi_ptr` class with a `std::nullptr_t`. |
| `bool operator>(const multi_ptr& lhs, std::nullptr_t)` | Comparison operator > for `multi_ptr` class with a `std::nullptr_t`. |
| `bool operator<=(const multi_ptr& lhs, std::nullptr_t )` | Comparison operator <= for `multi_ptr` class with a `std::nullptr_t`. |

Continued on next page

Table 4.65: Hidden friend functions of the `multi_ptr` class.

| Hidden friend function | Description |
|---|---|
| `bool operator>=(const multi_ptr& lhs, std::nullptr_t )` | Comparison operator >= for `multi_ptr` class with a `std::nullptr_t`. |
| `bool operator==(std::nullptr_t, const multi_ptr& rhs )` | Comparison operator == for `multi_ptr` class with a `std::nullptr_t`. |
| `bool operator!=(std::nullptr_t, const multi_ptr& rhs )` | Comparison operator != for `multi_ptr` class with a `std::nullptr_t`. |
| `bool operator<(std::nullptr_t, const multi_ptr& rhs)` | Comparison operator < for `multi_ptr` class with a `std::nullptr_t`. |
| `bool operator>(std::nullptr_t, const multi_ptr& rhs)` | Comparison operator > for `multi_ptr` class with a `std::nullptr_t`. |
| `bool operator<=(std::nullptr_t, const multi_ptr& rhs )` | Comparison operator <= for `multi_ptr` class with a `std::nullptr_t`. |
| `bool operator>=(std::nullptr_t, const multi_ptr& rhs )` | Comparison operator >= for `multi_ptr` class with a `std::nullptr_t`. |
| | End of table |

Table 4.65: Hidden friend functions of the `multi_ptr` class.

#### 4.7.7.2 Explicit pointer aliases

SYCL provides aliases to the `multi_ptr` class template (see Section 4.7.7.1) for each specialization of `access::address_space`.

A synopsis of the SYCL `multi_ptr` class template aliases is provided below.

```
namespace sycl {

template <typename ElementType, access::address_space Space, access::decorated IsDecorated>
class multi_ptr;

// Template specialization aliases for different pointer address spaces

// Deprecated, raw_global_ptr or decorated_global_ptr should be used instead.
template <typename ElementType>
using global_ptr = multi_ptr<ElementType, access::address_space::global_space,
                             access::decorated::no>;

// Deprecated, raw_local_ptr or decorated_local_ptr should be used instead.
template <typename ElementType>
using local_ptr = multi_ptr<ElementType, access::address_space::local_space,
                            access::decorated::no>;

// Deprecated, raw_constant_ptr or decorated_constant_ptr should be used instead.
template <typename ElementType>
using constant_ptr = multi_ptr<ElementType, access::address_space::constant_space,
                               access::decorated::no>;

// Deprecated, raw_private_ptr or decorated_private_ptr should be used instead.
template <typename ElementType>
using private_ptr = multi_ptr<ElementType, access::address_space::private_space,
                              access::decorated::no>;
```

```

// Template specialization aliases for different pointer address spaces.
// The interface exposes non decorated pointer while keeping the
// address space information internally.

template <typename ElementType>
using raw_global_ptr = multi_ptr<ElementType, access::address_space::global_space,
                                 access::decorated::no>;

template <typename ElementType>
using raw_local_ptr = multi_ptr<ElementType, access::address_space::local_space,
                                access::decorated::no>;

template <typename ElementType>
using raw_constant_ptr = multi_ptr<ElementType, access::address_space::constant_space,
                                   access::decorated::no>;

template <typename ElementType>
using raw_private_ptr = multi_ptr<ElementType, access::address_space::private_space,
                                  access::decorated::no>;

// Template specialization aliases for different pointer address spaces.
// The interface exposes decorated pointer.

template <typename ElementType>
using decorated_global_ptr = multi_ptr<ElementType, access::address_space::global_space,
                                       access::decorated::yes>;

template <typename ElementType>
using decorated_local_ptr = multi_ptr<ElementType, access::address_space::local_space,
                                      access::decorated::yes>;

template <typename ElementType>
using decorated_constant_ptr = multi_ptr<ElementType, access::address_space::constant_space,
                                         access::decorated::yes>;

template <typename ElementType>
using decorated_private_ptr = multi_ptr<ElementType, access::address_space::private_space,
                                        access::decorated::yes>;

} // namespace sycl
```

### 4.7.8 Samplers

The SYCL sampler struct encapsulates a configuration for sampling a sampled_image. A SYCL sampler can map to one native backend object.

```
namespace sycl {
enum class addressing_mode: unsigned int {
  mirrored_repeat,
  repeat,
  clamp_to_edge,
  clamp,
  none
```

```
};

enum class filtering_mode: unsigned int {
  nearest,
  linear
};

enum class coordinate_normalization_mode : unsigned int {
  normalized,
  unnormalized
};

struct image_sampler {
  addressing_mode addressing;
  coordinate_mode coordinate;
  filtering_mode filtering;
};
}  // namespace sycl
```

| addressing_mode | Description |
| --- | --- |
| mirrored_repeat | Out of range coordinates will be flipped at every integer junction. This addressing mode can only be used with normalized coordinates. If normalized coordinates are not used, this addressing mode may generate image coordinates that are undefined. |
| repeat | Out of range image coordinates are wrapped to the valid range. This addressing mode can only be used with normalized coordinates. If normalized coordinates are not used, this addressing mode may generate image coordinates that are undefined. |
| clamp_to_edge | Out of range image coordinates are clamped to the extent. |
| clamp | Out of range image coordinates will return a border color. |
| none | For this addressing mode the programmer guarantees that the image coordinates used to sample elements of the image refer to a location inside the image; otherwise the results are undefined. |
| | End of table |

Table 4.66: Addressing modes description.

| filtering_mode | Description |
| --- | --- |
| nearest | Chooses a color of nearest pixel. |
| linear | Performs a linear sampling of adjacent pixels. |
| | End of table |

Table 4.67: Filtering modes description.

| coordinate_normalization_mode | Description |
|---|---|
| `normalized` | Normalizes image coordinates. |
| `unnormalized` | Does not normalize image coordinates. |
| | End of table |

Table 4.68: Coordinate normalization modes description.

| Constructor | Description |
|---|---|
| `sampler( coordinate_normalization_mode normalizationMode, addressing_mode addressingMode, filtering_mode filteringMode, const property_list &propList = {})` | Constructs a SYCL `sampler` instance with address mode, filtering mode and coordinate normalization mode specified by the respective parameters. It is not valid to construct a SYCL `sampler` within a SYCL kernel function. The optional `property_list` provides properties for the constructed SYCL `sampler` object. |
| | End of table |

Table 4.69: Constructors the `sampler` class.

| Member function | Description |
|---|---|
| `addressing_mode get_addressing_mode()const` | Return the addressing mode used to construct this SYCL `sampler`. |
| `filtering_mode get_filtering_mode()const` | Return the filtering mode used to construct this SYCL `sampler`. |
| `coordinate_normalization_mode get_coordinate_normalization_mode()const` | Return the coordinate normalization mode used to construct this SYCL `sampler`. |
| | End of table |

Table 4.70: Member functions for the `sampler` class.

## 4.8 Unified shared memory

### 4.8.1 USM introduction

Unified Shared Memory (USM) provides a pointer-based alternative to the buffer programming model. USM enables:

- Easier integration into existing code bases by representing allocations as pointers rather than buffers, with full support for pointer arithmetic into allocations.
- Fine-grain control over ownership and accessibility of allocations, to optimally choose between performance and programmer convenience.
- A simpler programming model, by automatically migrating some allocations between SYCL devices and the host.

### 4.8.2 SYCL memory management

This section describes new properties and routines for pointer-based memory management interfaces in SYCL. These routines augment, rather than replace, the existing buffer-based interfaces in SYCL.

### 4.8.3 Unified addressing

Unified Addressing guarantees that all devices will use a unified address space. Pointer values in the unified address space will always refer to the same location in memory. The unified address space encompasses the host and one or more devices. Note that this does not require addresses in the unified address space to be accessible on all devices, just that pointer values will be consistent.

### 4.8.4 Unified shared memory

Unified Shared Memory (USM) is a capability that, when available, provides the ability to create allocations that are visible to both host and device(s). USM builds upon Unified Addressing to define a shared address space where pointer values in this space always refer to the same location in memory. USM defines multiple tiers of increasing capability described in the following sections:

- Explicit USM
- Restricted USM
- Concurrent USM
- System USM

USM is an optional feature which may not be supported by all devices, and devices that support USM may only support some of these tiers. A SYCL application can use the `device::has()` function to determine the level of USM support for a device. See Table 4.20 in Section 4.6.4.3 for more details.

#### 4.8.4.1 Explicit USM

Explicit USM defines capabilities for explicitly managing device memory. Programmers directly allocate device memory, and data must be explicitly copied between the host and a device. Device allocations are obtained through SYCL USM device allocation routines instead of system allocation routines like `std::malloc` or C++ `new`. Device allocations are not accessible on the host, but the pointer values remain consistent on account of Unified Addressing. Greater detail about how allocations are used is described by the following tables.

#### 4.8.4.2 Restricted USM

Restricted USM defines capabilities for implicitly sharing data between host and devices. However, Restricted USM, as the name implies, is limited in that host and device may not concurrently compute on memory in the shared address space. Restricted USM builds upon Explicit USM by adding two new types of allocations, `host` and `shared`. Allocations are obtained through SYCL allocator instead of the system allocator. `shared` allocations may be limited by device memory. Greater detail about the allocation types defined in Restricted USM and their usage is described by the following tables.

#### 4.8.4.3 Concurrent USM

Concurrent USM builds upon Restricted USM by enabling concurrent access to `shared` allocations between host and devices. Additionally, some implementations may support a working set of `shared` allocations larger than device memory.

#### 4.8.4.4 System USM

System USM extends upon the previous tiers by performing all shared allocations with the normal system memory allocation routines. In particular, programmers may now use std::malloc or C++ new instead of USM allocation routines to create shared allocations. Likewise, std::free and delete are used instead of sycl::free. Note that host and device allocations are unaffected by this change and must still be allocated using their respective USM functions in order to guarantee their behavior.

### 4.8.5 USM allocations

```
namespace sycl {
  namespace usm {
    enum class alloc {
      host,
      device,
      shared,
      unknown
    };
  }
}
```

| | |
|---|---|
| host | Allocations in host memory that are accessible by a device. |
| device | Allocations in device memory that are **not** accessible by the host. |
| shared | Allocations in shared memory that are accessible by both host and device. |

| Allocation Type | Initial Location | Accessible By | | Migratable To | |
|---|---|---|---|---|---|
| device | device | host | No | host | No |
| | | device | Yes | device | N/A |
| | | Another device | Optional (P2P) | Another device | No |
| host | host | host | Yes | host | N/A |
| | | Any device | Yes | device | No |
| shared | Unspecified | host | Yes | host | Yes |
| | | device | Yes | device | Yes |
| | | Another device | Optional (P2P) | Another device | Optional |

### 4.8.6 C++ allocator interface

```
template <typename T, usm::alloc AllocKind, size_t Alignment = 0>
class usm_allocator {
public:
  using value_type = T;

public:
  template <typename U> struct rebind {
    typedef usm_allocator<U, AllocKind, Alignment> other;
  };

  usm_allocator() noexcept = delete;
```

```
  usm_allocator(const context &ctxt, const device &dev) noexcept;
  usm_allocator(const queue &q) noexcept;
  usm_allocator(const usm_allocator &other) noexcept;

  template <class U> usm_allocator(usm_allocator<U, AllocKind, Alignment> const &) noexcept;

  /// Allocate memory
  T *allocate(size_t Size);

  /// Deallocate memory
  void deallocate(T *Ptr, size_t size);

  /// Constructs an object on memory pointed by Ptr.
  ///
  /// Note: AllocKind == alloc::device is not allowed.
  template <
      usm::alloc AllocT = AllocKind,
      typename std::enable_if<AllocT != usm::alloc::device, int>::type = 0,
      class U, class... ArgTs>
  void construct(U *Ptr, ArgTs &&... Args);

  /// Throws an error when trying to construct a device allocation
  /// on the host
  template <
      usm::alloc AllocT = AllocKind,
      typename std::enable_if<AllocT == usm::alloc::device, int>::type = 0,
      class U, class... ArgTs>
  void construct(U *Ptr, ArgTs &&... Args);

  /// Destroys an object.
  ///
  /// Note:: AllocKind == alloc::device is not allowed
  template <
      usm::alloc AllocT = AllocKind,
      typename std::enable_if<AllocT != usm::alloc::device, int>::type = 0>
  void destroy(T *Ptr);

  /// Throws an error when trying to destroy a device allocation
  /// on the host
  template <
      usm::alloc AllocT = AllocKind,
      typename std::enable_if<AllocT == usm::alloc::device, int>::type = 0>
  void destroy(T *Ptr);
};

/// Equality Comparison
///
/// Allocators only compare equal if they are of the same USM kind, alignment,
/// context, and device (when kind is not host)
template <class T, usm::alloc AllocKindT, size_t AlignmentT, class U,
          usm::alloc AllocKindU, size_t AlignmentU>
bool operator==(const usm_allocator<T, AllocKindT, AlignmentT> &,
                const usm_allocator<U, AllocKindU, AlignmentU> &) noexcept;

/// Inequality Comparison
```

```
///
/// Allocators only compare unequal if they are not of the same USM kind, alignment,
/// context, or device (when kind is not host)
template <class T, class U, usm::alloc AllocKind, size_t Alignment = 0>
bool operator!=(const usm_allocator<T, AllocKind, Alignment> &allocT,
                const usm_allocator<U, AllocKind, Alignment> &allocU) noexcept;
```

## 4.8.7 Utility functions

While the modern C++ usm_allocator interface is sufficient for specifying USM allocations and deallocations, many programmers may prefer C-style malloc-influenced APIs. As a convenience to programmers, malloc-style APIs are also defined. Additionally, other utility functions are specified in the following sections to perform various operations such as memory copies and initializations as well as to provide performance hints.

### 4.8.7.1 Explicit USM

#### 4.8.7.1.1 malloc

```
(1)
void* sycl::malloc_device(size_t num_bytes,
                          const sycl::device& dev,
                          const sycl::context& ctxt);

(2)
template <typename T>
T* sycl::malloc_device(size_t count,
                       const sycl::device& dev,
                       const sycl::context& ctxt);
```

**Parameters**

- (1) size_t num_bytes - number of bytes to allocate
- (2) size_t count - number of elements of type T to allocate
- const sycl::device& dev - the SYCL device to allocate on
- const sycl::context& ctxt - the SYCL context to which device belongs

**Return value** Returns a pointer to the newly allocated memory on the specified device on success. This memory is not accessible on the host. Memory allocated by sycl::malloc_device must be deallocated with sycl::free to avoid memory leaks. If ctxt is a host context, should behave as if calling malloc_host. On failure, returns nullptr.

```
(1)
void* sycl::malloc_device(size_t num_bytes,
                          const sycl::queue& q);

(2)
template <typename T>
T* sycl::malloc_device(size_t count,
                       const sycl::queue& q);
```

**Parameters**

- (1) size_t num_bytes - number of bytes to allocate

- (2) size_t count - number of elements of type T to allocate
- const sycl::queue& q - the SYCL q that provides the device and context to allocate against

**Return value** Returns a pointer to the newly allocated memory on the device associated with q on success. This memory is not accessible on the host. Memory allocated by sycl::malloc_device must be deallocated with sycl::free to avoid memory leaks. If ctxt is a host context, should behave as if calling malloc_host.On failure, returns nullptr.

#### 4.8.7.1.2 aligned_alloc

```
(1)
void* sycl::aligned_alloc_device(size_t alignment,
                                 size_t num_bytes,
                                 const sycl::device& dev,
                                 const sycl::context& ctxt);

(2)
template <typename T>
T* sycl::aligned_alloc_device(size_t alignment,
                              size_t count,
                              const sycl::device& dev,
                              const sycl::context& ctxt);
```

**Parameters**

- size_t alignment - specifies the byte alignment. Must be a valid alignment supported by the implementation.
- (1) size_t num_bytes - number of bytes to allocate
- (2) size_t count - number of elements of type T to allocate
- const sycl::device& dev - the device to allocate on
- const sycl::context& ctxt - the SYCL context to which device belongs

**Return value** Returns a pointer to the newly allocated memory on the specified device on success. This memory is not accessible on the host. Memory allocated by sycl::aligned_alloc_device must be deallocated with sycl::free to avoid memory leaks. If ctxt is a host context, should behave as if calling aligned_alloc_host. On failure, returns nullptr.

```
(1)
void* sycl::aligned_alloc_device(size_t alignment,
                                 size_t size,
                                 const sycl::queue& q);

(2)
template <typename T>
T* sycl::aligned_alloc_device(size_t alignment,
                              size_t count,
                              const sycl::queue& q);
```

**Parameters**

- size_t alignment - specifies the byte alignment. Must be a valid alignment supported by the implementation.

- (1) `size_t size` - number of bytes to allocate
- (2) `size_t count` - number of elements of type `T` to allocate
- `const sycl::queue& q` - the SYCL `q` that provides the `device` and `context` to allocate against

**Return value** Returns a pointer to the newly allocated memory on the `device` associated with `q` on success. This memory is not accessible on the host. Memory allocated by `sycl::aligned_alloc_device` must be deallocated with `sycl::free` to avoid memory leaks. If `ctxt` is a host `context`, should behave as if calling `aligned_alloc_host`. On failure, returns `nullptr`.

#### 4.8.7.1.3 `memcpy`

```
class handler {
 ...
 public:
  ...
  void memcpy(void* dest, const void* src, size_t num_bytes);
};

class queue {
 ...
 public:
  ...
  event memcpy(void* dest, const void* src, size_t num_bytes);
};
```

**Parameters**

- `void* dest` - pointer to the destination memory
- `const void* src` - pointer to the source memory
- `size_t num_bytes` - number of bytes to copy

**Return value** Returns an event representing the copy operation.

#### 4.8.7.1.4 `memset`

```
class handler {
 ...
 public:
  ...
  void memset(void* ptr, int value, size_t num_bytes);
};

class queue {
 ...
 public:
  ...
  event memset(void* ptr, int value, size_t num_bytes);
};
```

**Parameters**
- `void* ptr` - pointer to the memory to fill
- `int value` - value to be set. Value is interpreted as an `unsigned char`
- `size_t num_bytes` - number of bytes to fill

**Return value** Returns an event representing the fill operation.

#### 4.8.7.1.5 `fill`

```
class handler {
 ...
 public:
  ...
  template <typename T>
  void fill(void* ptr, const T& pattern, size_t count)
};

class queue {
 ...
 public:
  ...
  template <typename T>
  event fill(void* ptr, const T& pattern, size_t count);
};
```

**Parameters**
- `void* ptr` - pointer to the memory to fill
- `const T& pattern` - pattern to be filled. `T` should be trivially copyable.
- `size_t count` - number of times to fill `pattern` into `ptr`

**Return value** Returns an event representing the fill operation or void if on the `handler`.

### 4.8.7.2 Restricted USM

Restricted USM includes all of the Utility Functions of Explicit USM. It additionally introduces new functions to support `host` and `shared` allocations.

#### 4.8.7.2.1 `malloc`

```
(1)
void* sycl::malloc_host(size_t num_bytes, const sycl::context& ctxt);
(2)
template <typename T>
T* sycl::malloc_host(size_t count, const sycl::context& ctxt);
```

**Parameters**
- (1) `size_t num_bytes` - number of bytes to allocate
- (2) `size_t count` - number of elements of type `T` to allocate
- `const sycl::context& ctxt` - the SYCL `context` that contains the devices that will access the `host`

allocation

**Return value** Returns a pointer to the newly allocated host memory on success. Memory allocated by sycl::malloc_host must be deallocated with sycl::free to avoid memory leaks. On failure, returns nullptr.

```
(1)
void* sycl::malloc_host(size_t num_bytes, const sycl::queue& q);
(2)
template <typename T>
T* sycl::malloc_host(size_t count, const sycl::queue& q);
```

**Parameters**
- (1) size_t num_bytes - number of bytes to allocate
- (2) size_t count - number of elements of type T to allocate
- const sycl::queue& q - the SYCL queue whose context contains the devices that will access the host allocation

**Return value** Returns a pointer to the newly allocated host memory on success. Memory allocated by sycl::malloc_host must be deallocated with sycl::free to avoid memory leaks. On failure, returns nullptr.

```
(1)
void* sycl::malloc_shared(size_t num_bytes,
                          const sycl::device& dev,
                          const sycl::context& ctxt);
(2)
template <typename T>
T* sycl::malloc_shared(size_t count,
                       const sycl::device& dev,
                       const sycl::context& ctxt);
```

**Parameters**
- (1) size_t num_bytes - number of bytes to allocate
- (2) size_t count - number of elements of type T to allocate
- const sycl::device& dev - the SYCL device to allocate on
- const sycl::context& ctxt - the SYCL context to which device belongs

**Return value** Returns a pointer to the newly allocated shared memory on the specified device on success. Memory allocated by sycl::malloc_shared must be deallocated with sycl::free to avoid memory leaks. If ctxt is a host context, should behave as if calling malloc_host. On failure, returns nullptr.

```
(1)
void* sycl::malloc_shared(size_t num_bytes,
                          const sycl::queue& q);
(2)
template <typename T>
T* sycl::malloc_shared(size_t count,
                       const sycl::queue& q);
```

**Parameters**
- (1) size_t num_bytes - number of bytes to allocate

- (2) size_t count - number of elements of type T to allocate
- const sycl::queue& q - the SYCL q that provides the device and context to allocate against

**Return value** Returns a pointer to the newly allocated shared memory on the device associated with q on success. Memory allocated by sycl::malloc_shared must be deallocated with sycl::free to avoid memory leaks. On failure, returns nullptr.

#### 4.8.7.2.2 aligned_alloc

```
(1)
void* sycl::aligned_alloc_host(size_t alignment, size_t num_bytes, const sycl::context& ctxt);
(2)
template <typename T>
T* sycl::aligned_alloc_host(size_t alignment, size_t count, const sycl::context& ctxt);
```

**Parameters**

- size_t alignment - specifies the byte alignment. Must be a valid alignment supported by the implementation.
- (1) size_t num_bytes - number of bytes to allocate
- (2) size_t count - number of elements of type T to allocate
- const sycl::context& ctxt - the SYCL context that contains the devices that will access the host allocation

**Return value** Returns a pointer to the newly allocated host memory on success. Memory allocated by sycl::aligned_alloc_host must be deallocated with sycl::free to avoid memory leaks. On failure, returns nullptr.

```
(1)
void* sycl::aligned_alloc_host(size_t alignment, size_t num_bytes, const sycl::queue& q);
(2)
template <typename T>
void* sycl::aligned_alloc_host(size_t alignment, size_t count, const sycl::queue& q);
```

**Parameters**

- size_t alignment - specifies the byte alignment. Must be a valid alignment supported by the implementation.
- (1) size_t num_bytes - number of bytes to allocate
- (2) size_t count - number of elements of type T to allocate
- const sycl::queue& q - the SYCL q whose context contains the devices that will access the host allocation

**Return value** Returns a pointer to the newly allocated host memory on success. Memory allocated by sycl::aligned_alloc_host must be deallocated with sycl::free to avoid memory leaks. On failure, returns nullptr.

```
(1)
```

```
void* sycl::aligned_alloc_shared(size_t alignment,
                                 size_t num_bytes,
                                 const sycl::device& dev,
                                 const sycl::context& ctxt);
(2)
template <typename T>
T* sycl::aligned_alloc_shared(size_t alignment,
                              size_t count,
                              const sycl::device& dev,
                              const sycl::context& ctxt);
```

**Parameters**
- `size_t alignment` - specifies the byte alignment. Must be a valid alignment supported by the implementation.
- (1) `size_t num_bytes` - number of bytes to allocate
- (2) `size_t count` - number of elements of type `T` to allocate
- `const sycl::device& dev` - the SYCL `device` to allocate on
- `const sycl::context& ctxt` - the SYCL `context` to which `device` belongs

**Return value** Returns a pointer to the newly allocated `shared` memory on the specified `device` on success. Memory allocated by `sycl::aligned_alloc_shared` must be deallocated with `sycl::free` to avoid memory leaks. If `ctxt` is a host `context`, should behave as if calling `aligned_alloc_host`. On failure, returns `nullptr`.

```
(1)
void* sycl::aligned_alloc_shared(size_t alignment,
                                 size_t num_bytes,
                                 const sycl::queue& q);
(2)
template <typename T>
T* sycl::aligned_alloc_shared(size_t alignment,
                              size_t count,
                              const sycl::queue& q);
```

**Parameters**
- `size_t alignment` - specifies the byte alignment. Must be a valid alignment supported by the implementation.
- (1) `size_t num_bytes` - number of bytes to allocate
- (2) `size_t count` - number of elements of type `T` to allocate
- `const sycl::queue& q` - the SYCL `q` that provides the `device` and `context` to allocate against

**Return value** Returns a pointer to the newly allocated `shared` memory on the `device` associated with `q` on success. Memory allocated by `sycl::aligned_alloc_shared` must be deallocated with `sycl::free` to avoid memory leaks. If `ctxt` is a host `context`, should behave as if calling `aligned_alloc_host`. On failure, returns `nullptr`.

#### 4.8.7.2.3 Performance hints

Programmers may provide hints to the runtime that data should be made available on a device earlier than Unified Shared Memory would normally require it to be available. This can be accomplished through enqueueing prefetch commands. Prefetch commands may not be overlapped with kernel execution in Restricted USM.

##### 4.8.7.2.3.1 prefetch

```
class handler {
 ...
 public:
  ...
  void prefetch(const void* ptr, size_t num_bytes);
};

class queue {
 ...
 public:
  ...
  event prefetch(const void* ptr, size_t num_bytes);
};
```

**Parameters**
- `const void* ptr` - pointer to the memory to be prefetched to the device
- `size_t num_bytes` - number of bytes requested to be prefetched

**Return value** Returns an `event` representing the prefetch operation.

### 4.8.7.3 Concurrent USM

Concurrent USM contains all the utility functions of Explicit USM and Restricted USM. It introduces a new function, `sycl::queue::mem_advise`, that allows programmers to provide additional information to the underlying runtime about how different allocations are used.

#### 4.8.7.3.1 Performance hints

##### 4.8.7.3.1.1 prefetch

In Concurrent USM, prefetch commands may be overlapped with kernel execution.

##### 4.8.7.3.1.2 mem_advise

```
class handler {
  ...
  public:
   ...
   void mem_advise(const void *addr, size_t num_bytes, int advice);
};

class queue {
 ...
 public:
```

```
  ...
  event mem_advise(const void *addr, size_t num_bytes, int advice);
};
```

**Parameters**

- `void* addr` - address of allocation
- `size_t num_bytes` - number of bytes in the allocation
- `int advice` - device-defined advice for the specified allocation. A value of `0` reverts the advice for `addr` to the default behavior.

**Return Value** Returns an event representing the operation.

#### 4.8.7.4 General

##### 4.8.7.4.1 malloc

```
(1)
void *sycl::malloc(size_t num_bytes,
                   const sycl::device& dev,
                   const sycl::context& ctxt,
                   usm::alloc kind);
(2)
template <typename T>
T *sycl::malloc(size_t count,
                const sycl::device& dev,
                const sycl::context& ctxt,
                usm::alloc kind);
```

**Parameters**

- (1) `size_t num_bytes` - number of bytes to allocate
- (2) `size_t count` - number of elements of type `T` to allocate
- `const sycl::device& dev` - the SYCL device to allocate on (if applicable)
- `const sycl::context& ctxt` - the SYCL `context` to which `device` belongs
- `usm::alloc kind` - the type of allocation to perform

**Return value** Returns a pointer to the newly allocated `kind` memory on the specified `device` on success. If `kind` is `alloc::host`, `dev` is ignored. Memory allocated by `sycl::malloc` must be deallocated with `sycl::free` to avoid memory leaks. On failure, returns `nullptr`.

```
(1)
void *sycl::malloc(size_t num_bytes,
                   const sycl::queue& q,
                   usm::alloc kind);
(2)
template <typename T>
T *sycl::malloc(size_t count,
                const sycl::queue& q,
                usm::alloc kind);
```

**Parameters**

- (1) `size_t num_bytes` - number of bytes to allocate
- (2) `size_t count` - number of elements of type `T` to allocate
- `const sycl::queue& q` - the SYCL `q` that provides the `device` (if applicable) and `context` to allocate against
- `usm::alloc kind` - the type of allocation to perform

**Return value** Returns a pointer to the newly allocated `kind` memory on success. Memory allocated by `sycl::malloc` must be deallocated with `sycl::free` to avoid memory leaks. On failure, returns `nullptr`.

#### 4.8.7.4.2 `aligned_alloc`

```
(1)
void *sycl::aligned_alloc(size_t alignment,
                          size_t num_bytes,
                          const sycl::device& dev,
                          const sycl::context& ctxt,
                          usm::alloc kind);
(2)
template <typename T>
T* sycl::aligned_alloc(size_t alignment,
                       size_t count,
                       const sycl::device& dev,
                       const sycl::context& ctxt,
                       usm::alloc kind);
```

**Parameters**

- `size_t alignment` - specifies the byte alignment. Must be a valid alignment supported by the implementation.
- (1) `size_t num_bytes` - number of bytes to allocate
- (2) `size_t count` - number of elements of type `T` to allocate
- `const sycl::device& dev` - the SYCL device to allocate on (if applicable)
- `const sycl::context& ctxt` - the SYCL `context` to which `device` belongs
- `usm::alloc kind` - the type of allocation to perform

**Return value** Returns a pointer to the newly allocated `kind` memory on the specified `device` on success. If `kind` is `alloc::host`, `dev` is ignored. Memory allocated by `sycl::aligned_alloc` must be deallocated with `sycl::free` to avoid memory leaks. On failure, returns `nullptr`.

```
(1)
void *sycl::aligned_alloc(size_t alignment,
                          size_t num_bytes,
                          const sycl::queue& q,
                          usm::alloc kind);
(2)
template <typename T>
T* sycl::aligned_alloc(size_t alignment,
```

```
                          size_t count,
                          const sycl::queue& q,
                          usm::alloc kind);
```

**Parameters**

- `size_t alignment` - specifies the byte alignment. Must be a valid alignment supported by the implementation.
- (1) `size_t num_bytes` - number of bytes to allocate
- (2) `size_t count` - number of elements of type `T` to allocate
- `const sycl::queue& q` - the SYCL `q` that provides the `device` (if applicable) and `context` to allocate against.
- `usm::alloc kind` - the type of allocation to perform

**Return value** Returns a pointer to the newly allocated `kind` memory on success. Memory allocated by `sycl::aligned_alloc` must be deallocated with `sycl::free` to avoid memory leaks. On failure, returns `nullptr`.

#### 4.8.7.4.3 `free`

```
void sycl::free(void* ptr, sycl::context& context);
```

**Parameters**

- `void* ptr` - pointer to the memory to deallocate. Must have been allocated by a SYCL `malloc` or `aligned_alloc` function.
- `const sycl::context& ctxt` - the SYCL `context` in which `ptr` was allocated

**Return value** none

```
void sycl::free(void* ptr, sycl::queue& q);
```

**Parameters**

- `void* ptr` - pointer to the memory to deallocate. Must have been allocated by a SYCL `malloc` or `aligned_alloc` function.
- `const sycl::queue& q` - the SYCL `queue` that provides the `context` in which `ptr` was allocated

**Return value** none

## 4.8.8 Unified shared memory information

### 4.8.8.1 Pointer queries

#### 4.8.8.1.1 `get_pointer_type`

```
usm::alloc get_pointer_type(const void *ptr, const context &ctxt);
```

**Parameters**

- `const void* ptr` - the pointer to query.
- `const sycl::context& ctxt` - the SYCL `context` to which the USM allocation belongs

**Return value** Returns the USM allocation type for `ptr` if `ptr` falls inside a valid USM allocation. If `ctxt` is a host `context`, returns `usm::alloc::host`. Returns `usm::alloc::unknown` if `ptr` is not a valid USM allocation.

#### 4.8.8.1.2 `get_pointer_device`

```
sycl::device get_pointer_device(const void *ptr, const context &ctxt);
```

**Parameters**

- `const void* ptr` - the pointer to query
- `const sycl::context& ctxt` - the SYCL `context` to which the USM allocation belongs

**Return value** Returns the `device` associated with the USM allocation. If `ctxt` is a host `context`, returns the host `device` in `ctxt`. If `ptr` is an allocation of type `usm::alloc::host`, returns the first device in `ctxt`. Throws an error if `ptr` is not a valid USM allocation.

## 4.9 SYCL scheduling

SYCL 1.2.1 defines an execution model based on tasks submitted to Out-of-Order queues. Dependences between these tasks are constructed from the data they read and write. The data usage of a task is conveyed to the runtime by constructing accessors on buffer objects that specify their intent. Pointers obtained from using explicit memory management interfaces in SYCL cannot create accessors, so dependence graphs cannot be constructed in the same fashion. New methods are required to specify dependences between tasks.

### 4.9.1 DAGs without accessors

Unified Shared Memory changes how the SYCL runtime manages data movement. Since the runtime might no longer be responsible for orchestrating data movement, it makes sense to enable a way to build dependence graphs based on ordering computations rather than accesses to data inside them. Conveniently, a SYCL `queue` already returns an `event` upon calls to `submit`. These events can be used by the programmer to wait for the submitted task to complete.

```
queue q;
auto dev = q.get_device();
auto ctxt = q.get_context();
float* a = static_cast<float*>(malloc_shared(10*sizeof(float), dev, ctxt));
float* b = static_cast<float*>(malloc_shared(10*sizeof(float), dev, ctxt));
float* c = static_cast<float*>(malloc_shared(10*sizeof(float), dev, ctxt));

auto e = q.submit([&](handler& cgh) {
  cgh.parallel_for<class vec_add>(range<1> {10}, [=](id<1> ID) {
    size_t i = ID[0];
    c[i] = a[i] + b[i];
  });
});
e.wait();
```

### 4.9.2 Coarse grain DAGs with `depends_on`

While SYCL already defines the capability to wait on specific tasks, programmers should still be able to easily define relationships between tasks.

```
class handler {
 ...
 public:
  ...
  void depends_on(event e);
  void depends_on(const std::vector<event> &e);
};
```

**Parameters** `e` - event or vector of events representing task(s) required to complete before this task may begin

**Return value** none

# 4.10 Expressing parallelism through kernels

## 4.10.1 Ranges and index space identifiers

The data parallelism of the SYCL kernel execution model requires instantiation of a parallel execution over a range of iteration space coordinates. To achieve this, SYCL exposes types to define the range of execution and to identify a given execution instance's point in the iteration space.

The following types are defined: `range`, `nd_range`, `id`, `item`, `h_item`, `nd_item` and `group`.

When constructing ids or ranges from integers, the elements are written in row-major format.

| Type | Description |
|---|---|
| `id` | A point within a range |
| `range` | Bounds over which an `id` may vary |
| `item` | Pairing of an `id` (specific point) and the `range` that it is bounded by |
| `nd_range` | Encapsules both global and local (work-group size) `range`s over which work-item `id`s will vary |
| `nd_item` | Encapsulates two `item`s, one for global `id` and `range`, and one for local `id` and `range` |
| `h_item` | Index point queries within hierarchical parallelism (`parallel_for_work_item`). Encapsulates physical global and local `id`s and `range`s, as well as a logical local `id` and `range` defined by hierarchical parallelism |
| `group` | Work-group queries within hierarchical parallelism (`parallel_for_work_group`), and exposes the `parallel_for_work_item` construct that identifies code to be executed by each work-item. Encapsulates work-group `id`s and `range`s |
| | End of table |

Table 4.71: Summary of types used to identify points in an index space, and ranges over which those points can vary.

#### 4.10.1.1 range class

range<int dimensions> is a 1D, 2D or 3D vector that defines the iteration domain of either a single work-group in a parallel dispatch, or the overall dimensions of the dispatch. It can be constructed from integers.

The SYCL range class template provides the common by-value semantics (see Section 4.5.4).

A synopsis of the SYCL range class is provided below. The constructors, member functions and non-member functions of the SYCL range class are listed in Tables 4.77, 4.73 and 4.74 respectively. The additional common special member functions and common member functions are listed in 4.5.4 in Tables 4.3 and 4.4 respectively.

```
namespace sycl {
template <int dimensions = 1>
class range {
public:
  /* The following constructor is only available in the range class specialization where:
       dimensions==1 */
  range(size_t dim0);
  /* The following constructor is only available in the range class specialization where:
       dimensions==2 */
  range(size_t dim0, size_t dim1);
  /* The following constructor is only available in the range class specialization where:
       dimensions==3 */
  range(size_t dim0, size_t dim1, size_t dim2);

   /* -- common interface members -- */

  size_t get(int dimension) const;
  size_t &operator[](int dimension);
  size_t operator[](int dimension) const;

  size_t size() const;

  // OP is: +, -, *, /, %, <<, >>, &, |, ^, &&, ||, <, >, <=, >=
  friend range operatorOP(const range &lhs, const range &rhs) { /* ... */ }
  friend range operatorOP(const range &lhs, const size_t &rhs) { /* ... */ }

  // OP is: +=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=
  friend range & operatorOP(const range &lhs, const range &rhs) { /* ... */ }
  friend range & operatorOP(const range &lhs, const size_t &rhs) { /* ... */ }

  // OP is: +, -, *, /, %, <<, >>, &, |, ^, &&, ||, <, >, <=, >=
  friend range operatorOP(const size_t &lhs, const range &rhs) { /* ... */ }

};

// Deduction guides
range(size_t) -> range<1>;
range(size_t, size_t) -> range<2>;
range(size_t, size_t, size_t) -> range<3>;

}  // sycl
```

| Constructor | Description |
|---|---|
| `range(size_t dim0)` | Construct a 1D range with value dim0. Only valid when the template parameter `dimensions` is equal to 1. |
| `range(size_t dim0, size_t dim1)` | Construct a 2D range with values dim0 and dim1. Only valid when the template parameter `dimensions` is equal to 2. |
| `range(size_t dim0, size_t dim1, size_t dim2)` | Construct a 3D range with values dim0, dim1 and dim2. Only valid when the template parameter `dimensions` is equal to 3. |
| | End of table |

Table 4.72: Constructors of the `range` class template.

| Member function | Description |
|---|---|
| `size_t get(int dimension)const` | Return the value of the specified dimension of the `range`. |
| `size_t &operator[](int dimension)` | Return the l-value of the specified dimension of the `range`. |
| `size_t operator[](int dimension)const` | Return the value of the specified dimension of the `range`. |
| `size_t size()const` | Return the size of the range computed as dimension0*...*dimensionN. |
| | End of table |

Table 4.73: Member functions of the `range` class template.

| Hidden friend function | Description |
|---|---|
| `range operatorOP(const range &lhs, const range &rhs)` | Where `OP` is: +, -, *, /, %, <<, >>, &, \|, ^, &&, \|\|, <, >, <=, >=. Constructs and returns a new instance of the SYCL `range` class template with the same dimensionality as `lhs` `range`, where each element of the new SYCL `range` instance is the result of an element-wise `OP` operator between each element of `lhs` `range` and each element of the `rhs` `range`. If the operator returns a `bool` the result is the cast to `size_t`. |
| | Continued on next page |

Table 4.74: Hidden friend functions of the SYCL `range` class template.

| Hidden friend function | Description |
|---|---|
| `range operatorOP(const range &lhs, const size_t &rhs )` | Where `OP` is: +, -, *, /, %, <<, >>, &, \|, ^, &&, \|\|, <, >, <=, >=.<br>Constructs and returns a new instance of the SYCL `range` class template with the same dimensionality as `lhs` `range`, where each element of the new SYCL `range` instance is the result of an element-wise `OP` operator between each element of this SYCL `range` and the `rhs` `size_t`. If the operator returns a `bool` the result is the cast to `size_t`. |
| `range &operatorOP(range &lhs, const range &rhs)` | Where `OP` is: +=, -=,*=, /=, %=, <<=, >>=, &=, \|=, ^=.<br>Assigns each element of `lhs` `range` instance with the result of an element-wise `OP` operator between each element of `lhs` `range` and each element of the `rhs` `range` and returns `lhs` `range`. If the operator returns a `bool` the result is the cast to `size_t`. |
| `range &operatorOP(range &lhs, const size_t &rhs)` | Where `OP` is: +=, -=,*=, /=, %=, <<=, >>=, &=, \|=, ^=.<br>Assigns each element of `lhs` `range` instance with the result of an element-wise `OP` operator between each element of `lhs` `range` and the `rhs` `size_t` and returns `lhs` `range`. If the operator returns a `bool` the result is the cast to `size_t`. |
| `range operatorOP(const size_t &lhs, const range &rhs )` | Where `OP` is: +, -, *, /, %, <<, >>, &, \|, ^, &&, \|\|, <, >, <=, >=.<br>Constructs and returns a new instance of the SYCL `range` class template with the same dimensionality as the `rhs` SYCL `range` , where each element of the new SYCL `range` instance is the result of an element-wise `OP` operator between the `lhs` `size_t` and each element of the `rhs` SYCL `range`. If the operator returns a `bool` the result is the cast to `size_t`. |
| | End of table |

Table 4.74: Hidden friend functions of the SYCL `range` class template.

#### 4.10.1.2 `nd_range` class

```
namespace sycl {
template <int dimensions = 1>
class nd_range {
public:

  /* -- common interface members -- */
```

```

  nd_range(range<dimensions> globalSize, range<dimensions> localSize,
           id<dimensions> offset = id<dimensions>());

  range<dimensions> get_global_range() const;
  range<dimensions> get_local_range() const;
  range<dimensions> get_group_range() const;
  id<dimensions> get_offset() const;
};
}  // namespace sycl
```

`nd_range<int` dimensions> defines the iteration domain of both the work-groups and the overall dispatch. To define this the `nd_range` comprises two ranges: the whole range over which the kernel is to be executed, and the range of each work group.

The SYCL `nd_range` class template provides the common by-value semantics (see Section 4.5.4).

A synopsis of the SYCL `nd_range` class is provided below. The constructors and member functions of the SYCL `nd_range` class are listed in Tables 4.75 and 4.76 respectively. The additional common special member functions and common member functions are listed in 4.5.4 in Tables 4.3 and 4.4 respectively.

| Constructor | Description |
|---|---|
| `nd_range<dimensions>( range<dimensions> globalSize, range<dimensions> localSize) id<dimensions> offset = id<dimensions>())` | Construct an `nd_range` from the local and global constituent ranges as well as an optional offset. If the offset is not provided it will default to no offset. |
| | End of table |

Table 4.75: Constructors of the `nd_range` class.

| Member function | Description |
|---|---|
| `range<dimensions> get_global_range()const` | Return the constituent global range. |
| `range<dimensions> get_local_range()const` | Return the constituent local range. |
| `range<dimensions> get_group_range()const` | Return a range representing the number of groups in each dimension. This range would result from `globalSize/localSize` as provided on construction. |
| `id<dimensions> get_offset()const` | Return the constituent offset. |
| | End of table |

Table 4.76: Member functions for the `nd_range` class.

#### 4.10.1.3 `id` class

`id<int` dimensions> is a vector of dimensions that is used to represent an id into a global or local `range`. It can be used as an index in an accessor of the same rank. The `[n]` operator returns the component `n` as an `size_t`.

The SYCL `id` class template provides the common by-value semantics (see Section 4.5.4).

A synopsis of the SYCL `id` class is provided below. The constructors, member functions and non-member functions of the SYCL `id` class are listed in Tables 4.77, 4.78 and 4.79 respectively. The additional common special member functions and common member functions are listed in 4.5.4 in Tables 4.3 and 4.4 respectively.

```
namespace sycl {
template <int dimensions = 1>
class id {
public:
  id();

  /* The following constructor is only available in the id class
   * specialization where: dimensions==1 */
  id(size_t dim0);
  /* The following constructor is only available in the id class
   * specialization where: dimensions==2 */
  id(size_t dim0, size_t dim1);
  /* The following constructor is only available in the id class
   * specialization where: dimensions==3 */
  id(size_t dim0, size_t dim1, size_t dim2);

   /* -- common interface members -- */

  id(const range<dimensions> &range);
  id(const item<dimensions> &item);

  size_t get(int dimension) const;
  size_t &operator[](int dimension);
  size_t operator[](int dimension) const;

  // OP is: +, -, *, /, %, <<, >>, &, |, ^, &&, ||, <, >, <=, >=
    friend id operatorOP(const id &lhs, const id &rhs) { /* ... */ }
    friend id operatorOP(const id &lhs, const size_t &rhs) { /* ... */ }

  // OP is: +=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=
    friend id &operatorOP(id &lhs, const id &rhs) { /* ... */ }
    friend id &operatorOP(id &lhs, const size_t &rhs) { /* ... */ }

  // OP is: +, -, *, /, %, <<, >>, &, |, ^, &&, ||, <, >, <=, >=
    friend id operatorOP(const size_t &lhs, const id &rhs) { /* ... */ }

};

// Deduction guides
id(size_t) -> id<1>;
id(size_t, size_t) -> id<2>;
id(size_t, size_t, size_t) -> id<3>;

}  // namespace sycl
```

| Constructor | Description |
|---|---|
| `id()` | Construct a SYCL `id` with the value `0` for each dimension. |

Continued on next page

Table 4.77: Constructors of the `id` class template.

| Constructor | Description |
| --- | --- |
| `id(size_t dim0)` | Construct a 1D `id` with value dim0. Only valid when the template parameter `dimensions` is equal to 1. |
| `id(size_t dim0, size_t dim1)` | Construct a 2D `id` with values dim0, dim1. Only valid when the template parameter `dimensions` is equal to 2. |
| `id(size_t dim0, size_t dim1, size_t dim2)` | Construct a 3D `id` with values dim0, dim1, dim2. Only valid when the template parameter `dimensions` is equal to 3. |
| `id(const range<dimensions> &range)` | Construct an `id` from the dimensions of `range`. |
| `id(const item<dimensions> &item)` | Construct an `id` from `item.get_id()`. |
| | End of table |

Table 4.77: Constructors of the `id` class template.

| Member function | Description |
| --- | --- |
| `size_t get(int dimension)const` | Return the value of the `id` for dimension `dimension`. |
| `size_t &operator[](int dimension)` | Return a reference to the requested dimension of the `id` object. |
| `size_t operator[](int dimension)const` | Return the value of the requested dimension of the `id` object. |
| | End of table |

Table 4.78: Member functions of the `id` class template.

| Hidden friend function | Description |
| --- | --- |
| `id operatorOP(const id &lhs, const id &rhs)` | Where `OP` is: +, -, *, /, %, <<, >>, &, \|, ^, &&, \|\|, <, >, <=, >=. Constructs and returns a new instance of the SYCL `id` class template with the same dimensionality as `lhs` `id`, where each element of the new SYCL `id` instance is the result of an element-wise `OP` operator between each element of `lhs` `id` and each element of the `rhs` `id`. If the operator returns a `bool` the result is the cast to `size_t`. |
| | Continued on next page |

Table 4.79: Hidden friend functions of the `id` class template.

<table>
<tr><th>Hidden friend function</th><th>Description</th></tr>
<tr><td><code>id operatorOP(const id &lhs, const size_t &rhs)</code></td><td>Where <code>OP</code> is: +, -, *, /, %, <<, >>, &, |, ^, &&, ||, <, >, <=, >=.<br>Constructs and returns a new instance of the SYCL <code>id</code> class template with the same dimensionality as <code>lhs</code> <code>id</code>, where each element of the new SYCL <code>id</code> instance is the result of an element-wise <code>OP</code> operator between each element of <code>lhs</code> <code>id</code> and the <code>rhs</code> <code>size_t</code>. If the operator returns a <code>bool</code> the result is the cast to <code>size_t</code>.</td></tr>
<tr><td><code>id &operatorOP(id &lhs, const id &rhs)</code></td><td>Where <code>OP</code> is: +=, -=,*=, /=, %=, <<=, >>=, &=, |=, ^=.<br>Assigns each element of <code>lhs</code> <code>id</code> instance with the result of an element-wise <code>OP</code> operator between each element of <code>lhs</code> <code>id</code> and each element of the <code>rhs</code> <code>id</code> and returns <code>lhs</code> <code>id</code>. If the operator returns a <code>bool</code> the result is the cast to <code>size_t</code>.</td></tr>
<tr><td><code>id &operatorOP(id &lhs, const size_t &rhs)</code></td><td>Where <code>OP</code> is: +=, -=,*=, /=, %=, <<=, >>=, &=, |=, ^=.<br>Assigns each element of <code>lhs</code> <code>id</code> instance with the result of an element-wise <code>OP</code> operator between each element of <code>lhs</code> <code>id</code> and the <code>rhs</code> <code>size_t</code> and returns <code>lhs</code> <code>id</code>. If the operator returns a <code>bool</code> the result is the cast to <code>size_t</code>.</td></tr>
<tr><td><code>id operatorOP(const size_t &lhs, const id &rhs)</code></td><td>Where <code>OP</code> is: +, -, *, /, %, <<, >>, &, |, ^, &&, ||, <, >, <=, >=.<br>Constructs and returns a new instance of the SYCL <code>id</code> class template with the same dimensionality as the <code>rhs</code> SYCL <code>id</code>, where each element of the new SYCL <code>id</code> instance is the result of an element-wise <code>OP</code> operator between the <code>lhs</code> <code>size_t</code> and each element of the <code>rhs</code> SYCL <code>id</code>. If the operator returns a <code>bool</code> the result is the cast to <code>size_t</code>.</td></tr>
<tr><td colspan="2" align="right">End of table</td></tr>
</table>

Table 4.79: Hidden friend functions of the `id` class template.

#### 4.10.1.4 `item` class

item identifies an instance of the function object executing at each point in a `range`. It is passed to a `parallel_for` call or returned by member functions of `h_item`. It encapsulates enough information to identify the work-item's range of possible values and its ID in that range. It can optionally carry the offset of the range if provided to the `parallel_for`. Instances of the `item` class are not user-constructible and are passed by the runtime to each instance of the function object.

The SYCL `item` class template provides the common by-value semantics (see Section 4.5.4).

A synopsis of the SYCL `item` class is provided below. The member functions of the SYCL `item` class are listed in Table 4.78. The additional common special member functions and common member functions are listed in 4.5.4 in Tables 4.3 and 4.4 respectively.

```
namespace sycl {
template <int dimensions = 1, bool with_offset = true>
class item {
public:
  item() = delete;

   /* -- common interface members -- */

  id<dimensions> get_id() const;

  size_t get_id(int dimension) const;

  size_t operator[](int dimension) const;

  range<dimensions> get_range() const;

  size_t get_range(int dimension) const;

  // only available if with_offset is true
  id<dimensions> get_offset() const;

  // only available if with_offset is false
  operator item<dimensions, true>() const;

  // only available if dimensions == 1
  operator size_t() const;

  size_t get_linear_id() const;
};
}  // namespace sycl
```

| Member function | Description |
| --- | --- |
| `id<dimensions> get_id()const` | Return the constituent `id` representing the work-item's position in the iteration space. |
| `size_t get_id(int dimension)const` | Return the same value as `get_id()[dimension]`. |
| `size_t operator[](int dimension)const` | Return the same value as `get_id(dimension)`. |
| `range<dimensions> get_range()const` | Returns a `range` representing the dimensions of the range of possible values of the `item`. |
| `size_t get_range(int dimension)const` | Return the same value as `get_range().get(dimension)`. |

Continued on next page

Table 4.80: Member functions for the `item` class.

| Member function | Description |
|---|---|
| `id<dimensions> get_offset()const` | Returns an `id` representing the *n*-dimensional offset provided to the `parallel_for` and that is added by the runtime to the global-ID of each work-item, if this item represents a global range. For an item converted from an item with no offset this will always return an `id` of all 0 values. This member function is only available if `with_offset` is `true`. |
| `operator item<dimensions, true>()const` | Available only when: `with_offset == false` Returns an `item` representing the same information as the object holds but also includes the offset set to 0. This conversion allow users to seamlessly write code that assumes an offset and still provides an offset-less `item`. |
| `operator size_t()const` | Available only when: `dimensions == 1` Returns the index representing the work-item position in the iteration space. |
| `size_t get_linear_id()const` | Return the id as a linear index value. Calculating a linear address from the multi-dimensional index follow the equation 4.3. |
| | End of table |

Table 4.80: Member functions for the `item` class.

#### 4.10.1.5 `nd_item` class

`nd_item<int dimensions>` identifies an instance of the function object executing at each point in an `nd_range<int dimensions>` passed to a `parallel_for` call. It encapsulates enough information to identify the work-item's local and global ids, the work-group id and also provides access to the `group` and `sub_group` classes. Instances of the `nd_item<int dimensions>` class are not user-constructible and are passed by the runtime to each instance of the function object.

The SYCL `nd_item` class template provides the common by-value semantics (see Section 4.5.4).

A synopsis of the SYCL `nd_item` class is provided below. The member functions of the SYCL `nd_item` class are listed in Table 4.81. The additional common special member functions and common member functions are listed in 4.5.4 in Tables 4.3 and 4.4 respectively.

```
namespace sycl {
template <int dimensions = 1>
class nd_item {
public:
  nd_item() = delete;

   /* -- common interface members -- */

```

```
  id<dimensions> get_global_id() const;

  size_t get_global_id(int dimension) const;

  size_t get_global_linear_id() const;

  id<dimensions> get_local_id() const;

  size_t get_local_id(int dimension) const;

  size_t get_local_linear_id() const;

  group<dimensions> get_group() const;

  size_t get_group(int dimension) const;

  size_t get_group_linear_id() const;

  range<dimensions> get_group_range() const;

  size_t get_group_range(int dimension) const;

  range<dimensions> get_global_range() const;

  size_t get_global_range(int dimension) const;

  range<dimensions> get_local_range() const;

  size_t get_local_range(int dimension) const;

  id<dimensions> get_offset() const;

  nd_range<dimensions> get_nd_range() const;

  template <typename dataT>
  device_event async_work_group_copy(decorated_local_ptr<dataT> dest,
    decorated_global_ptr<dataT> src, size_t numElements) const;

  template <typename dataT>
  device_event async_work_group_copy(decorated_global_ptr<dataT> dest,
    decorated_local_ptr<dataT> src, size_t numElements) const;

  template <typename dataT>
  device_event async_work_group_copy(decorated_local_ptr<dataT> dest,
    decorated_global_ptr<dataT> src, size_t numElements, size_t srcStride) const;

  template <typename dataT>
  device_event async_work_group_copy(decorated_global_ptr<dataT> dest,
    decorated_local_ptr<dataT> src, size_t numElements, size_t destStride) const;

  template <typename... eventTN>
  void wait_for(eventTN... events) const;
};
} // namespace sycl
```

| Member function | Description |
|---|---|
| `id<dimensions> get_global_id()const` | Return the constituent global id representing the work-item's position in the global iteration space. |
| `size_t get_global_id(int dimension)const` | Return the constituent element of the global id representing the work-item's position in the nd-range in the given `dimension`. |
| `size_t get_global_linear_id()const` | Return the flattened id of the current work-item after subtracting the offset. Calculating a linear id from a multi-dimensional index follows the equation 4.3. |
| `id<dimensions> get_local_id()const` | Return the constituent local id representing the work-item's position within the current work-group. |
| `size_t get_local_id(int dimension)const` | Return the constituent element of the local id representing the work-item's position within the current work-group in the given `dimension`. |
| `size_t get_local_linear_id()const` | Return the flattened id of the current work-item within the current work-group. Calculating a linear address from a multi-dimensional index follows the equation 4.3. |
| `group<dimensions> get_group()const` | Return the constituent work-group, `group` representing the work-group's position within the overall nd-range. |
| `sub_group get_sub_group()const` | Return a `sub_group` representing the sub-group to which the work-item belongs. |
| `size_t get_group(int dimension)const` | Return the constituent element of the group `id` representing the work-group's position within the overall `nd_range` in the given `dimension`. |
| `size_t get_group_linear_id()const` | Return the group id as a linear index value. Calculating a linear address from a multi-dimensional index follows the equation 4.3. |
| `range<dimensions> get_group_range()const` | Returns the number of work-groups in the iteration space. |
| `size_t get_group_range(int dimension)const` | Return the number of work-groups for `dimension` in the iteration space. |
| `range<dimensions> get_global_range()const` | Returns a `range` representing the dimensions of the global iteration space. |
| `size_t get_global_range(int dimension)const` | Return the same value as get_global_range().get(dimension) |
| `range<dimensions> get_local_range()const` | Returns a `range` representing the dimensions of the current work-group. |
| `size_t get_local_range(int dimension)const` | Return the same value as get_local_range().get(dimension) |

Continued on next page

Table 4.81: Member functions for the `nd_item` class.

| Member function | Description |
|---|---|
| `id<dimensions> get_offset()const` | Returns an id representing the n-dimensional offset provided to the constructor of the nd_range and that is added by the runtime to the global id of each work-item. |
| `nd_range<dimensions> get_nd_range()const` | Returns the nd_range of the current execution. |
| `template <typename dataT> device_event async_work_group_copy( decorated_local_ptr<dataT> dest, decorated_global_ptr<dataT> src, size_t numElements)const` | Permitted types for dataT are all scalar and vector types. Asynchronously copies a number of elements specified by numElements from the source pointer src to destination pointer dest and returns a SYCL device_event which can be used to wait on the completion of the copy. |
| `template <typename dataT> device_event async_work_group_copy( decorated_global_ptr<dataT> dest, decorated_local_ptr<dataT> src, size_t numElements)const` | Permitted types for dataT are all scalar and vector types. Asynchronously copies a number of elements specified by numElements from the source pointer src to destination pointer dest and returns a SYCL device_event which can be used to wait on the completion of the copy. |
| `template <typename dataT> device_event async_work_group_copy( decorated_local_ptr<dataT> dest, decorated_global_ptr<dataT> src, size_t numElements, size_t srcStride)const` | Permitted types for dataT are all scalar and vector types. Asynchronously copies a number of elements specified by numElements from the source pointer src to destination pointer dest with a source stride specified by srcStride and returns a SYCL device_event which can be used to wait on the completion of the copy. |
| `template <typename dataT> device_event async_work_group_copy( decorated_global_ptr<dataT> dest, decorated_local_ptr<dataT> src, size_t numElements, size_t destStride)const` | Permitted types for dataT are all scalar and vector types. Asynchronously copies a number of elements specified by numElements from the source pointer src to destination pointer dest with a destination stride specified by destStride and returns a SYCL device_event which can be used to wait on the completion of the copy. |
| `template <typename... eventTN> void wait_for(eventTN... events)const` | Permitted type for eventTN is device_event. Waits for the asynchronous operations associated with each device_event to complete. |
| | End of table |

Table 4.81: Member functions for the nd_item class.

#### 4.10.1.6 h_item class

h_item<int dimensions> identifies an instance of a group::parallel_for_work_item function object executing at each point in a local range<int dimensions> passed to a parallel_for_work_item call or to the corresponding parallel_for_work_group call if no range is passed to the parallel_for_work_item call. It encapsulates enough information to identify the work-item's local and global items according to the information given to `parallel_for_work_group` (physical ids) as well as the work-item's logical local items in the logical local range. All returned items objects are offset-less. Instances of the `h_item<int dimensions>` class are not user-constructible and are passed by the runtime to each instance of the function object.

The SYCL `h_item` class template provides the common by-value semantics (see Section 4.5.4).

A synopsis of the SYCL `h_item` class is provided below. The member functions of the SYCL `h_item` class are listed in Table 4.82. The additional common special member functions and common member functions are listed in 4.5.4 in Tables 4.3 and 4.4 respectively.

```
namespace sycl {
template <int dimensions>
class h_item {
public:
  h_item() = delete;

  /* -- common interface members -- */

  item<dimensions, false> get_global() const;

  item<dimensions, false> get_local() const;

  item<dimensions, false> get_logical_local() const;

  item<dimensions, false> get_physical_local() const;

  range<dimensions> get_global_range() const;

  size_t get_global_range(int dimension) const;

  id<dimensions> get_global_id() const;

  size_t get_global_id(int dimension) const;

  range<dimensions> get_local_range() const;

  size_t get_local_range(int dimension) const;

  id<dimensions> get_local_id() const;

  size_t get_local_id(int dimension) const;

  range<dimensions> get_logical_local_range() const;

  size_t get_logical_local_range(int dimension) const;

  id<dimensions> get_logical_local_id() const;

  size_t get_logical_local_id(int dimension) const;

  range<dimensions> get_physical_local_range() const;

  size_t get_physical_local_range(int dimension) const;
```

```

  id<dimensions> get_physical_local_id() const;

  size_t get_physical_local_id(int dimension) const;

};
} // namespace sycl
```

| Member function | Description |
|---|---|
| `item<dimensions, false> get_global()const` | Return the constituent global item representing the work-item's position in the global iteration space as provided upon kernel invocation. |
| `item<dimensions, false> get_local()const` | Return the same value as `get_logical_local()`. |
| `item<dimensions, false> get_logical_local()const` | Return the constituent element of the logical local item work-item's position in the local iteration space as provided upon the invocation of the `group::parallel_for_work_item`. If the `group::parallel_for_work_item` was called without any logical local range then the member function returns the physical local item. A physical id can be computed from a logical id by getting the remainder of the integer division of the logical id and the physical range: `get_logical_local().get()% get_physical_local.get_range()== get_physical_local().get()`. |
| `item<dimensions, false> get_physical_local()const` | Return the constituent element of the physical local item work-item's position in the local iteration space as provided (by the user or the runtime) upon the kernel invocation. |
| `range<dimensions> get_global_range()const` | Return the same value as `get_global().get_range()` |
| `size_t get_global_range(int dimension)const` | Return the same value as `get_global().get_range(dimension)` |
| `id<dimensions> get_global_id()const` | Return the same value as `get_global().get_id()` |
| `size_t get_global_id(int dimension)const` | Return the same value as `get_global().get_id(dimension)` |
| `range<dimensions> get_local_range()const` | Return the same value as `get_local().get_range()` |
| `size_t get_local_range(int dimension)const` | Return the same value as `get_local().get_range(dimension)` |
| `id<dimensions> get_local_id()const` | Return the same value as `get_local().get_id()` |

Continued on next page

Table 4.82: Member functions for the `h_item` class.

| Member function | Description |
|---|---|
| size_t get_local_id(int dimension)const | Return the same value as get_local().get_id(dimension) |
| range<dimensions> get_logical_local_range()const | Return the same value as get_logical_local().get_range() |
| size_t get_logical_local_range(int dimension)const | Return the same value as get_logical_local().get_range(dimension) |
| id<dimensions> get_logical_local_id()const | Return the same value as get_logical_local().get_id() |
| size_t get_logical_local_id(int dimension)const | Return the same value as get_logical_local().get_id(dimension) |
| range<dimensions> get_physical_local_range()const | Return the same value as get_physical_local().get_range() |
| size_t get_physical_local_range(int dimension)const | Return the same value as get_physical_local().get_range(dimension) |
| id<dimensions> get_physical_local_id()const | Return the same value as get_physical_local().get_id() |
| size_t get_physical_local_id(int dimension)const | Return the same value as get_physical_local().get_id(dimension) |
| | End of table |

Table 4.82: Member functions for the h_item class.

#### 4.10.1.7 group class

The group<int dimensions> encapsulates all functionality required to represent a particular work-group within a parallel execution. It is not user-constructable.

The local range stored in the group class is provided either by the programmer, when it is passed as an optional parameter to parallel_for_work_group, or by the runtime system when it selects the optimal work-group size. This allows the developer to always know how many concurrent work-items are active in each executing work-group, even through the abstracted iteration range of the parallel_for_work_item loops.

The SYCL group class template provides the common by-value semantics (see Section 4.5.4).

A synopsis of the SYCL group class is provided below. The member functions of the SYCL group class are listed in Table 4.83. The additional common special member functions and common member functions are listed in 4.5.4 in Tables 4.3 and 4.4 respectively.

```
namespace sycl {
template <int Dimensions = 1>
class group {
public:

  using id_type = id<Dimensions>;
  using range_type = range<Dimensions>;
  using linear_id_type = size_t;
  static constexpr int dimensions = Dimensions;
```

```

   /* -- common interface members -- */

  id<Dimensions> get_group_id() const;

  size_t get_group_id(int dimension) const;

  id<Dimensions> get_local_id() const;

  size_t get_local_id(int dimension) const;

  range<Dimensions> get_local_range() const;

  size_t get_local_range(int dimension) const;

  range<Dimensions> get_group_range() const;

  size_t get_group_range(int dimension) const;

  range<Dimensions> get_max_local_range() const;

  range<Dimensions> get_uniform_group_range() const;

  size_t operator[](int dimension) const;

  size_t get_group_linear_id() const;

  size_t get_local_linear_id() const;

  size_t get_group_linear_range() const;

  size_t get_local_linear_range() const;

  template<typename workItemFunctionT>
  void parallel_for_work_item(const workItemFunctionT &func) const;

  template<typename workItemFunctionT>
  void parallel_for_work_item(range<dimensions> logicalRange,
    const workItemFunctionT &func) const;

  template <typename dataT>
  device_event async_work_group_copy(decorated_local_ptr<dataT> dest,
    decorated_global_ptr<dataT> src, size_t numElements) const;

  template <typename dataT>
  device_event async_work_group_copy(decorated_global_ptr<dataT> dest,
    decorated_local_ptr<dataT> src, size_t numElements) const;

  template <typename dataT>
  device_event async_work_group_copy(decorated_local_ptr<dataT> dest,
    decorated_global_ptr<dataT> src, size_t numElements, size_t srcStride) const;

  template <typename dataT>
  device_event async_work_group_copy(decorated_global_ptr<dataT> dest,
    decorated_local_ptr<dataT> src, size_t numElements, size_t destStride) const;
```

```

  template <typename... eventTN>
  void wait_for(eventTN... events) const;
};
}  // sycl
```

| Member function | Description |
|---|---|
| `id<dimensions> get_group_id()const` | Return an id representing the index of the work-group within the nd-range for every dimension. |
| `size_t get_group_id(int dimension)const` | Return the same value as `get_id()[dimension]`. |
| `id<dimensions> get_local_id()const` | Return a SYCL id representing the calling work-item's position within the work-group. It is undefined behavior for this member function to be invoked from within a `parallel_for_work_item` context. |
| `size_t get_local_id(int dimension)const` | Return the calling work-item's position within the work-group in the specified dimension. It is undefined behavior for this member function to be invoked from within a `parallel_for_work_item` context. |
| `range<dimensions> get_local_range()const` | Return a SYCL range representing all dimensions of the local range. This local range may have been provided by the programmer, or chosen by the SYCL runtime. |
| `size_t get_local_range(int dimension)const` | Return the dimension of the local range specified by the `dimension` parameter. |
| `range<dimensions> get_group_range()const` | Return a range representing the number of work-groups in the `nd_range`. |
| `size_t get_group_range(int dimension)const` | Return element `dimension` from the constituent group range. |
| `size_t operator[](int dimension)const` | Return the same value as `get_id(dimension)`. |
| `range<dimensions> get_max_local_range()const` | Return a range representing the maximum number of work-items in any work-group in the `nd_range`. |
| `range<dimensions> get_uniform_group_range()const` | Return a range representing the number of work-groups in the uniform region of the `nd_range`. |
| `size_t get_group_linear_id()const` | Get a linearized version of the work-group id. Calculating a linear work-group id from a multi-dimensional index follows the equation 4.3. |
| `size_t get_group_linear_range()const` | Return the total number of work-groups in the `nd_range`. |

Continued on next page

Table 4.83: Member functions for the `group` class.

| Member function | Description |
|---|---|
| `size_t get_local_linear_id()const` | Get a linearized version of the calling work-item's local id. Calculating a linear local id from a multi-dimensional index follows the equation 4.3.<br>It is undefined behavior for this member function to be invoked from within a `parallel_for_work_item` context. |
| `size_t get_local_linear_range()const` | Return the total number of work-items in the work-group. |
| `template <typename workItemFunctionT> void parallel_for_work_item(const workItemFunctionT &func)const` | Launch the work-items for this work-group.<br>`func` is a function object type with a public member function `void F::operator()(h_item<dimensions>)` representing the work-item computation.<br>This member function can only be invoked within a `parallel_for_work_group` context.<br>It is undefined behavior for this member function to be invoked from within the `parallel_for_work_group` form that does not define work-group size, because then the number of work-items that should execute the code is not defined. It is expected that this form of `parallel_for_work_item` is invoked within the `parallel_for_work_group` form that specifies the size of a work-group. |

Continued on next page

Table 4.83: Member functions for the `group` class.

| Member function | Description |
|---|---|
| `template <typename workItemFunctionT> void parallel_for_work_item(range<dimensions> logicalRange, const workItemFunctionT &func) const` | Launch the work-items for this work-group using a logical local range. The function object `func` is executed as if the kernel where invoked with `logicalRange` as the local range. This new local range is emulated and may not map one-to-one with the physical range.<br>`logicalRange` is the new local range to be used. This range can be smaller or larger than the one used to invoke the kernel. `func` is a function object type with a public member function `void F::operator()(h_item<dimensions>)` representing the work-item computation.<br>Note that the logical range does not need to be uniform across all work-groups in a kernel. For example the logical range may depend on a work-group varying query (e.g. `group::get_linear_id`), such that different work-groups in the same kernel invocation execute different logical range sizes.<br>This member function can only be invoked within a `parallel_for_work_group` context. |
| `template <typename dataT> device_event async_work_group_copy( decorated_local_ptr<dataT> dest, decorated_global_ptr<dataT> src, size_t numElements)const` | Permitted types for `dataT` are all scalar and vector types. Asynchronously copies a number of elements specified by `numElements` from the source pointer `src` to destination pointer `dest` and returns a SYCL `device_event` which can be used to wait on the completion of the copy. |
| `template <typename dataT> device_event async_work_group_copy( decorated_global_ptr<dataT> dest, decorated_local_ptr<dataT> src, size_t numElements)const` | Permitted types for `dataT` are all scalar and vector types. Asynchronously copies a number of elements specified by `numElements` from the source pointer `src` to destination pointer `dest` and returns a SYCL `device_event` which can be used to wait on the completion of the copy. |
| `template <typename dataT> device_event async_work_group_copy( decorated_local_ptr<dataT> dest, decorated_global_ptr<dataT> src, size_t numElements, size_t srcStride)const` | Permitted types for `dataT` are all scalar and vector types. Asynchronously copies a number of elements specified by `numElements` from the source pointer `src` to destination pointer `dest` with a source stride specified by `srcStride` and returns a SYCL `device_event` which can be used to wait on the completion of the copy. |

Continued on next page

Table 4.83: Member functions for the group class.

| Member function | Description |
|---|---|
| `template <typename dataT> device_event async_work_group_copy( decorated_global_ptr<dataT> dest, decorated_local_ptr<dataT> src, size_t numElements, size_t destStride)const` | Permitted types for `dataT` are all scalar and vector types. Asynchronously copies a number of elements specified by `numElements` from the source pointer `src` to destination pointer `dest` with a destination stride specified by `destStride` and returns a SYCL `device_event` which can be used to wait on the completion of the copy. |
| `template <typename... eventTN> void wait_for(eventTN... events)const` | Permitted type for `eventTN` is `device_event`. Waits for the asynchronous operations associated with each `device_event` to complete. |
| | End of table |

Table 4.83: Member functions for the `group` class.

#### 4.10.1.8 `sub_group` class

The `sub_group` class encapsulates all functionality required to represent a particular sub-group within a parallel execution. It is not user-constructible.

The SYCL `sub_group` class provides the common by-value semantics (see Section 4.5.4).

A synopsis of the SYCL `sub_group` class is provided below. The member functions of the SYCL `sub_group` class are listed in Table 4.84. The additional common special member functions and common member functions are listed in 4.5.4 in Tables 4.3 and 4.4 respectively.

```
namespace sycl {
class sub_group {
public:

  using id_type = id<1>;
  using range_type = range<1>;
  using linear_id_type = uint32_t;
  static constexpr int dimensions = 1;

   /* -- common interface members -- */

  id<1> get_group_id() const;

  id<1> get_local_id() const;

  range<1> get_local_range() const;

  range<1> get_group_range() const;

  range<1> get_max_local_range() const;

  uint32_t get_group_linear_id() const;

  uint32_t get_local_linear_id() const;

  uint32_t get_group_linear_range() const;
```

```

  uint32_t get_local_linear_range() const;

};
} // sycl
```

| Member function | Description |
| --- | --- |
| id<1> get_group_id()const | Return an id representing the index of the sub-group within the work-group. |
| id<1> get_local_id()const | Return a SYCL id representing the calling work-item's position within the sub-group. |
| range<1> get_local_range()const | Return a SYCL range representing the size of the sub-group. This size may have been chosen by the programmer via an attribute, or chosen by the device compiler. |
| range<1> get_group_range()const | Return a range representing the number of sub-groups in the work-group. |
| range<1> get_max_local_range()const | Return a range representing the maximum number of work-items in any sub-group in the work-group. |
| uint32_t get_group_linear_id()const | Equivalent to get_group_id(). |
| uint32_t get_group_linear_range()const | Equivalent to get_group_range(). |
| uint32_t get_local_linear_id()const | Equivalent to get_local_id(). |
| uint32_t get_local_linear_range()const | Equivalent to get_local_range(). |
| | End of table |

Table 4.84: Member functions for the sub_group class.

#### 4.10.1.9 device_event class

The SYCL device_event class encapsulates a single SYCL device event which is available only within SYCL kernel functions and can be used to wait for asynchronous operations within a SYCL kernel function to complete. A SYCL device event can map to All member functions of the device_event class must not throw a SYCL exception.

A synopsis of the SYCL device_event class is provided below. The constructors and member functions of the SYCL device_event class are listed in Table 4.86 and 4.85 respectively.

```
namespace sycl {
class device_event {

  device_event(__unspecified__);

 public:
  void wait() noexcept;
};
} // namespace sycl
```

| Member function | Description |
|---|---|
| void wait()noexcept | Waits for the asynchronous operation associated with this SYCL device_event to complete. |
| | End of table |

Table 4.85: Member functions of the SYCL device_event class.

| Constructor | Description |
|---|---|
| device_event(___unspecified___) | Unspecified implementation defined constructor. |
| | End of table |

Table 4.86: Constructors of the device_event class.

### 4.10.2 Reduction variables

**Note for this provisional version:** The reduction features described in this section support two alternative approaches for creating reducer objects and launching reduction kernels. Both alternatives are shown here to encourage feedback from implementers and developers, and it is expected that the final version of the SYCL specification will include only one approach. Please provide feedback on your preference or issues with either approach, through an issue at https://github.com/KhronosGroup/SYCL-Docs/issues.

SYCL kernels support variables captured by parallel_for kernels being treated as reduction variables. All functionality related to reductions is captured by the reducer class, the reduction function, and the parallel_reduce function.

The example below demonstrates how to write a parallel_reduce kernel that performs two reductions simultaneously on the same input values, computing both the sum of all values in a buffer and the maximum value in the buffer. One reducer is created explicitly and captured by the kernel lambda, and the other is created using the reduction function and passed to parallel_reduce as an argument.

```
buffer<int> valuesBuf { 1024 };
{
  // Initialize buffer on the host with 0, 1, 2, 3, ..., 1023
  host_accessor a { valuesBuf };
  std::iota(a.begin(), a.end(), 0);
}

// Buffers with just 1 element to get the reduction results
buffer<int> sumBuf { 1 };
buffer<int> maxBuf { 1 };

myQueue.submit([&](handler& cgh) {

  // Input values to reductions are standard accessors
  auto inputValues = valuesBuf.get_access<access::mode::read>(cgh);

  // Create a reducer explicitly for a variable with reduction semantics
  auto sum = reducer(sumBuf.get_access(cgh), plus<>());
```

```

  // Create a temporary object describing a variable with reduction semantics
  auto max_reduction = reduction(maxBuf.get_access(cgh), maximum<>());

  // parallel_reduce operates on two reducers:
  // - sum is captured directly by the lambda
  // - max is created from max_reduction and passed to the lambda call operator
  cgh.parallel_reduce<class example_reduction>(range<1>{1024},
    max_reduction,
    [=](id<1> idx, auto& max) {
      // plus<>() corresponds to += operator, so sum can be updated via += or combine()
      sum += inputValues[idx];

      // maximum<>() has no shorthand operator, so max can only be updated via combine()
      max.combine(inputValues[idx]);
  });
});

// sumBuf and maxBuf contain the reduction results once the kernel completes
assert(maxBuf.get_host_access()[0] == 1023 && sumBuf.get_host_access()[0] == 523776);
```

Reductions are supported for all trivially copyable types. If the reduction operator is non-associative or non-commutative, the behavior of a reduction may be non-deterministic. If multiple reductions reference the same reduction variable, or a reduction variable is accessed directly during the lifetime of a reduction (e.g. via an `accessor` or USM pointer), the behavior is undefined.

For user-defined reduction operators, an implementation should issue a compile-time warning if the functor does not contain a `static constexpr` member called `identity_value`, an identity is not specified in the call to `reduction`, and this is known to negatively impact performance (e.g. as a result of the implementation choosing a different reduction algorithm). For standard binary operations (e.g. `plus`) on arithmetic types, the implementation must determine the correct identity automatically in order to avoid performance penalties.

A reduction operation associated with a multi-dimensional `accessor` or a `span` represents an array reduction. An array reduction of size $N$ is functionally equivalent to specifying $N$ independent scalar reductions. The combination operations performed by an array reduction are limited to the accessible region of a buffer described by an `accessor` or the extent of a USM allocation described by a `span`, and access to elements outside of these regions results in undefined behavior.

#### 4.10.2.1 `reduction` interface

The `reduction` interface is used to attach reduction semantics to a variable, by specifying: the reduction variable, the reduction operator and an optional identity value associated with the operator. The overloads of the interface are described in Table 4.87. The return value of the `reduction` interface is an implementation-defined object of unspecified type, which is interpreted by `parallel_reduce` to construct an appropriate `reducer` type as detailed in Section 4.10.2.2.

```
template <typename AccessorT, typename BinaryOperation>
__unspecified__ reduction(AccessorT vars, BinaryOperation combiner);

template <typename AccessorT, typename BinaryOperation>
__unspecified__ reduction(AccessorT vars, const T& identity, BinaryOperation combiner);

template <typename T, typename BinaryOperation>
```

```
__unspecified__ reduction(T* var, BinaryOperation combiner);

template <typename T, typename BinaryOperation>
__unspecified__ reduction(T* var, T& identity, BinaryOperation combiner);

template <typename T, typename Extent, typename BinaryOperation>
__unspecified__ reduction(span<T, Extent> vars, BinaryOperation combiner);

template <typename T, typename Extent, typename BinaryOperation>
__unspecified__ reduction(span<T, Extent> vars, const T& identity, BinaryOperation combiner);
```

| Function | Description |
|---|---|
| `reduction<AccessorT, BinaryOperation>(AccessorT vars, BinaryOperation combiner)` | Construct an unspecified object representing a reduction of the variable(s) described by `vars` using the combination operation specified by `combiner`. If the access mode of `vars` is `access::mode::read_write` then the reduction operation includes the original value(s) of the variable(s) described by `vars`. `vars` must not be a placeholder accessor.<br><br>Available only when: `(accessMode == access::mode::read_write \|\| accessMode == access::mode::discard_write)&& accessTarget == access::target::global_buffer` |
| `reduction<AccessorT, BinaryOperation>(AccessorT vars, AccessorT::value_type identity, BinaryOperation combiner)` | Construct an unspecified object representing a reduction of the variable(s) described by `vars` using the combination operation specified by `combiner`. The value of `identity` may be used by the implementation to initialize temporary accumulation variables; using an `identity` value that is not the identity value of the combination operation specified by `combiner` results in undefined behavior. If the access mode of `vars` is `access::mode::read_write` then the reduction operation includes the original value(s) of the variable(s) described by `vars`. `vars` must not be a placeholder accessor.<br><br>Available only when: `(accessMode == access::mode::read_write \|\| accessMode == access::mode::discard_write)&& accessTarget == access::target::global_buffer` |

Continued on next page

Table 4.87: Overloads of the `reduction` interface.

| Function | Description |
|---|---|
| `reduction<T, BinaryOperation>(T* var, BinaryOperation combiner)` | Construct an unspecified object representing a reduction of the variable described by `var` using the combination operation specified by `combiner`. The reduction operation includes the original value of the variable described by `var`. |
| `reduction<T, BinaryOperation>(T* var, T identity, BinaryOperation combiner)` | Construct an unspecified object representing a reduction of the variable described by `var` using the combination operation specified by `combiner`. The value of `identity` may be used by the implementation to initialize temporary accumulation variables; using an `identity` value that is not the identity value of the combination operation specified by `combiner` results in undefined behavior. The reduction operation includes the original value of the variable described by `var`. |
| `reduction<T, BinaryOperation>(span<T, Extent> vars, BinaryOperation combiner)` | Construct an unspecified object representing a reduction of the variable(s) described by `vars` using the combination operation specified by `combiner`. The reduction operation includes the original value(s) of the variable described by `vars`. |
| `reduction<T, BinaryOperation>(span<T, Extent> vars, T identity, BinaryOperation combiner)` | Construct an unspecified object representing a reduction of the variable(s) described by `vars` using the combination operation specified by `combiner`. The value of `identity` may be used by the implementation to initialize temporary accumulation variables; using an `identity` value that is not the identity value of the combination operation specified by `combiner` results in undefined behavior. The reduction operation includes the original value(s) of the variable(s) described by `vars`. |
| | End of table |

Table 4.87: Overloads of the `reduction` interface.

#### 4.10.2.2 `reducer` class

The `reducer` class defines the interface between a work-item and a reduction variable during the execution of a SYCL kernel, restricting access to the underlying reduction variable. The intermediate values of a reduction variable cannot be inspected during kernel execution, and the variable cannot be updated using anything other than the reduction's specified combination operation. The combination order of different reducers is unspecified, as are when and how the value of each reducer is combined with the original reduction variable.

A `reducer` can be constructed explicitly by a user and bound to a command group handler, or constructed by an implementation of `parallel_reduce` given the return value of a call to the `reduction` function. To enable compile-time specialization of reduction algorithms, the implementation of the `reducer` class is unspecified, except for the functions and operators defined in Tables 4.88, 4.89 and 4.90. It is recommended that developers use `auto` in place of specifying the template arguments of a `reducer` directly.

An implementation must guarantee that it is safe for each concurrently executing work-item in a kernel to call the combine function of a `reducer` in parallel. An implementation is free to re-use reducer variables (e.g. across work-groups scheduled to the same compute unit) if it can guarantee that it is safe to do so.

The member functions of the `reducer` class are listed in Table 4.89. Additional shorthand operators may be made available for certain combinations of reduction variable type and combination operation, as described in Table 4.90.

```
// Exposition only
template <typename T, typename BinaryOperation, int Dimensions>
class reducer {

  template <access_mode Mode>
  reducer(accessor<T, Dimensions, Mode> vars, BinaryOperation combiner);

  template <access_mode Mode>
  reducer(accessor<T, Dimensions, Mode> vars, const T& identity, BinaryOperation combiner);

  reducer(T* var, BinaryOperation combiner, handler& cgh);

  reducer(T* var, const T& identity, BinaryOperation combiner, handler& cgh);

  template <typename Extent>
  reducer(span<T, Extent> vars, BinaryOperation combiner, handler& cgh);

  template <typename Extent>
  reducer(span<T, Extent> vars, const T& identity, BinaryOperation combiner, handler& cgh);

  reducer(const reducer<T,BinaryOperation,Dimensions>&) = delete;
  reducer<T,BinaryOperation,Dimensions>& operator(const reducer<T,BinaryOperation,Dimensions>&) =
      delete;

  /* Only available if Dimensions == 0 */
  void combine(const T& partial);

  /* Only available if Dimensions > 1 */
  __unspecified__ &operator[](size_t index) const;

  /* Only available if identity value is known */
  T identity() const;

};

template <typename T>
void operator+=(reducer<T,plus<T>,0>&, const T&);

template <typename T>
void operator*=(reducer<T,multiplies<T>,0>&, const T&);

/* Only available for integral types */
```

```
template <typename T>
void operator&=(reducer<T,bit_and<T>,0>&, const T&);

/* Only available for integral types */
template <typename T>
void operator|=(reducer<T,bit_or<T>,0>&, const T&);

/* Only available for integral types */
template <typename T>
void operator^=(reducer<T,bit_xor<T>,0>&, const T&);

/* Only available for integral types */
template <typename T>
void operator++(reducer<T,plus<T>,0>&);
```

| Constructor | Description |
|---|---|
| `reducer<access_mode Mode>(accessor<T, Dimensions, Mode> vars, BinaryOperation combiner)` | Construct a `reducer` representing a reduction of the variable(s) described by `vars` using the combination operation specified by `combiner`. If the access mode of `vars` is `access::mode::read_write` then the reduction operation includes the original value(s) of the variable(s) described by `vars`. `vars` must not be a placeholder accessor.<br><br>Available only when: `(accessMode == access::mode::read_write \|\| accessMode == access::mode::discard_write)&& accessTarget == access::target::global_buffer` |
| `reducer<access_mode Mode>(accessor<T, Dimensions, Mode> vars, const T& identity, BinaryOperation combiner)` | Construct a `reducer` representing a reduction of the variable(s) described by `vars` using the combination operation specified by `combiner`. The value of `identity` may be used by the implementation to initialize temporary accumulation variables; using an `identity` value that is not the identity value of the combination operation specified by `combiner` results in undefined behavior. If the access mode of `vars` is `access::mode::read_write` then the reduction operation includes the original value(s) of the variable(s) described by `vars`. `vars` must not be a placeholder accessor.<br><br>Available only when: `(accessMode == access::mode::read_write \|\| accessMode == access::mode::discard_write)&& accessTarget == access::target::global_buffer` |

Continued on next page

Table 4.88: Constructors of the reducer class.

| Constructor | Description |
|---|---|
| `reducer(T* var, BinaryOperation combiner, handler& cgh)` | Construct a `reducer` representing a reduction of the variable described by `var` using the combination operation specified by `combiner`. The reduction operation includes the original value of the variable described by `var`. |
| `reducer(T* var, const T& identity, BinaryOperation combiner, handler& cgh)` | Construct a `reducer` representing a reduction of the variable described by `var` using the combination operation specified by `combiner`. The value of `identity` may be used by the implementation to initialize temporary accumulation variables; using an `identity` value that is not the identity value of the combination operation specified by `combiner` results in undefined behavior. The reduction operation includes the original value of the variable described by `var`. |
| `reducer<Extent>(span<T, Extent> vars, BinaryOperation combine, handler& cgh)` | Construct a `reducer` representing a reduction of the variable(s) described by `vars` using the combination operation specified by `combiner`. The reduction operation includes the original value(s) of the variable described by `vars`. |
| `reducer<Extent>(span<T, Extent> vars, const T& identity, BinaryOperation combiner, handler& cgh)` | Construct a `reducer` representing a reduction of the variable(s) described by `vars` using the combination operation specified by `combiner`. The value of `identity` may be used by the implementation to initialize temporary accumulation variables; using an `identity` value that is not the identity value of the combination operation specified by `combiner` results in undefined behavior. The reduction operation includes the original value(s) of the variable(s) described by `vars`. |
| | End of table |

Table 4.88: Constructors of the `reducer` class.

| Member function | Description |
|---|---|
| `void combine(const T& partial)const` | Combine the value of `partial` with the reduction variable associated with this `reducer`. |
| | Continued on next page |

Table 4.89: Member functions of the `reducer` class.

| Member function | Description |
|---|---|
| `__unspecified__ &operator[](size_t index)const` | Available only when: `Dimensions > 1`. Returns an instance of an undefined intermediate type representing a `reducer` of the same type as this `reducer`, with the dimensionality `Dimensions-1` and containing an implicit SYCL `id` with index `Dimensions` set to `index`. The intermediate type returned must provide all member functions and operators defined by the `reducer` class that are appropriate for the type it represents (including this subscript operator). |
| `T identity()const` | Return the identity value of the combination operation associated with this `reducer`. Only available if the identity value is known to the implementation, or was specified explicitly in the call to `reduction` that returned this `reducer`. |
| | End of table |

Table 4.89: Member functions of the `reducer` class.

| Operator | Description |
|---|---|
| `template <typename T> void operator+=(reducer<T,plus<T>,0>& accum, const T& partial)` | Equivalent to calling `accum.combine(partial)`. |
| `template <typename T> void operator*=(reducer<T,multiplies<T>,0>& accum, const T& partial)` | Equivalent to calling `accum.combine(partial)`. |
| `template <typename T> void operator\|=(reducer<T,bit_or<T>,0>& accum, const T& partial)` | Equivalent to calling `accum.combine(partial)`. Only available for integral types. |
| `template <typename T> void operator&=(reducer<T,bit_and<T>,0>& accum, const T& partial)` | Equivalent to calling `accum.combine(partial)`. Only available for integral types. |
| `template <typename T> void operator^=(reducer<T,bit_xor<T>,0>& accum, const T& partial)` | Equivalent to calling `accum.combine(partial)`. Only available for integral types. |
| `template <typename T> void operator++(reducer<T,bit_xor<T>,0>& accum)` | Equivalent to calling `accum.combine(1)`. Only available for integral types. |
| | End of table |

Table 4.90: Operators of the `reducer` class.

### 4.10.3 Command group scope

A command group scope in SYCL, as it is defined in Section 3.6.1, consists of a single kernel or explicit memory operation (`handler` methods such as `copy`, `update_host`, `fill`), together with its **requirements**. The commands that enqueue a kernel or explicit memory operation and the requirements for its execution form the command group function object. The command group function object takes as a parameter an instance of the command group handler class which encapsulates all the member functions executed in the command group scope. The methods and objects defined in this scope will define the requirements for the kernel execution or explicit memory operation, and will be used by the SYCL runtime to evaluate if the operation is ready for execution. Host code within a command group function object (typically setting up requirements) is executed once, before the command group submit call returns. This abstraction of the kernel execution unifies the data with its processing, and consequently allows more abstraction and flexibility in the parallel programming models that can be implemented on top of SYCL.

The command group function object and the `handler` class serve as an interface for the encapsulation of command group scope. A SYCL kernel function is defined as a function object. All the device data accesses are defined inside this group and any transfers are managed by the SYCL runtime. The rules for the data transfers regarding device and host data accesses are better described in the data management section (4.7), where buffers (4.7.2) and accessor (4.7.6) classes are described. The overall memory model of the SYCL application is described in Section 3.7.1.

It is possible to obtain events for the start of the command group function object, the kernel starting, and the command group completing. These events are most useful for profiling, because safe synchronization in SYCL requires synchronization on buffer availability, not on kernel completion. This is because the memory that data is stored in upon kernel completion is not rigidly specified. The events are provided at the submission of the command group function object to the queue to be executed on.

It is possible for a command group function object to fail to enqueue to a queue, or for it to fail to execute correctly. A user can therefore supply a secondary queue when submitting a command group to the primary queue. If the SYCL runtime fails to enqueue or execute a command group on a primary queue, it can attempt to run the command group on the secondary queue. The circumstances in which it is, or is not, possible for a SYCL runtime to fall-back from primary to secondary queue are unspecified in the specification. Even if a command group is run on the secondary queue, the requirement that host code within the command group is executed exactly once remains, regardless of whether the fallback queue is used for execution.

The command group `handler` class provides the interface for all of the member functions that are able to be executed inside the command group scope, and it is also provided as a scoped object to all of the data access requests. The command group handler class provides the interface in which every command in the command group scope will be submitted to a queue.

### 4.10.4 Command group `handler` class

A command group handler object can only be constructed by the SYCL runtime. All of the accessors defined in command group scope take as a parameter an instance of the command group handler, and all the kernel invocation functions are member functions of this class.

The constructors of the SYCL `handler` class are described in Table 4.91.

It is disallowed for an instance of the SYCL `handler` class to be moved or copied.

```
namespace sycl {

class handler {
 private:

  // implementation defined constructor
  handler(___unspecified___);
```

```

public:

 template <typename dataT, int dimensions, access::mode accessMode,
   access::target accessTarget, access::placeholder isPlaceholder>
 void require(accessor<dataT, dimensions, accessMode, accessTarget,
              isPlaceholder> acc);

 //----- OpenCL interoperability interface
 //
 template <typename T>
 void set_arg(int argIndex, T && arg);

 template <typename... Ts>
 void set_args(Ts &&... args);

 //------ Kernel dispatch API
 //
 // Note: In all Kernel dispatch functions,
 // when using a functor with a globally visible name
 // the template parameter:"typename kernelName" can be ommitted
 // and the kernelType can be used instead.
 //
 template <typename KernelName, typename KernelType>
 void single_task(const KernelType &kernelFunc);

 template <typename KernelName, typename KernelType, int dimensions>
 void parallel_for(range<dimensions> numWorkItems, const KernelType &kernelFunc);

 template <typename KernelName, typename KernelType, int dimensions>
 void parallel_for(range<dimensions> numWorkItems,
                   id<dimensions> workItemOffset, const KernelType &kernelFunc);

 template <typename KernelName, typename KernelType, int dimensions>
 void parallel_for(nd_range<dimensions> executionRange, const KernelType &kernelFunc);

 template <typename KernelName, typename WorkgroupFunctionType, int dimensions>
 void parallel_for_work_group(range<dimensions> numWorkGroups,
                              const WorkgroupFunctionType &kernelFunc);

 template <typename KernelName, typename WorkgroupFunctionType, int dimensions>
 void parallel_for_work_group(range<dimensions> numWorkGroups,
                              range<dimensions> workGroupSize,
                              const WorkgroupFunctionType *kernelFunc);

 void single_task(kernel syclKernel);

 template <int dimensions>
 void parallel_for(range<dimensions> numWorkItems, kernel syclKernel);

 template <int dimensions>
 void parallel_for(range<dimensions> numWorkItems,
                   id<dimensions> workItemOffset, kernel syclKernel);

 template <int dimensions>
```

```
void parallel_for(nd_range<dimensions> ndRange, kernel syclKernel);

//------ Explicit memory operation APIs
//
template <typename T_src, int dim_src, access::mode mode_src, access::target tgt_src, access::placeholder isPlaceholder,
          typename T_dest>
void copy(accessor<T_src, dim_src, mode_src, tgt_src, isPlaceholder> src,
          std::shared_ptr<T_dest> dest);

template <typename T_src,
          typename T_dest, int dim_dest, access::mode mode_dest, access::target tgt_dest, access::placeholder isPlaceholder>
void copy(std::shared_ptr<T_src> src,
          accessor<T_dest, dim_dest, mode_dest, tgt_dest, isPlaceholder> dest);

template <typename T_src, int dim_src, access::mode mode_src, access::target tgt_src, access::placeholder isPlaceholder,
          typename T_dest>
void copy(accessor<T_src, dim_src, mode_src, tgt_src, isPlaceholder> src,
          T_dest *dest);

template <typename T_src,
          typename T_dest, int dim_dest, access::mode mode_dest, access::target tgt_dest, access::placeholder isPlaceholder>
void copy(const T_src *src,
          accessor<T_dest, dim_dest, mode_dest, tgt_dest, isPlaceholder> dest);

template <typename T_src, int dim_src, access::mode mode_src, access::target tgt_src, access::placeholder isPlaceholder_src,
          typename T_dest, int dim_dest, access::mode mode_dest, access::target tgt_dest, access::placeholder isPlaceholder_dest>
void copy(accessor<T_src, dim_src, mode_src, tgt_src, isPlaceholder_src> src,
          accessor<T_dest, dim_dest, mode_dest, tgt_dest, isPlaceholder_dest> dest);

template <typename T, int dim, access::mode mode, access::target tgt, access::placeholder isPlaceholder>
void update_host(accessor<T, dim, mode, tgt, isPlaceholder> acc);

template <typename T, int dim, access::mode mode, access::target tgt, access::placeholder isPlaceholder>
void fill(accessor<T, dim, mode, tgt, isPlaceholder> dest, const T& src);

void use_module(const module<module_state::executable> &execModule);

template <typename T>
void use_module(const module<module_state::executable> &execModule
                T deviceImageSelector);

template<auto& S>
bool has_specialization_constant() const noexcept;

template<auto& S>
typename std::remove_reference_t<decltype(S)>::type get_specialization_constant();

```

```
};
} // namespace sycl
```

| Constructor | Description |
|---|---|
| `handler(___unspecified___)` | Unspecified implementation defined constructor. |
| | End of table |

Table 4.91: Constructors of the `handler` class.

## 4.10.5 Class `kernel_handler`

Functionality and queries that are unique to the invocation of a SYCL kernel function are made available at the kernel scope via a kernel handler. A kernel handler is associated with the SYCL kernel function this is being invoked and the module and device image used by the SYCL runtime. A kernel handler is represented by the class `kernel_handler`.

The kernel handler is optional, and is only used if the SYCL kernel function has a `kernel_handler` as an additional parameter, in which case the SYCL runtime will construct an instance of `kernel_handler` and pass it to the SYCL kernel function as an argument. A `kernel_handler` is not user constructible and can only be constructed by the SYCL runtime.

```
class kernel_handler;
```

### 4.10.5.1 Constructors

```
kernel_handler(__unspecified__); // (1)
```

1. *Effects:* Unspecified private constructor for the SYCL runtime to use to construct a kernel handler.

### 4.10.5.2 Member functions

```
template<auto& S>
bool has_specialization_constant() const noexcept; // (1)
```

1. *Returns:* `true` if any of the SYCL kernel functions represented by the associated module contains the specialization constant represented by the `specialization_id` at the address `S`, otherwise returns `false`.

```
template<auto& S>
typename std::remove_reference_t<decltype(S)>::type get_specialization_constant(); // (1)
```

1. *Returns:* The value of the specialization constant associated with the `specialization_id` at the address `S`, from the associated module, if the specialization constant has been set, otherwise returns the default value.

   *Effects:* If the associated module is associated with a host context or `this->has_specialization_constant<S>()` evaluates to `false` this member function is undefined.

### 4.10.6 SYCL functions for adding requirements

Requirements for execution of SYCL kernels can be specified directly using handler methods.

| Member function | Description |
|---|---|
| `template <typename dataT, int dimensions, access::mode accessMode, access::target accessTarget , access::placeholder isPlaceholder> void require(accessor<dataT, dimensions, accessMode, accessTarget, isPlaceholder> acc)` | Requires access to the memory object associated with the accessor.<br>The command group now has a **requirement** to gain access to the given memory object before executing the kernel. If the accessor has already been registered with the command group, calling this function has no effect.<br>Throws `exception` with the `errc::accessor_error` error code if `(acc.empty()== true)`. |
| | End of table |

Table 4.92: Member functions of the `handler` class.

### 4.10.7 SYCL functions for invoking kernels

Kernels can be invoked as *single tasks*, basic *data-parallel* kernels, nd-range in work-groups, or *hierarchical parallelism*.

Each function takes an optional kernel name template parameter. The user may optionally provide a kernel name, otherwise an implementation defined name will be generated for the kernel.

All the functions for invoking kernels are member functions of the command group `handler` class 4.10.4, which is used to encapsulate all the member functions provided in a command group scope. Table 4.93 lists all the members of the `handler` class related to the kernel invocation.

| Member function | Description |
|---|---|
| `template <typename T> void set_arg(int argIndex, T &&arg)` | Set a kernel argument for a kernel through the SYCL interoperability interface. The `index` value specifies which parameter of the low-level kernel is being set and `arg` specifies the kernel argument.<br>Index 0 is the first parameter.<br>The argument can be either a SYCL accessor, a SYCL sampler or a trivially copyable C++ type.<br>Note it is invalid to set arguments to SYCL kernel function objects. |
| `template <typename... Ts> void set_args(Ts &&... args)` | Set all the given kernel `args` arguments for an kernel, as if `set_arg()` was used with each of them in the same order and increasing index always starting at 0. |
| | Continued on next page |

Table 4.93: Member functions of the `handler` class.

| Member function | Description |
|---|---|
| `template <typename KernelName, typename KernelType>`<br>`void single_task(const KernelType &kernelFunc)` | Defines and invokes a SYCL kernel function as a lambda function or a named function object type. Specification of a kernel name (`typename KernelName`), as described in Section 4.10.7, is optional. The callable `KernelType` can optionally take a `kernel_handler` in which case the SYCL runtime will construct an instance of `kernel_handler` and pass it to `KernelType`. |
| `template <typename KernelName, typename KernelType, int dimensions>`<br>`void parallel_for(`<br>`range<dimensions> numWorkItems,`<br>`const KernelType &kernelFunc)` | Defines and invokes a SYCL kernel function as a lambda function or a named function object type, for the specified range and given an item or integral type (e.g `int`, `size_t`), if range is 1-dimensional, for indexing in the indexing space defined by range. Generic kernel functions are permitted, in that case the argument type is an `item`. Specification of a kernel name (`typename KernelName`), as described in Section 4.10.7, is optional. The callable `KernelType` can optionally take a `kernel_handler` as it's last parameter, in which case the SYCL runtime will construct an instance of `kernel_handler` and pass it to `KernelType`. |
| `template <typename KernelName, typename KernelType, int dimensions>`<br>`void parallel_for(`<br>`range<dimensions> numWorkItems,`<br>`id<dimensions> workItemOffset,`<br>`const KernelType &kernelFunc)` | Defines and invokes a SYCL kernel function as a lambda function or a named function object type, for the specified range and offset and given an an item or integral type (e.g `int`, `size_t`), if range is 1-dimensional, for indexing in the indexing space defined by range. Generic kernel functions are permitted, in that case the argument type is an `item`. Specification of a kernel name (`typename KernelName`), as described in Section 4.10.7, is optional. The callable `KernelType` can optionally take a `kernel_handler` as it's last parameter, in which case the SYCL runtime will construct an instance of `kernel_handler` and pass it to `KernelType`. |

Continued on next page

Table 4.93: Member functions of the `handler` class.

| Member function | Description |
|---|---|
| `template <typename KernelName, typename KernelType, int dimensions> void parallel_for( nd_range<dimensions> executionRange, const KernelType &kernelFunc)` | Defines and invokes a SYCL kernel function as a lambda function or a named function object type, for the specified nd-range and given an nd-item for indexing in the indexing space defined by the nd-range. Generic kernel functions are permitted, in that case the argument type is an nd-item. Specification of a kernel name (`typename KernelName`), as described in Section 4.10.7, is optional. The callable `KernelType` can optionally take a `kernel_handler` as it's last parameter, in which case the SYCL runtime will construct an instance of `kernel_handler` and pass it to `KernelType`. |
| `template <typename KernelName, typename WorkgroupFunctionType, int dimensions> void parallel_for_work_group( range<dimensions> numWorkGroups, const WorkgroupFunctionType &kernelFunc)` | Hierarchical kernel invocation method of a kernel defined as a lambda encoding the body of each work-group to launch. Generic kernel functions are permitted, in that case the argument type is a `group`. May contain multiple calls to `parallel_for_work_item(..)` methods representing the execution on each work-item. Launches `num_work_groups` work-groups of runtime-defined size. Described in detail in 4.10.7. The callable `WorkgroupFunctionType` can optionally take a `kernel_handler` as it's last parameter, in which case the SYCL runtime will construct an instance of `kernel_handler` and pass it to `KernelType`. |
| `template <typename KernelName, typename WorkgroupFunctionType, int dimensions> void parallel_for_work_group( range<dimensions> numWorkGroups, range<dimensions> workGroupSize, const WorkgroupFunctionType &kernelFunc)` | Hierarchical kernel invocation method of a kernel defined as a lambda encoding the body of each work-group to launch. Generic kernel functions are permitted, in that case the argument type is a `group`. May contain multiple calls to `parallel_for_work_item` methods representing the execution on each work-item. Launches `num_work_groups` work-groups of `work_group_size` work-items each. Described in detail in 4.10.7. The callable `WorkgroupFunctionType` can optionally take a `kernel_handler` as it's last parameter, in which case the SYCL runtime will construct an instance of `kernel_handler` and pass it to `KernelType` . |

Continued on next page

Table 4.93: Member functions of the `handler` class.

| Member function | Description |
| --- | --- |
| `template <typename KernelName, typename KernelType, int dimensions, typename... Reductions> void parallel_reduce( range<dimensions> numWorkItems, Reductions... reductions, KernelType kernelFunc)` | Defines and invokes a SYCL kernel function as a lambda function or a named function object type, for the specified range and given an id or item for indexing in the indexing space defined by range. If it is a named function object and the function object type is globally visible there is no need for the developer to provide a kernel name (`typename KernelName`) for it, as described in 4.10.7. The callable `KernelType` can optionally take a `kernel_handler` as it's last parameter, in which case the SYCL runtime will construct an instance of `kernel_handler` and pass it to `KernelType`.<br><br>The `reductions` parameter pack consists of 1 or more objects created by the `reduction` function. For each object in `reductions`, the kernel functor should take an additional parameter corresponding to that object's `reducer` type. |
| `template <typename KernelName, typename KernelType, int dimensions, typename... Reductions> void parallel_reduce( range<dimensions> numWorkItems, id<dimensions> workItemOffset, Reductions... reductions, KernelType kernelFunc)` | Defines and invokes a SYCL kernel function as a lambda function or a named function object type, for the specified range and offset and given an id or item for indexing in the indexing space defined by range. If it is a named function object and the function object type is globally visible there is no need for the developer to provide a kernel name (`typename KernelName`) for it, as described in 4.10.7. The callable `KernelType` can optionally take a `kernel_handler` as it's last parameter, in which case the SYCL runtime will construct an instance of `kernel_handler` and pass it to `KernelType`.<br><br>The `reductions` parameter pack consists of 1 or more objects created by the `reduction` function. For each object in `reductions`, the kernel functor should take an additional parameter corresponding to that object's `reducer` type. |

Continued on next page

Table 4.93: Member functions of the `handler` class.

| Member function | Description |
|---|---|
| `template <typename KernelName, typename KernelType, int dimensions, typename... Reductions> void parallel_reduce( nd_range<dimensions> executionRange, Reductions ... reductions, KernelType kernelFunc)` | Defines and invokes a SYCL kernel function as a lambda function or a named function object type, for the specified nd-range and given an nd-item for indexing in the indexing space defined by the nd-range. If it is a named function object and the function object type is globally visible there is no need for the developer to provide a kernel name (`typename KernelName`) for it, as described in 4.10.7. The callable `KernelType` can optionally take a `kernel_handler` as it's last parameter, in which case the SYCL runtime will construct an instance of `kernel_handler` and pass it to `KernelType`. as described in 4.10.7.<br><br>The `reductions` parameter pack consists of 1 or more objects created by the `reduction` function. For each object in `reductions`, the kernel functor should take an additional parameter corresponding to that object's `reducer` type. |
| `void single_task(kernel syclKernel)` | Defines and invokes a SYCL kernel function as a lambda function or a named function object type, executes exactly once. |
| `template <int dimensions> void parallel_for( range<dimensions> numWorkItems, kernel syclKernel)` | Kernel invocation method of a pre-compiled kernel defined by SYCL `sycl-kernel-function` instance, for the specified range and given an item or integral type (e.g `int` , `size_t`), if range is 1-dimensional, for indexing in the indexing space defined by range. Generic kernel functions are permitted, in that case the argument type is an `item` . Described in detail in 4.10.7. |
| `template <int dimensions> void parallel_for( range<dimensions> numWorkItems, id<dimensions> workItemOffset, kernel syclKernel )` | Kernel invocation method of a pre-compiled kernel defined by SYCL `sycl-kernel-function` instance, for the specified range and offset and given an item or integral type (e.g `int`, `size_t`), if range is 1-dimensional, for indexing in the indexing space defined by range. Generic kernel functions are permitted, in that case the argument type is an `item`. Described in detail in 4.10.7. |

Continued on next page

Table 4.93: Member functions of the `handler` class.

| Member function | Description |
|---|---|
| `template <int dimensions> void parallel_for( nd_range<dimensions> ndRange, kernel syclKernel)` | Kernel invocation method of a pre-compiled kernel defined by SYCL `kernel` instance, for the specified `ndrange` and given an `nd_item` for indexing in the indexing space defined by the `nd_range`. Generic kernel functions are permitted, in that case the argument type is an `nd_item`. Described in detail in 4.10.7. |
| | End of table |

Table 4.93: Member functions of the `handler` class.

#### 4.10.7.1 `single_task` invoke

SYCL provides a simple interface to enqueue a kernel that will be sequentially executed on a device. Only one instance of the kernel will be executed. This interface is useful as a primitive for more complicated parallel algorithms, as it can easily create a chain of sequential tasks on a SYCL device with each of them managing its own data transfers.

This function can only be called inside a command group using the `handler` object created by the runtime. Any accessors that are used in a kernel should be defined inside the same command group.

Local accessors are disallowed for single task invocations.

```
myQueue.submit([&](handler & cgh) {
cgh.single_task<class kernel_name>(
    [=] () {
      // [kernel code]
    }));
});
```

For single tasks, the kernel method takes no parameters, as there is no need for index space classes in a unary index space.

A `kernel_handler` can optionally be passed as a parameter to the SYCL kernel function that is invoked by `single_task`.

```
myQueue.submit([&](handler & cgh) {
cgh.single_task<class kernel_name>(
    [=] (kernel_handler kh) {
      // [kernel code]
    }));
});
```

#### 4.10.7.2 `parallel_for` invoke

The `parallel_for` member function of the SYCL `handler` class provides an interface to define and invoke a SYCL kernel function in a command group, to execute in parallel execution over a 3 dimensional index space. There are three overloads of the `parallel_for` member function which provide variations of this interface, each with a different level of complexity and providing a different set of features.

For the simplest case, users need only provide the global range (the total number of work-items in the index space) via a SYCL range parameter, and the SYCL runtime will select a local range (the number of work-items in each work-group). The local range chosen by the SYCL runtime is entirely implementation defined. In this case the function object that represents the SYCL kernel function must take one of: 1) a single SYCL item parameter, 2) single generic parameter (template parameter or auto), 3) any other type implicitly converted from SYCL item, representing the currently executing work-item within the range specified by the range parameter.

The execution of the kernel function is the same whether the parameter to the SYCL kernel function is a SYCL id or a SYCL item. What differs is the functionality that is available to the SYCL kernel function via the respective interfaces.

Below is an example of invoking a SYCL kernel function with parallel_for using a lambda function, and passing a SYCL id parameter. In this case only the global id is available. This variant of parallel_for is designed for when it is not necessary to query the global range of the index space being executed across, or the local (work-group) size chosen by the implementation.

```
myQueue.submit([&](handler & cgh) {
    accessor acc { myBuffer, cgh, write_only };

    cgh.parallel_for<class myKernel>(range<1>(numWorkItems),
                                     [=] (id<1> index) {
        acc[index] = 42.0f;
    });
});
```

Below is an example of invoking a SYCL kernel function with parallel_for using a lambda function and passing a SYCL item parameter. In this case both the global id and global range are queryable. This variant of parallel_for is designed for when it is necessary to query the global range within which the global id will vary. No information is queryable on the local (work-group) size chosen by the implementation.

```
myQueue.submit([&](handler & cgh) {
    accessor acc { myBuffer, cgh, write_only };

    cgh.parallel_for<class myKernel>(range<1>(numWorkItems),
                                     [=] (item<1> item) {
        // kernel argument type is item
        size_t index = item.get_linear_id();
        acc[index] = index;
    });
});
```

Below is an example of invoking a SYCL kernel function with parallel_for using a lambda function and passing auto parameter, treated as item. In this case both the global id and global range are queryable. The same effect can be achieved using class with templatized operator(). This variant of parallel_for is designed for when it is necessary to query the global range within which the global id will vary. No information is queryable on the local (work-group) size chosen by the implementation.

```
myQueue.submit([&](handler & cgh) {
    auto acc = myBuffer.get_access<access::mode::write>(cgh);

    cgh.parallel_for<class myKernel>(range<1>(numWorkItems),
```

```
                                      [=] (auto item) {
            // kernel agrument type is auto treated as an item
            size_t index = item.get_linear_id();
            acc[index] = index;
        });
    });
```

Below is an example of invoking a SYCL kernel function with `parallel_for` using a lambda function and passing integral type (e.g. `int`, `size_t`) parameter. This example is only valid when calling `parallel_for` with `range<1>`. In this case only the global id is available. This variant of `parallel_for` is designed for when it is not necessary to query the global range of the index space being executed across, or the local (workgroup) size chosen by the implementation.

```
myQueue.submit([&](handler & cgh) {
    auto acc = myBuffer.get_access<access::mode::write>(cgh);

    cgh.parallel_for<class myKernel>(range<1>(numWorkItems),
                                     [=] (size_t index) {
        // kernel argument type is size_t
        acc[index] = index;
    });
});
```

The `parallel_for` overload without offset can be called with either number or `braced-init-list` with 1-3 elements. In that case the following calls are equivalent:

- `parallel_for<class MyKernel>(N, some_kernel)` has same effect as `parallel_for<class MyKernel>(range<1>(N), some_kernel)`
- `parallel_for<class MyKernel>({N}, some_kernel)` has same effect as `parallel_for<class MyKernel>(range<1>(N), some_kernel)`
- `parallel_for<class MyKernel>({N1, N2}, some_kernel)` has same effect as `parallel_for<class MyKernel>(range<2>(N1, N2), some_kernel)`
- `parallel_for<class MyKernel>({N1, N2, N3}, some_kernel)` has same effect as `parallel_for<class MyKernel>(range<3>(N1, N2, N3), some_kernel)`

Below is an example of invoking `parallel_for` with nubmer instead of explicit `range` object.

```
myQueue.submit([&](handler & cgh) {
    auto acc = myBuffer.get_access<access::mode::write>(cgh);

    // parallel_for may be called with number (with numWorkItems)
    cgh.parallel_for<class myKernel>(numWorkItems,
                                     [=] (auto item) {
        size_t index = item.get_linear_id();
        acc[index] = index;
    });
});
```

For SYCL kernel functions invoked via the above described overload of the `parallel_for` member function, it is disallowed to use local accessors or to use a work-group barrier.

The following two examples show how a kernel function object can be launched over a 3D grid, with 3 elements in each dimension. In the first case work-item ids range from 0 to 2 inclusive, and in the second case work-item ids run from 1 to 3.

```
myQueue.submit([&](handler & cgh) {
 cgh.parallel_for<class example_kernel1>(
    range<3>(3,3,3), // global range
      [=] (item<3> it) {
        //[kernel code]
      });
});
myQueue.submit([&](handler & cgh) {
 cgh.parallel_for<class example_kernel2>(
    range<3>(3,3,3), // global range
    id<3>(1,1,1), // offset
      [=] (item<3> it) {
        //[kernel code]
      });
});
```

The last case of a parallel_for invocation enables low-level functionality of work-items and work-groups. This becomes valuable when an execution requires groups of work-items that communicate and synchronize. These are exposed in SYCL through `parallel_for (nd_range,...)` and the `nd_item` class. In this case, the developer needs to define the `nd_range` that the kernel will execute on in order to have fine grained control of the enqueing of the kernel. This variation of parallel_for expects an `nd_range`, specifying both local and global ranges, defining the global number of work-items and the number in each cooperating work-group. The resulting function object is passed an `nd_item` instance making all the information available, as well as work-group barrier to synchronize between the work-items in the work-group.

The following example shows how sixty-four work-items may be launched in a three-dimensional grid with four in each dimension, and divided into eight work-groups. Each group of work-items synchronizes with a work-group barrier.

```
myQueue.submit([&](handler & cgh) {
  cgh.parallel_for<class example_kernel>(
      nd_range<3>(range<3>(4, 4, 4), range<3>(2, 2, 2)), [=](nd_item<3> item) {
        //[kernel code]
        // Internal synchronization
        item.barrier(access::fence_space::global_space);
        //[kernel code]
      });
});
```

Optionally, in any of these variations of parallel_for invocations, the developer may also pass an offset. An offset is an instance of the `id` class added to the identifier for each point in the range.

In all of these cases the underlying nd-range will be created and the kernel defined as a function object will be created and enqueued as part of the command group scope.

A `kernel_handler` can optionally be passed as a parameter to the SYCL kernel function that is invoked by both variants of parallel_for.

```
myQueue.submit([&](handler & cgh) {
 cgh.parallel_for<class example_kernel1>(
    range<3>(3,3,3), // global range
      [=] (item<3> it, kernel_handler kh) {
        //[kernel code]
      });
});
myQueue.submit([&](handler & cgh) {
 cgh.parallel_for<class example_kernel2>(
    range<3>(3,3,3), // global range
    id<3>(1,1,1), // offset
      [=] (item<3> it, kernel_handler kh) {
        //[kernel code]
      });
});
```

#### 4.10.7.3 Parallel for hierarchical invoke

The hierarchical parallel kernel execution interface provides the same functionality as is available from the nd-range interface, but exposed differently. To execute the same sixty-four work-items in sixteen work-groups that we saw in the previous example, we execute an outer parallel_for_work_group call to create the groups. The member function handler::parallel_for_work_group is parameterized by the number of work-groups, such that the size of each group is chosen by the runtime, or by the number of work-groups and number of work-items for users who need more control.

The body of the outer parallel_for_work_group call consists of a lambda function or function object. The body of this function object contains code that is executed only once for the entire work-group. If the code has no side-effects and the compiler heuristic suggests that it is more efficient to do so, this code will be executed for each work-item.

Within this region any variable declared will have the semantics of local memory, shared between all work-items in the work-group. If the device compiler can prove that an array of such variables is accessed only by a single work-item throughout the lifetime of the work-group, for example if access is derived from the id of the work-item with no transformation, then it can allocate the data in private memory or registers instead.

To guarantee use of private per-work-item memory, the private_memory class can be used to wrap the data. This class very simply constructs private data for a given group across the entire group. The id of the current work-item is passed to any access to grab the correct data.

The private_memory class has the following interface:

```
namespace sycl {
template <typename T, int Dimensions = 1>
class private_memory {
 public:
  // Construct based directly off the number of work-items
  private_memory(const group<Dimensions> &);

  // Access the instance for the current work-item
  T &operator()(const h_item<Dimensions> &id);
};
```

```
}
```

| Constructor | Description |
|---|---|
| `private_memory(const group<Dimensions> &)` | Place an object of type `T` in the underlying private memory of each work-items. The type `T` must be default constructible. The underlying constructor will be called for each work-item. |
| | End of table |

Table 4.94: Constructor of the `private_memory` class.

| Member functions | Description |
|---|---|
| `T &operator()(const h_item<Dimensions> &id)` | Retrieve a reference to the object for the work-items. |
| | End of table |

Table 4.95: Member functions of the `private_memory` class.

Private memory is allocated per underlying work-item, not per iteration of the `parallel_for_work_item` loop. The number of instances of a private memory object is only under direct control if a work-group size is passed to the `parallel_for_work_group` call. If the underlying work-group size is chosen by the runtime, the number of private memory instances is opaque to the program. Explicit private memory declarations should therefore be used with care and with a full understanding of which instances of a `parallel_for_work_item` loop will share the same underlying variable.

Also within the lambda body can be a sequence of calls to `parallel_for_work_item`. At the edges of these inner parallel executions the work-group synchronizes. As a result the pair of `parallel_for_work_item` calls in the code below is equivalent to the parallel execution with a work-group barrier in the earlier example.

```
myQueue.submit([&](handler & cgh) {
  // Issue 8 work-groups of 8 work-items each
  cgh.parallel_for_work_group<class example_kernel>(
      range<3>(2, 2, 2), range<3>(2, 2, 2), [=](group<3> myGroup) {

    //[workgroup code]
    int myLocal;  // this variable is shared between workitems
    // this variable will be instantiated for each work-item separately
    private_memory<int> myPrivate(myGroup);

    // Issue parallel work-items.  The number issued per work-group is determined
    // by the work-group size range of parallel_for_work_group.  In this case,
    // 8 work-items will execute the parallel_for_work_item body for each of the
    // 8 work-groups, resulting in 64 executions globally/total.
    myGroup.parallel_for_work_item([&](h_item<3> myItem) {
      //[work-item code]
      myPrivate(myItem) = 0;
    });

    // Implicit work-group barrier
```

```

    // Carry private value across loops
    myGroup.parallel_for_work_item([&](h_item<3> myItem) {
      //[work-item code]
      output[myItem.get_global_id()] = myPrivate(myItem);
    });
    //[workgroup code]
  });
});
```

It is valid to use more flexible dimensions of the work-item loops. In the following example we issue 8 work-groups but let the runtime choose their size, by not passing a work-group size to the `parallel_for_work_group` call. The `parallel_for_work_item` loops may also vary in size, with their execution ranges unrelated to the dimensions of the work-group, and the compiler generating an appropriate iteration space to fill the gap. In this case, the `h_item` provides access to local ids and ranges that reflect both kernel and `parallel_for_work_item` invocation ranges.

```
myQueue.submit([&](handler & cgh) {
  // Issue 8 work-groups.  The work-group size is chosen by the runtime because unspecified
  cgh.parallel_for_work_group<class example_kernel>(
      range<3>(2, 2, 2), [=](group<3> myGroup) {

    // Launch a set of work-items for each work-group.  The number of work-items is chosen
    // by the runtime because the work-group size was not specified to parallel_for_work_group
    // and a logical range is not specified to parallel_for_work_item.
    myGroup.parallel_for_work_item([=](h_item<3> myItem) {
      //[work-item code]
    });

    // Implicit work-group barrier

    // Launch 512 logical work-items that will be executed by the underlying work-group size
    // chosen by the runtime.  myItem allows the logical and physical work-item IDs to be
    // queried.  512 logical work-items will execute for each work-group, and the parallel_for
    // body will therefore be executed 8*512 = 4096 times globally/total.
    myGroup.parallel_for_work_item(range<3>(8, 8, 8), [=](h_item<3> myItem) {
      //[work-item code]
    });
    //[workgroup code]
  });
});
```

This interface offers a more intuitive way for tiling parallel programming paradigms. In summary, the hierarchical model allows a developer to distinguish the execution at work-group level and at work-item level using the `parallel_for_work_group` and the nested `parallel_for_work_item` functions. It also provides this visibility to the compiler without the need for difficult loop fission such that host execution may be more efficient.

A `kernel_handler` can optionally be passed as a parameter to the SYCL kernel function that is invoked by any variant of `parallel_for_work_group`.

```
myQueue.submit([&](handler & cgh) {
  // Issue 8 work-groups of 8 work-items each
  cgh.parallel_for_work_group<class example_kernel>(
```

```
      range<3>(2, 2, 2), range<3>(2, 2, 2), [=](group<3> myGroup,
      kernel_handler kh) {

    //[workgroup code]
    int myLocal;  // this variable is shared between workitems
    // this variable will be instantiated for each work-item separately
    private_memory<int> myPrivate(myGroup);

    // Issue parallel work-items.  The number issued per work-group is determined
    // by the work-group size range of parallel_for_work_group.  In this case,
    // 8 work-items will execute the parallel_for_work_item body for each of the
    // 8 work-groups, resulting in 64 executions globally/total.
    myGroup.parallel_for_work_item([&](h_item<3> myItem) {
      //[work-item code]
      myPrivate(myItem) = 0;
    });

    // Implicit work-group barrier

    // Carry private value across loops
    myGroup.parallel_for_work_item([&](h_item<3> myItem) {
      //[work-item code]
      output[myItem.get_global_id()] = myPrivate(myItem);
    });
    //[workgroup code]
  });
});
```

### 4.10.8 SYCL functions for explicit memory operations

In addition to kernels, command group objects can also be used to perform manual operations on host and device memory by using the *copy* API of the command group handler. Manual copy operations can be seen as specialized kernels executing on the device, except that typically this operations will be implemented using the OpenCL host API (e.g, enqueue copy operations).

The SYCL memory objects involved in a copy operation are specified using accessors. Explicit copy operations have a source and a destination. When an accessor is the *source* of the operation, the destination can be a host pointer or another accessor. The *source* accessor can have either `read` or `read_write` access mode.

When an accessor is the *destination* of the explicit copy operation, the source can be a host pointer or another accessor. The *destination* accessor can have either `write`, `read_write`, `discard_write`, `discard_read_write` access modes.

When accessors are both the origin and the destination, the operation is executed on objects controlled by the SYCL runtime. The SYCL runtime is allowed to not perfom an explicit in-copy operation if a different path to update the data is available according to the SYCL Application memory model.

The most recent copy of the memory object may reside on any context controlled by the SYCL runtime, or on the host in a pointer controlled by the SYCL runtime. The SYCL runtime will ensure that data is copied to the destination once the command group has completed execution.

Whenever a host pointer is used as either the host or the destination of these explicit memory operations, it is the responsibility of the user for that pointer to have at least as much memory allocated as the accessor is giving access to, e.g: if an accessor accesses a range of 10 elements of `int` type, the host pointer must at least have `10 * sizeof(int)` bytes of memory allocated.

A special case is the `update_host` method. This method only requires an accessor, and instructs the runtime to update the internal copy of the data in the host, if any. This is particularly useful when users use manual synchronization with host pointers, e.g. via mutex objects on the `buffer` constructors.

Table 4.96 describes the interface for the explicit copy operations.

| Member function | Description |
|---|---|
| `template <typename T_src, int dim_src, access::mode mode_src, access::target tgt_src, typename T_dest, access::placeholder isPlaceholder>`<br>`void copy(accessor<T_src, dim_src, mode_src, tgt_src, isPlaceholder> src, std::shared_ptr<T_dest> dest)` | Copies the contents of the memory object accessed by `src` into the memory pointed to by `dest`. `dest` must have at least as many bytes as the range accessed by `src`. |
| `template <typename T_src, typename T_dest, int dim_dest, access::mode mode_dest, access::target tgt_dest, access::placeholder isPlaceholder>`<br>`void copy(std::shared_ptr<T_src> src, accessor< T_dest, dim_dest, mode_dest, tgt_dest, isPlaceholder > dest)` | Copies the contents of the memory pointed to by `src` into the memory object accessed by `dest`. `src` must have at least as many bytes as the range accessed by `dest`. |
| `template <typename T_src, int dim_src, access::mode mode_src, access::target tgt_src, typename T_dest, access::placeholder isPlaceholder>`<br>`void copy(accessor<T_src, dim_src, mode_src, tgt_src, isPlaceholder> src, T_dest * dest)` | Copies the contents of the memory object accessed by `src` into the memory pointed to by `dest`. `dest` must have at least as many bytes as the range accessed by `src`. |
| `template <typename T_src, typename T_dest, int dim_dest, access::mode mode_dest, access::target tgt_dest, access::placeholder isPlaceholder>`<br>`void copy(const T_src * src, accessor<T_dest, dim_dest, mode_dest, tgt_dest, isPlaceholder> dest)` | Copies the contents of the memory pointed to by `src` into the memory object accessed by `dest`. `src` must have at least as many bytes as the range accessed by `dest`. |
| `template <typename T_src, int dim_src, access::mode mode_src, access::target tgt_src, access::placeholder isPlaceholder_src, typename T_dest, int dim_dest, access::mode mode_dest, access::target tgt_dest, access::placeholder isPlaceholder_dest>`<br>`void copy(accessor<T_src, dim_src, mode_src, tgt_src, isPlaceholder_src> src, accessor<T_dest, dim_dest, mode_dest, tgt_dest, isPlaceholder_dest> dest)` | Copies the contents of the memory object accessed by `src` into the memory object accessed by `dest`. `src` must have at least as many bytes as the range accessed by `dest`. |
| `template <typename T, int dim, access::mode mode, access::target tgt, access::placeholder isPlaceholder>`<br>`void update_host(accessor<T, dim, mode, tgt, isPlaceholder> acc)` | The contents of the memory object accessed via `acc` on the host are guaranteed to be up-to-date after this command group object execution is complete. |

Continued on next page

Table 4.96: Member functions of the `handler` class.

| Member function | Description |
|---|---|
| `template <typename T, int dim, access::mode mode, access::target tgt, access::placeholder isPlaceholder> void fill(accessor<T, dim, mode, tgt, isPlaceholder> dest, const T& src)` | Replicates the value of `src` into the memory object accessed by `dest`. T must be a scalar value or a SYCL vector type. |
| | End of table |

Table 4.96: Member functions of the `handler` class.

The listing below illustrates how to use explicit copy operations in SYCL. The example copies half of the contents of a `std::vector` into the device, leaving the rest of the contents of the buffer on the device unchanged.

```
const size_t nElems = 10u;

// Create a vector and fill it with values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
std::vector<int> v { nElems };
std::iota(std::begin(v), std::end(v), 0);

// Create a buffer with no associated user storage
sycl::buffer<int, 1> b { range<1>(nElems) };

// Create a queue
queue myQueue;

myQueue.submit([&](handler &cgh) {
  // Retrieve a ranged write accessor to a global buffer with access to the
  // first half of the buffer
  accessor acc { b, cgh, range<1>(nElems / 2), id<1>(0), write_only };
  // Copy the first five elements of the vector into the buffer associated with
  // the accessor
  cgh.copy(v.data(), acc);
});
```

## 4.11 Host tasks

### 4.11.1 Overview

A host task is a native C++ callable which is scheduled by the SYCL runtime. A host task is submitted to a queue via a command group by a host task command.

When a host task command is submitted to a queue it is scheduled based on its data dependencies with other commands including kernel invocation commands and asynchronous copies, resolving any requisites created by accessors attached to the command group as defined in Section 3.7.1.

Since a host task is invoked directly by the SYCL runtime rather than being compiled as a SYCL kernel function, it does not have the same restrictions as a SYCL kernel function, and can therefore contain any arbitrary C++ code. However, capturing or using any SYCL class with reference semantics (see Section 4.5.3) is undefined behaviour.

A host task can be enqueued on any queue including a host queue and the callable will be invoked directly by the SYCL runtime, regardless of which device the queue is associated with.

A host task is enqueued on a queue via the `host_task` member function of the `handler` class.

A host task can optionally be used to interoperate with the native backend objects associated with the queue executing the host task, the context that the queue is associated with, the device that the queue is associated with and the accessors that have been captured in the callable, via an optional `interop_handle` parameter.

This allows host task to be used for two purposes: either as a task which can perform arbitrary C++ code within the scheduling of the SYCL runtime or as a task which can perform interoperability at a point within the scheduling of the SYCL runtime.

For the former use case host accessors should be used to request that a buffer or image be made available on the host so that it can be accessed directly via the accessor.

For the later use case device accessors should be used to request that a buffer or image be made available on the device associated with the queue used to submit the host task so that is can be accessed via interoperability member functions provided by the `interop_handle` class.

Local accessors cannot be used within a host task.

```
namespace sycl {

class interop_handle {
 private:

  interop_handle(__unspecified__);

 public:

  interop_handle() = delete;

  template <backend Backend, typename dataT, int dims, access::mode accessMode,
            access::target accessTarget, access::placeholder isPlaceholder>
  backend_traits<Backend>::native_type<buffer>
  get_native_mem(const accessor<dataT, dims, accessMode, accessTarget,
                                isPlaceholder> &bufferAccessor) const;

  template <backend Backend, typename dataT, int dims, access::mode accessMode,
            access::target accessTarget, access::placeholder isPlaceholder>
  backend_traits<Backend>::native_type<image>
  get_native_mem(const accessor<dataT, dims, accessMode, accessTarget,
                                isPlaceholder> &imageAccessor) const;

  template <backend Backend>
  backend_traits<Backend>::native_type<queue> get_native_queue() const noexcept;

  template <backend Backend>
  backend_traits<Backend>::native_type<device> get_native_device() const noexcept;

  template <backend Backend>
  backend_traits<Backend>::native_type<context> get_native_context() const noexcept;

```

```
};

class handler {
  ...

 public:

  template <typename T>
  void host_task(T &&hostTaskCallable);

  ...
};

}  // namespace sycl
```

### 4.11.2 Class interop_handle

The `interop_handle` class is an abstraction over the queue which is being used to invoke the host task and its associated device and context. It also represents the state of the SYCL runtime dependency model at the point the host task is invoked.

The `interop_handle` class provides access to the native backend object associated with the queue, device, context and any buffers or images that are captured in the callable being invoked in order to allow a host task to be used for interoperability purposes.

An `interop_handle` cannot be constructed by user-code, only by the SYCL runtime.

```
class interop_handle;
```

#### 4.11.2.1 Constructors

```
private:

 interop_handle(__unspecified__); // (1)

public:

 interop_handle() = delete; // (2)
```

1. Private implementation defined constructor with unspecified arguments so that the SYCL runtime can construct a `interop_handle`.

2. Explicitly deleted default constructor.

#### 4.11.2.2 Template member functions get_native_*

```
template <backend Backend, typename dataT, int dims, access::mode accMode,
          access::target accTarget, access::placeholder isPlaceholder>
backend_traits<Backend>::native_type<buffer>
get_native_mem(const accessor<dataT, dims, accMode, accTarget, // (1)
                              isPlaceholder> &bufferAccessor) const;
```

```

template <backend Backend, typename dataT, int dims, access::mode accMode,
          access::target accTarget, access::placeholder isPlaceholder>
backend_traits<Backend>::native_type<image>
get_native_mem(const accessor<dataT, dims, accMode, accTarget, // (2)
                              isPlaceholder> &imageAccessor) const;

template <backend Backend>
backend_traits<Backend>::native_type<queue> get_native_queue() const noexcept; // (3)

template <backend Backend>
backend_traits<Backend>::native_type<device> get_native_device() const noexcept; // (4)

template <backend Backend>
backend_traits<Backend>::native_type<context> get_native_context() const noexcept; // (5)
```

1. *Constraints:* Available only if the optional interoperability function `get_native` taking a `buffer` is available and if `accTarget` is `access::target::global_buffer` or `access::target::constant_buffer`.

   *Returns:* The SYCL application interoperability native backend object associated with the accessor `bufferAccessor`. The native backend object returned must be in a state where it represents the memory in its current state within the SYCL runtime dependency model be capable of being used in a way appropriate for the associated SYCL backend. It is undefined behaviour to use the native backend object outside of the scope of the host task.

   *Throws:* `errc::invalid_object_error` if the accessor `bufferAccessor` was not registered with the command group which contained the host task.

2. *Constraints:* Available only if the optional interoperability function `get_native` taking an `unsampled_image` or `sampled_image` is available and if `accTarget` is `access::target::unsampled_image` or `access::target::sampled_image`.

   *Returns:* The SYCL application interoperability native backend object associated with the accessor `imageAccessor`. The native backend object returned must be in a state where it represents the memory in its current state within the SYCL runtime dependency model and is capable of being used in a way appropriate for the associated SYCL backend. It is undefined behaviour to use the native backend object outside of the scope of the host task.

   *Throws:* `errc::invalid_object_error` if the accessor `imageAccessor` was not registered with the command group which contained the host task.

3. *Constraints:* Available only if the optional interoperability function `get_native` taking a `queue` is available.

   *Returns:* The SYCL application interoperability native backend object associated with the queue that the host task was submitted to. If the command group was submitted with a secondary queue and the fall-back was triggered, the queue that is associated with the `interop_handle` must be the fall-back queue. The native backend object returned must be in a state where it is capable of being used in a way appropriate for the associated SYCL backend. It is undefined behaviour to use the native backend object outside of the scope of the host task.

4. *Constraints:* Available only if the optional interoperability function `get_native` taking a `device` is available.

*Returns:* The SYCL application interoperability native backend object associated with the device that is associated with the queue that the host task was submitted to. The native backend object returned must be in a state where it is capable of being used in a way appropriate for the associated SYCL backend. It is undefined behaviour to use the native backend object outside of the scope of the host task.

5. *Constraints:* Available only if the optional interoperability function `get_native` taking a `context` is available.

   *Returns:* The SYCL application interoperability native backend object associated with the context that is associated with the queue that the host task was submitted to. The native backend object returned must be in a state where it is capable of being used in a way appropriate for the associated SYCL backend. It is undefined behaviour to use the native backend object outside of the scope of the host task.

### 4.11.3 Handler class

```
class handler;
```

#### 4.11.3.1 Member function `host_task`

```
template <typename T>
void host_task(T &&hostTaskCallable); // (1)
```

1. *Effects:* Enqueues an implementation defined command to the SYCL runtime to invoke `hostTaskCallable` exactly once. The scheduling of the invocation of `hostTaskCallable` in relation to other commands enqueued to the SYCL runtime must be in accordance with the dependency model described in Section 3.7.1. Initialises a `interop_handle` object and passes it to `hostTaskCallable` when it is invoked if `std::is_invocable_v<T, interop_handle>` evaluates to `true`, otherwise invokes invokes `hostTaskCallable` as a nullary function.

### 4.11.4 Functions for using a module

```
void use_module(const module<module_state::executable> &execModule); // (1)

template <typename T>
void use_module(const module<module_state::executable> &execModule // (2)
                T deviceImageSelector);
```

1. *Effects:* The command group associated with the `handler` will use a device images of the `module` `execModule` in any kernel invocation commands for all SYCL kernel functions represented by the `module`. If the `module` contains multiple device images then the device image chosen is implementation defined.

   *Throws:* `errc::invalid_object_error` if the context associated with the command group handler via its associated queue is different from the context associated with the module specified by `execModule`.

2. *Effects:* The command group associated with the `handler` will use a device images of the `module` `execModule` in any kernel invocation commands for all SYCL kernel functions represented by the `module`. All device images in the `module` will be passed to the device image selection function and the device image with the highest score will be chosen.

*Throws:* errc::invalid_object_error if the context associated with the command group handler via its associated queue is different from the context associated with the module specified by execModule.

*Throws:* errc::device_image_selection_error if no device image could be selected.

### 4.11.5 Functions for using specialization constants

```
template<auto& S>
bool has_specialization_constant() const noexcept; // (1)
```

1. *Returns:* true if any of the SYCL kernel functions represented by the module associated with the command group handler contains the specialization constant represented by the specialization_id at the address S, otherwise returns false.

```
template<auto& S>
typename std::remove_reference_t<decltype(S)>::type get_specialization_constant(); // (1)
```

1. *Returns:* From the module associated with the command group handler, the value of the specialization constant associated with the specialization_id at the address S if the specialization constant has been set, otherwise returns the default value.

   *Throws:* errc::invalid_object_error if this->has_specialization_constant<S>() evaluates to false.

## 4.12 Kernel class

The kernel class is an abstraction of a kernel object in SYCL. In the most common case the kernel object will contain the compiled version of a kernel invoked inside a command group using one of the parallel interface functions as described in 4.10.7. The SYCL runtime will create a kernel object, when it needs to enqueue the kernel on a command queue.

In the case where a developer would like to pre-compile a kernel or compile and link it with an existing program, then the kernel object will be created and contain that kernel using the module class, as defined in 4.13.5. In both of the above cases, the developer cannot instantiate a kernel object but can instantiate a named function object type

that they could use, or create a function object from a kernel method using C++ features. The kernel class object needs a parallel_for(...) invocation or an explicitly built SYCL kernel instance, for this compilation of the kernel to be triggered.

The SYCL kernel class provides the common reference semantics (see Section 4.5.3).

The member functions of the SYCL kernel class are listed in Table 4.97. The additional common special member functions and common member functions are listed in Tables 4.1 and 4.2, respectively.

```
namespace sycl {
class kernel {
 private:
  kernel();

```

```
 public:

  /* -- common interface members -- */

  backend get_backend() const;

  bool is_host() const;

  context get_context() const;

  module<module_state::executable> get_module() const;

  template <info::kernel param>
  typename info::param_traits<info::kernel, param>::return_type
  get_info() const;

  template <info::kernel_device_specific param>
  typename info::param_traits<info::kernel_device_specific, param>::return_type
  get_info(const device &dev) const;

  template <typename BackendEnum, BackendEnum param>
  typename info::param_traits<BackendEnum, param>::return_type
  get_backend_info() const;

  template <info::kernel_work_group param>
  typename info::param_traits<info::kernel_work_group, param>::return_type
  get_work_group_info(const device &dev) const;
};
}  // namespace sycl
```

| Member functions | Description |
|---|---|
| `backend get_backend()const` | Returns the a backend identifying the SYCL backend associated with this platform. |
| `bool is_host()const` | Returns true if this SYCL kernel is a host kernel. |
| `context get_context()const` | Return the context that this kernel is defined for. The value returned must be equal to that returned by `get_info<info::kernel::context>()`. |
| `module<module_state::executable> get_program()const` | Returns the module that this kernel is part of. The value returned must be equal to that returned by `get_info<info::kernel::module>()`. |
| `template <info::kernel param> typename info::param_traits< info::kernel, param>::return_type get_info()const` | Query information from the kernel object using the `info::kernel_info` descriptor. |

Continued on next page

Table 4.97: Member functions of the kernel class.

| Member functions | Description |
|---|---|
| `template <info::kernel_device_specific param> typename info::param_traits< info::kernel_device_specific, param>:: return_type get_info(const device &dev)const` | Query information from a kernel using the `info::kernel_device_specific` descriptor for a specific device. |
| `template <typename BackendEnum, BackendEnum param> typename info::param_traits<BackendEnum, param>:: return_type get_backend_info()const` | Queries this SYCL `platform` for SYCL backend-specific information requested by the template parameter `param`. `BackendEnum` can be any enum class type specified by the SYCL backend specification of a supported SYCL backend named according to the convention `info::<backend_name>::kernel` and `param` must be a valid enumeration of that enum class. Specializations of `info::param_traits` must be defined for `BackendEnum` in accordance with the SYCL backend specification. Must throw an `exception` with the `errc::invalid_object_error` error code if the SYCL backend that corresponds with `BackendEnum` is different from the SYCL backend that is associated with this `kernel`. |
| `template <info::kernel_work_group param> typename info::param_traits< info::kernel_work_group, param>::return_type get_work_group_info(const device &dev)const` | Query information from the work-group from a kernel using the `info::kernel_work_group` descriptor for a specific device. This query has been deprecated in SYCL 2020 and will likely be removed in a future version of SYCL. |
| | End of table |

Table 4.97: Member functions of the `kernel` class.

## 4.12.1 Kernel information descriptors

A kernel can be queried for information using the `get_info` member function of the `kernel` class, specifying one of the info parameters enumerated in `info::kernel`. Every kernel (including a host kernel) must produce a valid value for each info parameter. The possible values for each info parameter and any restriction are defined in the specification of the SYCL backend associated with the kernel. All info parameters in `info::kernel` are specified in Table 4.98 and the synopsis for `info::kernel` is described in appendix A.5.

| Kernel Descriptors | Return type | Description |
|---|---|---|
| `info::kernel::function_name` | `std::string` | Return the kernel function name. |
| `info::kernel::num_args` | `uint32_t` | Return the number of arguments of the extracted kernel. |
| `info::kernel::context` | `context` | Return the SYCL `context` associated with this SYCL `kernel`. |
| | | Continued on next page |

Table 4.98: Kernel class information descriptors.

| Kernel Descriptors | Return type | Description |
|---|---|---|
| info::kernel::module | module< module_state:: executable> | Return the SYCL module associated with this SYCL kernel. |
| info::kernel::attributes | std::string | Return any attributes specified on a kernel function (as defined in Section 5.7). |
| | | End of table |

Table 4.98: Kernel class information descriptors.

A kernel can be queried for device-specific information using the get_info member function of the kernel class, specifying one of the info parameters enumerated in info::kernel_device_specific. Every kernel (including a host kernel) must produce a valid value for each info parameter. The possible values for each info parameter and any restriction are defined in the specification of the SYCL backend associated with the kernel. All info parameters in info::kernel_device_specific are specified in Table 4.99. The synopsis for info::kernel is described in appendix A.5.

| Device-specific Kernel Information Descriptors | Return type | Description |
|---|---|---|
| info::kernel_device_specific:: global_work_size | range<3> | Returns the maximum global work size. Only valid if device is of device_type custom or the kernel is a built-in kernel. |
| info::kernel_device_specific:: work_group_size | size_t | Returns the maximum work-group size that can be used to execute a kernel on a specific device. |
| info::kernel_device_specific:: compile_work_group_size | range<3> | Returns the work-group size specified by the device compiler if applicable, otherwise returns $(0, 0, 0)$ |
| info::kernel_device_specific:: preferred_work_group_size_multiple | size_t | Returns a value, of which work-group size is preferred to be a multiple, for executing a kernel on a particular device. This is a performance hint. The value must be less than or equal to that returned by info:: kernel_device_specific::work_group_size. |
| info::kernel_device_specific:: private_mem_size | size_t | Returns the minimum amount of private memory, in bytes, used by each work-item in the kernel. This value may include any private memory needed by an implementation to execute the kernel, including that used by the language built-ins and variables declared inside the kernel in the private address space. |
| info::kernel_device_specific:: max_num_sub_groups | uint32_t | Returns the maximum number of sub-groups for this kernel. |
| info::kernel_device_specific:: compile_num_sub_groups | uint32_t | Returns the number of sub-groups specified by the kernel, or 0 (if not specified). |
| info::kernel_device_specific:: max_sub_group_size | uint32_t | Returns the maximum sub-group size for this kernel. |
| info::kernel_device_specific:: compile_sub_group_size | uint32_t | Returns the required sub-group size specified by the kernel, or 0 (if not specified). |
| | | End of table |

Table 4.99: Device-specific kernel information descriptors.

Alternatively, a kernel can be queried for work-group information using the `get_work_group_info` member functions of the `kernel` class, specifying one of the info parameters enumerated in `info::kernel_work_group`. This query has been deprecated in SYCL 2020, and will likely be removed in a future version of SYCL.

| Kernel Work-group Information Descriptors | Return type | Description |
|---|---|---|
| `info::kernel_work_group::global_work_size` | `range<3>` | Returns the maximum global work size. Only valid if device is of device_type custom or the kernel is a built-in kernel. |
| `info::kernel_work_group::work_group_size` | `size_t` | Returns the maximum work-group size that can be used to execute a kernel on a specific device. |
| `info::kernel_work_group::compile_work_group_size` | `range<3>` | Returns the work-group size specified by the device compiler if applicable, otherwise returns $(0, 0, 0)$ |
| `info::kernel_work_group::preferred_work_group_size_multiple` | `size_t` | Returns a value, of which work-group size is preferred to be a multiple, for executing a kernel on a particular device. This is a performance hint. The value must be less than or equal to that returned by `info::kernel_work_group::work_group_size`. |
| `info::kernel_work_group::private_mem_size` | `size_t` | Returns the minimum amount of private memory, in bytes, used by each work-item in the kernel. This value may include any private memory needed by an implementation to execute the kernel, including that used by the language built-ins and variables declared inside the kernel in the private address space. |
| | | End of table |

Table 4.100: Kernel work-group information descriptors.

## 4.13 Modules

### 4.13.1 Overview

A module is a high-level abstraction which represents a set of SYCL kernel functions which are associated with a context and can be executed on a number of devices, where each device is associated with that same context.

The SYCL kernel functions represented by a module can exist within one or more device images of implementation defined file formats. Each device image within a module must contain the necessary symbols and meta-data for each SYCL kernel function that the containing module represents.

*[Note: For example, a module associated with a context of an OpenCL SYCL backend, that represents the SYCL kernel function* `foo` *could contain two modules; one of SPIR-V and one of a vendor specific ISA, both containing* `foo` *in the relevant file format. – end note]*

Modules are used to express the explicit compilation of SYCL kernel functions in order to then be invoked via a kernel invocation command such as `parallel_for`. Normally this compilation is done implicitly by the kernel invocation command, however it can be useful to perform the compilation manually in order to add custom properties to the compilation or to link SYCL kernel functions with other libraries.

A module can be obtained either by requesting the module associated with the current translation unit or via some SYCL backend-specific operation.

Once a module has been obtained there are a number of free functions for performing compilation, linking and joining.

A module can then be bound to a command group so that the SYCL kernel functions represented by the module are used in any kernel invocation commands. See Section 4.11.4 for more details.

### 4.13.2 Specialization constants

A SYCL kernel function can contain specialization constants which represent a constant variable where the value is not known until compilation of the SYCL kernel function. Any specialization constants of a given SYCL kernel function are exposed via an input module representing that SYCL kernel function, where the value of the specialization constants can be set. However, once a module has been compiled, resulting in an object module or executable module the value of specialization constants can no longer be changed.

Any specialization constants in a SYCL kernel function which are not set before that function is invoked will take a default value. This includes invoking a kernel invocation command such as `parallel_for` or retrieving an object module or executable module directly triggering implicit compilation.

As a module may contain more than one device image, some of these may natively support specialization constants and some may not, however all device images must set the value of specialization constants.

*[Note: It is expected that a specialization constant either via an implementation-defined mechanism available to the file format of that device image such as SPIR-V specialization constants or by passing the value as an additional kernel argument to the SYCL kernel function via the SYCL runtime when it's invoked. –end note]*

A specialization id is an identifier which represents a reference to a specialization constant both in the SYCL application for setting the value prior to the compilation of an input module and in a SYCL kernel function for retrieving the value during invocation.

A module that is associated with a host context may not contain native-specialization constants, though it must still emulate all specialization constants the SYCL kernel functions contains.

#### 4.13.2.1 Synopsis

```
namespace sycl {

template <typename T>
class specialization_id {
 private:

  specialization_id(const specialization_id& rhs) = delete;

  specialization_id(specialization_id&& rhs) = delete;

  specialization_id &operator=(const specialization_id& rhs) = delete;

  specialization_id &operator=(specialization_id&& rhs) = delete;

 public:

```

```
  using value_type = T;

  template<class... Args >
  explicit constexpr specialization_id(Args&&... args);
};

enum class module_state {
  input,
  object,
  executable
};

template<module_state State>
class module {
 private:

  module(__unspecified__);

 public:

  using device_image_type = __unspecified__;

  using device_image_iterator = __unspecified__;

  module() = delete;

  context get_context() const noexcept;

  std::vector<device> get_devices() const noexcept;

  bool has_kernel(std::string kernelName) const noexcept;

  kernel get_kernel(std::string kernelName) const;

  std::vector<std::string> get_kernel_names() const;

  bool is_empty() const noexcept;

  device_image_iterator begin() const;

  device_image_iterator end() const;

  bool contains_specialization_constants() const noexcept;

  bool native_specialization_constant() const noexcept;

  template<auto& S>
  bool has_specialization_constant() const noexcept;

  template<auto& S>
  void set_specialization_constant(
    typename std::remove_reference_t<decltype(S)>::type value);

  template<auto& S>
  typename std::remove_reference_t<decltype(S)>::type get_specialization_constant() const;
```

```
};

module<module_state::executable>
build(const module<module_state::input> &inputModule,
      const property_list &propList = {});

module<module_state::object>
compile(const module<module_state::input> &inputModule,
        const property_list &propList = {});

module<module_state::executable>
link(const module<module_state::object> &objModule,
     const property_list &propList = {});

module<module_state::executable>
link(const std::vector<module<module_state::object>> &objModules,
     const property_list &propList = {});

template<module_state T>
module<T> join(const std::vector<module<T>> &modules);

namespace this_module {

template <typename T>
std::string kernel_name_v;

bool has_any_module(context ctxt);

template<module_state S>
bool has_module_in(context ctxt);

template<module_state S>
module<T> get(context ctxt);

} // this_module

} // namespace sycl
```

### 4.13.3 Enum class `module_state`

A module can be in one of three different module states; input, object and executable. The module states reflect the current state of the device images and subsequently the SYCL kernel functions. The module states also alters the capabilities of the module.

The three module states are represented by an enum class called `module_state`.

```
enum class module_state {
  input,
  object,
  executable
};
```

The values of each enumeration of `module_state` are implementation defined.

## 4.13.4 Class template specialization_id

A specialization id is represented by the class template `specialization_id` with a single template parameter `T` which specifies the unique name used to identify the associated specialization constant.

The template parameter `T` must be a forward-declarable type and `specialization_id` objects must be declared with automatic or static storage duration within the a namespace or class scope.

```
template <typename T>
class specialization_id;
```

#### 4.13.4.1 Constructors

```
template<class... Args>
explicit constexpr specialization_id(Args&&... args);
```

1. *Constraints:* Available only when `std::is_constructible_v<T, Args...>` evaluates to `true`.

   *Effects:* Constructs a `specialization_id` containing an instance of `T` initialized with `args...`, which represents the default value of the specialization constant.

#### 4.13.4.2 Special member functions

```
specialization_id(const specialization_id& rhs) = delete; // (1)

specialization_id(specialization_id&& rhs) = delete; // (2)

specialization_id &operator=(const specialization_id& rhs) = delete; // (3)

specialization_id &operator=(specialization_id&& rhs) = delete; // (4)
```

1. Deleted copy constructor.
2. Deleted move constructor.
3. Deleted copy assignment operator.
4. Deleted move assignment operator.

## 4.13.5 Class template module

A module is represented by the class template `module` with the single template parameter `State` of type `module_state` which specifies its module state.

There are three different module types, to reflect the three module states:

- An input module is a module of the input module state and represents SYCL kernel functions which are yet to be compiled such as a source or intermediate representation.
- A object module is a module of the object module state and represents SYCL kernel functions which have been compiled but are yet to be linked such as an intermediate object of the ISAs of the associated devices.

- A executable module is a module of the executable module state and represents SYCL kernel functions which have been compiled and linked such as an executable of the ISAs of the associated devices.

A `module` cannot be constructed by user-code, only by the SYCL runtime.

A module is permitted to be empty in which case it contains no device images and represents no SYCL kernel functions.

A module is considered to have reference semantics as specified in Section 4.5.3, therefore any module constructed as a copy of another and the module that was copied from are considered to be equal. Furthermore, two modules of the same module state are considered to be equal if they are associated with the same context and devices and contain the same device images and subsequently the same SYCL kernel function.

A module must contain a copy of the context and devices that are associated with it for the duration of its lifetime. This means that the destructor of the associated context or devices will not be invoked if the module is still alive in accordance with Section 4.5.3.

```
template<module_state State>
class module;
```

#### 4.13.5.1 Constructors

```
private:

 module(___unspecified___); // (1)

public:

 module() = delete; // (2)
```

1. Private implementation defined constructor with unspecified arguments so that the SYCL runtime can construct a `module`.
2. Explicitly deleted default constructor.

#### 4.13.5.2 Member functions

```
context get_context() const noexcept; // (1)
```

1. *Returns:* A `context` object representing the associated context.

```
std::vector<device> get_devices() const noexcept; // (1)
```

1. *Returns:* A `std::vector` of `device` objects representing the associated devices.

```
bool has_kernel(std::string kernelName) const noexcept; // (1)
```

1. *Constraints:* Available only when `State == module_state::executable`.

*Returns:* `true` if the module represents a SYCL kernel function with the value of string kernel name `kernelName`, otherwise returns `false`. Only available when the module is a executable module.

```
kernel get_kernel(std::string kernelName) const; // (1)
```

1. *Constraints:* Available only when `State == module_state::executable`.

   *Returns:* a `kernel` object representing the SYCL kernel function with the string kernel name with the value of `kernelName` if `this->has_kernel(kernelName)` evaluates to true, otherwise throws `exception` with the `errc::invalid_object` error code.

```
std::vector<std::string> get_kernel_names() const; // (1)
```

1. *Returns:* A `std::vector` of `std::string` objects representing each of the SYCL kernel functions represented by the module.

```
bool is_empty() const noexcept; // (1)
```

1. *Returns:* `true` if the module contains no device images, otherwise returns `false`.

```
 device_image_iterator begin() const; // (1)

 device_image_iterator end() const; // (2)
```

1. *Returns:* An iterator of type `device_image_iterator` pointing to the beginning of a sequence of device images of type `std::iterator_traits<device_image_iterator>::value_type`.
2. *Returns:* An iterator of type `device_image_iterator` pointing to the end of a sequence of device images of type `std::iterator_traits<device_image_iterator>::value_type`.

```
bool use_specialization_constant() const noexcept; // (1)
```

1. *Returns:* `true` if any SYCL kernel function represented by the module contains a specialization constant, otherwise returns `false`.

```
bool native_specialization_constant() const noexcept; // (1)
```

1. *Returns:* `true` if all of the specialization constants contained in the module support are native-specialization constants for all device images.

```
template<auto& S>
bool has_specialization_constant() const noexcept; // (1)
```

1. *Returns:* `true` if any of the SYCL kernel functions represented by the module contains the specialization constant represented by the `specialization_id` at the address `S`, otherwise returns `false`.

```
template<auto& S>
void set_specialization_constant(
  typename std::remove_reference_t<decltype(S)>::type value); // (1)
```

1. *Constraints:* Available only when `State == module_state::input`.

   *Effects:* Sets the value of the specialization constant represented by the `specialization_id` at the address `S` in all SYCL kernel functions which contain that specialization constant and for all device images. If the specialization constant was already set, then the previous value is overwritten. If two or more SYCL kernel functions contain the same specialization constant they are assume to be the same and will have the same value.

   *Throws:* `errc::invalid_object_error` if `this->has_specialization_constant<S>()` evaluates to `false`.

```
template<auto& S>
typename std::remove_reference_t<decltype(S)>::type get_specialization_constant() const;  // (1)
```

1. *Returns:* the value of the specialization constant associated with the `specialization_id` at the address `S` if the specialization constant has been set, otherwise returns the default value.

   *Throws:* `errc::invalid_object_error` if `this->has_specialization_constant<S>()` evaluates to `false`.

### 4.13.6 Free functions

Modules can be compiled, linked, built or joined together using free functions which operate on modules of different module states.

When a module is being created as a results of one of these operations it is permitted to perform ad-hoc implementation-defined operations such as just-in-time compilation or translations to alter the file format of the device image in order to create a module of a resulting module state, however this is not required.

*[Note: For example, if a SYCL kernel function was compiled by a device compiler to generate the file format SPIR-V a module associated with a context of an OpenCL SYCL backend could be created as an input module using the SPIR-V file format directly or it could be created as a executable module implicitly triggering online compilation via the OpenCL runtime. – end note]*

```
namespace sycl {

module<module_state::executable>
build(const module<module_state::input> &inputModule,
      const property_list &propList = {}); // (1)

module<module_state::object>
compile(const module<module_state::input> &inputModule,
        const property_list &propList = {}); // (2)

module<module_state::executable>
link(const module<module_state::object> &objModule,
     const property_list &propList = {}); // (3)

module<module_state::executable>
link(const std::vector<module<module_state::object>> &objModules,
```

```
    const property_list &propList = {}); // (4)

template<module_state T>
module<T> join(const std::vector<module<T>> &modules); // (5)

} // namespace sycl
```

1. *Effects:* Performs implementation defined build operation(s), including compilation and linking, on the input module input, applying any properties provided via `propList`.

   *Returns:* A `module<module_state::executable>` object containing the result of the build operation(s) performed. The `module` returned must represent the same SYCL kernel functions and be associated with the same context and devices as that of `inputModule`, however the device images may differ.

   *Throws:* `errc::build_error` if none of the devices associated with the module have `aspect::online_compiler`.

   *Throws:* `errc::build_error` if the build operation(s) fail.

2. *Effects:* Performs implementation defined compilation operation(s) on the input module `inputModule`, applying any properties provided via `propList`.

   *Returns:* A `module<module_state::object>` object containing the result of the compilation operation(s) performed. The `module` returned must represent the same SYCL kernel functions and be associated with the same context and devices as that of `inputModule`, however the device images may differ.

   *Throws:* `errc::compile_error` if none of the devices associated with the module have `aspect::online_compiler`.

   *Throws:* `errc::compile_error` if the compilation operation(s) fail.

3. *Effects:* Performs implementation defined linking operation(s) on the input module `objModule`, applying any properties provided via `propList`. If two or more modules contain the same SYCL kernel functions it is assumed that they are the same so one of them is selected and the rest are discard. The one which is selected is implementation defined.

   *Returns:* A `module<module_state::executable>` object containing the result of the linking operation(s) performed. The `module` returned must represent the same SYCL kernel functions and be associated with the same context and devices as that of `objModule`, however the device images may differ.

   *Throws:* `errc::link_error` if none of the devices associated with the module have `aspect::online_linker`.

   *Throws:* `errc::link_error` if the compilation operation(s) fail.

4. *Preconditions:* Each `module` in `moduleObjects` must be associated with the same context and devices.

   *Effects:* Performs implementation defined linking operation(s) on the input modules in `moduleObjects`, applying any properties provided via `propList`. If two or more modules contain the same SYCL kernel functions it is assumed that they are the same so one of them is selected and the rest are discard. The one which is selected is implementation defined.

   *Returns:* A `module<module_state::executable>` object containing the result of the linking operation(s) performed. The module returned must represent the same SYCL kernel functions and be associated with the same context and devices as that of objModules, however the device images may differ.

*Throws:* errc::link_error if none of the devices associated with the module have aspect::online_linker.

*Throws:* errc::link_error if the compilation operation(s) fail.

*Throws:* errc::invalid_object_error if objModules.empty() evaluates to true or any module in objModules is associated with a different context or devices than another module in objModules.

5. *Preconditions:* Each module in modules must be associated with the same context and devices.

   *Effects:* Performs implementation defined joining operation(s) on the modules in modules, applying any properties provided via propList. If two or more modules contain the same SYCL kernel functions it is assumed that they are the same so one of them is selected and the rest are discard. The one which is selected is implementation defined.

   *Returns:* A module object of module_state T, containing the result of the linking operation(s) performed. The module returned must represent all of the combined SYCL kernel functions associated each module in modules and must be associated with the same context and devices as that of modules, however the device images may differ.

   *Throws:* errc::link_error if the joining operation(s) fail.

   *Throws:* errc::invalid_object_error if modules.empty() evaluates to true or any module in modules is associated with a different context or devices than another module in modules.

### 4.13.7 Namespace this_module

The namespace this_module provides additional free functions which can be used for querying and retrieving modules available to the current translation unit.

#### 4.13.7.1 Type traits

```
template <typename T>
std::string kernel_name_v; // (1)
```

1. Template variable that takes a type T specifying the type kernel name of a SYCL kernel function in the current translation unit and whose value is the corresponding string kernel name.

#### 4.13.7.2 Free functions

```
bool has_any_module(context ctxt); // (1)

template<module_state S> // (2)
bool has_module_in(context ctxt);

template<module_state S> // (3)
module<T> get(context ctxt);
```

1. *Returns:* `true` if is a module of any module state, available within the current translation unit that is compatible with the context `ctxt`, otherwise returns `false`. Must not perform any compilation, linking, building or joining operation(s).

2. *Returns:* `true` if is a module of the module state specified by the `module_state` value `S`, available or retrievable within the current translation unit that is compatible with the context `ctxt`, otherwise returns `false`. Must not perform any compilation, linking, building or joining operation(s).

3. *Effects:* May perform implementation defined compilation, linking or building operations in order to transform a module that is available in another module state into the module state specified by the `module_state` value `S`

   *Returns:* A `module` of `module_state` `S` representing a module associated with the current translation unit and the context `ctxt` if `this->get_module_in<S>()` evaluates to `true`, otherwise returns a `module` that is empty. If the returned `module` if not empty, it must represent the set of SYCL kernel functions available to the current translation unit and may also contain a further implementation defined set of SYCL kernel functions.

*[Note: A `module` returned from `module<State>::get` may contain additional SYCL kernel functions to those available in the current translation unit in order to facilitate different single-source compilation methods. This does not impact user code as `link` and `join` are required to assume any duplicate SYCL kernel functions are the same. – end note]*

## 4.14 Defining kernels

In SYCL, functions that are executed on a SYCL device are referred to as SYCL kernel functions. A kernel containing such a SYCL kernel function is enqueued on a device queue in order to be executed on that particular device.

The return type of the SYCL kernel function is `void`, and `operator()` of the SYCL kernel function must be const-qualified. A SYCL kernel function may be copied zero or more times by an implementation, and it is undefined behavior if a kernel writes to any member of the SYCL kernel function.

All kernel accesses between host and device are through the accessor class 4.7.6 or USM pointers.

There are three ways of defining kernels: as named function objects, as lambda functions, or through backend-specific interoperability interfaces for modules and kernels, where available.

### 4.14.1 Defining kernels as named function objects

A kernel can be defined as a named function object type. These function objects provide the same functionality as any C++ function object, with the restriction that they need to follow C++ rules to be trivially copyable. The kernel function can be templated via templating the kernel function object type. The `operator()` function may take different parameters depending on the data accesses defined for the specific kernel. For details on restrictions for kernel naming, please refer to 5.2.

The following example defines a SYCL kernel function, *RandomFiller*, which initializes a buffer with a random number. The random number is generated during the construction of the function object while processing the command group. The `operator()` member function of the function object receives an `item` object. This method will be called for each work item of the execution range. The value of the random number will be assigned to each element of the buffer. In this case, the accessor and the scalar random number are members of the function object and therefore will be parameters to the device kernel. Usual restrictions of passing parameters to kernels apply.

```
class RandomFiller {
 public:
  RandomFiller(accessor<int> ptr)
      : ptr_ { ptr } {
    std::random_device hwRand;
    std::uniform_int_distribution<> r { 1, 100 };
    randomNum_ = r(hwRand);
  }
  void operator()(item<1> item) const { ptr_[item.get_id()] = get_random(); }
  int get_random() { return randomNum_; }

 private:
  accessor<int> ptr_;
  int randomNum_;
};

void workFunction(buffer<int, 1>& b, queue& q, const range<1> r) {
    myQueue.submit([&](handler& cgh) {
      accessor ptr { buf, cgh };
      RandomFiller filler { ptr };

      cgh.parallel_for(r, filler);
    });
}
```

### 4.14.2 Defining kernels as lambda functions

In C++, function objects can be defined using lambda functions. Kernels may be defined as lambda functions in SYCL. The name of a lambda function in SYCL may optionally be specified by passing it as a template parameter to the invoking method, and in that case, the lambda name is a *C++ typename*. If the lambda function relies on template arguments, then if specified, the name of the lambda function must contain those template arguments. The class used for the name of a lambda function is only used for naming purposes and is not required to be defined. For details on restrictions for kernel naming, please refer to 5.2.

The kernel function for the lambda function is the lambda function itself. The kernel lambda must use copy for all of its captures (i.e. [=]).

```
class MyKernel;

myQueue.submit([&](handler& cmdGroup) {
  cmdgroup.single_task<class MyKernel>([=]() {
    // [kernel code]
  });
});
```

### 4.14.3 Defining kernels using modules

In case the developer needs to specify compiler flags or special linkage options for a kernel, then a kernel object can be used, as described in 4.13.5.2. The SYCL kernel function is defined as a named function object 4.14.1 or lambda function 4.14.2. The user can obtain a `module` object for the kernel with the `get_kernel` method. This method is templated by the kernel name, so that the user can specify the kernel whose associated kernel they wish to obtain.

In the following example, the kernel is defined as a lambda function. The example obtains the `module` object for the lambda function kernel and then passes it to the `parallel_for`.

```
class my_kernel; // Forward declaration of the name of the lambda functor

sycl::queue myQueue;
auto myContext = myQueue.get_context();

auto myModule = sycl::this_module::get<module_state::executable>(myContext);

auto myRange = sycl::nd_range<2>(range<2>(1024, 1024),range<2>(64, 64));

myQueue.submit([&](sycl::handler& cgh) {
  cgh.use_module(myModule);
  cgh.parallel_for<my_kernel>(myRange, ([=](sycl::nd_item<2> index) {
    // kernel code
  }));
});
```

In the above example, the SYCL kernel function is defined in the `parallel_for` invocation as part of a lambda function which is named using the type of the forward declared class “myKernel”. The type of the function object and the `module` object enable the compilation and linking of the kernel in the `module` class, *a priori* of its actual invocation as a kernel object. For more details on the SYCL device compiler please refer to chapter 5.

In the following example, the SYCL kernel function performs a convolution and uses specialization constants to set the values of the coefficients.

```
#include <CL/sycl.hpp>

using namespace sycl;

using coeff_t = std::array<std::array<float, 3>, 3>;

// Read coefficients from somewhere.
coeff_t get_coefficients();

// Identify the specialization constant.
specialization_id<coeff_t> coeff_id;

void do_conv(buffer<float, 2> in, buffer<float, 2> out) {
  queue myQueue;

  myQueue.submit([&](handler &cgh) {
    accessor in_acc { in, cgh, read_only };
    accessor out_acc { out, cgh, write_only };
```

```

    // Set the coefficient of the convolution as constant.
    // This will build a specific kernel the coefficient available as literals.
    cgh.set_specialization_constant<coeff_id>(get_coefficients());

    cgh.parallel_for<class Convolution>(
        in.get_range(), [=](item<2> item_id, kernel_handler h) {
         float acc = 0;
         coeff_t coeff = h.get_specialization_constant<coeff_id>();
         for (int i = -1; i <= 1; i++) {
          if (item_id[0] + i < 0 || item_id[0] + i >= in_acc.get_range()[0])
           continue;
          for (int j = -1; j <= 1; j++) {
           if (item_id[1] + j < 0 || item_id[1] + j >= in_acc.get_range()[1])
            continue;
           // the underlying JIT can see all the values of the array returned by coeff.get().
           acc += coeff[i + 1][j + 1] *
                  in_acc[item_id[0] + i][item_id[1] + j];
          }
         }
         out_acc[item_id] = acc;
        });
   });

   myQueue.wait();
 }
```

### 4.14.4 Rules for parameter passing to kernels

In a case where a kernel is a named function object or a lambda function, any member variables encapsulated within the function object or variables captured by the lambda function must be treated according to the following rules:

- Any accessor must be passed as an argument to the device kernel in a form that allows the device kernel to access the data in the specified way.
- The SYCL runtime and compiler(s) must produce the necessary conversions to enable accessor arguments from the host to be converted to the correct type of parameter on the device.
- The device compiler(s) must validate that the layout of any data shared between the host and the device(s) (e.g. value kernel arguments or data accessed through an accessor or USM) is compatible with the layout of that data on the host. If there is a layout mismatch that the implementation cannot or will not correct for (to make the layouts compatible), then the device compiler must issue an error and compilation must fail.
- A local accessor provides access to work-group-local memory. The accessor is not constructed with any buffer, but instead constructed with a size and base data type. The runtime must ensure that the work-group-local memory is allocated per work-group and available to be used by the kernel via the local accessor.
- C++ trivially copyable types must be passed by value to the kernel.
- C++ non-trivially copyable types must not be passed as arguments to a kernel that is compiled for a device.
- It is illegal to pass a buffer or image (instead of an accessor class) as an argument to a kernel. Generation of a compiler error in this illegal case is optional.

- Sampler objects (`sampler`) can be passed as parameters to kernels.
- It is illegal to pass a pointer or reference argument to a kernel. Generation of a compiler error in this illegal case is optional.
- Any aggregate types such as structs or classes should follow the rules above recursively. It is not necessary to separate struct or class members into separate kernel parameters if all members of the aggregate type are unaffected by the rules above.

# 4.15 Error handling

## 4.15.1 Error handling rules

Error handling in a SYCL application (host code) uses C++ exceptions. If an error occurs, it will be thrown by the API function call and may be caught by the user through standard C++ exception handling mechanisms.

SYCL applications are asynchronous in the sense that host and device code executions are decoupled from one another except at specific points. For example, device code executions often begin when dependencies in the SYCL task graph are satisfied, which occurs asynchronously from host code execution. As a result of this the errors that occur on a device cannot be thrown directly from a host API call, because the call enqueueing a device action has typically already returned by the time that the error occurs. Such errors are not detected until the error-causing task executes or tries to execute, and we refer to these as asynchronous errors.

### 4.15.1.1 Asynchronous error handler

The queue and context classes can optionally take an asynchronous handler object `async_handler` on construction, which is a callable such as a function class or lambda, with an `exception_list` as a parameter. Invocation of an `async_handler` may be triggered by the queue member functions `queue::wait_and_throw()` or `queue::throw_asynchronous()`, by the event member function `event::wait_and_throw()`, or automatically on destruction of a queue or context that contains unconsumed asynchronous errors. When invoked, an `async_handler` is called and receives an `exception_list` argument containing a list of exception objects representing any unconsumed asynchronous errors associated with the queue or context.

When an asynchronous error instance has been passed to an `async_handler`, then that instance of the error has been consumed for handling and is not reported on any subsequent invocations of the `async_handler`.

The `async_handler` may be a named function object type, a lambda function or a `std::function`. The `exception_list` object passed to the `async_handler` is constructed by the SYCL runtime.

### 4.15.1.2 Behavior without an `async_handler`

If an asynchronous error occurs in a queue or context that has no user-supplied asynchronous error handler object `async_handler`, then an implementation defined default `async_handler` is called to handle the error in the same situations that a user-supplied `async_handler` would be, as defined in 4.15.1.1. The default `async_handler` must in some way report all errors passed to it, when possible, and must then invoke `std::terminate` or equivalent.

### 4.15.1.3 Priorities of `async_handlers`

If the SYCL runtime can associate an asynchronous error with a specific queue, then:

- If the queue was constructed with an `async_handler`, that handler is invoked to handle the error.

- Otherwise if the context enclosing the queue was constructed with an async_handler, that handler is invoked to handle the error.
- Otherwise when no handler was passed to either queue or context on construction, then a default handler is invoked to handle the error, as described by 4.15.1.2.
- All handler invocations in this list occur at times as defined by 4.15.1.1.

If the SYCL runtime cannot associate an asynchronous error with a specific queue, then:

- If the context in which the error occurred was constructed with an async_handler, then that handler is invoked to handle the error.
- Otherwise when no handler was passed to the associated context on construction, then a default handler is invoked to handle the error, as described by 4.15.1.2.
- All handler invocations in this paragraph occur at times as defined by 4.15.1.1.

#### 4.15.1.4 Asynchronous errors with a secondary queue

If an asynchronous error occurs when running or enqueuing a command group which has a secondary queue specified, then the command group may be enqueued to the secondary queue instead of the primary queue. The error handling in this case is also configured using the async_handler provided for both queues. If there is no async_handler given on any of the queues, then no asynchronous error reporting is done and no exceptions are thrown. If the primary queue fails and there is an async_handler given at this queue's construction, which populates the exception_list parameter, then any errors will be added and can be thrown whenever the user chooses to handle those exceptions. Since there were errors on the primary queue and a secondary queue was given, then the execution of the kernel is re-scheduled to the secondary queue and any error reporting for the kernel execution on that queue is done through that queue, in the same way as described above. The secondary queue may fail as well, and the errors will be thrown if there is an async_handler and either wait_and_throw() or throw() are called on that queue. The command group function object event returned by that function will be relevant to the queue where the kernel has been enqueued.

Below is an example of catching a SYCL exception and printing out the error message.

```
void catch_any_errors(sycl::context const& ctx) {
  try {
    do_something_to_invoke_error(ctx);
  }
  catch(sycl::exception const& e) {
    std::cerr << e.what();
  }
}
```

Below is an example of catching a SYCL exception with the invalid_object error code and printing out the error message.

```
void catch_invalid_object_errors(sycl::context const& ctx) {
  try {
    do_something_to_invoke_error(ctx);
  }
  catch(sycl::exception const& e) {
```

```
    if(e.code() == sycl::errc::invalid_object) {
      std::cerr << "Invalid object error: " << e.what();
    }
    else {
      throw;
    }
  }
}
```

Below is an example of catching a SYCL exception, checking for the SYCL backend by inspecting the category and handling the OpenCL SYCL backend error codes if the category is that of the OpenCL SYCL backend otherwise checking the standard error code.

```
void catch_backend_errors(sycl::context const& ctx) {
  try {
    do_something_to_invoke_error(ctx);
  }
  catch(sycl::exception const& e) {
    if(e.category() == sycl::error_category_for<sycl::backend::opencl>()) {
      switch(e.code().value()) {
        case CL_INVALID_PROGRAM:
          std::cerr << "OpenCL invalid program error: " << e.what();
        /* ...*/
      }
      else {
        throw;
      }
    }
    else {
      if(e.code() == sycl::errc::invalid_object) {
        std::cerr << "Invalid object error: " << e.what();
      }
      else {
        throw;
      }
    }
  }
}
```

### 4.15.2 Exception class interface

case is also configured using the `async_handler` provided on construction of one or both queues or enclosing contexts. If execution on the primary queue fails then execution of the kernel is re-scheduled to the secondary queue, but regardless of this behavior, any asynchronous errors occurring on either queue are associated with the queue and are handled as defined in 4.15.1.1 and 4.15.1.2.

```
namespace sycl {

using async_handler = std::function<void(sycl::exception_list)>;

class exception : public virtual std::exception {
 public:
```

```
    exception(std::error_code ec, const std::string& what_arg);
    exception(std::error_code ec, const char * what_arg);
    exception(std::error_code ec);
    exception(int ev, const std::error_category& ecat, const std::string& what_arg);
    exception(int ev, const std::error_category& ecat, const char* what_arg);
    exception(int ev, const std::error_category& ecat);

    exception(context ctx, std::error_code ec, const std::string& what_arg);
    exception(context ctx, std::error_code ec, const char* what_arg);
    exception(context ctx, std::error_code ec);
    exception(context ctx, int ev, const std::error_category& ecat, const std::string& what_arg);
    exception(context ctx, int ev, const std::error_category& ecat, const char* what_arg);
    exception(context ctx, int ev, const std::error_category& ecat);

    const std::error_code& code() const noexcept;
    const std::error_category& category() const noexcept;

    bool has_context() const noexcept;
    context get_context() const;
};

class exception_list {
  // Used as a container for a list of asynchronous exceptions
 public:
  using value_type = std::exception_ptr;
  using reference = value_type&;
  using const_reference = const value_type&;
  using size_type = std::size_t;
  using iterator = /*unspecified*/;
  using const_iterator = /*unspecified*/;

  size_type size() const;
  iterator begin() const;  // first asynchronous exception
  iterator end() const;    // refer to past-the-end last asynchronous exception
};

enum class errc {
  runtime_error = /* implementation-defined */,
  kernel = /* implementation-defined */,
  accessor = /* implementation-defined */,
  nd_range = /* implementation-defined */,
  event = /* implementation-defined */,
  invalid_parameter = /* implementation-defined */,
  compile_program = /* implementation-defined */,
  link_program = /* implementation-defined */,
  invalid_object = /* implementation-defined */,
  memory_allocation = /* implementation-defined */,
  platform = /* implementation-defined */,
  profiling = /* implementation-defined */,
  feature_not_supported = /* implementation-defined */
};

template<backend b>
using errc_for = typename backend_traits<b>::errc;
```

```
std::error_condition make_error_condition(errc e) noexcept;
std::error_code make_error_code(errc e) noexcept;

const std::error_category& sycl_category() noexcept;

template<backend b>
const std::error_category& error_category_for() noexcept;

}  // namespace sycl

namespace std {

  template <>
  struct is_error_condition_enum<sycl::errc> : true_type {};

  template <>
  struct is_error_code_enum<see-below> : true_type {};

}  // namespace std
```

The SYCL `exception_list` class is also available in order to provide a list of synchronous and asynchronous exceptions.

There are two categories of errors, the `runtime_error` that refers to the scheduling errors that may happen during execution, and the `device_error` that refers to the execution errors on a SYCL device.

Errors can occur both in the SYCL library and SYCL host side, or may come directly from a SYCL backend. The member functions on these exceptions provide the corresponding information. SYCL backends can provide additional exception class objects as long as they derive from `sycl::exception` object, or any of its derived classes.

The asynchronous handler object `async_handler` is a `std::function` with an `exception_list` as a parameter. The asynchronous handler is an optional parameter to a constructor of the `queue` class and it is the only way to handle asynchronous errors occurring on a SYCL device. The asynchronous handler may be a named function object type, a lambda function or a `std::function`, that can be given to the queue and will be executed on error. The `exception_list` object is constructed from the SYCL runtime and is populated with the errors caught during the execution of all the kernels running on the same queue.

A specialization of `std::is_error_condition_enum` must be defined for `sycl::errc` inheriting from `std::true_type`.

A specialization of `std::is_error_code_enum` must be defined for `sycl::errc` and `backend_traits<Backend>::errc` inheriting from `std::true_type` for each `Backend`, where `backend` is each enumeration of the enum class `backend`.

| Member function | Description |
|---|---|
| `exception(std::error_code ec, const std::string& what_arg)` | Constructs an `exception`. The string returned by `what()` is guaranteed to contain `what_arg` as a substring. |

Continued on next page

Table 4.101: Member functions of the SYCL `exception` class.

| Member function | Description |
| --- | --- |
| exception(std::error_code ec, const char* what_arg) | Constructs an exception. The string returned by what() is guaranteed to contain what_arg as a substring. |
| exception(std::error_code ec) | Constructs an exception. |
| exception(int ev, const std::error_category& ecat, const std::string& what_arg) | Constructs an exception with the error code ev and the underlying error category ecat. The string returned by what() is guaranteed to contain what_arg as a substring. |
| exception(int ev, const std::error_category& ecat, const char* what_arg) | Constructs an exception with the error code ev and the underlying error category ecat. The string returned by what() is guaranteed to contain what_arg as a substring. |
| exception(int ev, const std::error_category& ecat) | Constructs an exception with the error code ev and the underlying error category ecat. |
| exception(context ctx, std::error_code ec, const std ::string& what_arg) | Constructs an exception with an associated SYCL context ctx. The string returned by what() is guaranteed to contain what_arg as a substring. |
| exception(context ctx, std::error_code ec, const char* what_arg) | Constructs an exception with an associated SYCL context ctx. The string returned by what() is guaranteed to contain what_arg as a substring. |
| exception(context ctx, std::error_code ec) | Constructs an exception with an associated SYCL context ctx. |
| exception(context ctx, int ev, const std:: error_category& ecat, const std::string& what_arg) | Constructs an exception with an associated SYCL context ctx, the error code ev and the underlying error category ecat. The string returned by what() is guaranteed to contain what_arg as a substring. |
| exception(context ctx, int ev, const std:: error_category& ecat, const char* what_arg) | Constructs an exception with an associated SYCL context ctx, the error code ev and the underlying error category ecat. The string returned by what() is guaranteed to contain what_arg as a substring. |
| exception(context ctx, int ev, const std:: error_category& ecat) | Constructs an exception with an associated SYCL context ctx, the error code ev and the underlying error category ecat. |
| const std::error_code& code()const noexcept | Returns the error code stored inside the exception. |
| const std::error_categeory& category()const noexcept | Returns the error category of the error code stored inside the exception. |
| const char *what()const | Returns an implementation defined non-null constant C-style string that describes the error that triggered the exception. |
| bool has_context()const | Returns true if this SYCL exception has an associated SYCL context and false if it does not. |

Continued on next page

Table 4.101: Member functions of the SYCL exception class.

| Member function | Description |
|---|---|
| context get_context()const | Returns the SYCL context that is associated with this SYCL exception if one is available. Must throw an exception with the errc::invalid_object_error error code if this SYCL exception does not have a SYCL context. |
| | End of table |

Table 4.101: Member functions of the SYCL exception class.

| Member function | Description |
|---|---|
| size_t size()const | Returns the size of the list |
| iterator begin()const | Returns an iterator to the beginning of the list of asynchronous exceptions. |
| iterator end()const | Returns an iterator to the end of the list of asynchronous exceptions. |
| | End of table |

Table 4.102: Member functions of the exception_list.

| Standard SYCL Error Codes | Description |
|---|---|
| runtime_error | Generic runtime error. |
| kernel_error | Error that occurred before or while enqueuing the SYCL kernel. |
| nd_range_error | Error regarding the SYCL nd_range specified for the SYCL kernel |
| accessor_error | Error regarding the SYCL accessor objects defined. |
| event_error | Error regarding associated SYCL event objects. |
| invalid_parameter_error | Error regarding parameters to the SYCL kernel, it may apply to any captured parameters to the kernel lambda. |
| compile_program_error | Error while compiling the SYCL kernel to a SYCL device. |
| link_program_error | Error while linking the SYCL kernel to a SYCL device. |
| invalid_object_error | Error regarding any memory objects being used inside the kernel |
| memory_allocation_error | Error on memory allocation on the SYCL device for a SYCL kernel. |
| platform_error | The SYCL platform will trigger this exception on error. |
| profiling_error | The SYCL runtime will trigger this error if there is an error when profiling info is enabled. |
| | Continued on next page |

Table 4.103: Values of the SYCL errc enum.

| Standard SYCL Error Codes | Description |
|---|---|
| `feature_not_supported` | Exception thrown when an optional feature or extension is used in a kernel but it's not available on the device the SYCL kernel is being enqueued on. |
| | End of table |

Table 4.103: Values of the SYCL `errc` enum.

| SYCL Error Code Helpers | Description |
|---|---|
| `std::error_condition make_error_condition(errc e) noexcept;` | Constructs an error condition using `e` and `sycl_category()`. |
| `std::error_code make_error_code(errc e)noexcept;` | Constructs an error code using `e` and `sycl_category()`. |
| | End of table |

Table 4.104: SYCL error code helper functions.

# 4.16 Data types

SYCL as a C++ programming model supports the C++ core language data types, and it also provides the ability for all SYCL applications to be executed on SYCL compatible devices. The scalar and vector data types that are supported by the SYCL system are defined below. More details about the SYCL device compiler support for fundamental and OpenCL interoperability types are found in 5.5.

## 4.16.1 Scalar data types

The fundamental C++ data types which are supported in SYCL are described in Table 5.1. Note these types are fundamental and therefore do not exist within the `sycl` namespace.

Additional scalar data types which are supported by SYCL within the `sycl` namespace are described in Table 4.105.

| Scalar data type | Description |
|---|---|
| `byte` | A signed or unsigned 8-bit integer, as defined by the C++ core language. |
| `half` | A 16-bit floating-point. The half data type must conform to the IEEE 754-2008 half precision storage format. A SYCL `feature_not_supported` exception must be thrown if the `half` type is used in a SYCL kernel function which executes on a SYCL `device` that does not have `aspect::fp16`. |
| | End of table |

Table 4.105: Additional scalar data types supported by SYCL.

## 4.16.2 Vector types

SYCL provides a cross-platform class template that works efficiently on SYCL devices as well as in host C++ code. This type allows sharing of vectors between the host and its SYCL devices. The vector supports methods that allow construction of a new vector from a swizzled set of component elements.

`vec<typename` dataT, `int` numElements> is a vector type that compiles down to a SYCL backend built-in vector types on SYCL devices, where possible, and provides compatible support on the host or when it is not possible. The `vec` class is templated on its number of elements and its element type. The number of elements parameter, `numElements`, can be one of: 1, 2, 3, 4, 8 or 16. Any other value should produce a compilation failure. The element type parameter, `dataT`, must be one of the basic scalar types supported in device code.

The SYCL `vec` class template provides interoperability with the underlying vector type defined by `vector_t` which is available only when compiled for the device. The SYCL `vec` class can be constructed from an instance of `vector_t` and can implicitly convert to an instance of `vector_t` in order to support interoperability with native SYCL backend functions from a SYCL kernel function.

An instance of the SYCL `vec` class template can also be implicitly converted to an instance of the data type when the number of elements is `1` in order to allow single element vectors and scalars to be convertible with each other.

#### 4.16.2.1 Vec interface

The constructors, member functions and non-member functions of the SYCL `vec` class template are listed in Tables 4.106, 4.107 and 4.108 respectively.

```
namespace sycl {

enum class rounding_mode {
  automatic,
  rte,
  rtz,
  rtp,
  rtn
};

struct elem {
  static constexpr int x = 0;
  static constexpr int y = 1;
  static constexpr int z = 2;
  static constexpr int w = 3;
  static constexpr int r = 0;
  static constexpr int g = 1;
  static constexpr int b = 2;
  static constexpr int a = 3;
  static constexpr int s0 = 0;
  static constexpr int s1 = 1;
  static constexpr int s2 = 2;
  static constexpr int s3 = 3;
  static constexpr int s4 = 4;
  static constexpr int s5 = 5;
  static constexpr int s6 = 6;
  static constexpr int s7 = 7;
  static constexpr int s8 = 8;
  static constexpr int s9 = 9;
```

```
  static constexpr int sA = 10;
  static constexpr int sB = 11;
  static constexpr int sC = 12;
  static constexpr int sD = 13;
  static constexpr int sE = 14;
  static constexpr int sF = 15;
};

template <typename dataT, int numElements>
class vec {
 public:
  using element_type = dataT;

#ifdef __SYCL_DEVICE_ONLY__
  using vector_t = __unspecified__;
#endif

  vec();

  explicit vec(const dataT &arg);

  template <typename... argTN>
  vec(const argTN&... args);

  vec(const vec<dataT, numElements> &rhs);

#ifdef __SYCL_DEVICE_ONLY__
  vec(vector_t openclVector);

  operator vector_t() const;
#endif

  // Available only when: numElements == 1
  operator dataT() const;

  static constexpr int get_count();

  static constexpr size_t get_size();

  template <typename convertT, rounding_mode roundingMode = rounding_mode::automatic>
  vec<convertT, numElements> convert() const;

  template <typename asT>
  asT as() const;

  template<int... swizzleIndexes>
  __swizzled_vec__ swizzle() const;

  // Available only when numElements <= 4.
  // XYZW_ACCESS is: x, y, z, w, subject to numElements.
  __swizzled_vec__ XYZW_ACCESS() const;

  // Available only numElements == 4.
  // RGBA_ACCESS is: r, g, b, a.
  __swizzled_vec__ RGBA_ACCESS() const;
```

```

  // INDEX_ACCESS is: s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, sA, sB, sC, sD,
  // sE, sF, subject to numElements.
  __swizzled_vec__ INDEX_ACCESS() const;

#ifdef SYCL_SIMPLE_SWIZZLES
  // Available only when numElements <= 4.
  // XYZW_SWIZZLE is all permutations with repetition of: x, y, z, w, subject to
  // numElements.
  __swizzled_vec__ XYZW_SWIZZLE() const;

  // Available only when numElements == 4.
  // RGBA_SWIZZLE is all permutations with repetition of: r, g, b, a.
  __swizzled_vec__ RGBA_SWIZZLE() const;

#endif  // #ifdef SYCL_SIMPLE_SWIZZLES

  // Available only when: numElements > 1.
  __swizzled_vec__ lo() const;
  __swizzled_vec__ hi() const;
  __swizzled_vec__ odd() const;
  __swizzled_vec__ even() const;

  // load and store member functions
  template <access::address_space addressSpace, access::decorated IsDecorated>
  void load(size_t offset, multi_ptr<const dataT, addressSpace, IsDecorated> ptr);
  template <access::address_space addressSpace, access::decorated IsDecorated>
  void store(size_t offset, multi_ptr<dataT, addressSpace, IsDecorated> ptr) const;

  // subscript operator
  dataT &operator[](int index);
  const dataT &operator[](int index) const;

  // OP is: +, -, *, /, %
  /* When OP is % available only when: dataT != float && dataT != double
  && dataT != half. */
  friend vec operatorOP(const vec &lhs, const vec &rhs) { /* ... */ }
  friend vec operatorOP(const vec &lhs, const dataT &rhs) { /* ... */ }

  // OP is: +=, -=, *=, /=, %=
  /* When OP is %= available only when: dataT != float && dataT != double
  && dataT != half. */
  friend vec &operatorOP(vec &lhs, const vec &rhs) { /* ... */ }
  friend vec &operatorOP(vec &lhs, const dataT &rhs) { /* ... */ }

  // OP is: ++, --
  friend vec &operatorOP(vec &lhs) { /* ... */ }
  friend vec operatorOP(vec& lhs, int) { /* ... */ }

  // OP is: +, -
  friend vec operatorOP(vec &lhs) const { /* ... */ }

  // OP is: &, |, ^
  /* Available only when: dataT != float && dataT != double
  && dataT != half. */
```

```
friend vec operatorOP(const vec &lhs, const vec &rhs) { /* ... */ }
friend vec operatorOP(const vec &lhs, const dataT &rhs) { /* ... */ }

// OP is: &=, |=, ^=
/* Available only when: dataT != float && dataT != double
&& dataT != half. */
friend vec &operatorOP(vec &lhs, const vec &rhs) { /* ... */ }
friend vec &operatorOP(vec &lhs, const dataT &rhs) { /* ... */ }

// OP is: &&, ||
friend vec<RET, numElements> operatorOP(const vec &lhs, const vec &rhs) { /* ... */ }
friend vec<RET, numElements> operatorOP(const vec& lhs, const dataT &rhs) { /* ... */ }

// OP is: <<, >>
/* Available only when: dataT != float && dataT != double
&& dataT != half. */
friend vec operatorOP(const vec &lhs, const vec &rhs) { /* ... */ }
friend vec operatorOP(const vec &lhs, const dataT &rhs) { /* ... */ }

// OP is: <<=, >>=
/* Available only when: dataT != float && dataT != double
&& dataT != half. */
friend vec &operatorOP(vec &lhs, const vec &rhs) { /* ... */ }
friend vec &operatorOP(vec &lhs, const dataT &rhs) { /* ... */ }

// OP is: ==, !=, <, >, <=, >=
friend vec<RET, numElements> operatorOP(const vec &lhs, const vec &rhs) { /* ... */ }
friend vec<RET, numElements> operatorOP(const vec &lhs, const dataT &rhs) { /* ... */ }

vec &operator=(const vec<dataT, numElements> &rhs);
vec &operator=(const dataT &rhs);

/* Available only when: dataT != float && dataT != double
&& dataT != half. */
friend vec operator~(const vec &v) { /* ... */ }
friend vec<RET, numElements> operator!(const vec &v) { /* ... */ }

// OP is: +, -, *, /, %
/* operator% is only available when: dataT != float && dataT != double &&
dataT != half. */
friend vec operatorOP(const dataT &lhs, const vec &rhs) { /* ... */ }

// OP is: &, |, ^
/* Available only when: dataT != float && dataT != double
&& dataT != half. */
friend vec operatorOP(const dataT &lhs, const vec &rhs) { /* ... */ }

// OP is: &&, ||
friend vec<RET, numElements> operatorOP(const dataT &lhs, const vec &rhs) { /* ... */ }

// OP is: <<, >>
/* Available only when: dataT != float && dataT != double
&& dataT != half. */
friend vec operatorOP(const dataT &lhs, const vec &rhs) { /* ... */ }
```

```
  // OP is: ==, !=, <, >, <=, >=
  friend vec<RET, numElements> operatorOP(const dataT &lhs, const vec &rhs) { /* ... */ }

};

// Deduction guides
// Available only when: (std::is_same_v<T, U> && ...)
template <class T, class... U>
vec(T, U...) -> vec<T, sizeof...(U) + 1>;

} // namespace sycl
```

| Constructor | Description |
|---|---|
| `vec()` | Default construct a vector with element type `dataT` and with `numElements` dimensions by default construction of each of its elements. |
| `explicit vec(const dataT &arg)` | Construct a vector of element type `dataT` and `numElements` dimensions by setting each value to `arg` by assignment. |
| `template <typename... argTN>`<br>`vec(const argTN&... args)` | Construct a SYCL `vec` instance from any combination of scalar and SYCL `vec` parameters of the same element type, providing the total number of elements for all parameters sum to `numElements` of this `vec` specialization. |
| `vec(const vec<dataT, numElements> &rhs)` | Construct a vector of element type `dataT` and number of elements `numElements` by copy from another similar vector. |
| `vec(vector_t openclVector)` | Available only when: compiled for the device.<br>Constructs a SYCL `vec` instance from an instance of the underlying OpenCL vector type defined by `vector_t`. |

End of table

Table 4.106: Constructors of the SYCL `vec` class template.

| Member function | Description |
|---|---|
| `operator vector_t()const` | Available only when: compiled for the device.<br>Converts this SYCL `vec` instance to the underlying OpenCL vector type defined by `vector_t`. |

Continued on next page

Table 4.107: Member functions for the SYCL `vec` class template.

| Member function | Description |
|---|---|
| `operator dataT()const` | Available only when: `numElements == 1`. Converts this SYCL `vec` instance to an instance of `dataT` with the value of the single element in this SYCL `vec` instance. The SYCL `vec` instance shall be implicitly convertible to the same data types, to which `dataT` is implicitly convertible. Note that conversion operator shall not be templated to allow standard conversion sequence for implicit conversion. |
| `static constexpr int get_count()` | Returns the number of elements of this SYCL `vec`. |
| `static constexpr size_t get_size()` | Returns the size of this SYCL `vec` in bytes. 3-element vector size matches 4-element vector size to provide interoperability with OpenCL vector types. The same rule applies to vector alignment as described in 4.16.2.6. |
| `template<typename convertT, rounding_mode roundingMode = rounding_mode::automatic> vec<convertT, numElements> convert()const` | Converts this SYCL `vec` to a SYCL `vec` of a different element type specified by `convertT` using the rounding mode specified by `roundingMode`. The new SYCL `vec` type must have the same number of elements as this SYCL `vec`. The different rounding modes are described in Table 4.109. |
| `template<typename asT> asT as()const` | Bitwise reinterprets this SYCL `vec` as a SYCL `vec` of a different element type and number of elements specified by `asT`. The new SYCL `vec` type must have the same storage size in bytes as this SYCL `vec`. |
| `template<int... swizzleIndexes> __swizzled_vec__ swizzle()const` | Return an instance of the implementation defined intermediate class template `__swizzled_vec__` representing an index sequence which can be used to apply the swizzle in a valid expression as described in 4.16.2.4. |
| `__swizzled_vec__ XYZW_ACCESS()const` | Available only when `numElements <= 4`. Returns an instance of the implementation defined intermediate class template `__swizzled_vec__` representing an index sequence which can be used to apply the swizzle in a valid expression as described in 4.16.2.4.<br><br>Where `XYZW_ACCESS` is: `x` for `numElements == 1`, `x, y` for `numElements == 2`, `x, y, z` for `numElements == 3` and `x, y, z, w` for `numElements == 4`. |

Continued on next page

Table 4.107: Member functions for the SYCL vec class template.

| Member function | Description |
|---|---|
| `__swizzled_vec__ RGBA_ACCESS()const` | Available only when `numElements == 4`. Returns an instance of the implementation defined intermediate class template `__swizzled_vec__` representing an index sequence which can be used to apply the swizzle in a valid expression as described in 4.16.2.4.<br><br>Where `RGBA_ACCESS` is: `r, g, b, a`. |
| `__swizzled_vec__ INDEX_ACCESS()const` | Returns an instance of the implementation defined intermediate class template `__swizzled_vec__` representing an index sequence which can be used to apply the swizzle in a valid expression as described in 4.16.2.4.<br><br>Where `INDEX_ACCESS` is: `s0` for `numElements == 1`, `s0, s1` for `numElements == 2`, `s0, s1, s2` for `numElements == 3`, `s0, s1, s2, s3` for `numElements == 4`, `s0, s1, s2, s3, s4, s5, s6, s7, s8` for `numElements == 8` and `s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, sA, sB, sC, sD, sE, sF` for `numElements == 16`. |
| `__swizzled_vec__ XYZW_SWIZZLE()const` | Available only when `numElements <= 4`, and when the macro `SYCL_SIMPLE_SWIZZLES` is defined before including `sycl.hpp`. Returns an instance of the implementation defined intermediate class template `__swizzled_vec__` representing an index sequence which can be used to apply the swizzle in a valid expression as described in 4.16.2.4.<br><br>Where XYZW_SWIZZLE is all permutations with repetition, of any subset with length greater than `1`, of `x, y` for `numElements == 2`, `x, y, z` for `numElements == 3` and `x, y, z, w` for `numElements == 4`. For example a four element `vec` provides permutations including `xzyw`, `xyyy` and `xz`. |

Continued on next page

Table 4.107: Member functions for the SYCL `vec` class template.

| Member function | Description |
|---|---|
| `__swizzled_vec__ RGBA_SWIZZLE()const` | Available only when `numElements == 4`, and when the macro `SYCL_SIMPLE_SWIZZLES` is defined before including `sycl.hpp`. Returns an instance of the implementation defined intermediate class template `__swizzled_vec__` representing an index sequence which can be used to apply the swizzle in a valid expression as described in 4.16.2.4.<br><br>Where RGBA_SWIZZLE is all permutations with repetition, of any subset with length greater than `1`, of `r`, `g`, `b`, `a`. For example a four element `vec` provides permutations including `rbga`, `rggg` and `rb`. |
| `__swizzled_vec__ lo()const` | Available only when: `numElements > 1`. Return an instance of the implementation defined intermediate class template `__swizzled_vec__` representing an index sequence made up of the lower half of this SYCL vec which can be used to apply the swizzle in a valid expression as described in 4.16.2.4. When `numElements == 3` this SYCL `vec` is treated as though `numElements == 4` with the fourth element undefined. |
| `__swizzled_vec__ hi()const` | Available only when: `numElements > 1`. Return an instance of the implementation defined intermediate class template `__swizzled_vec__` representing an index sequence made up of the upper half of this SYCL vec which can be used to apply the swizzle in a valid expression as described in 4.16.2.4. When `numElements == 3` this SYCL `vec` is treated as though `numElements == 4` with the fourth element undefined. |
| `__swizzled_vec__ odd()const` | Available only when: `numElements > 1`. Return an instance of the implementation defined intermediate class template `__swizzled_vec__` representing an index sequence made up of the odd indexes of this SYCL vec which can be used to apply the swizzle in a valid expression as described in 4.16.2.4. When `numElements == 3` this SYCL `vec` is treated as though `numElements == 4` with the fourth element undefined. |

Continued on next page

Table 4.107: Member functions for the SYCL `vec` class template.

| Member function | Description |
|---|---|
| `__swizzled_vec__ even()const` | Available only when: `numElements > 1`. Return an instance of the implementation defined intermediate class template `__swizzled_vec__` representing an index sequence made up of the even indexes of this SYCL vec which can be used to apply the swizzle in a valid expression as described in 4.16.2.4. When `numElements == 3` this SYCL `vec` is treated as though `numElements == 4` with the fourth element undefined. |
| `template <access::address_space addressSpace, access::decorated IsDecorated> void load(size_t offset, multi_ptr<const dataT, addressSpace, IsDecorated> ptr)` | Loads the values at the address of `ptr` offset in elements of type `dataT` by `numElements * offset`, into the components of this SYCL `vec`. |
| `template <access::address_space addressSpace, access::decorated IsDecorated> void store(size_t offset, multi_ptr<dataT, addressSpace, IsDecorated> ptr)const` | Stores the components of this SYCL `vec` into the values at the address of `ptr` offset in elements of type `dataT` by `numElements * offset`. |
| `dataT &operator[](int index)` | Returns a reference to the element stored within this SYCL `vec` at the index specified by `index`. |
| `const dataT &operator[](int index)const` | Returns a const reference to the element stored within this SYCL `vec` at the index specified by `index`. |
| `vec &operator=(const vec &rhs)` | Assign each element of the `rhs` SYCL `vec` to each element of this SYCL `vec` and return a reference to this SYCL `vec`. |
| `vec &operator=(const dataT &rhs)` | Assign each element of the `rhs` scalar to each element of this SYCL `vec` and return a reference to this SYCL `vec`. |
| | End of table |

Table 4.107: Member functions for the SYCL `vec` class template.

| Hidden friend function | Description |
|---|---|
| `vec operatorOP(const vec &lhs, const vec &rhs)` | When `OP` is `%` available only when: `dataT != float && dataT != double && dataT != half`. Construct a new instance of the SYCL `vec` class template with the same template parameters as `lhs` `vec` with each element of the new SYCL `vec` instance the result of an element-wise `OP` arithmetic operation between each element of `lhs` `vec` and each element of the `rhs` SYCL `vec`. <br><br> Where `OP` is: `+`, `-`, `*`, `/`, `%`. |
| `vec operatorOP(const vec &lhs, const dataT &rhs)` | When `OP` is `%` available only when: `dataT != float && dataT != double && dataT != half`. Construct a new instance of the SYCL `vec` class template with the same template parameters as `lhs` `vec` with each element of the new SYCL `vec` instance the result of an element-wise `OP` arithmetic operation between each element of `lhs` `vec` and the `rhs` scalar. <br><br> Where `OP` is: `+`, `-`, `*`, `/`, `%`. |
| `vec &operatorOP(vec &lhs, const vec &rhs)` | When `OP` is `%=` available only when: `dataT != float && dataT != double && dataT != half`. Perform an in-place element-wise `OP` arithmetic operation between each element of `lhs` `vec` and each element of the `rhs` SYCL `vec` and return `lhs` `vec`. <br><br> Where `OP` is: `+=`, `-=`, `*=`, `/=`, `%=`. |
| `vec &operatorOP(vec &lhs, const dataT &rhs)` | When `OP` is `%=` available only when: `dataT != float && dataT != double && dataT != half`. Perform an in-place element-wise `OP` arithmetic operation between each element of `lhs` `vec` and `rhs` scalar and return `lhs` `vec`. <br><br> Where `OP` is: `+=`, `-=`, `*=`, `/=`, `%=`. |
| `vec &operatorOP(vec &v)` | Perform an in-place element-wise `OP` prefix arithmetic operation on each element of `lhs` `vec`, assigning the result of each element to the corresponding element of `lhs` `vec` and return `lhs` `vec`. <br><br> Where `OP` is: `++`, `--`. |

Continued on next page

Table 4.108: Hidden friend functions of the `vec` class template.

| Hidden friend function | Description |
|---|---|
| `vec operatorOP(vec &v, int)` | Perform an in-place element-wise `OP` post-fix arithmetic operation on each element of `lhs vec`, assigning the result of each element to the corresponding element of `lhs vec` and returns a copy of `lhs vec` before the operation is performed.<br><br>Where `OP` is: `++`, `--`. |
| `vec operatorOP(vec &v)` | Construct a new instance of the SYCL `vec` class template with the same template parameters as this SYCL `vec` with each element of the new SYCL `vec` instance the result of an element-wise `OP` unary arithmetic operation on each element of this SYCL `vec`.<br><br>Where `OP` is: `+`, `-`. |
| `vec operatorOP(const vec &lhs, const vec &rhs)` | Available only when: `dataT != float && dataT != double && dataT != half`. Construct a new instance of the SYCL `vec` class template with the same template parameters as `lhs vec` with each element of the new SYCL `vec` instance the result of an element-wise `OP` bitwise operation between each element of `lhs vec` and each element of the `rhs` SYCL `vec`.<br><br>Where `OP` is: `&`, `\|`, `^`. |
| `vec operatorOP(const vec &lhs, const dataT &rhs)` | Available only when: `dataT != float && dataT != double && dataT != half`. Construct a new instance of the SYCL `vec` class template with the same template parameters as `lhs vec` with each element of the new SYCL `vec` instance the result of an element-wise `OP` bitwise operation between each element of `lhs vec` and the `rhs` scalar.<br><br>Where `OP` is: `&`, `\|`, `^`. |
| `vec &operatorOP(vec &lhs, const vec &rhs)` | Available only when: `dataT != float && dataT != double && dataT != half`. Perform an in-place element-wise `OP` bitwise operation between each element of `lhs vec` and the `rhs` SYCL `vec` and return `lhs vec`.<br><br>Where `OP` is: `&=`, `\|=`, `^=`. |

Continued on next page

Table 4.108: Hidden friend functions of the `vec` class template.

| Hidden friend function | Description |
|---|---|
| `vec &operatorOP(vec &lhs, const dataT &rhs)` | Available only when: `dataT != float && dataT != double && dataT != half`. Perform an in-place element-wise `OP` bitwise operation between each element of `lhs` `vec` and the `rhs` scalar and return a `lhs` `vec`.<br><br>Where `OP` is: `&=`, `\|=`, `^=`. |
| `vec<RET, numElements> operatorOP(const vec &lhs, const vec &rhs)` | Construct a new instance of the SYCL `vec` class template with the same template parameters as `lhs` `vec` with each element of the new SYCL `vec` instance the result of an element-wise `OP` logical operation between each element of `lhs` `vec` and each element of the `rhs` SYCL `vec`.<br><br>The `dataT` template parameter of the constructed SYCL `vec`, `RET`, varies depending on the `dataT` template parameter of this SYCL `vec`. For a SYCL `vec` with `dataT` of type `int8_t` or `uint8_t` `RET` must be `int8_t`. For a SYCL `vec` with `dataT` of type `int16_t`, `uint16_t` or `half` `RET` must be `int16_t`. For a SYCL `vec` with `dataT` of type `int32_t`, `uint32_t` or `float` `RET` must be `int32_t`. For a SYCL `vec` with `dataT` of type `int64_t`, `uint64_t` or `double` `RET` must be `int64_t`.<br><br>Where `OP` is: `&&`, `\|\|`. |

Continued on next page

Table 4.108: Hidden friend functions of the vec class template.

| Hidden friend function | Description |
|---|---|
| `vec<RET, numElements> operatorOP(const vec &lhs, const dataT &rhs)` | Construct a new instance of the SYCL `vec` class template with the same template parameters as this SYCL `vec` with each element of the new SYCL `vec` instance the result of an element-wise `OP` logical operation between each element of `lhs` `vec` and the `rhs` scalar.<br><br>The `dataT` template parameter of the constructed SYCL `vec`, `RET`, varies depending on the `dataT` template parameter of this SYCL `vec`. For a SYCL `vec` with `dataT` of type `int8_t` or `uint8_t` `RET` must be `int8_t`. For a SYCL `vec` with `dataT` of type `int16_t`, `uint16_t` or `half` `RET` must be `int16_t`. For a SYCL `vec` with `dataT` of type `int32_t`, `uint32_t` or `float` `RET` must be `int32_t`. For a SYCL `vec` with `dataT` of type `int64_t`, `uint64_t` or `double` `RET` must be `uint64_t`.<br><br>Where `OP` is: &&, `\|\|`. |
| `vec operatorOP(const vec &lhs, const vec &rhs)` | Available only when: `dataT != float && dataT != double && dataT != half`. Construct a new instance of the SYCL `vec` class template with the same template parameters as `lhs` `vec` with each element of the new SYCL `vec` instance the result of an element-wise `OP` bitshift operation between each element of `lhs` `vec` and each element of the `rhs` SYCL `vec`. If `OP` is >>, `dataT` is a signed type and `lhs` `vec` has a negative value any vacated bits viewed as an unsigned integer must be assigned the value `1`, otherwise any vacated bits viewed as an unsigned integer must be assigned the value `0`.<br><br>Where `OP` is: <<, >>. |

Continued on next page

Table 4.108: Hidden friend functions of the `vec` class template.

| Hidden friend function | Description |
|---|---|
| `vec operatorOP(const vec &lhs, const dataT &rhs)` | Available only when: `dataT != float && dataT != double && dataT != half`. Construct a new instance of the SYCL `vec` class template with the same template parameters as `lhs` `vec` with each element of the new SYCL `vec` instance the result of an element-wise `OP` bitshift operation between each element of `lhs` `vec` and the `rhs` scalar. If `OP` is >>, `dataT` is a signed type and `lhs` `vec` has a negative value any vacated bits viewed as an unsigned integer must be assigned the value `1`, otherwise any vacated bits viewed as an unsigned integer must be assigned the value `0`.<br><br>Where `OP` is: <<, >>. |
| `vec &operatorOP(vec &lhs, const vec &rhs)` | Available only when: `dataT != float && dataT != double && dataT != half`. Perform an in-place element-wise `OP` bitshift operation between each element of `lhs` `vec` and the `rhs` SYCL `vec` and returns `lhs` `vec`. If `OP` is >>=, `dataT` is a signed type and `lhs` `vec` has a negative value any vacated bits viewed as an unsigned integer must be assigned the value `1`, otherwise any vacated bits viewed as an unsigned integer must be assigned the value `0`.<br><br>Where `OP` is: <<=, >>=. |
| `vec &operatorOP(vec &lhs, const dataT &rhs)` | Available only when: `dataT != float && dataT != double && dataT != half`. Perform an in-place element-wise `OP` bit-shift operation between each element of `lhs` `vec` and the `rhs` scalar and returns a reference to this SYCL `vec`. If `OP` is >>=, `dataT` is a signed type and `lhs` `vec` has a negative value any vacated bits viewed as an unsigned integer must be assigned the value `1`, otherwise any vacated bits viewed as an unsigned integer must be assigned the value `0`.<br><br>Where `OP` is: <<=, >>=. |

Continued on next page

Table 4.108: Hidden friend functions of the vec class template.

| Hidden friend function | Description |
|---|---|
| `vec<RET, numElements> operatorOP(const vec& lhs, const vec &rhs)` | Construct a new instance of the SYCL `vec` class template with the element type `RET` with each element of the new SYCL `vec` instance the result of an element-wise `OP` relational operation between each element of `lhs` `vec` and each element of the `rhs` SYCL `vec`. Each element of the SYCL `vec` that is returned must be `-1` if the operation results in `true` and `0` if the operation results in `false` or either this SYCL `vec` or the `rhs` SYCL `vec` is a NaN.<br><br>The `dataT` template parameter of the constructed SYCL `vec`, `RET`, varies depending on the `dataT` template parameter of this SYCL `vec`. For a SYCL `vec` with `dataT` of type `int8_t` or `uint8_t` `RET` must be `int8_t`. For a SYCL `vec` with `dataT` of type `int16_t`, `uint16_t` or `half` `RET` must be `int16_t`. For a SYCL `vec` with `dataT` of type `int32_t`, `uint32_t` or `float` `RET` must be `int32_t`. For a SYCL `vec` with `dataT` of type `int64_t`, `uint64_t` or `double` `RET` must be `uint64_t`.<br><br>Where `OP` is: ==, !=, <, >, <=, >=. |
| | Continued on next page |

Table 4.108: Hidden friend functions of the `vec` class template.

| Hidden friend function | Description |
|---|---|
| `vec<RET, numElements> operatorOP(const vec &lhs, const dataT &rhs)` | Construct a new instance of the SYCL `vec` class template with the `dataT` parameter of `RET` with each element of the new SYCL `vec` instance the result of an element-wise `OP` relational operation between each element of `lhs` `vec` and the `rhs` scalar. Each element of the SYCL `vec` that is returned must be `-1` if the operation results in `true` and `0` if the operation results in `false` or either `lhs` `vec` or the `rhs` SYCL `vec` is a NaN.<br><br>The `dataT` template parameter of the constructed SYCL `vec`, `RET`, varies depending on the `dataT` template parameter of this SYCL `vec`. For a SYCL `vec` with `dataT` of type `int8_t` or `uint8_t` `RET` must be `int8_t`. For a SYCL `vec` with `dataT` of type `int16_t`, `uint16_t` or `half` `RET` must be `int16_t`. For a SYCL `vec` with `dataT` of type `int32_t`, `uint32_t` or `float` `RET` must be `int32_t`. For a SYCL `vec` with `dataT` of type `int64_t`, `uint64_t` or `double` `RET` must be `uint64_t`.<br><br>Where `OP` is: ==, !=, <, >, <=, >=. |
| `vec operatorOP(const dataT &lhs, const vec &rhs)` | When `OP` is `%` available only when: `dataT != float && dataT != double && dataT != half`.<br>Construct a new instance of the SYCL `vec` class template with the same template parameters as the `rhs` SYCL `vec` with each element of the new SYCL `vec` instance the result of an element-wise `OP` arithmetic operation between the `lhs` scalar and each element of the `rhs` SYCL `vec`.<br><br>Where `OP` is: `+`, `-`, `*`, `/`, `%`. |
| `vec operatorOP(const dataT &lhs, const vec &rhs)` | Available only when: `dataT != float && dataT != double && dataT != half`.<br>Construct a new instance of the SYCL `vec` class template with the same template parameters as the `rhs` SYCL `vec` with each element of the new SYCL `vec` instance the result of an element-wise `OP` bitwise operation between the `lhs` scalar and each element of the `rhs` SYCL `vec`.<br><br>Where `OP` is: `&`, `\|`, `^`. |

Continued on next page

Table 4.108: Hidden friend functions of the vec class template.

| Hidden friend function | Description |
|---|---|
| `vec<RET, numElements> operatorOP(const dataT &lhs, const vec &rhs)` | Available only when: `dataT != float && dataT != double && dataT != half`. Construct a new instance of the SYCL `vec` class template with the same template parameters as the `rhs` SYCL `vec` with each element of the new SYCL `vec` instance the result of an element-wise `OP` logical operation between the `lhs` scalar and each element of the `rhs` SYCL `vec`.<br><br>The `dataT` template parameter of the constructed SYCL `vec`, `RET`, varies depending on the `dataT` template parameter of this SYCL `vec`. For a SYCL `vec` with `dataT` of type `int8_t` or `uint8_t` `RET` must be `int8_t`. For a SYCL `vec` with `dataT` of type `int16_t`, `uint16_t` or `half` `RET` must be `int16_t`. For a SYCL `vec` with `dataT` of type `int32_t`, `uint32_t` or `float` `RET` must be `int32_t`. For a SYCL `vec` with `dataT` of type `int64_t`, `uint64_t` or `double` `RET` must be `int64_t`.<br><br>Where `OP` is: `&&`, `\|\|`. |
| `vec operatorOP(const dataT &lhs, const vec &rhs)` | Construct a new instance of the SYCL `vec` class template with the same template parameters as the `rhs` SYCL `vec` with each element of the new SYCL `vec` instance the result of an element-wise `OP` bitshift operation between the `lhs` scalar and each element of the `rhs` SYCL `vec`. If `OP` is `>>`, `dataT` is a signed type and this SYCL `vec` has a negative value any vacated bits viewed as an unsigned integer must be assigned the value `1`, otherwise any vacated bits viewed as an unsigned integer must be assigned the value `0`.<br><br>Where `OP` is: `<<`, `>>`. |

Continued on next page

Table 4.108: Hidden friend functions of the `vec` class template.

| Hidden friend function | Description |
|---|---|
| `vec<RET, numElements> operatorOP(const dataT &lhs, const vec &rhs)` | Available only when: `dataT != float && dataT != double && dataT != half`. Construct a new instance of the SYCL `vec` class template with the element type `RET` with each element of the new SYCL `vec` instance the result of an element-wise `OP` relational operation between the `lhs` scalar and each element of the `rhs` SYCL `vec`. Each element of the SYCL `vec` that is returned must be `-1` if the operation results in `true` and `0` if the operation results in `false` or either this SYCL `vec` or the `rhs` SYCL `vec` is a NaN.<br><br>The `dataT` template parameter of the constructed SYCL `vec`, `RET`, varies depending on the `dataT` template parameter of this SYCL `vec`. For a SYCL `vec` with `dataT` of type `int8_t` or `uint8_t` `RET` must be `int8_t`. For a SYCL `vec` with `dataT` of type `int16_t`, `uint16_t` or `half` `RET` must be `int16_t`. For a SYCL `vec` with `dataT` of type `int32_t`, `uint32_t` or `float` `RET` must be `int32_t`. For a SYCL `vec` with `dataT` of type `int64_t`, `uint64_t` or `double` `RET` must be `int64_t`.<br><br>Where `OP` is: ==, !=, <, >, <=, >=. |
| `vec &operator~(const vec &v)` | Available only when: `dataT != float && dataT != double && dataT != half`. Construct a new instance of the SYCL `vec` class template with the same template parameters as `v` `vec` with each element of the new SYCL `vec` instance the result of an element-wise `OP` bitwise operation on each element of `v` `vec`. |

Continued on next page

Table 4.108: Hidden friend functions of the `vec` class template.

| Hidden friend function | Description |
|---|---|
| `vec<RET, numElements> operator!(const vec &v)` | Construct a new instance of the SYCL `vec` class template with the same template parameters as `v` `vec` with each element of the new SYCL `vec` instance the result of an element-wise `OP` logical operation on each element of `v` `vec`. Each element of the SYCL `vec` that is returned must be `-1` if the operation results in `true` and `0` if the operation results in `false` or this SYCL `vec` is a NaN.<br><br>The `dataT` template parameter of the constructed SYCL `vec`, `RET`, varies depending on the `dataT` template parameter of this SYCL `vec`. For a SYCL `vec` with `dataT` of type `int8_t` or `uint8_t` `RET` must be `int8_t`. For a SYCL `vec` with `dataT` of type `int16_t`, `uint16_t` or `half` `RET` must be `int16_t`. For a SYCL `vec` with `dataT` of type `int32_t`, `uint32_t` or `float` `RET` must be `int32_t`. For a SYCL `vec` with `dataT` of type `int64_t`, `uint64_t` or `double` `RET` must be `int64_t`. |
| | End of table |

Table 4.108: Hidden friend functions of the `vec` class template.

#### 4.16.2.2 Aliases

The SYCL programming API provides all permutations of the type alias:

```
using <type><elems> = vec<<storage-type>, <elems>>
```

where `<elems>` is `2`, `3`, `4`, `8` and `16`, and pairings of `<type>` and `<storage-type>` for integral types are `char` and `int8_t`, `uchar` and `uint8_t`, `short` and `int16_t`, `ushort` and `uint16_t`, `int` and `int32_t`, `uint` and `uint32_t`, `long` and `int64_t`, `ulong` and `uint64_t`, and for floating point types are both `half`, `float` and `double`.

For example `uint4` is the alias to `vec<uint64_t, 4>` and `float16` is the alias to `vec<float, 16>`.

#### 4.16.2.3 Swizzles

Swizzle operations can be performed in two ways. Firstly by calling the `swizzle` member function template, which takes a variadic number of integer template arguments between `0` and `numElements-1`, specifying swizzle indexes. Secondly by calling one of the simple swizzle member functions defined in 4.107 as `XYZW_SWIZZLE` and `RGBA_SWIZZLE`. Note that the simple swizzle functions are only available for up to 4 element vectors and are only available when the macro `SYCL_SIMPLE_SWIZZLES` is defined before including `sycl.hpp`.

In both cases the return type is always an instance of `__swizzled_vec__`, an implementation defined temporary class representing a swizzle of the original SYCL `vec` instance. Both kinds of swizzle member functions must not perform the swizzle operation themselves, instead the swizzle operation must be performed by the returned instance of `__swizzled_vec__` when used within an expression, meaning if the returned `__swizzled_vec__` is never used in an expression no swizzle operation is performed.

Both the `swizzle` member function template and the simple swizzle member functions allow swizzle indexes to be repeated.

A series of static constexpr values are provided within the `elem` struct to allow specifying named swizzle indexes when calling the `swizzle` member function template.

#### 4.16.2.4 Swizzled `vec` class

The `__swizzled_vec__` class must define an unspecified temporary which provides the entire interface of the SYCL `vec` class template, including swizzled member functions, with the additions and alterations described below:

- The `__swizzled_vec__` class template must be readable as an r-value reference on the RHS of an expression. In this case the swizzle operation is performed on the RHS of the expression and then the result is applied to the LHS of the expression.
- The `__swizzled_vec__` class template must be assignable as an l-value reference on the LHS of an expression. In this case the RHS of the expression is applied to the original SYCL `vec` which the `__swizzled_vec__` represents via the swizzle operation. Note that a `__swizzled_vec__` that is used in an l-value expression may not contain any repeated element indexes. For example: `f4.xxxx()= fx.wzyx()` would not be valid.
- The `__swizzled_vec__` class template must be convertible to an instance of SYCL `vec` with the type `dataT` and number of elements specified by the swizzle member function, if `numElements > 1`, and must be convertible to an instance of type `dataT`, if `numElements == 1`.
- The `__swizzled_vec__` class template must be non-copyable, non-moveable, non-user constructible and may not be bound to a l-value or escape the expression it was constructed in. For example `auto x = f4.x()` would not be valid.
- The `__swizzled_vec__` class template should return `__swizzled_vec__ &` for each operator inhetired from the `vec` class template interface which would return `vec<dataT, numElements> &`.

#### 4.16.2.5 Rounding modes

The various rounding modes that can be used in the `as` member function template are described in Table 4.109.

| Rounding mode | Description |
|---|---|
| `automatic` | Default rounding mode for the SYCL `vec` class element type. `rtz` (round toward zero) for integer types and `rte` (round to nearest even) for floating-point types. |
| `rte` | Round to nearest even. |
| `rtz` | Round toward zero. |
| `rtp` | Round toward positive infinity. |
| `rtn` | Round toward negative infinity. |
| | End of table |

Table 4.109: Rounding modes for the SYCL `vec` class template.

#### 4.16.2.6 Memory layout and alignment

The elements of an instance of the SYCL vec class template are stored in memory sequentially and contiguously and are aligned to the size of the element type in bytes multiplied by the number of elements:

$$\texttt{sizeof(dataT)} \cdot \texttt{numElements} \tag{4.6}$$

The exception to this is when the number of element is three in which case the SYCL vec is aligned to the size of the element type in bytes multiplied by four:

$$\texttt{sizeof(dataT)} \cdot 4 \tag{4.7}$$

This is true for both host and device code in order to allow for instances of the vec class template to be passed to SYCL kernel functions.

#### 4.16.2.7 Considerations for endianness

As SYCL supports both big-endian and little-endian on OpenCL devices, users must take care to ensure kernel arguments are processed correctly. This is particularly true for SYCL vec arguments as the order in which a SYCL vec is loaded differs between big-endian and little-endian.

Users should consult vendor documentation for guidance on how to handle kernel arguments in these situations.

#### 4.16.2.8 Performance note

The usage of the subscript `operator[]` may not be efficient on some devices.

### 4.16.3 Math array types

SYCL provides a `marray<typename` dataT, `std::size_t` numElements> class template to represent a contiguous fixed-size container. This type allows sharing of containers between the host and its SYCL devices.

The `marray` class is templated on its element type and number of elements. The number of elements parameter, `numElements`, is a positive value of the `std::size_t` type. The element type parameter, `dataT`, must be a `Numeric type` as it is defined by C++ standard.

An instance of the `marray` class template can also be implicitly converted to an instance of the data type when the number of elements is `1` in order to allow single element arrays and scalars to be convertible with each other.

Logical and comparison operators for `marray` class template return `marray<bool, numElements>`.

#### 4.16.3.1 Math array interface

The constructors, member functions and non-member functions of the SYCL `marray` class template are listed in Tables 4.110, 4.111 and 4.112 respectively.

```
namespace sycl {

template <typename dataT, std::size_t numElements>
class marray {
```

```
 public:
  using value_type = dataT;
  using reference = dataT&;
  using const_reference = const dataT&;
  using iterator = dataT*;
  using const_iterator = const dataT*;

  marray();

  explicit marray(const dataT &arg);

  template <typename... argTN>
  marray(const argTN&... args);

  marray(const marray<dataT, numElements> &rhs);
  marray(marray<dataT, numElements> &&rhs);

  // Available only when: numElements == 1
  operator dataT() const;

  static constexpr std::size_t size();

  // subscript operator
  reference operator[](std::size_t index);
  const_reference operator[](std::size_t index) const;

  marray &operator=(const marray<dataT, numElements> &rhs);
  marray &operator=(const dataT &rhs);
};

// iterator functions
iterator begin(marray &v);
const_iterator begin(const marray &v);

iterator end(marray &v);
const_iterator end(const marray &v);

// OP is: +, -, *, /, %
/* When OP is % available only when: dataT != float && dataT != double && dataT != half. */
marray operatorOP(const marray &lhs, const marray &rhs) { /* ... */ }
marray operatorOP(const marray &lhs, const dataT &rhs) { /* ... */ }

// OP is: +=, -=, *=, /=, %=
/* When OP is %= available only when: dataT != float && dataT != double && dataT != half. */
marray &operatorOP(marray &lhs, const marray &rhs) { /* ... */ }
marray &operatorOP(marray &lhs, const dataT &rhs) { /* ... */ }

// OP is: ++, --
marray &operatorOP(marray &lhs) { /* ... */ }
marray operatorOP(marray& lhs, int) { /* ... */ }

// OP is: +, -
marray operatorOP(marray &lhs) const { /* ... */ }

// OP is: &, |, ^
```

```
/* Available only when: dataT != float && dataT != double && dataT != half. */
marray operatorOP(const marray &lhs, const marray &rhs) { /* ... */ }
marray operatorOP(const marray &lhs, const dataT &rhs) { /* ... */ }

// OP is: &=, |=, ˆ=
/* Available only when: dataT != float && dataT != double && dataT != half. */
marray &operatorOP(marray &lhs, const marray &rhs) { /* ... */ }
marray &operatorOP(marray &lhs, const dataT &rhs) { /* ... */ }

// OP is: &&, ||
marray<bool, numElements> operatorOP(const marray &lhs, const marray &rhs) { /* ... */ }
marray<bool, numElements> operatorOP(const marray& lhs, const dataT &rhs) { /* ... */ }

// OP is: <<, >>
/* Available only when: dataT != float && dataT != double && dataT != half. */
marray operatorOP(const marray &lhs, const marray &rhs) { /* ... */ }
marray operatorOP(const marray &lhs, const dataT &rhs) { /* ... */ }

// OP is: <<=, >>=
/* Available only when: dataT != float && dataT != double && dataT != half. */
marray &operatorOP(marray &lhs, const marray &rhs) { /* ... */ }
marray &operatorOP(marray &lhs, const dataT &rhs) { /* ... */ }

// OP is: ==, !=, <, >, <=, >=
marray<bool, numElements> operatorOP(const marray &lhs, const marray &rhs) { /* ... */ }
marray<bool, numElements> operatorOP(const marray &lhs, const dataT &rhs) { /* ... */ }

/* Available only when: dataT != float && dataT != double && dataT != half. */
marray operator~(const marray &v) { /* ... */ }
marray<bool, numElements> operator!(const marray &v) { /* ... */ }

// OP is: +, -, *, /, %
/* operator% is only available when: dataT != float && dataT != double && dataT != half. */
marray operatorOP(const dataT &lhs, const marray &rhs) { /* ... */ }

// OP is: &, |, ˆ
/* Available only when: dataT != float && dataT != double
&& dataT != half. */
marray operatorOP(const dataT &lhs, const marray &rhs) { /* ... */ }

// OP is: &&, ||
marray<bool, numElements> operatorOP(const dataT &lhs, const marray &rhs) { /* ... */ }

// OP is: <<, >>
/* Available only when: dataT != float && dataT != double && dataT != half. */
marray operatorOP(const dataT &lhs, const marray &rhs) { /* ... */ }

// OP is: ==, !=, <, >, <=, >=
marray<bool, numElements> operatorOP(const dataT &lhs, const marray &rhs) { /* ... */ }

marray operator~(const marray &v) const { /* ... */ }

marray<bool, numElements> operator!(const marray &v) const { /* ... */ }

}  // namespace sycl
```

| Constructor | Description |
|---|---|
| `marray()` | Default construct an array with element type `dataT` and with `numElements` dimensions by default construction of each of its elements. |
| `explicit marray(const dataT &arg)` | Construct an array of element type `dataT` and `numElements` dimensions by setting each value to `arg` by assignment. |
| `template <typename... argTN> marray(const argTN&... args)` | Construct a SYCL `marray` instance from any combination of scalar and SYCL `marray` parameters of the same element type, providing the total number of elements for all parameters sum to `numElements` of this `marray` specialization. |
| `marray(const marray<dataT, numElements> &rhs)` | Construct an array of element type `dataT` and number of elements `numElements` by copy from another similar vector. |
| | End of table |

Table 4.110: Constructors of the SYCL `marray` class template.

| Member function | Description |
|---|---|
| `operator dataT()const` | Available only when: `numElements == 1`. Converts this SYCL `marray` instance to an instance of `dataT` with the value of the single element in this SYCL `marray` instance. The SYCL `marray` instance shall be implicitly convertible to the same data types, to which `dataT` is implicitly convertible. Note that conversion operator shall not be templated to allow standard conversion sequence for implicit conversion. |
| `static constexpr std::size_t size()` | Returns the size of this SYCL `marray` in bytes. 3-element vector size matches 4-element vector size to provide interoperability with OpenCL vector types. The same rule applies to vector alignment as described in 4.16.2.6. |
| `dataT &operator[](std::size_t index)` | Returns a reference to the element stored within this SYCL `marray` at the index specified by `index`. |
| `const dataT &operator[](std::size_t index)const` | Returns a const reference to the element stored within this SYCL `marray` at the index specified by `index`. |
| `marray &operator=(const marray &rhs)` | Assign each element of the `rhs` SYCL `marray` to each element of this SYCL `marray` and return a reference to this SYCL `marray`. |
| | Continued on next page |

Table 4.111: Member functions for the SYCL `marray` class template.

| Member function | Description |
|---|---|
| `marray &operator=(const dataT &rhs)` | Assign each element of the `rhs` scalar to each element of this SYCL `marray` and return a reference to this SYCL `marray`. |
| | End of table |

Table 4.111: Member functions for the SYCL `marray` class template.

| Non-member function | Description |
|---|---|
| `iterator begin(marray &v)` | Returns an iterator referring to the first element stored within the `v` `marray`. |
| `const_iterator begin(const marray &v)` | Returns a const iterator referring to the first element stored within the `v` `marray`. |
| `iterator end(marray &v)` | Returns an iterator referring to the one past the last element stored within the `v` `marray`. |
| `const_iterator end(const marray &v)` | Returns a const iterator referring to the one past the last element stored within the`v` `marray`. |
| `marray operatorOP(const marray &lhs, const marray & rhs)` | When `OP` is `%` available only when: `dataT != float && dataT != double && dataT != half`.<br>Construct a new instance of the SYCL `marray` class template with the same template parameters as `lhs` `marray` with each element of the new SYCL `marray` instance the result of an element-wise `OP` arithmetic operation between each element of `lhs` `marray` and each element of the `rhs` SYCL `marray`.<br><br>Where `OP` is: `+`, `-`, `*`, `/`, `%`. |
| `marray operatorOP(const marray &lhs, const dataT & rhs)` | When `OP` is `%` available only when: `dataT != float && dataT != double && dataT != half`.<br>Construct a new instance of the SYCL `marray` class template with the same template parameters as `lhs` `marray` with each element of the new SYCL `marray` instance the result of an element-wise `OP` arithmetic operation between each element of `lhs` `marray` and the `rhs` scalar.<br><br>Where `OP` is: `+`, `-`, `*`, `/`, `%`. |
| | Continued on next page |

Table 4.112: Non-member functions of the `marray` class template.

| Non-member function | Description |
|---|---|
| `marray &operatorOP(marray &lhs, const marray &rhs)` | When `OP` is `%=` available only when: `dataT != float && dataT != double && dataT != half`.<br>Perform an in-place element-wise `OP` arithmetic operation between each element of `lhs` `marray` and each element of the `rhs` SYCL `marray` and return `lhs` `marray`.<br><br>Where `OP` is: `+=`, `-=`, `*=`, `/=`, `%=`. |
| `marray &operatorOP(marray &lhs, const dataT &rhs)` | When `OP` is `%=` available only when: `dataT != float && dataT != double && dataT != half`.<br>Perform an in-place element-wise `OP` arithmetic operation between each element of `lhs` `marray` and `rhs` scalar and return `lhs` `marray`.<br><br>Where `OP` is: `+=`, `-=`, `*=`, `/=`, `%=`. |
| `marray &operatorOP(marray &v)` | Perform an in-place element-wise `OP` prefix arithmetic operation on each element of `lhs` `marray`, assigning the result of each element to the corresponding element of `lhs` `marray` and return `lhs` `marray`.<br><br>Where `OP` is: `++`, `--`. |
| `marray operatorOP(marray &v, int)` | Perform an in-place element-wise `OP` postfix arithmetic operation on each element of `lhs` `marray`, assigning the result of each element to the corresponding element of `lhs` `marray` and returns a copy of `lhs` `marray` before the operation is performed.<br><br>Where `OP` is: `++`, `--`. |
| `marray operatorOP(marray &v)` | Construct a new instance of the SYCL `marray` class template with the same template parameters as this SYCL `marray` with each element of the new SYCL `marray` instance the result of an element-wise `OP` unary arithmetic operation on each element of this SYCL `marray`.<br><br>Where `OP` is: `+`, `-`. |

Continued on next page

Table 4.112: Non-member functions of the `marray` class template.

| Non-member function | Description |
|---|---|
| `marray operatorOP(const marray &lhs, const marray &rhs)` | Available only when: `dataT != float && dataT != double && dataT != half`. Construct a new instance of the SYCL `marray` class template with the same template parameters as `lhs` `marray` with each element of the new SYCL `marray` instance the result of an element-wise `OP` bitwise operation between each element of `lhs` `marray` and each element of the `rhs` SYCL `marray`.<br><br>Where `OP` is: `&`, `\|`, `^`. |
| `marray operatorOP(const marray &lhs, const dataT &rhs)` | Available only when: `dataT != float && dataT != double && dataT != half`. Construct a new instance of the SYCL `marray` class template with the same template parameters as `lhs` `marray` with each element of the new SYCL `marray` instance the result of an element-wise `OP` bitwise operation between each element of `lhs` `marray` and the `rhs` scalar.<br><br>Where `OP` is: `&`, `\|`, `^`. |
| `marray &operatorOP(marray &lhs, const marray &rhs)` | Available only when: `dataT != float && dataT != double && dataT != half`. Perform an in-place element-wise `OP` bitwise operation between each element of `lhs` `marray` and the `rhs` SYCL `marray` and return `lhs` `marray`.<br><br>Where `OP` is: `&=`, `\|=`, `^=`. |
| `marray &operatorOP(marray &lhs, const dataT &rhs)` | Available only when: `dataT != float && dataT != double && dataT != half`. Perform an in-place element-wise `OP` bitwise operation between each element of `lhs` `marray` and the `rhs` scalar and return a `lhs` `marray`.<br><br>Where `OP` is: `&=`, `\|=`, `^=`. |
| `marray<bool, numElements> operatorOP(const marray &lhs, const marray &rhs)` | Construct a new instance of the `marray` class template with `dataT = bool` and same numElements as `lhs` `marray` with each element of the new `marray` instance the result of an element-wise `OP` logical operation between each element of `lhs` `marray` and each element of the `rhs` `marray`.<br><br>Where `OP` is: `&&`, `\|\|`. |

Continued on next page

Table 4.112: Non-member functions of the marray class template.

| Non-member function | Description |
|---|---|
| `marray<bool, numElements> operatorOP(const marray & lhs, const dataT &rhs)` | Construct a new instance of the `marray` class template with `dataT = bool` and same numElements as `lhs` `marray` with each element of the new `marray` instance the result of an element-wise `OP` logical operation between each element of `lhs` `marray` and the `rhs` scalar.<br><br>Where `OP` is: `&&`, `\|\|`. |
| `marray operatorOP(const marray &lhs, const marray & rhs)` | Available only when: `dataT != float && dataT != double && dataT != half`. Construct a new instance of the SYCL `marray` class template with the same template parameters as `lhs` `marray` with each element of the new SYCL `marray` instance the result of an element-wise `OP` bitshift operation between each element of `lhs` `marray` and each element of the `rhs` SYCL `marray`. If `OP` is `>>`, `dataT` is a signed type and `lhs` `marray` has a negative value any vacated bits viewed as an unsigned integer must be assigned the value `1`, otherwise any vacated bits viewed as an unsigned integer must be assigned the value `0`.<br><br>Where `OP` is: `<<`, `>>`. |
| `marray operatorOP(const marray &lhs, const dataT & rhs)` | Available only when: `dataT != float && dataT != double && dataT != half`. Construct a new instance of the SYCL `marray` class template with the same template parameters as `lhs` `marray` with each element of the new SYCL `marray` instance the result of an element-wise `OP` bitshift operation between each element of `lhs` `marray` and the `rhs` scalar. If `OP` is `>>`, `dataT` is a signed type and `lhs` `marray` has a negative value any vacated bits viewed as an unsigned integer must be assigned the value `1`, otherwise any vacated bits viewed as an unsigned integer must be assigned the value `0`.<br><br>Where `OP` is: `<<`, `>>`. |

Continued on next page

Table 4.112: Non-member functions of the `marray` class template.

| Non-member function | Description |
|---|---|
| `marray &operatorOP(marray &lhs, const marray &rhs)` | Available only when: `dataT != float && dataT != double && dataT != half`. Perform an in-place element-wise `OP` bit-shift operation between each element of `lhs marray` and the `rhs` SYCL `marray` and returns `lhs marray`. If `OP` is >>=, `dataT` is a signed type and `lhs marray` has a negative value any vacated bits viewed as an unsigned integer must be assigned the value `1`, otherwise any vacated bits viewed as an unsigned integer must be assigned the value `0`.<br><br>Where `OP` is: <<=, >>=. |
| `marray &operatorOP(marray &lhs, const dataT &rhs)` | Available only when: `dataT != float && dataT != double && dataT != half`. Perform an in-place element-wise `OP` bit-shift operation between each element of `lhs marray` and the `rhs` scalar and returns a reference to this SYCL `marray`. If `OP` is >>=, `dataT` is a signed type and `lhs marray` has a negative value any vacated bits viewed as an unsigned integer must be assigned the value `1`, otherwise any vacated bits viewed as an unsigned integer must be assigned the value `0`.<br><br>Where `OP` is: <<=, >>=. |
| `marray<bool, numElements> operatorOP(const marray& lhs, const marray &rhs)` | Construct a new instance of the `marray` class template with `dataT = bool` and same numElements as `lhs marray` with each element of the new `marray` instance the result of an element-wise `OP` relational operation between each element of `lhs marray` and each element of the `rhs marray`. Corresponding element of the `marray` that is returned must be `false` if the operation results is a NaN.<br><br>Where `OP` is: ==, !=, <, >, <=, >=. |

Continued on next page

Table 4.112: Non-member functions of the `marray` class template.

| Non-member function | Description |
|---|---|
| `marray<bool, numElements> operatorOP(const marray & lhs, const dataT &rhs)` | Construct a new instance of the `marray` class template with `dataT = bool` and same numElements as `lhs` `marray` with each element of the new `marray` instance the result of an element-wise `OP` relational operation between each element of `lhs` `marray` and the `rhs` scalar. Corresponding element of the `marray` that is returned must be `false` if the operation results is a NaN.<br><br>Where `OP` is: ==, !=, <, >, <=, >=. |
| `marray operatorOP(const dataT &lhs, const marray & rhs)` | When `OP` is `%` available only when: `dataT != float && dataT != double && dataT != half`.<br>Construct a new instance of the SYCL `marray` class template with the same template parameters as the `rhs` SYCL `marray` with each element of the new SYCL `marray` instance the result of an element-wise `OP` arithmetic operation between the `lhs` scalar and each element of the `rhs` SYCL `marray`.<br><br>Where `OP` is: `+`, `-`, `*`, `/`, `%`. |
| `marray operatorOP(const dataT &lhs, const marray & rhs)` | Available only when: `dataT != float && dataT != double && dataT != half`.<br>Construct a new instance of the SYCL `marray` class template with the same template parameters as the `rhs` SYCL `marray` with each element of the new SYCL `marray` instance the result of an element-wise `OP` bitwise operation between the `lhs` scalar and each element of the `rhs` SYCL `marray`.<br><br>Where `OP` is: `&`, `\|`, `^`. |
| `marray<RET, numElements> operatorOP(const dataT &lhs , const marray &rhs)` | Available only when: `dataT != float && dataT != double && dataT != half`.<br>Construct a new instance of the `marray` class template with `dataT = bool` and same numElements as `lhs` `marray` with each element of the new `marray` instance the result of an element-wise `OP` logical operation between the `lhs` scalar and each element of the `rhs` `marray`.<br><br>Where `OP` is: `&&`, `\|\|`. |

Continued on next page

Table 4.112: Non-member functions of the marray class template.

| Non-member function | Description |
|---|---|
| `marray operatorOP(const dataT &lhs, const marray &rhs)` | Construct a new instance of the SYCL `marray` class template with the same template parameters as the `rhs` SYCL `marray` with each element of the new SYCL `marray` instance the result of an element-wise `OP` bitshift operation between the `lhs` scalar and each element of the `rhs` SYCL `marray`. If `OP` is >>, `dataT` is a signed type and this SYCL `marray` has a negative value any vacated bits viewed as an unsigned integer must be assigned the value `1`, otherwise any vacated bits viewed as an unsigned integer must be assigned the value `0`.<br><br>Where `OP` is: <<, >>. |
| `marray<bool, numElements> operatorOP(const dataT &lhs, const marray &rhs)` | Available only when: `dataT != float && dataT != double && dataT != half`. Construct a new instance of the `marray` class template with `dataT = bool` and same numElements as `lhs` `marray` with each element of the new SYCL `marray` instance the result of an element-wise `OP` relational operation between the `lhs` scalar and each element of the `rhs` `marray`. Corresponding element of the `marray` that is returned must be `false` if the operation results is a NaN.<br><br>Where `OP` is: ==, !=, <, >, <=, >=. |
| `marray &operator~(const marray &v)` | Available only when: `dataT != float && dataT != double && dataT != half`. Construct a new instance of the SYCL `marray` class template with the same template parameters as `v` `marray` with each element of the new SYCL `marray` instance the result of an element-wise `OP` bitwise operation on each element of `v` `marray`. |

Continued on next page

Table 4.112: Non-member functions of the `marray` class template.

| Non-member function | Description |
|---|---|
| `marray<bool, numElements> operator!(const marray &v)` | Construct a new instance of the `marray` class template with `dataT = bool` and same numElements as `v` `marray` with each element of the new `marray` instance the result of an element-wise logical `!` operation on each element of `v` `marray`.<br><br>The `dataT` template parameter of the constructed SYCL `marray`, `RET`, varies depending on the `dataT` template parameter of this SYCL `marray`. For a SYCL `marray` with `dataT` of type `int8_t` or `uint8_t` `RET` must be `int8_t`. For a SYCL `marray` with `dataT` of type `int16_t`, `uint16_t` or `half` `RET` must be `int16_t`. For a SYCL `marray` with `dataT` of type `int32_t`, `uint32_t` or `float` `RET` must be `int32_t`. For a SYCL `marray` with `dataT` of type `int64_t`, `uint64_t` or `double` `RET` must be `int64_t`. |
| | End of table |

Table 4.112: Non-member functions of the `marray` class template.

#### 4.16.3.2 Aliases

The SYCL programming API provides all permutations of the type alias:

```
using m<type><elems> = marray<<storage-type>, <elems>>
```

where `<elems>` is `2`, `3`, `4`, `8` and `16`, and pairings of `<type>` and `<storage-type>` for integral types are `char` and `int8_t`, `uchar` and `uint8_t`, `short` and `int16_t`, `ushort` and `uint16_t`, `int` and `int32_t`, `uint` and `uint32_t`, `long` and `int64_t`, `ulong` and `uint64_t`, for floating point types are both `half`, `float` and `double`, and for boolean type `bool`.

For example `muint4` is the alias to `marray<uint64_t, 4>` and `mfloat16` is the alias to `marray<float, 16>`.

#### 4.16.3.3 Memory layout and alignment

The elements of an instance of the `marray` class template as if stored in `std::array<dataT, numElements>`.

## 4.17 Synchronization and atomics

The available features are:

- Accessor classes: Accessor classes specify acquisition and release of buffer and image data structures to provide points at which underlying queue synchronization primitives must be generated.
- Atomic operations: SYCL devices support a restricted subset of C++ atomics and SYCL uses the library syntax from the next C++ specification to make this available.

- Fences: Fence primitives are made available to order loads and stores. They are exposed through the `atomic_fence` function. Fences can have acquire semantics, release semantics or both.

- Barriers: Barrier primitives are made available to synchronize sets of work-items within individual groups. They are exposed through the `group_barrier` function.

- Hierarchical parallel dispatch: In the hierarchical parallelism model of describing computations, synchronization within the work-group is made explicit through multiple instances of the `parallel_for_work_item` function call, rather than through the use of explicit work-group barrier operations.

## 4.17.1 Barriers and fences

A group barrier or mem-fence may provide ordering semantics over the local address space, global address space or both. All memory operations initiated before the group barrier or mem-fence operation in the specified address space(s) will be completed before any memory operation after the operation.

```
namespace sycl {

void atomic_fence(memory_order order, memory_scope scope);

}  // namespace sycl
```

The effects of a call to `atomic_fence` depend on the value of the `order` parameter:

- `memory_order::relaxed`: No effect
- `memory_order::acquire`: Acquire fence
- `memory_order::release`: Release fence
- `memory_order::acq_rel`: Both an acquire fence and a release fence
- `memory_order::seq_cst`: A sequentially consistent acquire and release fence

A group barrier acts as both an acquire fence and a release fence: all work-items in the group execute a release fence prior to synchronizing at the barrier, and all work-items in the group execute an acquire fence afterwards.

## 4.17.2 Atomic references

The SYCL specification provides atomic operations based on the library syntax from the next C++ specification. The set of supported orderings is specific to a device, but every device is guaranteed to support at least `memory_order::relaxed`. Since different devices have different capabilities, there is no default ordering in SYCL and the default order used by each instance of `sycl::atomic_ref` is set by a template argument. If the default order is set to `memory_order::relaxed`, all memory order arguments default to `memory_order::relaxed`. If the default order is set to `memory_order::acq_rel`, memory order arguments default to `memory_order::acquire` for load operations, `memory_order::release` for store operations and `memory_order::acq_rel` for read-modify-write operations. If the default order is set to `memory_order::seq_cst`, all memory order arguments default to `memory_order::seq_cst`.

The SYCL atomic library may map directly to the underlying C++ library in SYCL application code, and must interact safely with the host C++ atomic library when used in host code. The SYCL library must be used in device code to ensure that only the limited subset of functionality is available. SYCL device compilers should give a compilation error on use of the std::atomic and std::atomic_ref classes and functions in device code.

The template parameter addressSpace is permitted to be access::address_space::generic_space, access::address_space::global_space or access::address_space::local_space.

The data type T is permitted to be int, unsigned int, long, unsigned long, long long, unsigned long long , float or double. For floating-point types, the member functions of the atomic_ref class may be emulated, and may use a different floating-point environment to those defined by info::device::single_fp_config and info::device::double_fp_config (i.e. floating-point atomics may use different rounding modes and may have different exception behavior).

As detailed in Tables 4.114, 4.115, 4.116 and 4.117, not all devices support atomic operations for 64-bit data types. The member functions of the atomic_ref class are required to compile even if the device does not support 64-bit data types, however they are only guaranteed to execute if the device has that support. If a member function is called with a 64-bit data type and the device does not have the necessary support, the SYCL runtime must throw an exception with the errc::feature_not_supported error code.

The atomic types are defined as follows.

```
namespace sycl {
enum class memory_order : /* unspecified */ {
  relaxed, acquire, release, acq_rel, seq_cst
};
inline constexpr memory_order memory_order_relaxed = memory_order::relaxed;
inline constexpr memory_order memory_order_acquire = memory_order::acquire;
inline constexpr memory_order memory_order_release = memory_order::release;
inline constexpr memory_order memory_order_acq_rel = memory_order::acq_rel;
inline constexpr memory_order memory_order_seq_cst = memory_order::seq_cst;

enum class memory_scope : /* unspecified */ {
  work_item, sub_group, work_group, device, system
};
inline constexpr memory_scope memory_scope_work_item = memory_scope::work_item;
inline constexpr memory_scope memory_scope_sub_group = memory_scope::sub_group;
inline constexpr memory_scope memory_scope_work_group = memory_scope::work_group;
inline constexpr memory_scope memory_scope_device = memory_scope::device;
inline constexpr memory_scope memory_scope_system = memory_scope::system;

// Exposition only
template <memory_order ReadModifyWriteOrder>
struct memory_order_traits;

template <>
struct memory_order_traits<memory_order::relaxed> {
  static constexpr memory_order read_order = memory_order::relaxed;
  static constexpr memory_order write_order = memory_order::relaxed;
};

template <>
struct memory_order_traits<memory_order::acq_rel> {
  static constexpr memory_order read_order = memory_order::acquire;
  static constexpr memory_order write_order = memory_order::release;
};
```

```

template <>
struct memory_order_traits<memory_order::seq_cst> {
  static constexpr memory_order read_order = memory_order::seq_cst;
  static constexpr memory_order write_order = memory_order::seq_cst;
};

template <typename T, memory_order DefaultOrder, memory_scope DefaultScope, access::address_space Space = access::address_space::generic_space>
class atomic_ref {
 public:

  using value_type = T;
  static constexpr size_t required_alignment = /* implementation-defined */;
  static constexpr bool is_always_lock_free = /* implementation-defined */;
  static constexpr memory_order default_read_order = memory_order_traits<DefaultOrder>::read_order;
  static constexpr memory_order default_write_order = memory_order_traits<DefaultOrder>::write_order;
  static constexpr memory_order default_read_modify_write_order = DefaultOrder;
  static constexpr memory_scope default_scope = DefaultScope;

  bool is_lock_free() const noexcept;

  explicit atomic_ref(T&);
  atomic_ref(const atomic_ref&) noexcept;
  atomic_ref& operator=(const atomic_ref&) = delete;

  void store(T operand,
    memory_order order = default_write_order,
    memory_scope scope = default_scope) const noexcept;

  T operator=(T desired) const noexcept;

  T load(memory_order order = default_read_order,
    memory_scope scope = default_scope) const noexcept;

  operator T() const noexcept;

  T exchange(T operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

  bool compare_exchange_weak(T &expected, T desired,
    memory_order success,
    memory_order failure,
    memory_scope scope = default_scope) const noexcept;

  bool compare_exchange_weak(T &expected, T desired,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

  bool compare_exchange_strong(T &expected, T desired,
    memory_order success,
    memory_order failure,
```

```
    memory_scope scope = default_scope) const noexcept;

  bool compare_exchange_strong(T &expected, T desired,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;
};

// Partial specialization for integral types
template <memory_order DefaultOrder, memory_scope DefaultScope, access::address_space Space =
    access::address_space::generic_space>
class atomic_ref<Integral, DefaultOrder, DefaultScope, Space> {

  /* All other members from atomic_ref<T> are available */

  using difference_type = value_type;

  Integral fetch_add(Integral operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

  Integral fetch_sub(Integral operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

  Integral fetch_and(Integral operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

  Integral fetch_or(Integral operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

  Integral fetch_min(Integral operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

  Integral fetch_max(Integral operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

  Integral operator++(int) const noexcept;
  Integral operator--(int) const noexcept;
  Integral operator++() const noexcept;
  Integral operator--() const noexcept;
  Integral operator+=(Integral) const noexcept;
  Integral operator-=(Integral) const noexcept;
  Integral operator&=(Integral) const noexcept;
  Integral operator|=(Integral) const noexcept;
  Integral operator^=(Integral) const noexcept;

};

// Partial specialization for floating-point types
template <memory_order DefaultOrder, memory_scope DefaultScope, access::address_space Space =
    access::address_space::generic_space>
```

```
class atomic_ref<Floating, DefaultOrder, DefaultScope, Space> {

  /* All other members from atomic_ref<T> are available */

  using difference_type = value_type;

  Floating fetch_add(Floating operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

  Floating fetch_sub(Floating operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

  Floating fetch_min(Floating operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

  Floating fetch_max(Floating operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

  Floating operator++(int) const noexcept;
  Floating operator--(int) const noexcept;
  Floating operator++() const noexcept;
  Floating operator--() const noexcept;
  Floating operator+=(Floating) const noexcept;
  Floating operator-=(Floating) const noexcept;

};

// Partial specialization for pointers
template <typename T, memory_order DefaultOrder, memory_scope DefaultScope, access::address_space
    Space = access::address_space::generic_space>
class atomic_ref<T*, DefaultOrder, DefaultScope, Space> {

  using value_type = T*;
  using difference_type = ptrdiff_t;
  static constexpr size_t required_alignment = /* implementation-defined */;
  static constexpr bool is_always_lock_free = /* implementation-defined */;
  static constexpr memory_order default_read_order = memory_order_traits<DefaultOrder>::read_order
      ;
  static constexpr memory_order default_write_order = memory_order_traits<DefaultOrder>::
      write_order;
  static constexpr memory_order default_read_modify_write_order = DefaultOrder;
  static constexpr memory_scope default_scope = DefaultScope;

  bool is_lock_free() const noexcept;

  explicit atomic_ref(T*&);
  atomic_ref(const atomic_ref&) noexcept;
  atomic_ref& operator=(const atomic_ref&) = delete;

  void store(T* operand,
    memory_order order = default_write_order,
```

```
    memory_scope scope = default_scope) const noexcept;

  T* operator=(T* desired) const noexcept;

  T* load(memory_order order = default_read_order,
    memory_scope scope = default_scope) const noexcept;

  operator T*() const noexcept;

  T* exchange(T* operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

  bool compare_exchange_weak(T* &expected, T* desired,
    memory_order success,
    memory_order failure,
    memory_scope scope = default_scope) const noexcept;

  bool compare_exchange_weak(T* &expected, T* desired,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

  bool compare_exchange_strong(T* &expected, T* desired,
    memory_order success,
    memory_order failure,
    memory_scope scope = default_scope) const noexcept;

  bool compare_exchange_strong(T* &expected, T* desired,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

  T* fetch_add(difference_type,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

  T* fetch_sub(difference_type,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

  T* operator++(int) const noexcept;
  T* operator--(int) const noexcept;
  T* operator++() const noexcept;
  T* operator--() const noexcept;
  T* operator+=(difference_type) const noexcept;
  T* operator-=(difference_type) const noexcept;

};

} // namespace sycl
```

The constructors and member functions for instances of the SYCL `atomic_ref` class using any compatible type are listed in Tables 4.113 and 4.114 respectively. Additional member functions for integral, floating-point and pointer types are listed in Tables 4.115, 4.116 and 4.117 respectively.

The static member `required_alignment` describes the minimum required alignment in bytes of an object that can be referenced by an `atomic_ref<T>`, which must be at least `alignof(T)`.

The static member `is_always_lock_free` is true if all atomic operations for type `T` are always lock-free. A SYCL implementation is not guaranteed to support atomic operations that are not lock-free.

The static members `default_read_order`, `default_write_order` and `default_read_modify_write_order` reflect the default memory order values for each type of atomic operation, consistent with the `DefaultOrder` template.

The atomic operations and member functions behave as described in the C++ specification, barring the restrictions discussed above. Note that care must be taken when using atomics to implement synchronization routines due to the lack of forward progress guarantees between work-items in SYCL. No work-item may be dependent on another work-item to make progress if the code is to be portable.

| Constructor | Description |
|---|---|
| `atomic_ref(T& ref)` | Constructs an instance of SYCL `atomic_ref` which is associated with the reference `ref`. |
| | End of table |

Table 4.113: Constructors of the SYCL `atomic_ref` class template.

| Member function | Description |
|---|---|
| `bool is_lock_free()const` | Return `true` if the atomic operations provided by this `atomic_ref` are lock-free. |
| `void store(T operand,`<br>`    memory_order order = default_write_order,`<br>`    memory_scope scope = default_scope)const` | Atomically stores `operand` to the object referenced by this `atomic_ref`. The memory order of this atomic operation must be `memory_order::relaxed`, `memory_order::release` or `memory_order::seq_cst`. This function is only supported for 64-bit data types on devices that have `aspect::int64_base_atomics`. |
| `T operator=(T desired)const` | Equivalent to `store(desired)`. Returns `desired`. |
| `T load(`<br>`    memory_order order = default_read_order`<br>`    memory_odrer scope = default_scope)const` | Atomically loads the value of the object referenced by this `atomic_ref`. The memory order of this atomic operation must be `memory_order::relaxed`, `memory_order::acquire`, or `memory_order::seq_cst`. This function is only supported for 64-bit data types on devices that have `aspect::int64_base_atomics`. |
| `operator T()const` | Equivalent to `load()`. |
| `T exchange(T operand,`<br>`    memory_order order =`<br>`default_read_modify_write_order,`<br>`    memory_scope scope = default_scope)const` | Atomically replaces the value of the object referenced by this `atomic_ref` with value `operand` and returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have `aspect::int64_base_atomics`. |
| | Continued on next page |

Table 4.114: Member functions available on any object of type `atomic_ref<T>`.

| Member function | Description |
|---|---|
| `bool compare_exchange_weak(T &expected, T desired, memory_order success, memory_order failure, memory_scope scope = default_scope)const` | Atomically compares the value of the object referenced by this `atomic_ref` against the value of `expected`. If the values are equal attempts to replace the value of the referenced object with the value of `desired`, otherwise assigns the original value of the referenced object to `expected`.<br>Returns `true` if the comparison operation and replacement operation were successful. The `failure` memory order of this atomic operation must be `memory_order::relaxed`, `memory_order::acquire` or `memory_order::seq_cst`.<br>This function is only supported for 64-bit data types on devices that have `aspect::int64_base_atomics`. |
| `bool compare_exchange_weak(T &expected, T desired, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope)const` | Equivalent to `compare_exchange_weak(expected, desired, order, order, scope)`. |
| `bool compare_exchange_strong(T &expected, T desired, memory_order success, memory_order failure, memory_scope scope = default_scope)const` | Atomically compares the value of the object referenced by this `atomic_ref` against the value of `expected`. If the values are equal replaces the value of the referenced object with the value of `desired`, otherwise assigns the original value of the referenced object to `expected`.<br>Returns `true` if the comparison operation was successful. The `failure` memory order of this atomic operation must be `memory_order::relaxed`, `memory_order::acquire` or `memory_order::seq_cst`.<br>This function is only supported for 64-bit data types on devices that have `aspect::int64_base_atomics`. |
| `bool compare_exchange_strong(T &expected, T desired, memory_order order = default_read_modify_write_order)const` | Equivalent to `compare_exchange_strong(expected, desired, order, order, scope)`. |
| | End of table |

Table 4.114: Member functions available on any object of type `atomic_ref<T>`.

| Member function | Description |
|---|---|
| `T fetch_add(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope)const` | Atomically adds `operand` to the value of the object referenced by this `atomic_ref` and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have `aspect::int64_base_atomics`. |
| `T operator+=(T operand)const` | Equivalent to `fetch_add(operand)`. |
| `T operator++(int)const` | Equivalent to `fetch_add(1)`. |
| `T operator++()const` | Equivalent to `fetch_add(1)+ 1`. |
| `T fetch_sub(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope)const` | Atomically subtracts `operand` from the value of the object referenced by this `atomic_ref` and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have `aspect::int64_base_atomics`. |
| `T operator-=(T operand)const` | Equivalent to `fetch_sub(operand)`. |
| `T operator--(int)const` | Equivalent to `fetch_sub(1)`. |
| `T operator--()const` | Equivalent to `fetch_sub(1)- 1`. |
| `T fetch_and(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope)const` | Atomically performs a bitwise AND between `operand` and the value of the object referenced by this `atomic_ref`, and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have `aspect::int64_extended_atomics`. |
| `T operator&=(T operand)const` | Equivalent to `fetch_and(operand)`. |
| `T fetch_or(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope)const` | Atomically performs a bitwise OR between `operand` and the value of the object referenced by this `atomic_ref`, and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have `aspect::int64_extended_atomics`. |
| `T operator\|=(T operand)const` | Equivalent to `fetch_or(operand)`. |
| `T fetch_xor(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope)const` | Atomically performs a bitwise XOR between `operand` and the value of the object referenced by this `atomic_ref`, and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have `aspect::int64_extended_atomics`. |
| `T operator^=(T operand)const` | Equivalent to `fetch_xor(operand)`. |

Continued on next page

Table 4.115: Additional member functions available on an object of type `atomic_ref<T>` for integral `T`.

| Member function | Description |
|---|---|
| `T fetch_min(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope)const` | Atomically computes the minimum of operand and the value of the object referenced by this atomic_ref, and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have aspect::int64_extended_atomics. |
| `T fetch_max(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope)const` | Atomically computes the maximum of operand and the value of the object referenced by this atomic_ref, and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have aspect::int64_extended_atomics. |
| | End of table |

Table 4.115: Additional member functions available on an object of type atomic_ref<T> for integral T.

| Member function | Description |
|---|---|
| `T fetch_add(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope)const` | Atomically adds operand to the value of the object referenced by this atomic_ref and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have aspect::int64_base_atomics. |
| `T operator+=(T operand)const` | Equivalent to fetch_add(operand). |
| `T fetch_sub(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope)const` | Atomically subtracts operand from the value of the object referenced by this atomic_ref and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have aspect::int64_base_atomics. |
| `T operator-=(T operand)const` | Equivalent to fetch_sub(operand). |
| `T fetch_min(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope)const` | Atomically computes the minimum of operand and the value of the object referenced by this atomic_ref, and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have aspect::int64_extended_atomics. |
| | Continued on next page |

Table 4.116: Additional member functions available on an object of type atomic_ref<T> for floating-point T.

| Member function | Description |
|---|---|
| `T fetch_max(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope)const` | Atomically computes the maximum of operand and the value of the object referenced by this `atomic_ref`, and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have `aspect::int64_extended_atomics`. |
| | End of table |

Table 4.116: Additional member functions available on an object of type `atomic_ref<T>` for floating-point `T`.

| Member function | Description |
|---|---|
| `T* fetch_add(ptrdiff_t operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope)const` | Atomically adds operand to the value of the object referenced by this `atomic_ref` and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit pointers on devices that have `aspect::int64_base_atomics`. |
| `T* operator+=(ptrdiff_t operand)const` | Equivalent to `fetch_add(operand)`. |
| `T* operator++(int)const` | Equivalent to `fetch_add(1)`. |
| `T* operator++()const` | Equivalent to `fetch_add(1)+ 1`. |
| `T* fetch_sub(ptrdiff_t operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope)const` | Atomically subtracts operand from the value of the object referenced by this `atomic_ref` and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit pointers on devices that have `aspect::int64_base_atomics`. |
| `T* operator-=(ptrdiff_t operand)const` | Equivalent to `fetch_sub(operand)`. |
| `T* operator--(int)const` | Equivalent to `fetch_sub(1)`. |
| `T* operator--()const` | Equivalent to `fetch_sub(1)- 1`. |
| | End of table |

Table 4.117: Additional member functions available on an object of type `atomic_ref<T*>`.

### 4.17.3 Atomic types (deprecated)

The atomic types and operations on atomic types provided by SYCL 1.2.1 are deprecated in SYCL 2020, and will be removed in a future version of SYCL. The types and operations are made available in the `cl::sycl::` namespace for backwards compatibility.

The constructors and member functions for the `cl::sycl::atomic` class are listed in Tables 4.118 and 4.119 respectively.

```
namespace cl {
```

```
namespace sycl {
/* Deprecated in SYCL 2020 */
enum class memory_order : int {
  relaxed
};

/* Deprecated in SYCL 2020 */
template <typename T, access::address_space addressSpace =
  access::address_space::global_space>
class atomic {
 public:
  template <typename pointerT, access::decorated IsDecorated>
  atomic(multi_ptr<pointerT, addressSpace, IsDecorated> ptr);

  void store(T operand, memory_order memoryOrder =
    memory_order::relaxed);

  T load(memory_order memoryOrder = memory_order::relaxed) const;

  T exchange(T operand, memory_order memoryOrder =
   memory_order::relaxed);

  /* Available only when: T != float */
  bool compare_exchange_strong(T &expected, T desired,
    memory_order successMemoryOrder = memory_order::relaxed,
    memory_order failMemoryOrder = memory_order::relaxed);

  /* Available only when: T != float */
  T fetch_add(T operand, memory_order memoryOrder =
    memory_order::relaxed);

  /* Available only when: T != float */
  T fetch_sub(T operand, memory_order memoryOrder =
    memory_order::relaxed);

  /* Available only when: T != float */
  T fetch_and(T operand, memory_order memoryOrder =
    memory_order::relaxed);

  /* Available only when: T != float */
  T fetch_or(T operand, memory_order memoryOrder =
    memory_order::relaxed);

  /* Available only when: T != float */
  T fetch_xor(T operand, memory_order memoryOrder =
    memory_order::relaxed);

  /* Available only when: T != float */
  T fetch_min(T operand, memory_order memoryOrder =
    memory_order::relaxed);

  /* Available only when: T != float */
  T fetch_max(T operand, memory_order memoryOrder =
    memory_order::relaxed);
};
```

```
} // namespace sycl
} // namespace cl
```

The global functions are as follows and described in Table 4.120.

```
namespace cl {
namespace sycl {
/* Deprecated in SYCL 2020 */
template <typename T, access::address_space addressSpace>
void atomic_store(atomic<T, addressSpace> object, T operand, memory_order memoryOrder =
  memory_order::relaxed);

/* Deprecated in SYCL 2020 */
template <typename T, access::address_space addressSpace>
T atomic_load(atomic<T, addressSpace> object, memory_order memoryOrder =
  memory_order::relaxed);

/* Deprecated in SYCL 2020 */
template <typename T, access::address_space addressSpace>
T atomic_exchange(atomic<T, addressSpace> object, T operand, memory_order memoryOrder =
  memory_order::relaxed);

/* Deprecated in SYCL 2020 */
template <typename T, access::address_space addressSpace>
bool atomic_compare_exchange_strong(atomic<T, addressSpace> object, T &expected, T desired,
    memory_order successMemoryOrder = memory_order::relaxed,
    memory_order failMemoryOrder = memory_order::relaxed);

/* Deprecated in SYCL 2020 */
template <typename T, access::address_space addressSpace>
T atomic_fetch_add(atomic<T, addressSpace> object, T operand, memory_order memoryOrder =
    memory_order::relaxed);

/* Deprecated in SYCL 2020 */
template <typename T, access::address_space addressSpace>
T atomic_fetch_sub(atomic<T, addressSpace> object, T operand, memory_order memoryOrder =
  memory_order::relaxed);

/* Deprecated in SYCL 2020 */
template <typename T, access::address_space addressSpace>
T atomic_fetch_and(atomic<T, addressSpace> object, T operand, memory_order memoryOrder =
  memory_order::relaxed);

/* Deprecated in SYCL 2020 */
template <typename T, access::address_space addressSpace>
T atomic_fetch_or(atomic<T, addressSpace> object, T operand, memory_order memoryOrder =
  memory_order::relaxed);

/* Deprecated in SYCL 2020 */
template <typename T, access::address_space addressSpace>
T atomic_fetch_xor(atomic<T, addressSpace> object, T operand, memory_order memoryOrder =
  memory_order::relaxed);

/* Deprecated in SYCL 2020 */
```

```
template <typename T, access::address_space addressSpace>
T atomic_fetch_min(atomic<T, addressSpace> object, T operand, memory_order memoryOrder =
  memory_order::relaxed);

/* Deprecated in SYCL 2020 */
template <typename T, access::address_space addressSpace>
T atomic_fetch_max(atomic<T, addressSpace> object, T operand, memory_order memoryOrder =
  memory_order::relaxed);
} // namespace sycl
} // namespace cl
```

| Constructor | Description |
|---|---|
| `template <typename pointerT>` `atomic(multi_ptr<pointerT, addressSpace> ptr)` | Deprecated in SYCL 2020. Permitted data types for `pointerT` are any valid scalar data type which is the same size in bytes as `T`. Constructs an instance of SYCL `atomic` which is associated with the pointer `ptr`, converted to a pointer of data type `T`. |
| | End of table |

Table 4.118: Constructors of the `cl::sycl::atomic` class template.

| Member function | Description |
|---|---|
| `void store(T operand, memory_order memoryOrder = memory_order::relaxed)` | Deprecated in SYCL 2020. Atomically stores the value `operand` at the address of the `multi_ptr` associated with this SYCL `atomic`. The memory order of this atomic operation must be `memory_order::relaxed`. This function is only supported for 64-bit data types on devices that have `aspect::int64_base_atomics`. |
| `T load(memory_order memoryOrder = memory_order::relaxed)const` | Deprecated in SYCL 2020. Atomically loads the value at the address of the `multi_ptr` associated with this SYCL `atomic`. Returns the value at the address of the `multi_ptr` associated with this SYCL `atomic` before the call. The memory order of this atomic operation must be `memory_order::relaxed`. This function is only supported for 64-bit data types on devices that have `aspect::int64_base_atomics`. |
| | Continued on next page |

Table 4.119: Member functions available on an object of type `cl::sycl::atomic<T>`.

| Member function | Description |
|---|---|
| `T exchange(T operand, memory_order memoryOrder = memory_order::relaxed)` | Deprecated in SYCL 2020.<br>Atomically replaces the value at the address of the `multi_ptr` associated with this SYCL `atomic` with value `operand` and returns the value at the address of the `multi_ptr` associated with this SYCL `atomic` before the call. The memory order of this atomic operation must be `memory_order::relaxed`. This function is only supported for 64-bit data types on devices that have `aspect::int64_base_atomics`. |
| `bool compare_exchange_strong(T &expected, T desired, memory_order successMemoryOrder = memory_order::relaxed, memory_order failMemoryOrder = memory_order::relaxed)` | Deprecated in SYCL 2020.<br>Available only when: `T != float`.<br>Atomically compares the value at the address of the `multi_ptr` associated with this SYCL `atomic` against the value of `expected`. If the values are equal replaces value at address of the `multi_ptr` associated with this SYCL `atomic` with the value of `desired`, otherwise assigns the original value at the address of the `multi_ptr` associated with this SYCL `atomic` to `expected`. Returns `true` if the comparison operation was successful. The memory order of this atomic operation must be `memory_order::relaxed` for both success and fail. This function is only supported for 64-bit data types on devices that have `aspect::int64_base_atomics`. |
| `T fetch_add(T operand, memory_order memoryOrder = memory_order::relaxed)` | Deprecated in SYCL 2020.<br>Available only when: `T != float`.<br>Atomically adds the value `operand` to the value at the address of the `multi_ptr` associated with this SYCL `atomic` and assigns the result to the value at the address of the `multi_ptr` associated with this SYCL `atomic`. Returns the value at the address of the `multi_ptr` associated with this SYCL `atomic` before the call. The memory order of this atomic operation must be `memory_order::relaxed`. This function is only supported for 64-bit data types on devices that have `aspect::int64_base_atomics`. |

Continued on next page

Table 4.119: Member functions available on an object of type `cl::sycl::atomic<T>`.

| Member function | Description |
|---|---|
| T fetch_sub(T operand, memory_order memoryOrder = memory_order::relaxed) | Deprecated in SYCL 2020.<br>Available only when: T != float.<br>Atomically subtracts the value operand to the value at the address of the multi_ptr associated with this SYCL atomic and assigns the result to the value at the address of the multi_ptr associated with this SYCL atomic. Returns the value at the address of the multi_ptr associated with this SYCL atomic before the call. The memory order of this atomic operation must be memory_order::relaxed. This function is only supported for 64-bit data types on devices that have aspect::int64_base_atomics. |
| T fetch_and(T operand, memory_order memoryOrder = memory_order::relaxed) | Deprecated in SYCL 2020.<br>Available only when: T != float.<br>Atomically performs a bitwise AND between the value operand and the value at the address of the multi_ptr associated with this SYCL atomic and assigns the result to the value at the address of the multi_ptr associated with this SYCL atomic. Returns the value at the address of the multi_ptr associated with this SYCL atomic before the call. The memory order of this atomic operation must be memory_order::relaxed. This function is only supported for 64-bit data types on devices that have aspect::int64_extended_atomics. |
| T fetch_or(T operand, memory_order memoryOrder = memory_order::relaxed) | Deprecated in SYCL 2020.<br>Available only when: T != float.<br>Atomically performs a bitwise OR between the value operand and the value at the address of the multi_ptr associated with this SYCL atomic and assigns the result to the value at the address of the multi_ptr associated with this SYCL atomic. Returns the value at the address of the multi_ptr associated with this SYCL atomic before the call. The memory order of this atomic operation must be memory_order::relaxed. This function is only supported for 64-bit data types on devices that have aspect::int64_extended_atomics. |

Continued on next page

Table 4.119: Member functions available on an object of type cl::sycl::atomic<T>.

| Member function | Description |
|---|---|
| T fetch_xor(T operand, memory_order memoryOrder = memory_order::relaxed) | Deprecated in SYCL 2020.<br>Available only when: T != float.<br>Atomically performs a bitwise XOR between the value operand and the value at the address of the multi_ptr associated with this SYCL atomic and assigns the result to the value at the address of the multi_ptr associated with this SYCL atomic. Returns the value at the address of the multi_ptr associated with this SYCL atomic before the call. The memory order of this atomic operation must be memory_order::relaxed . This function is only supported for 64-bit data types on devices that have aspect ::int64_extended_atomics. |
| T fetch_min(T operand, memory_order memoryOrder = memory_order::relaxed) | Deprecated in SYCL 2020.<br>Atomically computes the minimum of the value operand and the value at the address of the multi_ptr associated with this SYCL atomic and assigns the result to the value at the address of the multi_ptr associated with this SYCL atomic. Returns the value at the address of the multi_ptr associated with this SYCL atomic before the call. The memory order of this atomic operation must be memory_order::relaxed . This function is only supported for 64-bit data types on devices that have aspect ::int64_extended_atomics. |
| T fetch_max(T operand, memory_order memoryOrder = memory_order::relaxed) | Deprecated in SYCL 2020.<br>Available only when: T != float.<br>Atomically computes the maximum of the value operand and the value at the address of the multi_ptr associated with this SYCL atomic and assigns the result to the value at the address of the multi_ptr associated with this SYCL atomic. Returns the value at the address of the multi_ptr associated with this SYCL atomic before the call. The memory order of this atomic operation must be memory_order::relaxed . This function is only supported for 64-bit data types on devices that have aspect ::int64_extended_atomics. |
| | End of table |

Table 4.119: Member functions available on an object of type cl::sycl::atomic<T>.

<table>
<tr><th>Functions</th><th>Description</th></tr>
<tr><td><pre>template <typename T, access::address_space
addressSpace>
T atomic_load(atomic<T, addressSpace> object,
    memory_order memoryOrder = memory_order::relaxed
)</pre></td><td>Deprecated in SYCL 2020.<br>Equivalent to calling <code>object.load( memoryOrder)</code>.</td></tr>
<tr><td><pre>template <typename T, access::address_space
addressSpace>
void atomic_store(atomic<T, addressSpace> object, T
operand,
    memory_order memoryOrder = memory_order::relaxed
)</pre></td><td>Deprecated in SYCL 2020.<br>Equivalent to calling <code>object.store( operand, memoryOrder)</code>.</td></tr>
<tr><td><pre>template <typename T, access::address_space
addressSpace>
T atomic_exchange(atomic<T, addressSpace> object, T
operand,
    memory_order memoryOrder = memory_order::relaxed
)</pre></td><td>Deprecated in SYCL 2020.<br>Equivalent to calling <code>object.exchange( operand, memoryOrder)</code>.</td></tr>
<tr><td><pre>template <typename T, access::address_space
addressSpace>
bool atomic_compare_exchange_strong(
    atomic<T, addressSpace> object, T &expected, T
desired,
     memory_order successMemoryOrder =
     memory_order::relaxed
     memory_order failMemoryOrder =
     memory_order::relaxed)</pre></td><td>Deprecated in SYCL 2020.<br>Equivalent to calling <code>object. compare_exchange_strong(expected , desired, successMemoryOrder, failMemoryOrders)</code>.</td></tr>
<tr><td><pre>template <typename T, access::address_space
addressSpace>
T atomic_fetch_add(atomic<T, addressSpace> object, T
 operand,
    memory_order memoryOrder = memory_order::relaxed
)</pre></td><td>Deprecated in SYCL 2020.<br>Equivalent to calling <code>object.fetch_add( operand, memoryOrder)</code>.</td></tr>
<tr><td><pre>template <typename T, access::address_space
addressSpace>
T atomic_fetch_sub(atomic<T, addressSpace> object, T
 operand,
    memory_order memoryOrder = memory_order::relaxed
)</pre></td><td>Deprecated in SYCL 2020.<br>Equivalent to calling <code>object.fetch_sub( operand, memoryOrder)</code>.</td></tr>
<tr><td><pre>template <typename T, access::address_space
addressSpace>
T atomic_fetch_and(atomic<T> operand, T object,
    memory_order memoryOrder = memory_order::relaxed
)</pre></td><td>Deprecated in SYCL 2020.<br>Equivalent to calling <code>object.fetch_add( operand, memoryOrder)</code>.</td></tr>
<tr><td colspan="2" align="right">Continued on next page</td></tr>
</table>

Table 4.120: Global functions available on atomic types.

| Functions | Description |
|---|---|
| `template <typename T, access::address_space addressSpace>`<br>`T atomic_fetch_or(atomic<T, addressSpace> object, T operand,`<br>`    memory_order memoryOrder = memory_order::relaxed`<br>`)` | Deprecated in SYCL 2020.<br>Equivalent to calling `object.fetch_or(operand, memoryOrder)`. |
| `template <typename T, access::address_space addressSpace>`<br>`T atomic_fetch_xor(atomic<T, addressSpace> object, T operand,`<br>`    memory_order memoryOrder = memory_order::relaxed`<br>`)` | Deprecated in SYCL 2020.<br>Equivalent to calling `object.fetch_xor(operand, memoryOrder)`. |
| `template <typename T, access::address_space addressSpace>`<br>`T atomic_fetch_min(atomic<T, addressSpace> object, T operand,`<br>`    memory_order memoryOrder = memory_order::relaxed`<br>`)` | Deprecated in SYCL 2020.<br>Equivalent to calling `object.fetch_min(operand, memoryOrder)`. |
| `template <typename T, access::address_space addressSpace>`<br>`T atomic_fetch_max(atomic<T, addressSpace> object, T operand,`<br>`    memory_order memoryOrder = memory_order::relaxed`<br>`)` | Deprecated in SYCL 2020.<br>Equivalent to calling `object.fetch_max(operand, memoryOrder)`. |
| | End of table |

Table 4.120: Global functions available on atomic types.

## 4.18 Stream class

The SYCL `stream` class is a buffered output stream that allows outputting the values of built-in, vector and SYCL types to the console. The implementation of how values are streamed to the console is left as an implementation detail.

The way in which values are output by an instance of the SYCL `stream` class can also be altered using a range of manipulators.

An instance of the SYCL `stream` class has a maximum buffer size (`totalBufferSize`) that specifies maximum size of the overall character stream that can be output in characters (a character is of size `sizeof(char)` bytes) during a kernel invocation, and a maximum stream size (`workItemBufferSize`) that specifies the maximum size of the character stream (number of characters) that can be output within a work item before a flush must be performed (a character is of size `sizeof(char)` bytes). `totalBufferSize` must be sufficient to contain the characters output by all stream statements during execution of a kernel invocation (the aggregate of outputs from all work items), and `workItemBufferSize` must be sufficient to contain the characters output within a work item between stream flush operations.

A stream flush operation is defined that synchronizes the work item stream buffer with the global stream buffer, and then empties the work item stream buffer. A flush can be explicitly triggered using the `flush` stream manipulator, or implicitly triggered by the `endl` stream manipulator or by kernel completion (from the perspective of each completing work item).

If the `totalBufferSize` or `workItemBufferSize` limits are exceeded, it is implementation defined whether the streamed characters exceeding the limit are output, or silently ignored/discarded, and if output it is implementation defined whether those extra characters exceeding the `workItemBufferSize` limit count toward the `totalBufferSize` limit. Regardless of this implementations defined behavior of output exceeding the limits, no undefined or erroneous behavior is permitted of an implementation when the limits are exceeded. Unused characters within `workItemBufferSize` (any portion of the `workItemBufferSize` capacity that has not been used at the time of a stream flush) do not count toward the `totalBufferSize` limit, in that only characters flushed count toward the `totalBufferSize` limit.

The SYCL `stream` class provides the common reference semantics (see Section 4.5.3).

### 4.18.1 Stream class interface

The constructors and member functions of the SYCL `stream` class are listed in Tables 4.123, 4.124, and 4.125 respectively. The additional common special member functions and common member functions are listed in Tables 4.1 and 4.2, respectively.

The operand types that are supported by the SYCL `stream` class `operator<<()` operator are listed in Table 4.121.

The manipulators that are supported by the SYCL `stream` class `operator<<()` operator are listed in Table 4.122.

```
namespace sycl {

enum class stream_manipulator {
  flush,
  dec,
  hex,
  oct,
  noshowbase,
  showbase,
  noshowpos,
  showpos,
  endl,
  fixed,
  scientific,
  hexfloat,
  defaultfloat
};


const stream_manipulator flush = stream_manipulator::flush;

const stream_manipulator dec = stream_manipulator::dec;

const stream_manipulator hex = stream_manipulator::hex;

const stream_manipulator oct = stream_manipulator::oct;

const stream_manipulator noshowbase = stream_manipulator::noshowbase;

const stream_manipulator showbase = stream_manipulator::showbase;
```

```

const stream_manipulator noshowpos = stream_manipulator::noshowpos;

const stream_manipulator showpos = stream_manipulator::showpos;

const stream_manipulator endl = stream_manipulator::endl;

const stream_manipulator fixed = stream_manipulator::fixed;

const stream_manipulator scientific = stream_manipulator::scientific;

const stream_manipulator hexfloat = stream_manipulator::hexfloat;

const stream_manipulator defaultfloat = stream_manipulator::defaultfloat;

__precision_manipulator__ setprecision(int precision);

__width_manipulator__ setw(int width);

class stream {
 public:

  stream(size_t totalBufferSize, size_t workItemBufferSize, handler& cgh);

  /* -- common interface members -- */

  size_t get_size() const;

  size_t get_work_item_buffer_size() const;

  /* get_max_statement_size() has the same functionality as get_work_item_buffer_size(),
     and is provided for backward compatibility.  get_max_statement_size() is a deprecated
     query. */
  size_t get_max_statement_size() const;
};

template <typename T>
const stream& operator<<(const stream& os, const T &rhs);

}  // namespace sycl
```

| Stream operand type | Description |
|---|---|
| `char`, `signed char`, `unsigned char`, `int`, `unsigned int`, `short`, `unsigned short`, `long int`, `unsigned long int`, `long long int`, `unsigned long long int`, `byte` | Outputs the value as a stream of characters. |
| `float`, `double`, `half` | Outputs the value according to the precision of the current statement as a stream of characters. |
| `char *`, `const char *` | Outputs the string. |
| `T *`, `const T *`, `multi_ptr` | Outputs the address of the pointer as a stream of characters. |
| | Continued on next page |

Table 4.121: Operand types supported by the stream class.

| Stream operand type | Description |
|---|---|
| vec | Outputs the value of each component of the vector as a stream of characters. |
| id, range, item, nd_item, group, nd_range, h_item | Outputs the value of each component of each id or range as a stream of characters. |
| | End of table |

Table 4.121: Operand types supported by the stream class.

| Stream manipulator | Description |
|---|---|
| flush | Triggers a flush operation, which synchronizes the work item stream buffer with the global stream buffer, and then empties the work item stream buffer. After a flush, the full workItemBufferSize is available again for subsequent streaming within the work item. |
| endl | Outputs a new-line character and then triggers a flush operation. |
| dec | Outputs any subsequent values in the current statement in decimal base. |
| hex | Outputs any subsequent values in the current statement in hexadecimal base. |
| oct | Outputs any subsequent values in the current statement in octal base. |
| noshowbase | Outputs any subsequent values without the base prefix. |
| showbase | Outputs any subsequent values with the base prefix. |
| noshowpos | Outputs any subsequent values without a plus sign if the value is positive. |
| showpos | Outputs any subsequent values with a plus sign if the value is positive. |
| setw(int) | Sets the field width of any subsequent values in the current statement. |
| setprecision(int) | Sets the precision of any subsequent values in the current statement. |
| fixed | Outputs any subsequent floating-point values in the current statement in fixed notation. |
| scientific | Outputs any subsequent floating-point values in the current statement in scientific notation. |
| hexfloat | Outputs any subsequent floating-point values in the current statement in hexadecimal notation. |
| defaultfloat | Outputs any subsequent floating-point values in the current statement in the default notation. |
| | End of table |

Table 4.122: Manipulators supported by the stream class.

| Constructor | Description |
|---|---|
| `stream(size_t totalBufferSize, size_t workItemBufferSize, handler& cgh)` | Constructs a SYCL `stream` instance associated with the command group specified by `cgh`, with a maximum buffer size in characters per kernel invocation specified by the parameter `totalBufferSize`, and a maximum stream size that can be buffered by a work item between stream flushes specified by the parameter `workItemBufferSize`. `totalBufferSize` and `workItemBufferSize` relate to memory storage size in that they are the term `n` in `alloc_size = n*sizeof(char)`. |
| | End of table |

Table 4.123: Constructors of the `stream` class.

| Member function | Description |
|---|---|
| `size_t get_size()const` | Returns the total buffer size, in characters. |
| `size_t get_work_item_buffer_size()const` | Returns the buffer size per work item, in characters. |
| `size_t get_max_statement_size()const` | Deprecated query with same functionality as `get_work_item_buffer_size()`. |
| | End of table |

Table 4.124: Member functions of the `stream` class.

| Global function | Description |
|---|---|
| `template <typename T> const stream& operator<<(const stream& os, const T &rhs)` | Outputs any valid values (see 4.121) as a stream of characters and applies any valid manipulator (see 4.122) to the current stream. |
| | End of table |

Table 4.125: Global functions of the `stream` class.

### 4.18.2 Synchronization

An instance of the SYCL `stream` class is required to synchronize with the host, and must output everything that is streamed to it via the `operator<<()` operator before a flush operation (that doesn't exceed the `workItemBufferSize` or `totalBufferSize` limits) within a SYCL kernel function by the time that the event associated with a command group submission enters the completed state. The point at which this synchronization occurs and the method by which this synchronization is performed are implementation defined. For example it is valid for an implementation to use `printf()`.

The SYCL `stream` class is required to output the content of each stream, between flushes (up to `workItemBufferSize`), without mixing with content from the same stream in other work items. There are no other output order guarantees between work items or between streams. The stream flush operation therefore delimits the unit of output that is guaranteed to be displayed without mixing with other work items, with respect to a single stream.

If an instance of the SYCL `stream` class is used on a SYCL kernel function executed on a SYCL context that is not from the host SYCL backend, there is no guarantee that statements are output in their entirety. If an instance of the SYCL `stream` class is used on a SYCL kernel function executed on a host context, then the SYCL `stream` class is required to output each statement in full without mixing with statements of other work items.

### 4.18.3 Implicit flush

There is guaranteed to be an implicit flush of each stream used by a kernel, at the end of kernel execution, from the perspective of each work item. There is also an implicit flush when the endl stream manipulator is executed. No other implicit flushes are permitted in an implementation.

### 4.18.4 Performance note

The usage of the `stream` class is designed for debugging purposes and is therefore not recommended for performance critical applications.

## 4.19 SYCL built-in functions for SYCL host and device

SYCL kernels may execute on any SYCL device, which requires the functions used in the kernels to be compiled and linked for both device and host. In the SYCL programming model, the built-ins are available for the entire SYCL Application within the `sycl` namespace, although their semantics may be different. This section follows the OpenCL 1.2 specification document [1, ch. 6.12] - except that for SYCL, all functions are located within the `sycl` namespace - and describes the behavior of these functions for SYCL host and device. The expected precision and any other semantic requirements are defined in the backend specification.

The SYCL built-in functions are available throughout the SYCL application, and depending on where they execute, they are either implemented using their host implementation or the device implementation. The SYCL system guarantees that all of the built-in functions fulfill the same requirements for both host and device.

### 4.19.1 Description of the built-in types available for SYCL host and device

All of the OpenCL built-in types are available in the namespace `sycl`. For the purposes of this document we use generic type names for describing sets of valid SYCL types. The generic type names themselves are not valid SYCL types, but they represent a set of valid types, as defined in Tables 4.126. Each generic type within a section is comprised of a combination of scalar, SYCL `vec` and/or `marray` class specializations class specializations. `n` and `N` define valid sizes for class specializations, where `n` means 2,3,4,8,16 and `N` means any positive value of size_t type. Note that any reference to the base type refers to the type of a scalar or the element type of a SYCL `vec` or `marray` specialization.

In the OpenCL 1.2 specification document [1, ch. 6.12.1] in Table 6.7 the work-item functions are defined where they provide the size of the enqueued kernel NDRange. These functions are available in SYCL through the item and group classes see sections 4.10.1.4, 4.10.1.5 and 4.10.1.7.

| Generic type name | Description |
|---|---|
| floatn | floatn, mfloatn, marray<N,float> |
| genfloatf | float, floatn |
| doublen | doublen, mdoublen, marray<N,double> |
| genfloatd | double, doublen |
| halfn | halfn, mhalfn, marray<N,half> |
| genfloath | half, halfn |
| genfloat | genfloatf, genfloatd, genfloath |
| sgenfloat | float, double, half |
| gengeofloat | float, float2, float3, float4, mfloat2, mfloat3, mfloat4 |
| gengeodouble | double, double2, double3, double4, mdouble2, mdouble3, mdouble4 |
| charn | charn, mcharn, marray<N,char> |
| scharn | scharn, mscharn, marray<N,signed char > |
| ucharn | ucharn, mucharn, marray<N,unsigned char> |
| igenchar | signed char, scharn |
| ugenchar | unsigned char, ucharn |
| genchar | char, charn, igenchar, ugenchar |
| shortn | shortn, mshortn, marray<N,short> |
| genshort | short, shortn |
| ushortn | ushortn, mushortn, marray<N,unsigned short> |
| ugenshort | unsigned short, ushortn |
| uintn | uintn, muintn, marray<N,unsigned int> |
| ugenint | unsigned int, uintn |
| intn | intn, mintn, marray<N,int> |
| genint | int, intn |
| ulongn | ulongn, mulongn, marray<N,unsigned long int> |
| ugenlong | unsigned long int, ulongn |
| longn | longn, mlongn, marray<N,long int> |
| genlong | long int, longn |
| ulonglongn | ulonglongn, mulonglongn, marray<N, unsigned long long int> |
| ugenlonglong | unsigned long long int, ulonglongn |
| longlongn | longlongn, mlonglongn, marray<N,long long int> |
| genlonglong | long long int, longlongn |
| igenlonginteger | genlong, genlonglong |
| ugenlonginteger | ugenlong, ugenlonglong |
| geninteger | genchar, genshort, ugenshort, genint, ugenint, igenlonginteger, ugenlonginteger |

Continued on next page

Table 4.126: Generic type name description, which serves as a description for all valid types of parameters to kernel functions [1].

| Generic type name | Description |
|---|---|
| genintegerNbit | All types within geninteger whose base type are N bits in size, where N = 8, 16, 32, 64. |
| igeninteger | igenchar, genshort, genint, igenlonginteger |
| igenintegerNbit | All types within igeninteger whose base type are N bits in size, where N = 8, 16, 32, 64. |
| ugeninteger | ugenchar, ugenshort, ugenint, ugenlonginteger |
| ugenintegerNbit | All types within ugeninteger whose base type are N bits in size, where N = 8, 16, 32, 64. |
| sgeninteger | char, signed char, unsigned char, short, unsigned short, int, unsigned int, long int, unsigned long int, long long int, unsigned long long int |
| gentype | genfloat, geninteger |
| genfloatptr | All permutations of multi_ptr<dataT, addressSpace, IsDecorated> where dataT is all types within genfloat, addressSpace is access::address_space::global_space, access::address_space::local_space and access::address_space::private_space and IsDecorated is access::decorated::yes and access::decorated::no. |
| genintptr | All permutations of multi_ptr<dataT, addressSpace, IsDecorated> where dataT is all types within genint, addressSpace is access::address_space::global_space, access::address_space::local_space and access::address_space::private_space and IsDecorated is access::decorated::yes and access::decorated::no. |
| booln | marray<N,bool> |
| genbool | bool, booln |
| | End of table |

Table 4.126: Generic type name description, which serves as a description for all valid types of parameters to kernel functions [1].

### 4.19.2 Work-item functions

In the OpenCL 1.2 specification document [1, ch. 6.12.1] in Table 6.7 the work-item functions are defined where they provide the size of the enqueued kernel NDRange. These functions are available in SYCL through the nd_item and group classes see section 4.10.1.5 and 4.10.1.7.

### 4.19.3 Function objects

SYCL provides a number of function objects in the `sycl` namespace on host and device. All function objects obey C++ conversion and promotion rules. Each function object is additionally specialized for `void` as a *transparent* function object that deduces its parameter types and return type.

SYCL function objects can be identified using the `sycl::is_native_function_object` and `sycl::is_native_function_object_v` traits classes.

```
namespace sycl {

template <typename T=void>
struct plus {
  T operator()(const T& x, const T& y) const;
};

template <typename T=void>
struct multiplies {
  T operator()(const T& x, const T& y) const;
};

template <typename T=void>
struct bit_and {
  T operator()(const T& x, const T& y) const;
};

template <typename T=void>
struct bit_or {
  T operator()(const T& x, const T& y) const;
};

template <typename T=void>
struct bit_xor {
  T operator()(const T& x, const T& y) const;
};

template <typename T=void>
struct logical_and {
  T operator()(const T& x, const T& y) const;
};

template <typename T=void>
struct logical_or {
  T operator()(const T& x, const T& y) const;
};

template <typename T=void>
struct minimum {
  T operator()(const T& x, const T& y) const;
};

template <typename T=void>
struct maximum {
  T operator()(const T& x, const T& y) const;
};
```

```

} // namespace sycl
```

| Member function | Description |
| --- | --- |
| `T operator()(const T& x, const T& y)const` | Returns the sum of its arguments, equivalent to `x + y`. |
| | End of table |

Table 4.127: Member functions for the `plus` function object.

| Member function | Description |
| --- | --- |
| `T operator()(const T& x, const T& y)const` | Returns the product of its arguments, equivalent to `x * y`. |
| | End of table |

Table 4.128: Member functions for the `multiplies` function object.

| Member function | Description |
| --- | --- |
| `T operator()(const T& x, const T& y)const` | Returns the bitwise AND of its arguments, equivalent to `x & y`. |
| | End of table |

Table 4.129: Member functions for the `bit_and` function object.

| Member function | Description |
| --- | --- |
| `T operator()(const T& x, const T& y)const` | Returns the bitwise OR of its arguments, equivalent to `x \| y`. |
| | End of table |

Table 4.130: Member functions for the `bit_or` function object.

| Member function | Description |
| --- | --- |
| `T operator()(const T& x, const T& y)const` | Returns the bitwise XOR of its arguments, equivalent to `x ^ y`. |
| | End of table |

Table 4.131: Member functions for the `bit_xor` function object.

| Member function | Description |
| --- | --- |
| `T operator()(const T& x, const T& y)const` | Returns the logical AND of its arguments, equivalent to `x && y`. |
| | End of table |

Table 4.132: Member functions for the `logical_and` function object.

| Member function | Description |
|---|---|
| `T operator()(const T& x, const T& y)const` | Returns the logical OR of its arguments, equivalent to `x \|\| y`. |
| | End of table |

Table 4.133: Member functions for the `logical_or` function object.

| Member function | Description |
|---|---|
| `T operator()(const T& x, const T& y)const` | Applies `std::less` to its arguments, in the same order, then returns the lesser argument unchanged. |
| | End of table |

Table 4.134: Member functions for the `minimum` function object.

| Member function | Description |
|---|---|
| `T operator()(const T& x, const T& y)const` | Applies `std::greater` to its arguments, in the same order, then returns the lesser argument unchanged. |
| | End of table |

Table 4.135: Member functions for the `maximum` function object.

### 4.19.4 Algorithms library

SYCL provides an algorithms library based on the functions described in Section 28 of the C++17 specification. The first argument to each function is an execution policy, and data ranges are described using instances of `multi_ptr` (in place of more general iterators) in order to guarantee that address space information is visible to the compiler.

Any restrictions from the standard algorithms library apply. Some of the functions in the SYCL algorithms library introduce additional restrictions in order to maximize portability across different devices and to minimize the chances of encountering unexpected behavior.

All algorithms are supported for the fundamental scalar types supported by SYCL (see Table 5.1) and instances of the SYCL `vec` and `marray` classes. Functions with arguments of type `vec<T,N>` or `marray<T,N>` are applied component-wise and are semantically equivalent to $N$ calls to a scalar algorithm with type `T`.

The execution policy `sycl::execution::group` denotes that an algorithm should be performed collaboratively by the work-items in the specified group. All algorithms using this execution policy therefore act as group functions (as defined in Section 4.19.5), inheriting all restrictions of group functions. Unless the description of a function says otherwise, how the elements of a range are processed by the work-items in a group is undefined.

#### 4.19.4.1 `any_of`, `all_of` and `none_of`

The `any_of`, `all_of` and `none_of` functions test whether Boolean conditions hold for any of, all of or none of the values in a range, respectively.

| Function | Description |
|---|---|
| `template <typename ExecutionPolicy, typename Ptr, typename Predicate> bool any_of(ExecutionPolicy&& policy, Ptr first, Ptr last, Predicate pred)` | Return true if `pred` returns true for any element in the range $[first, last)$.<br><br>If `policy` is `sycl::execution::group`: `first` and `last` must be the same for all work-items in the group; and `pred` must be an immutable callable with the same type and state for all work-items in the group. |
| | End of table |

Table 4.136: Overloads for the `any_of` function.

| Function | Description |
|---|---|
| `template <typename ExecutionPolicy, typename Ptr, typename Predicate> bool all_of(ExecutionPolicy&& policy, Ptr first, Ptr last, Predicate pred)` | Return true if `pred` returns true for all elements in the range $[first, last)$.<br><br>If `policy` is `sycl::execution::group`: `first` and `last` must be the same for all work-items in the group; and `pred` must be an immutable callable with the same type and state for all work-items in the group. |
| | End of table |

Table 4.137: Overloads for the `all_of` function.

| Function | Description |
|---|---|
| `template <typename ExecutionPolicy, typename Ptr, typename Predicate> bool none_of(ExecutionPolicy&& policy, Ptr first, Ptr last, Predicate pred)` | Return true if `pred` returns true for no elements in the range $[first, last)$.<br><br>If `policy` is `sycl::execution::group`: `first` and `last` must be the same for all work-items in the group; and `pred` must be an immutable callable with the same type and state for all work-items in the group. |
| | End of table |

Table 4.138: Overloads for the `none_of` function.

#### 4.19.4.2 `reduce`

The `reduce` function combines values in an unspecified order using a binary operator. The result of a call to `reduce` is non-deterministic if the binary operator is not commutative and associative. Only the binary operators defined in Section 4.19.3 are supported by `reduce` in SYCL 2020, but the standard C++ syntax is used for forward compatibility with future SYCL versions.

| Function | Description |
|---|---|
| `template <typename ExecutionPolicy, typename Ptr, typename BinaryOperation> Ptr::value_type reduce(ExecutionPolicy&& policy, Ptr first, Ptr last, BinaryOperation binary_op)` | Combine the values in the range $[first, last)$ using the operator `binary_op`, which must be an instance of a SYCL function object. `binary_op(*first, *first)` must return a value of type `Ptr::value_type`.<br><br>If `policy` is `sycl::execution::group`: `first`, `last` and the type of `binary_op` must be the same for all work-items in the group. |
| `template <typename ExecutionPolicy, typename Ptr, typename T, typename BinaryOperation> T reduce(ExecutionPolicy&& policy, Ptr first, Ptr last, T init, BinaryOperation binary_op)` | Combine the values in the range $[first, last)$ using an initial value of `init` and the operator `binary_op`, which must be an instance of a SYCL function object. `binary_op(init, *first)` must return a value of type `T`.<br><br>If `policy` is `sycl::execution::group`: `first`, `last`, `init` and the type of `binary_op` must be the same for all work-items in the group. |
| | End of table |

Table 4.139: Overloads of the `reduce` function.

#### 4.19.4.3 `exclusive_scan` and `inclusive_scan`

The scan functions compute a generalized prefix sum using a binary operator. The result of a call to a scan is non-deterministic if the binary operator is not associative. Only the binary operators defined in Section 4.19.3 are supported by the scan functions in SYCL 2020, but the standard C++ syntax is used for forward compatibility with future SYCL versions.

A scan operation can be exclusive or inclusive. For a scan of elements $[x_0, ..., x_n]$, the $i$th result in an exclusive scan excludes $x_i$, whereas the $i$th result in an inclusive scan includes $x_i$.

| Function | Description |
|---|---|
| `template <typename ExecutionPolicy, typename InPtr, typename OutPtr, typename BinaryOperation> OutPtr exclusive_scan(ExecutionPolicy&& policy, InPtr first, InPtr last, OutPtr result, BinaryOperation binary_op)` | Perform an exclusive scan over the values in the range $[first, last)$ using the operator `binary_op`, which must be an instance of a SYCL function object. `binary_op(*first, *first)` must return a value of type `OutPtr::value_type`.<br><br>The value written to `result +` $i$ is the exclusive scan of the first $i$ values in the range and the identity value of `binary_op`. Returns a pointer to the end of the output range.<br><br>If `policy` is `sycl::execution::group` : `first`, `last`, `result` and the type of `binary_op` must be the same for all work-items in the group. |
| `template <typename ExecutionPolicy, typename InPtr, typename OutPtr, typename T, typename BinaryOperation> OutPtr exclusive_scan(ExecutionPolicy&& policy, InPtr first, InPtr last, OutPtr result, T init, BinaryOperation binary_op)` | Perform an exclusive scan over the values in the range $[first, last)$ using the operator `binary_op`, which must be an instance of a SYCL function object. `binary_op(init, *first)` must return a value of type `T`.<br><br>The value written to `result +` $i$ is the exclusive scan of the first $i$ values in the range and an initial value specified by `init`. Returns a pointer to the end of the output range.<br><br>If `policy` is `sycl::execution::group`: `first`, `last`, `result`, `init` and the type of `binary_op` must be the same for all work-items in the group. |
| | End of table |

Table 4.140: Overloads of the `exclusive_scan` function.

| Function | Description |
| --- | --- |
| `template <typename ExecutionPolicy, typename InPtr, typename OutPtr, typename BinaryOperation> OutPtr inclusive_scan(ExecutionPolicy&& policy, InPtr first, InPtr last, OutPtr result, BinaryOperation binary_op)` | Perform an inclusive scan over the values in the range $[first, last)$ using the operator `binary_op`, which must be an instance of a SYCL function object. `binary_op(*first, *first)` must return a value of type `OutPtr::value_type`.<br><br>The value written to `result` + $i$ is the inclusive scan of the first $i$ values in the range. Returns a pointer to the end of the output range.<br><br>If `policy` is `sycl::execution::group` : `first`, `last`, `result` and the type of `binary_op` must be the same for all work-items in the group. |
| `template <typename ExecutionPolicy, typename InPtr, typename OutPtr, typename BinaryOperation, typename T> OutPtr inclusive_scan(ExecutionPolicy&& policy, InPtr first, InPtr last, OutPtr result, BinaryOperation binary_op, T init)` | Perform an inclusive scan over the values in the range $[first, last)$ using the operator `binary_op`, which must be an instance of a SYCL function object. `binary_op(init, *first)` must return a value of type `T`.<br><br>The value written to `result` + $i$ is the inclusive scan of the first $i$ values in the range and an initial value specified by `init`. Returns a pointer to the end of the output range.<br><br>If `policy` is `sycl::execution::group`: `first`, `last`, `result`, `init` and the type of `binary_op` must be the same for all work-items in the group. |
| | End of table |

Table 4.141: Overloads of the `inclusive_scan` function.

### 4.19.5 Group functions

SYCL provides a number of functions that expose functionality tied to groups of work-items (such as group barriers and collective operations). These group functions act as synchronization points and must be encountered in converged control flow by all work-items in the group — if one work-item in the group reaches the function, then all work-items in the group must reach the function. Additionally, restrictions may be placed on the arguments passed to each function in order to ensure that all work-items in the group agree on the operation that is being performed. Any such restrictions on the arguments passed to a function are defined within the descriptions of those functions. Violating these restrictions results in undefined behavior.

All group functions are supported for the fundamental scalar types supported by SYCL (see Table 5.1) and instances of the SYCL `vec` and `marray` classes. Functions with arguments of type `vec<T,N>` or `marray<T,N>` are applied component-wise and are semantically equivalent to $N$ calls to a scalar function with type `T`.

Using a group function inside of a kernel may introduce additional limits on the resources available to user code inside the same kernel. The behavior of these limits is implementation-defined, but must be reflected by calls to kernel querying functions (such as `kernel::get_info`) as described in Section 4.12.

It is undefined behavior for any group function to be invoked within a `parallel_for_work_group` or `parallel_for_work_item` context.

#### 4.19.5.1 `group_broadcast`

The `group_broadcast` function communicates a value held by one work-item to all other work-items in the group.

| Function | Description |
|---|---|
| `template <typename Group, typename T>`<br>`T group_broadcast(Group g, T x)` | Broadcast the value of `x` from the work-item with the smallest linear id to all work-items within the group. |
| `template <typename Group, typename T>`<br>`T group_broadcast(Group g, T x, Group::linear_id_type local_linear_id)` | Broadcast the value of `x` from the work-item with the specified linear id to all work-items within the group.<br><br>The value of `local_linear_id` must be the same for all work-items in the group. |
| `template <typename Group, typename T>`<br>`T group_broadcast(Group g, T x, Group::id_type local_id)` | Broadcast the value of `x` from the work-item with the specified id to all work-items within the group.<br><br>The value of `local_id` must be the same for all work-items in the group, and its dimensionality must match the dimensionality of the group. |
| | End of table |

Table 4.142: Overloads of the `group_broadcast` function.

#### 4.19.5.2 `group_leader`

The `group_leader` function elects a single work-item of a group as leader, commonly in order to execute a task that should only be executed once per group.

| Function | Description |
|---|---|
| `template <typename Group>`<br>`bool group_leader(Group g)` | Return true for exactly one work-item in the group, if the calling work-item is the elected leader of the group. Every call to `leader` with the same group `g` must elect the same work-item. |
| | End of table |

Table 4.143: Overloads of the `group_leader` function.

#### 4.19.5.3 group_barrier

The group_barrier function synchronizes all work-items in a group, using a group barrier.

| Function | Description |
|---|---|
| `template <typename Group>`<br>`void group_barrier(Group g)` | Synchronizes all work-items in the group, with memory ordering on the local address space, global address space or both based on the value of accessSpace. The current work-item will wait at the barrier until all work-items in the group have reached the barrier. In addition the barrier performs a group mem-fence operation ensuring that all memory accesses issued before the barrier complete before those issued after the barrier. |
| | End of table |

Table 4.144: Overloads for the group_barrier function.

#### 4.19.5.4 group_any_of, group_all_of and group_none_of

The group_any_of, group_all_of and group_none_of functions correspond to the any_of, all_of and none_of functions from the algorithm library in Section 4.19.4, respectively. The group_ variants of the functions perform the same operations, but apply directly to values supplied by the work-items in a group instead of a range of values stored in memory.

| Function | Description |
|---|---|
| `template <typename Group>`<br>`bool group_any_of(Group g, bool pred)` | Return true if pred is true for any work-item in the group. |
| `template <typename Group, typename T, typename Predicate>`<br>`bool group_any_of(Group g, T x, Predicate pred)` | Return true if pred(x) is true for any work-item in the group.<br><br>pred must be an immutable callable with the same type and state for all work-items in the group. |
| | End of table |

Table 4.145: Overloads for the group_any_of function.

| Function | Description |
|---|---|
| `template <typename Group>`<br>`bool group_all_of(Group g, bool pred)` | Return true if pred is true for all work-items in the group. |
| | Continued on next page |

Table 4.146: Overloads for the group_all_of function.

| Function | Description |
|---|---|
| `template <typename Group, typename T, typename Predicate> bool group_all_of(Group g, T x, Predicate pred)` | Return true if `pred(x)` is true for all work-items in the group.<br><br>`pred` must be an immutable callable with the same type and state for all work-items in the group. |
| | End of table |

Table 4.146: Overloads for the `group_all_of` function.

| Function | Description |
|---|---|
| `template <typename Group> bool group_none_of(Group g, bool pred)` | Return true if `pred` is true for no work-item in the group. |
| `template <typename Group, typename T, typename Predicate> bool group_none_of(Group g, T x, Predicate pred)` | Return true if `pred(x)` is true for no work-item in the group.<br><br>`pred` must be an immutable callable with the same type and state for all work-items in the group. |
| | End of table |

Table 4.147: Overloads for the `group_none_of` function.

#### 4.19.5.5 `group_reduce`

The `group_reduce` function corresponds to the `reduce` function from the algorithms library in Section 4.19.4. The `group_` variant of the function performs the same operation, but applies directly to values supplied by the work-items in a group instead of a range of values stored in memory.

| Function | Description |
|---|---|
| `template <typename Group, typename T, typename BinaryOperation> T group_reduce(Group g, T x, BinaryOperation binary_op)` | Combine the values of `x` from all work-items in the group using the operator `binary_op`, which must be an instance of a SYCL function object. `binary_op(x, x)` must return a value of type `T`.<br><br>The type of `binary_op` must be the same for all work-items in the group. |
| `template <typename Group, typename V, typename T, typename BinaryOperation> T group_reduce(Group g, V x, T init, BinaryOperation binary_op)` | Combine the values of `x` from all work-items in the group using an initial value of `init` and the operator `binary_op`, which must be an instance of a SYCL function object. `binary_op(init, x)` must return a value of type `T`.<br><br>The type of `binary_op` must be the same for all work-items in the group. |
| | End of table |

Table 4.148: Overloads of the `group_reduce` function.

#### 4.19.5.6 group_exclusive_scan and group_inclusive_scan

The group_exclusive_scan and group_inclusive_scan functions correspond to the exclusive_scan and inclusive_scan functions from the algorithm library in Section 4.19.4, respectively. The group_ variants of the functions perform the same operations, but apply directly to values supplied by the work-items in a group instead of a range of values stored in memory.

| Function | Description |
|---|---|
| `template <typename Group, typename T, typename BinaryOperation>`<br>`T group_exclusive_scan(Group g, T x, BinaryOperation binary_op)` | Perform an exclusive scan over the values of `x` from all work-items in the group using the operator `binary_op`, which must be an instance of a SYCL function object. `binary_op(x, x)` must return a value of type `T`.<br><br>The value returned on work-item *i* is the exclusive scan of the first *i* work-items in the group and the identity value of `binary_op`. For multi-dimensional groups, the order of work-items in the group is determined by their linear id.<br><br>The type of `binary_op` must be the same for all work-items in the group. |
| `template <typename Group, typename V, typename T, typename BinaryOperation>`<br>`T group_exclusive_scan(Group g, V x, T init, BinaryOperation binary_op)` | Perform an exclusive scan over the values of `x` from all work-items in the group using the operator `binary_op`, which must be an instance of a SYCL function object. `binary_op(init, x)` must return a value of type `T`.<br><br>The value returned on work-item *i* is the exclusive scan of the first *i* work items in the group and an initial value specified by `init`. For multi-dimensional groups, the order of work-items in the group is determined by their linear id.<br><br>`init` and the type of `binary_op` must be the same for all work-items in the group. |
| | End of table |

Table 4.149: Overloads of the group_exclusive_scan function.

| Function | Description |
|---|---|
| `template <typename Group, typename T, typename BinaryOperation>`<br>`T group_inclusive_scan(Group g, T x, BinaryOperation binary_op)` | Perform an inclusive scan over the values of `x` from all work-items in the group using the operator `binary_op`, which must be an instance of a SYCL function object. `binary_op(x, x)` must return a value of type `T`.<br><br>The value returned on work-item $i$ is the inclusive scan of the first $i$ work items in the group. For multi-dimensional groups, the order of work-items in the group is determined by their linear id.<br><br>The type of `binary_op` must be the same for all work-items in the group. |
| `template <typename Group, typename V, typename BinaryOperation, typename T>`<br>`T group_inclusive_scan(Group g, V x, BinaryOperation binary_op, T init)` | Perform an inclusive scan over the values of `x` from all work-items in the group using the operator `binary_op`, which must be an instance of a SYCL function object. `binary_op(init, x)` must return a value of type `T`.<br><br>The value returned on work-item $i$ is the inclusive scan of the first $i$ work items in the group and an initial value specified by `init`. For multi-dimensional groups, the order of work-items in the group is determined by their linear id.<br><br>`init` and the type of `binary_op` must be the same for all work-items in the group. |
| | End of table |

Table 4.150: Overloads of the `group_inclusive_scan` function.

### 4.19.6 Math functions

In SYCL the OpenCL math functions are available in the namespace `sycl` on host and device with the same precision guarantees as defined in the OpenCL 1.2 specification document [1, ch. 7] for host and device. For a SYCL platform the numerical requirements for host need to match the numerical requirements of the OpenCL math built-in functions. The built-in functions can take as input float or optionally double and their `vec` and `marray` counterparts, for all supported dimensions including dimension 1.

The built-in functions available for SYCL host and device, with the same precision requirements for both host and device, are described in Table 4.151.

| Math Function | Description |
|---|---|
| `genfloat acos (genfloat x)` | Inverse cosine function. |
| `genfloat acosh (genfloat x)` | Inverse hyperbolic cosine. |
| `genfloat acospi (genfloat x)` | Compute $acos x/\pi$ |
| `genfloat asin (genfloat x)` | Inverse sine function. |
| `genfloat asinh (genfloat x)` | Inverse hyperbolic sine. |
| `genfloat asinpi (genfloat x)` | Compute $asin x/\pi$ |
| `genfloat atan (genfloat y_over_x)` | Inverse tangent function. |
| `genfloat atan2 (genfloat y, genfloat x)` | Compute atan( y / x). |
| `genfloat atanh (genfloat x)` | Hyperbolic inverse tangent. |
| `genfloat atanpi (genfloat x)` | Compute atan (x) / $\pi$. |
| `genfloat atan2pi (genfloat y, genfloat x)` | Compute atan2 (y, x) / $\pi$. |
| `genfloat cbrt (genfloat x)` | Compute cube-root. |
| `genfloat ceil (genfloat x)` | Round to integral value using the round to positive infinity rounding mode. |
| `genfloat copysign (genfloat x, genfloat y)` | Returns x with its sign changed to match the sign of y. |
| `genfloat cos (genfloat x)` | Compute cosine. |
| `genfloat cosh (genfloat x)` | Compute hyperbolic cosine. |
| `genfloat cospi (genfloat x)` | Compute cos $(\pi x)$. |
| `genfloat erfc (genfloat x)` | Complementary error function. |
| `genfloat erf (genfloat x)` | Error function encountered in integrating the normal distribution. |
| `genfloat exp (genfloat x )` | Compute the base-e exponential of x. |
| `genfloat exp2 (genfloat x)` | Exponential base 2 function. |
| `genfloat exp10 (genfloat x)` | Exponential base 10 function. |
| `genfloat expm1 (genfloat x)` | Compute $\exp(x) - 1.0$. |
| `genfloat fabs (genfloat x)` | Compute absolute value of a floating-point number. |
| `genfloat fdim (genfloat x, genfloat y)` | $x - y$ if $x > y$,+0 if x is less than or equal to y. |
| `genfloat floor (genfloat x)` | Round to integral value using the round to negative infinity rounding mode. |
| `genfloat fma (genfloat a, genfloat b, genfloat c)` | Returns the correctly rounded floating-point representation of the sum of c with the infinitely precise product of a and b. Rounding of intermediate products shall not occur. Edge case behavior is per the IEEE 754-2008 standard. |
| `genfloat fmax (genfloat x, genfloat y)`<br>`genfloat fmax (genfloat x, sgenfloat y)` | Returns y if $x < y$, otherwise it returns x. If one argument is a NaN, fmax() returns the other argument. If both arguments are NaNs, fmax() returns a NaN. |
| `genfloat fmin (genfloat x, genfloat y)`<br>`genfloat fmin (genfloat x, sgenfloat y)` | Returns y if $y < x$, otherwise it returns x. If one argument is a NaN, fmin() returns the other argument. If both arguments are NaNs, fmin() returns a NaN. |
| `genfloat fmod (genfloat x, genfloat y)` | Modulus. Returns $x–y * trunc(x/y)$. |
| | Continued on next page |

Table 4.151: Math functions which work on SYCL host and device. They correspond to Table 6.8 of the OpenCL 1.2 specification [1].

| Math Function | Description |
|---|---|
| `genfloat fract (genfloat x, genfloatptr iptr)` | Returns fmin( x − floor (x), nextafter(genfloat(1.0), genfloat(0.0)) ). floor(x) is returned in iptr. |
| `genfloat frexp (genfloat x, genintptr exp)` | Extract mantissa and exponent from x. For each component the mantissa returned is a float with magnitude in the interval [1/2, 1) or 0. Each component of x equals mantissa returned * 2exp. |
| `genfloat hypot (genfloat x, genfloat y)` | Compute the value of the square root of x2+ y2 without undue overflow or underflow. |
| `genint ilogb (genfloat x)` | Return the exponent as an integer value. |
| `genfloat ldexp (genfloat x, genint k)`<br>`genfloat ldexp (genfloat x, int k)` | Multiply x by 2 to the power k. |
| `genfloat lgamma (genfloat x)` | Log gamma function. Returns the natural logarithm of the absolute value of the gamma function. The sign of the gamma function is returned in the signp argument of `lgamma_r`. |
| `genfloat lgamma_r (genfloat x, genintptr signp)` | Log gamma function. Returns the natural logarithm of the absolute value of the gamma function. The sign of the gamma function is returned in the signp argument of `lgamma_r`. |
| `genfloat log (genfloat x)` | Compute natural logarithm. |
| `genfloat log2 (genfloat x)` | Compute a base 2 logarithm. |
| `genfloat log10 (genfloat x)` | Compute a base 10 logarithm. |
| `genfloat log1p (genfloat x)` | Compute $loge(1.0 + x)$. |
| `genfloat logb (genfloat x)` | Compute the exponent of x, which is the integral part of logr $(\|x\|)$. |
| `genfloat mad (genfloat a,genfloat b, genfloat c)` | mad approximates a * b + c. Whether or how the product of a * b is rounded and how supernormal or subnormal intermediate products are handled is not defined. mad is intended to be used where speed is preferred over accuracy. |
| `genfloat maxmag (genfloat x, genfloat y)` | Returns x if $\|x\| > \|y\|$, y if $\|y\| > \|x\|$, otherwise fmax(x, y). |
| `genfloat minmag (genfloat x, genfloat y)` | Returns x if $\|x\| < \|y\|$, y if $\|y\| < \|x\|$, otherwise fmin(x, y). |
| `genfloat modf (genfloat x, genfloatptr iptr)` | Decompose a floating-point number. The modf function breaks the argument x into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part in the object pointed to by iptr. |
| `genfloatf nan (ugenint nancode)`<br>`genfloatd nan (ugenlonginteger nancode)` | Returns a quiet NaN. The nancode may be placed in the significand of the resulting NaN. |

Continued on next page

Table 4.151: Math functions which work on SYCL host and device. They correspond to Table 6.8 of the OpenCL 1.2 specification [1].

| Math Function | Description |
|---|---|
| `genfloat nextafter (genfloat x, genfloat y)` | Computes the next representable single-precision floating-point value following x in the direction of y. Thus, if y is less than x, nextafter() returns the largest representable floating-point number less than x. |
| `genfloat pow (genfloat x, genfloat y)` | Compute x to the power y. |
| `genfloat pown (genfloat x, genint y)` | Compute x to the power y, where y is an integer. |
| `genfloat powr (genfloat x, genfloat y)` | Compute x to the power y, where $x >= 0$. |
| `genfloat remainder (genfloat x, genfloat y)` | Compute the value r such that r = x - n*y, where n is the integer nearest the exact value of x/y. If there are two integers closest to x/y, n shall be the even one. If r is zero, it is given the same sign as x. |
| `genfloat remquo (genfloat x, genfloat y, genintptr quo)` | The remquo function computes the value r such that r = x - k*y, where k is the integer nearest the exact value of x/y. If there are two integers closest to x/y, k shall be the even one. If r is zero, it is given the same sign as x. This is the same value that is returned by the remainder function. remquo also calculates the lower seven bits of the integral quotient x/y, and gives that value the same sign as x/y. It stores this signed value in the object pointed to by quo. |
| `genfloat rint (genfloat x)` | Round to integral value (using round to nearest even rounding mode) in floating-point format. Refer to section 7.1 of the OpenCL 1.2 specification document [1] for description of rounding modes. |
| `genfloat rootn (genfloat x, genint y)` | Compute x to the power 1/y. |
| `genfloat round (genfloat x)` | Return the integral value nearest to x rounding halfway cases away from zero, regardless of the current rounding direction. |
| `genfloat rsqrt (genfloat x)` | Compute inverse square root. |
| `genfloat sin (genfloat x)` | Compute sine. |
| `genfloat sincos (genfloat x, genfloatptr cosval)` | Compute sine and cosine of x. The computed sine is the return value and computed cosine is returned in `cosval`. |
| `genfloat sinh (genfloat x)` | Compute hyperbolic sine. |
| `genfloat sinpi (genfloat x)` | Compute sin ($\pi$ x). |
| `genfloat sqrt (genfloat x)` | Compute square root. |
| `genfloat tan (genfloat x)` | Compute tangent. |
| `genfloat tanh (genfloat x)` | Compute hyperbolic tangent. |
| `genfloat tanpi (genfloat x)` | Compute tan ($\pi$ x). |
| `genfloat tgamma (genfloat x)` | Compute the gamma function. |
| | Continued on next page |

Table 4.151: Math functions which work on SYCL host and device. They correspond to Table 6.8 of the OpenCL 1.2 specification [1].

| Math Function | Description |
|---|---|
| `genfloat trunc (genfloat x)` | Round to integral value using the round to zero rounding mode. |
| | End of table |

Table 4.151: Math functions which work on SYCL host and device. They correspond to Table 6.8 of the OpenCL 1.2 specification [1].

In SYCL the implementation defined precision math functions are defined in the namespace `sycl::native`. The functions that are available within this namespace are specified in Tables 4.152.

| Native Math Function | Description |
|---|---|
| `genfloatf cos (genfloatf x)` | Compute cosine over an implementation-defined range. The maximum error is implementation-defined. |
| `genfloatf divide (genfloatf x, genfloatf y)` | Compute x / y over an implementation-defined range. The maximum error is implementation-defined. |
| `genfloatf exp (genfloatf x)` | Compute the base- e exponential of x over an implementation-defined range. The maximum error is implementation-defined. |
| `genfloatf exp2 (genfloatf x)` | Compute the base- 2 exponential of x over an implementation-defined range. The maximum error is implementation-defined. |
| `genfloatf exp10 (genfloatf x)` | Compute the base- 10 exponential of x over an implementation-defined range. The maximum error is implementation-defined. |
| `genfloatf log (genfloatf x)` | Compute natural logarithm over an implementation defined range. The maximum error is implementation-defined. |
| `genfloatf log2 (genfloatf x)` | Compute a base 2 logarithm over an implementation-defined range. The maximum error is implementation-defined. |
| `genfloatf log10 (genfloatf x)` | Compute a base 10 logarithm over an implementation-defined range. The maximum error is implementation-defined. |
| `genfloatf powr (genfloatf x, genfloatf y)` | Compute x to the power y, where $x >= 0$. The range of x and y are implementation-defined. The maximum error is implementation-defined. |
| `genfloatf recip (genfloatf x)` | Compute reciprocal over an implementation-defined range. The maximum error is implementation-defined. |
| `genfloatf rsqrt (genfloatf x)` | Compute inverse square root over an implementation-defined range. The maximum error is implementation-defined. |
| | Continued on next page |

Table 4.152: Native math functions.

| Native Math Function | Description |
|---|---|
| `genfloatf sin (genfloatf x)` | Compute sine over an implementation-defined range. The maximum error is implementation-defined. |
| `genfloatf sqrt (genfloatf x)` | Compute square root over an implementation-defined range. The maximum error is implementation-defined. |
| `genfloatf tan (genfloatf x)` | Compute tangent over an implementation-defined range. The maximum error is implementation-defined. |
| | End of table |

Table 4.152: Native math functions.

In SYCL the half precision math functions are defined in `sycl::half_precision`. The functions that are available within this namespace are specified in Tables 4.153. These functions are implemented with a minimum of 10-bits of accuracy i.e. an ULP value is less than or equal to 8192 ulp.

| Half Math function | Description |
|---|---|
| `genfloatf cos (genfloatf x)` | Compute cosine. x must be in the range -216 to +216. |
| `genfloatf divide (genfloatf x, genfloatf y)` | Compute x / y. |
| `genfloatf exp (genfloatf x)` | Compute the base- e exponential of x. |
| `genfloatf exp2 (genfloatf x)` | Compute the base- 2 exponential of x. |
| `genfloatf exp10 (genfloatf x)` | Compute the base- 10 exponential of x. |
| `genfloatf log (genfloatf x)` | Compute natural logarithm. |
| `genfloatf log2 (genfloatf x)` | Compute a base 2 logarithm. |
| `genfloatf log10 (genfloatf x)` | Compute a base 10 logarithm. |
| `genfloatf powr (genfloatf x, genfloatf y)` | Compute x to the power y, where $x >= 0$. |
| `genfloatf recip (genfloatf x)` | Compute reciprocal. |
| `genfloatf rsqrt (genfloatf x)` | Compute inverse square root. |
| `genfloatf sin (genfloatf x)` | Compute sine. x must be in the range -216 to +216. |
| `genfloatf sqrt (genfloatf x)` | Compute square root. |
| `genfloatf tan (genfloatf x)` | Compute tangent. x must be in the range -216 to +216. |
| | End of table |

Table 4.153: Half precision math functions.

## 4.19.7 Integer functions

Integer math functions are available in SYCL in the namespace `sycl` on host and device. The built-in functions can take as input `char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long long int`, `unsigned long long int` and their `vec` and `marray` counterparts. The supported integer math functions are described in Table 4.154.

| Integer Function | Description |
|---|---|
| `ugeninteger abs (geninteger x)` | Returns $\|x\|$. |
| `ugeninteger abs_diff (geninteger x, geninteger y)` | Returns $\|x-y\|$ without modulo overflow. |
| `geninteger add_sat (geninteger x, geninteger y)` | Returns $x+y$ and saturates the result. |
| `geninteger hadd (geninteger x, geninteger y)` | Returns $(x+y) >> 1$. The intermediate sum does not modulo overflow. |
| `geninteger rhadd (geninteger x, geninteger y)` | Returns $(x+y+1) >> 1$. The intermediate sum does not modulo overflow. |
| `geninteger clamp (geninteger x, geninteger minval, geninteger maxval)` `geninteger clamp (geninteger x, sgeninteger minval, sgeninteger maxval)` | Returns min(max(x, minval), maxval). Results are undefined if minval > maxval. |
| `geninteger clz (geninteger x)` | Returns the number of leading 0-bits in x, starting at the most significant bit position. If x is 0, returns the size in bits of the type of x or component type of x, if x is a vector type. |
| `geninteger ctz (geninteger x)` | Returns the count of trailing 0-bits in x. If x is 0, returns the size in bits of the type of x or component type of x, if x is a vector type. |
| `geninteger mad_hi ( geninteger a, geninteger b, geninteger c)` | Returns `mul_hi(a, b)+c`. |
| `geninteger mad_sat (geninteger a, geninteger b, geninteger c)` | Returns `a * b + c` and saturates the result. |
| `geninteger max (geninteger x, geninteger y)` `geninteger max (geninteger x, sgeninteger y)` | Returns y if $x < y$, otherwise it returns x. |
| `geninteger min (geninteger x, geninteger y)` `geninteger min (geninteger x, sgeninteger y)` | Returns y if $y < x$, otherwise it returns x. |
| `geninteger mul_hi (geninteger x, geninteger y)` | Computes `x * y` and returns the high half of the product of x and y. |
| `geninteger rotate (geninteger v, geninteger i)` | For each element in v, the bits are shifted left by the number of bits given by the corresponding element in i (subject to usual shift modulo rules described in section 6.3). Bits shifted off the left side of the element are shifted back in from the right. |
| `geninteger sub_sat (geninteger x, geninteger y)` | Returns $x-y$ and saturates the result. |
| `ugeninteger16bit upsample (ugeninteger8bit hi, ugeninteger8bit lo)` | `result[i] = ((ushort)hi[i] << 8)\| lo[i]` |
| `igeninteger16bit upsample (igeninteger8bit hi, ugeninteger8bit lo)` | `result[i] = ((short)hi[i] << 8)\| lo[i]` |
| `ugeninteger32bit upsample (ugeninteger16bit hi, ugeninteger16bit lo)` | `result[i] = ((uint)hi[i] << 16)\| lo[i]` |
| `igeninteger32bit upsample (igeninteger16bit hi, ugeninteger16bit lo)` | `result[i] = ((int)hi[i] << 16)\| lo[i]` |
| `ugeninteger64bit upsample (ugeninteger32bit hi, ugeninteger32bit lo)` | `result[i] = ((ulonglong)hi[i] << 32)\| lo[i]` |
| | Continued on next page |

Table 4.154: Integer functions which work on SYCL host and device, are available in the `sycl` namespace.

| Integer Function | Description |
|---|---|
| `igeninteger64bit upsample (igeninteger32bit hi, ugeninteger32bit lo)` | `result[i] = ((longlong)hi[i] << 32)\| lo[i]` |
| `geninteger popcount (geninteger x)` | Returns the number of non-zero bits in x. |
| `geninteger32bit mad24 (geninteger32bit x, geninteger32bit y, geninteger32bit z)` | Multiply two 24-bit integer values x and y and add the 32-bit integer result to the 32-bit integer z. Refer to definition of mul24 to see how the 24-bit integer multiplication is performed. |
| `geninteger32bit mul24 (geninteger32bit x, geninteger32bit y)` | Multiply two 24-bit integer values x and y. x and y are 32-bit integers but only the low 24-bits are used to perform the multiplication. mul24 should only be used when values in x and y are in the range [- 223, 223-1] if x and y are signed integers and in the range [0, 224-1] if x and y are unsigned integers. If x and y are not in this range, the multiplication result is implementation-defined. |
| | End of table |

Table 4.154: Integer functions which work on SYCL host and device, are available in the `sycl` namespace.

### 4.19.8 Common functions

In SYCL the OpenCL *common* functions are available in the namespace `sycl` on host and device as defined in the OpenCL 1.2 specification document [1, par. 6.12.4]. They are described here in Table 4.155. The built-in functions can take as input `float` or optionally `double` and their `vec` and `marray` counterparts.

| Common Function | Description |
|---|---|
| `genfloat clamp (genfloat x, genfloat minval, genfloat maxval)`<br>`genfloatf clamp (genfloatf x, float minval, float maxval)`<br>`genfloatd clamp (genfloatd x, double minval, double maxval)` | Returns fmin(fmax(x, minval), maxval). Results are undefined if $minval > maxval$. |
| `genfloat degrees (genfloat radians)` | Converts radians to degrees, i.e.$(180/\pi) * radians$. |
| `genfloat max (genfloat x, genfloat y)`<br>`genfloatf max (genfloatf x, float y)`<br>`genfloatd max (genfloatd x, double y)` | Returns y if $x < y$, otherwise it returns x. If x or y are infinite or NaN, the return values are undefined. |
| `genfloat min (genfloat x, genfloat y)`<br>`genfloatf min (genfloatf x, float y)`<br>`genfloatd min (genfloatd x, double y)` | Returns y if $y < x$, otherwise it returns x. If x or y are infinite or NaN, the return values are undefined. |
| `genfloat mix (genfloat x, genfloat y, genfloat a)`<br>`genfloatf mix (genfloatf x, genfloatf y, float a)`<br>`genfloatd mix (genfloatd x, genfloatd y, double a)` | Returns the linear blend of $x\&y$ implemented as: $x+(y-x)*a$. $a$ must be a value in the range 0.0 ... 1.0. If a is not in the range 0.0 ... 1.0, the return values are undefined. |
| | Continued on next page |

Table 4.155: Common functions which work on SYCL host and device, are available in the `sycl` namespace. They correspond to Table 6.12 of the OpenCL 1.2 specification [1].

| Common Function | Description |
|---|---|
| `genfloat radians (genfloat degrees)` | Converts degrees to radians, i.e. $(\pi/180) * degrees$. |
| `genfloat step (genfloat edge, genfloat x)`<br>`genfloatf step (float edge, genfloatf x)`<br>`genfloatd step (double edge, genfloatd x)` | Returns 0.0 if $x < edge$, otherwise it returns 1.0. |
| `genfloat smoothstep (genfloat edge0, genfloat edge1, genfloat x)`<br>`genfloatf smoothstep (float edge0, float edge1, genfloatf x)`<br>`genfloatd smoothstep (double edge0, double edge1 , genfloatd x)` | Returns 0.0 if $x <= edge0$ and 1.0 if $x >= edge1$ and performs smooth Hermite interpolation between 0 and 1 when $edge0 < x < edge1$. This is useful in cases where you would want a threshold function with a smooth transition.<br>This is equivalent to:<br>`gentype t;`<br>`t = clamp ((x <= edge0)/ (edge1 >= edge0), 0, 1);`<br>`return t * t * (3 - 2 * t);`<br><br>Results are undefined if $edge0 >= edge1$ or if x, edge0 or edge1 is a NaN. |
| `genfloat sign (genfloat x)` | Returns 1.0 if $x > 0$, $-0.0$ if $x = -0.0$, $+0.0$ if $x = +0.0$, or $-1.0$ if $x < 0$. Returns 0.0 if x is a NaN. |
| | End of table |

Table 4.155: Common functions which work on SYCL host and device, are available in the sycl namespace. They correspond to Table 6.12 of the OpenCL 1.2 specification [1].

### 4.19.9 Geometric functions

In SYCL the OpenCL *geometric* functions are available in the namespace sycl on host and device as defined in the OpenCL 1.2 specification document [1, par. 6.12.5]. The built-in functions can take as input float or optionally double and their vec and codeinlinemarray counterparts, for dimensions 2, 3 and 4. On the host the vector types use the vec class and on an SYCL device use the corresponding native SYCL backend vector types. All of the geometric functions use round-to-nearest-even rounding mode. Table 4.156 contains the definitions of supported geometric functions.

| Geometric Function | Description |
|---|---|
| `float4 cross (float4 p0, float4 p1)`<br>`float3 cross (float3 p0, float3 p1)`<br>`double4 cross (double4 p0, double4 p1)`<br>`double3 cross (double3 p0, double3 p1)` | Returns the cross product of p0.xyz and p1.xyz. The $w$ component of float4 result returned will be 0.0. |
| `mfloat4 cross (mfloat4 p0, mfloat4 p1)`<br>`mfloat3 cross (mfloat3 p0, mfloat3 p1)`<br>`mdouble4 cross (mdouble4 p0, mdouble4 p1)`<br>`mdouble3 cross (mdouble3 p0, mdouble3 p1)` | Returns the cross product of first 3 components of p0 and p1. The 4th component of result returned will be 0.0. |
| `float dot (gengeofloat p0, gengeofloat p1)`<br>`double dot (gengeodouble p0, gengeodouble p1)` | Compute dot product. |
| | Continued on next page |

Table 4.156: Geometric functions which work on SYCL host and device, are available in the sycl namespace. They correspond to Table 6.13 of the OpenCL 1.2 specification [1].

| Geometric Function | Description |
|---|---|
| `float distance (gengeofloat p0, gengeofloat p1)`<br>`double distance (gengeodouble p0, gengeodouble p1)` | Returns the distance between p0 and p1. This is calculated as `length(p0 - p1)`. |
| `float length (gengeofloat p)`<br>`double length (gengeodouble p)` | Return the length of vector p, i.e., $\sqrt{p.x^2 + p.y^2 + ...}$ |
| `gengeofloat normalize (gengeofloat p)`<br>`gengeodouble normalize (gengeodouble p)` | Returns a vector in the same direction as p but with a length of 1. |
| `float fast_distance (gengeofloat p0, gengeofloat p1)` | Returns `fast_length(p0 - p1)`. |
| `float fast_length (gengeofloat p)` | Returns the length of vector p computed as: `sqrt((half)(pow(p.x,2)+ pow(p.y,2) + ...))` |
| `gengeofloat fast_normalize (gengeofloat p)` | Returns a vector in the same direction as p but with a length of 1. fast_normalize is computed as: `p * rsqrt((half)(pow(p.x ,2)+ pow(p.y,2)+ ... ))`<br>The result shall be within 8192 ulps error from the infinitely precise result of `if (all (p == 0.0f))`<br>`result = p;`<br>`else`<br>`result = p / sqrt (pow(p.x,2)+ pow(p. y,2)+ ... );`<br>with the following exceptions:<br>1. If the sum of squares is greater than `FLT_MAX` then the value of the floating-point values in the result vector are undefined.<br>2. If the sum of squares is less than `FLT_MIN` then the implementation may return back p.<br>3. If the device is in "denorms are flushed to zero" mode, individual operand elements with magnitude less than `sqrt(FLT_MIN)` may be flushed to zero before proceeding with the calculation. |
| | End of table |

Table 4.156: Geometric functions which work on SYCL host and device, are available in the `sycl` namespace. They correspond to Table 6.13 of the OpenCL 1.2 specification [1].

### 4.19.10 Relational functions

In SYCL the OpenCL *relational* functions are available in the namespace `sycl` on host and device as defined in the OpenCL 1.2 specification document [1, par. 6.12.6]. The built-in functions can take as input `char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `float` or optionally `double` and their `vec` and `marray` counterparts. The relational functions are provided in addition to the the operators.

The available built-in functions for `vec` template class are described in Tables 4.157

| Relational Function | Description |
|---|---|
| `igeninteger32bit isequal (genfloatf x, genfloatf y)`<br>`igeninteger64bit isequal (genfloatd x, genfloatd y)` | Returns the component-wise compare of $x == y$. |
| `igeninteger32bit isnotequal (genfloatf x, genfloatf y)`<br>`igeninteger64bit isnotequal (genfloatd x, genfloatd y)` | Returns the component-wise compare of $x! = y$. |
| `igeninteger32bit isgreater (genfloatf x, genfloatf y )`<br>`igeninteger64bit isgreater (genfloatd x, genfloatd y )` | Returns the component-wise compare of $x > y$. |
| `igeninteger32bit isgreaterequal (genfloatf x, genfloatf y)`<br>`igeninteger64bit isgreaterequal (genfloatd x, genfloatd y)` | Returns the component-wise compare of $x >= y$. |
| `igeninteger32bit isless (genfloatf x, genfloatf y)`<br>`igeninteger64bit isless (genfloatd x, genfloatd y)` | Returns the component-wise compare of $x < y$. |
| `igeninteger32bit islessequal (genfloatf x, genfloatf y)`<br>`igeninteger64bit islessequal (genfloatd x, genfloatd y)` | Returns the component-wise compare of $x <= y$. |
| `igeninteger32bit islessgreater (genfloatf x, genfloatf y)`<br>`igeninteger64bit islessgreater (genfloatd x, genfloatd y)` | Returns the component-wise compare of $(x < y) \|\| (x > y)$. |
| `igeninteger32bit isfinite (genfloatf x)`<br>`igeninteger64bit isfinite (genfloatd x)` | Test for finite value. |
| `igeninteger32bit isinf (genfloatf x)`<br>`igeninteger64bit isinf (genfloatd x)` | Test for infinity value (positive or negative) . |
| `igeninteger32bit isnan (genfloatf x)`<br>`igeninteger64bit isnan (genfloatd x)` | Test for a NaN. |
| `igeninteger32bit isnormal (genfloatf x)`<br>`igeninteger64bit isnormal (genfloatd x)` | Test for a normal value. |
| `igeninteger32bit isordered (genfloatf x, genfloatf y )`<br>`igeninteger64bit isordered (genfloatd x, genfloatd y )` | Test if arguments are ordered. isordered() takes arguments x and y, and returns the result isequal(x, x) && isequal(y, y). |
| `igeninteger32bit isunordered (genfloatf x, genfloatf y)`<br>`igeninteger64bit isunordered (genfloatd x, genfloatd y)` | Test if arguments are unordered. isunordered() takes arguments x and y, returning non-zero if x or y is NaN, and zero otherwise. |
| `igeninteger32bit signbit (genfloatf x)`<br>`igeninteger64bit signbit (genfloatd x)` | Test for sign bit. The scalar version of the function returns a 1 if the sign bit in the float is set else returns 0.<br>The vector version of the function returns the following for each component in *floatn*: -1 (i.e all bits set) if the sign bit in the float is set else returns 0. |

Continued on next page

Table 4.157: Relational functions for `vec` template class which work on SYCL host and device, are available in the `sycl` namespace. They correspond to Table 6.14 of the OpenCL 1.2 specification [1].

| Relational Function | Description |
| --- | --- |
| `int any (igeninteger x)` | Returns 1 if the most significant bit in any component of x is set; otherwise returns 0. |
| `int all (igeninteger x)` | Returns 1 if the most significant bit in all components of x is set; otherwise returns 0. |
| `gentype bitselect (gentype a, gentype b, gentype c)` | Each bit of the result is the corresponding bit of a if the corresponding bit of c is 0. Otherwise it is the corresponding bit of b. |
| `geninteger select (geninteger a, geninteger b, igeninteger c)`<br>`geninteger select (geninteger a, geninteger b, ugeninteger c)`<br>`genfloatf select (genfloatf a, genfloatf b, genint c)`<br>`genfloatf select (genfloatf a, genfloatf b, ugenint c)`<br>`genfloatd select (genfloatd a, genfloatd b, igeninteger64 c)`<br>`genfloatd select (genfloatd a, genfloatd b, ugeninteger64 c)` | For each component of a vector type:<br>`result[i] = (MSB of c[i] is set)? b[i] : a[i].`<br>For a scalar type:<br>`result = c ? b : a.`<br>`geninteger` must have the same number of elements and bits as `gentype`. |
| | End of table |

Table 4.157: Relational functions for `vec` template class which work on SYCL host and device, are available in the `sycl` namespace. They correspond to Table 6.14 of the OpenCL 1.2 specification [1].

The available built-in functions for `marray` template class are described in Tables 4.158

| Relational Function | Description |
| --- | --- |
| `genbool isequal (genfloatf x, genfloatf y)`<br>`genbool isequal (genfloatd x, genfloatd y)` | Returns the component-wise compare of $x == y$. |
| `genbool isnotequal (genfloatf x, genfloatf y)`<br>`genbool isnotequal (genfloatd x, genfloatd y)` | Returns the component-wise compare of $x! = y$. |
| `genbool isgreater (genfloatf x, genfloatf y)`<br>`genbool isgreater (genfloatd x, genfloatd y)` | Returns the component-wise compare of $x > y$. |
| `genbool isgreaterequal (genfloatf x, genfloatf y)`<br>`genbool isgreaterequal (genfloatd x, genfloatd y)` | Returns the component-wise compare of $x >= y$. |
| `genbool isless (genfloatf x, genfloatf y)`<br>`genbool isless (genfloatd x, genfloatd y)` | Returns the component-wise compare of $x < y$. |
| `genbool islessequal (genfloatf x, genfloatf y)`<br>`genbool islessequal (genfloatd x, genfloatd y)` | Returns the component-wise compare of $x <= y$. |
| `genbool islessgreater (genfloatf x, genfloatf y)`<br>`genbool islessgreater (genfloatd x, genfloatd y)` | Returns the component-wise compare of $(x < y) \vert\vert (x > y)$. |
| `genbool isfinite (genfloatf x)`<br>`genbool isfinite (genfloatd x)` | Test for finite value. |
| `genbool isinf (genfloatf x)`<br>`genbool isinf (genfloatd x)` | Test for infinity value (positive or negative) . |
| `genbool isnan (genfloatf x)`<br>`genbool isnan (genfloatd x)` | Test for a NaN. |
| | Continued on next page |

Table 4.158: Relational functions for scalar data types and `marray` template class template class which work on SYCL host and device, are available in the `sycl` namespace.

| Relational Function | Description |
|---|---|
| `genbool isnormal (genfloatf x)`<br>`genbool isnormal (genfloatd x)` | Test for a normal value. |
| `genbool isordered (genfloatf x, genfloatf y)`<br>`genbool isordered (genfloatd x, genfloatd y)` | Test if arguments are ordered. isordered() takes arguments x and y, and returns the result isequal(x, x) && isequal(y, y). |
| `genbool isunordered (genfloatf x, genfloatf y)`<br>`genbool isunordered (genfloatd x, genfloatd y)` | Test if arguments are unordered. isunordered() takes arguments x and y, returning `true` if x or y is NaN, and `false` otherwise. |
| `genbool signbit (genfloatf x)`<br>`genbool signbit (genfloatd x)` | Test for sign bit, returning `true` if the sign bit in the float is set, and `false` otherwise. |
| `bool any (genbool x)` | Returns `true` if the most significant bit in any component of x is set; otherwise returns `false`. |
| `int all (igeninteger x)` | Returns `true` if the most significant bit in all components of x is set; otherwise returns `false`. |
| `gentype bitselect (gentype a, gentype b, gentype c)` | Each bit of the result is the corresponding bit of a if the corresponding bit of c is 0. Otherwise it is the corresponding bit of b. |
| `gentype select (gentype a, gentype b, genbool c)` | Returns the component-wise `result = c ? b : a`. |
| | End of table |

Table 4.158: Relational functions for scalar data types and `marray` template class template class which work on SYCL host and device, are available in the `sycl` namespace..

## 4.19.11 Vector data load and store functions

The functionality from the OpenCL functions as defined in the OpenCL 1.2 specification document [1, par. 6.12.7] is available in SYCL through the `vec` class in section 4.16.2.

## 4.19.12 Synchronization functions

In SYCL the OpenCL *synchronization* functions are available through the `nd_item` class 4.10.1.5, as they are applied to work-items for local or global address spaces. Please see 4.81.

## 4.19.13 `printf` function

The functionality of the `printf` function is covered by the `stream` class 4.18, which has the capability to print to standard output all of the SYCL classes and primitives, and covers the capabilities defined in the OpenCL 1.2 specification document [1, par. 6.12.13].

# 5. SYCL Device Compiler

This section specifies the requirements of the SYCL device compiler. Most features described in this section relate to underlying SYCL backend capabilities of target devices and limiting the requirements of device code to ensure portability.

## 5.1 Offline compilation of SYCL source files

There are two alternatives for a SYCL device compiler: a *single-source device compiler* and a device compiler that supports the technique of SMCP.

A SYCL device compiler takes in a C++ source file, extracts only the SYCL kernels and outputs the device code in a form that can be enqueued from host code by the associated SYCL runtime. How the SYCL runtime invokes the kernels is implementation defined, but a typical approach is for a device compiler to produce a header file with the compiled kernel contained within it. By providing a command-line option to the host compiler, it would cause the implementation's SYCL header files to `#include` the generated header file. The SYCL specification has been written to allow this as an implementation approach in order to allow SMCP. However, any of the mechanisms needed from the SYCL compiler, the SYCL runtime and build system are implementation defined, as they can vary depending on the platform and approach.

A SYCL single-source device compiler takes in a C++ source file and compiles both host and device code at the same time. This specification specifies how a SYCL single-source device compiler sees and outputs device code for kernels, but does not specify the host compilation.

## 5.2 Naming of kernels

SYCL kernels are extracted from C++ source files and stored in an implementation- defined format. In the case of the shared-source compilation model, the kernels have to be uniquely identified by both host and device compiler. This is required in order for the host runtime to be able to load the kernel by using the OpenCL host runtime interface.

From this requirement the following rules apply for naming the kernels:

- The kernel name is a *C++ typename*.
- The kernel name type may not be forward declared other than in namespace scope (including global namespace scope). If it isn't forward declared but is specified as a template argument in a kernel invoking interface, as described in 4.10.7, then it may not conflict with a name in any enclosing namespace scope.
- If the kernel is defined as a named function object type, the name can be the typename of the function object as long as it is either declared at namespace scope, or does not conflict with any name in an enclosing namespace scope.
- If the kernel is defined as a lambda, a typename can optionally be provided to the kernel invoking interface as described in 4.10.7, so that the developer can control the kernel name for purposes such as debugging or

referring to the kernel when applying build options.

In both single-source and shared-source implementations, a device compiler should detect the kernel invocations (e.g. `parallel_for<kernelname>`) in the source code and compile the enclosed kernels, storing them with their associated type name.

The format of the kernel and the compilation techniques are implementation defined. The interface between the compiler and the runtime for extracting and executing SYCL kernels on the device is implementation defined.

## 5.3 Language restrictions for kernels

The extracted SYCL kernels need to be compiled by an OpenCL online or offline compiler and be executed by the OpenCL 1.2 runtime. The extracted kernels need to be OpenCL 1.2 compliant kernels and as such there are certain restrictions that apply to them.

The following restrictions are applied to device functions and kernels:

- Structures containing pointers may be shared but the value of any pointer passed between SYCL devices or between the host and a SYCL device is undefined.
- Memory storage allocation is not allowed in kernels. All memory allocation for the device is done on the host using accessor classes. Consequently, the default allocation `operator new` overloads that allocate storage are disallowed in a SYCL kernel. The placement `new` operator and any user-defined overloads that do not allocate storage are permitted.
- Kernel functions must always have a `void` return type. A kernel lambda trailing-return-type that is not `void` is therefore illegal, as is a return statement (that would return from the kernel function) with an expression that does not convert to `void`.
- The odr-use of polymorphic classes and classes with virtual inheritance is allowed. However, no virtual member functions are allowed to be called in a SYCL kernel or any functions called by the kernel.
- No function pointers or references are allowed to be called in a SYCL kernel or any functions called by the kernel.
- RTTI is disabled inside kernels.
- No variadic functions are allowed to be called in a SYCL kernel or any functions called by the kernel.
- Exception-handling cannot be used inside a SYCL kernel or any code called from the kernel. But of course `noexcept` is allowed.
- Recursion is not allowed in a SYCL kernel or any code called from the kernel.
- Variables with thread storage duration (`thread_local` storage class specifier) are not allowed to be odr-used in kernel code.
- Variables with static storage duration that are odr-used inside a kernel must be `const` or `constexpr` and zero-initialized or constant-initialized.
- The rules for kernels apply to both the kernel function objects themselves and all functions, operators, member functions, constructors and destructors called by the kernel. This means that kernels can only use library functions that have been adapted to work with SYCL. Implementations are not required to

support any library routines in kernels beyond those explicitly mentioned as usable in kernels in this spec. Developers should refer to the SYCL built-in functions in 4.19 to find functions that are specified to be usable in kernels.

- Interacting with a special SYCL runtime class (i.e. SYCL `accessor`, `sampler` or `stream`) that is stored within a C++ union is undefined behavior.

## 5.4 Compilation of functions

The SYCL device compiler parses an entire C++ source file supplied by the user. This also includes C++ header files, using `#include` directives. From this source file, the SYCL device compiler must compile kernels for the device, as well as any functions that the kernels call.

In SYCL, kernels are invoked using a kernel invoke function (e.g. `parallel_for`). The kernel invoke functions are templated by their kernel parameter, which is a function object. The code inside the function object that is invoked as a kernel is called the “kernel function”. The “kernel function” must always return void. Any function called by the kernel function is compiled for device and called a “device function”. Recursively, any function called by a device function is itself compiled as a device function.

For example, this source code shows three functions and a kernel invoke with comments explaining which functions need to be compiled for device.

```
void f ()
{
    // function "f" is not compiled for device

    single_task<class kernel_name>([=] ()
        {
            // This code compiled for device
            g (); // this line forces "g" to be compiled for device
        });
}

void g ()
{
    // called from kernel, so "g" is compiled for device
}

void h ()
{
    // not called from a device function, so not compiled for device
}
```

In order for the SYCL device compiler to correctly compile device functions, all functions in the source file, whether device functions or not, must be syntactically correct functions according to this specification. A syntactically correct function adheres to at least the minimum required C++ version defined in Section 3.8.1, and doesn’t violate any of the restrictions defined in Section 5.3.

## 5.5 Built-in scalar data types

In a SYCL device compiler, the device definition of all standard C++ fundamental types from Table 5.1 must match the host definition of those types, in both size and alignment. A device compiler may have this preconfigured so that it can match them based on the definitions of those types on the platform, or there may be a necessity for a device compiler command-line option to ensure the types are the same.

The standard C++ fixed width types, e.g. `int8_t`, `int16_t`, `int32_t`,`int64_t`, should have the same size as defined by the C++ standard for host and device.

| Fundamental data type | Description |
|---|---|
| `bool` | A conditional data type which can be either true or false. The value true expands to the integer constant 1 and the value false expands to the integer constant 0. |
| `char` | A signed or unsigned 8-bit integer, as defined by the C++ core language |
| `signed char` | A signed 8-bit integer, as defined by the C++ core language |
| `unsigned char` | An unsigned 8-bit integer, as defined by the C++ core language |
| `short int` | A signed integer of at least 16-bits, as defined by the C++ core language |
| `unsigned short int` | An unsigned integer of at least 16-bits, as defined by the C++ core language |
| `int` | A signed integer of at least 16-bits, as defined by the C++ core language |
| `unsigned int` | An unsigned integer of at least 16-bits, as defined by the C++ core language |
| `long int` | A signed integer of at least 32-bits, as defined by the C++ core language |
| `unsigned long int` | An unsigned integer of at least 32-bits, as defined by the C++ core language |
| `long long int` | An integer of at least 64-bits, as defined by the C++ core language |
| `unsigned long long int` | An unsigned integer of at least 64-bits, as defined by the C++ core language |
| `float` | A 32-bit floating-point. The float data type must conform to the IEEE 754 single precision storage format. |
| `double` | A 64-bit floating-point. The double data type must conform to the IEEE 754 double precision storage format. |
| | End of table |

Table 5.1: Fundamental data types supported by SYCL.

## 5.6 Preprocessor directives and macros

The standard C++ preprocessing directives and macros are supported. The following preprocessor macros must be defined by all conformant implementations:

- `SYCL_LANGUAGE_VERSION` substitutes an integer reflecting the version number of the SYCL language being supported by the device compiler. The version of SYCL defined in this document will have `SYCL_LANGUAGE_VERSION` substitute the integer 2020 pre-provisional;

- `__SYCL_DEVICE_ONLY__` is defined to 1 if the source file is being compiled with a SYCL device compiler which does not produce host binary;

- `__SYCL_SINGLE_SOURCE__` is defined to 1 if the source file is being compiled with a SYCL single-source compiler which produces host as well as device binary;

- `SYCL_EXTERNAL` is an optional macro which enables external linkage of SYCL functions and methods to be included in a SYCL kernel. The macro is only defined if the implementation supports external linkage. For more details see 5.9.1

In addition, for each SYCL backend supported, the preprocessor macros described in the backend section 4.1 must be defined by all conformant implementations.

## 5.7 Kernel attributes

The SYCL general programming interface defines attributes that augment the information available while generating the device code for a particular platform.

### 5.7.1 Core kernel attributes

The attributes in Table 5.2 are defined in the `[[sycl::]]` namespace and are applied to the function-type of kernel function declarations using C++ attribute specifier syntax.

A given attribute-token shall appear at most once in each attribute-list. The first declaration of a function shall specify an attribute if any declaration of that function specifies the same attribute. If a function is declared with an attribute in one translation unit and the same function is declared without the same attribute in another translation unit, the program is ill-formed and no diagnostic is required.

If there are any conflicts between different kernel attributes, then the behavior is undefined. The attributes have an effect when applied to a kernel function and no effect otherwise (i.e. no effect on non-kernel functions and on anything other than a function). If an attribute is applied to a device function that is not a kernel function (but that is potentially called from a kernel function), then the effect is implementation defined. It is implementation defined whether any diagnostic is produced when an attribute is applied to anything other than the function-type of a kernel function declaration.

| SYCL attribute | Description |
|---|---|
| `reqd_work_group_size(dim0)`<br>`reqd_work_group_size(dim0, dim1)`<br>`reqd_work_group_size(dim0, dim1, dim2)` | Indicates that the kernel must be launched with the specified work-group size. The order of the arguments matches the constructor of the `group` class. Each argument to the attribute must be an integral constant expression. The dimensionality of the attribute variant used must match the dimensionality of the work-group used to invoke the kernel.<br><br>SYCL device compilers should give a compilation error if the required work-group size is unsupported. If the kernel is submitted for execution using an incompatible work-group size, the SYCL runtime must throw an `exception` with the `errc::nd_range_error` error code. |
| `work_group_size_hint(dim0)`<br>`work_group_size_hint(dim0, dim1)`<br>`work_group_size_hint(dim0, dim1, dim2)` | Hint to the compiler on the work-group size most likely to be used when launching the kernel at runtime. Each argument must be an integral constant expression, and the number of dimensional values defined provide additional information to the compiler on the dimensionality most likely to be used when launching the kernel at runtime. The effect of this attribute, if any, is implementation defined. |
| `vec_type_hint(<type>)` | Hint to the compiler on the vector computational width of of the kernel. The argument must be one of the vector types defined in section 4.10.2. The effect of this attribute, if any, is implementation defined.<br><br>This attribute is deprecated (available for use, but will likely be removed in a future version of the specification and is not recommended for use in new code). |

Continued on next page

Table 5.2: Attributes supported by the SYCL General programming interface.

| SYCL attribute | Description |
|---|---|
| `reqd_sub_group_size(dim)` | Indicates that the kernel must be compiled and executed with the specified sub-group size. The argument to the attribute must be an integral constant expression.<br><br>SYCL device compilers should give a compilation error if the required sub-group size is unsupported by the device or incompatible with any language feature used by the kernel. The set of valid sub-group sizes for a kernel can be queried as described in Table 4.19 and Table 4.99. |
| | End of table |

Table 5.2: Attributes supported by the SYCL General programming interface.

Other attributes may be provided as part of SYCL backend-interop functionality.

### 5.7.2 Example attribute syntax

Using `[[sycl::reqd_work_group_size(16)]]` as an example attribute, but applying equally to all attributes in Table 5.2 and to attributes that are part of SYCL backend-interop or extensions, the following code clips demonstrate how to apply the attributes to the function-type of a kernel function.

```
// Kernel defined as a lambda
myQueue.submit([&](handler &h) {
 h.parallel_for( range<1>(16),
      [=] (item<1> it) [[sycl::reqd_work_group_size(16)]] {
        //[kernel code]
      });
});

// Kernel defined as a functor to be invoked later
class KernelFunctor {
  public:
  void operator()(item<1> it) const [[sycl::reqd_work_group_size(16)]] {
    //[kernel code]
  };
};
```

### 5.7.3 Deprecated attribute syntax

The SYCL 1.2.1 specification (superseded by this version) defined two mechanisms for kernel attributes to be specified, which are deprecated in this version of SYCL. The old syntaxes are supported but will be removed in a future version, and are therefore not recommended for use. Specifically, the following two attribute syntaxes defined by the SYCL 1.2.1 specification are deprecated:

1. The attribute syntax defined by the OpenCL C specification within device code (`__attribute__((attrib))`).

2. The C++ attribute specifier syntax in the `[[cl::]]` namespace applied to device functions (not the function-type of a kernel function), including automatic propagation of the attribute to any caller of such device functions.

## 5.8 Address-space deduction

C++ has no type-level support to represent address spaces. As a consequence, the SYCL generic programming model does not directly affect the C++ type of unannotated pointers and references.

Source level guarantees about addresse spaces in the SYCL generic programming model can only be achieved using pointer classes (instances of `multi_ptr`), which are regular classes that represent pointers to data stored in the corresponding address spaces.

In SYCL, the address space of pointer and references are derived from:

- Accessors that give access to shared data. They can be bound to a memory object in a command group and passed into a kernel. Accessors are used in scheduling of kernels to define ordering. Accessors to buffers have a compile-time address space based on their access mode.
- Explicit pointer classes (e.g. `global_ptr`) holds a pointer which is known to be addressing the address space represented by the `access::address_space`. This allows the compiler to determine whether the pointer references global, local, constant or private memory and generate code accordingly.
- Raw C++ pointer and reference types (e.g. `int*`) are allowed within SYCL kernels. They can be constructed from the address of local variables, explicit pointer classes, or accessors.

### 5.8.1 Address space assignement

In order to understand where data lives, the device compiler is expected to assign address spaces while lowering types for the underlying target based on the context. Depending on the SYCL backends and mode, address space deducing rules differ slightly.

If the target of the SYCL backend can represent the generic address space, then the common address space deduction rules and generic as default address space rules apply. If the target of the SYCL backend cannot represent the generic address space, then the common address space deduction rules and inferred address space rules apply.

Note: SYCL address space does not affect the type, address space shall be understood as memory segment in which data is allocated. For instance, if `int i;` is allocated to the global address space, then `decltype(&i)` shall evaluate to `int*`.

### 5.8.2 Common address space deduction rules

The variable declarations get assigned to a address space depending on their scope and storage class:

- Namespace scope
  - The declaration is assigned to global address space if the type is not `const`
  - The declaration is assigned to constant address space if the type is `const`
- Block scope and function parameter scope

  - Declarations with static storage are treated the same way as variables in namespace scope
  - Otherwise the declaration is assigned to the local address space if declared in a hierarchical context
  - Otherwise the declaration is assigned to the private address space

- Class scope:
  - Static data members are treated the same way as for variable in namespace scope

The result of a prvalue-to-xvalue conversion is assigned to the local address space if it happens in a hierarchical context or to the private address space otherwise.

### 5.8.3 Generic as default address space

Unannotated pointers and references are considered to be pointing to the generic address space.

### 5.8.4 Inferred address space

**Note for this provisional version:** The address space deduction feature described next is being reworked to better align with addition of generic address space and generic as default address space.

Inside kernels, the SYCL device compiler will need to auto-deduce the memory region of unannotated pointer and reference types during the lowering of types from C++ to the underlying representation.

If a kernel function or device function contains a pointer or reference type, then the address space deduction must be attempted using the following rules:

- If an explicit pointer class is converted into a C++ pointer value, then the C++ pointer value will point to same address space as the one represented by the explicit pointer class.
- If a variable is declared as a pointer type, but initialized in its declaration to a pointer value with an already-deduced address space, then that variable will have the same address space as its initializer.
- If a function parameter is declared as a pointer type, and the argument is a pointer value with a deduced address space, then the function will be compiled as if the parameter had the same address space as its argument. It is legal for a function to be called in different places with different address spaces for its arguments: in this case the function is said to be "duplicated" and compiled multiple times. Each duplicated instance of the function must compile legally in order to have defined behavior.
- If a function return type is declared as a pointer type and return statements use address space deduced expressions, then the function will be compiled as if the return type had the same address space. To compile legally, all return expressions must deduce to the same address space.
- The rules for pointer types also apply to reference types. i.e. a reference variable takes its address space from its initializer. A function with a reference parameter takes its address space from its argument.
- If no other rule above can be applied to a declaration of a pointer, then it is assumed to be in the `private` address space.

It is illegal to assign a pointer value addressing one address space to a pointer variable addressing a different address space.

# 5.9 SYCL offline linking

## 5.9.1 SYCL functions and methods linkage

The default behavior in SYCL applications is that all the definitions and declarations of the functions and methods are available to the SYCL compiler, in the same translation unit. When this is not the case, all the symbols that need to be exported to a SYCL library or from a C++ library to a SYCL application need to be defined using the macro: `SYCL_EXTERNAL`.

The `SYCL_EXTERNAL` macro will only be defined if the implementation supports offline linking. The macro is implementation-defined, but the following restrictions apply:

- `SYCL_EXTERNAL` can only be used on functions;
- if the SYCL backend does not support the generic address space then the function cannot use raw pointers as parameter or return types. Explicit pointer classes must be used instead;
- externally defined functions cannot call a `sycl::parallel_for_work_item` method;
- externally defined functions cannot be called from a `sycl::parallel_for_work_group` scope.

The SYCL linkage mechanism is optional and implementation defined.

# 6. SYCL Extensions

This section describes the mechanism by which the SYCL core specification can be extended. An extension can be either of two flavors: an extension ratified by the Khronos SYCL group or a vendor supplied extension. In both cases, an extension is an optional feature set which an implementation need not implement in order to be conformant with the core SYCL specification.

Vendors may choose to define extensions in order to expose custom features or to gather feedback on an API that is not yet ready for inclusion in the core specification. Since the APIs for extensions may change as feedback is gathered, the extension mechanism includes a way for application developers to test for the API version of each extension. Once a vendor extension has stabilized, vendors are encouraged to promote it to a future version of the core SYCL specification. Thus, extensions can be viewed as a pipeline of features for consideration in future SYCL versions.

This section does not describe any particular extension to SYCL. Rather, it just describes the *mechanism* for defining an extension. Each extension is defined by its own separate document. If an extension is ratified by the Khronos SYCL group, that group will release a document describing the extension. If a vendor defines an extension, the vendor is responsible for releasing its documentation.

## 6.1 Definition of an extension

An extension can be implemented by adding new types or free functions in a specific namespace, by adding functionality to an existing class that is defined in the core SYCL specification, or through a combination of the two.

New types or free functions for Khronos ratified extensions are defined in the namespace ::sycl::khr::<extensionname>. For example, ::sycl::khr::fancy could be the namespace for a Khronos extension named "fancy".

If a vendor specific extension adds new types or free functions, the vendor is encouraged to define them in the namespace ::sycl::ext::<vendorname> and they are encouraged to add another namespace layer according to the name of the extension. For example, ::sycl::ext::acme::fancier could be the namespace for an extension from the Acme vendor. However, vendors may also choose to define new types and free functions in another top-level namespace that is outside of ::sycl. This might be more appropriate, for example, when an extension integrates features from an existing non-SYCL API. A vendor may not define new types or free functions underneath ::sycl, unless they are in ::sycl::ext::<vendorname>.

*[Note: Vendors are discouraged from defining top level namespaces that start with the word "sycl" because we believe that application developers may want to use namespaces like this as namespace aliases. – end note]*

Extensions may only add functionality to existing SYCL classes in a limited way. When a Khronos ratified extension needs to add functionality to an existing class, it does so by adding a method named khr() to that class. For example, an extension on the device class would add a method like this:

```
class device {
```

```
  // ...
  sycl::khr::device khr();
};
```

The `khr()` method returns an object, and that object provides methods that are part of the extension.

Likewise, a vendor specific extension may add functionality to an existing SYCL class by adding a method named `ext_<vendorname>()` (e.g. `ext_acme()` for the Acme vendor) like this:

```
class device {
  // ...
  sycl::ext::acme::device ext_acme();
};
```

One motivation for this pattern is to reduce verbosity of application code that uses an extension and to facilitate application migration when an extension is promoted to the core SYCL specification. Consider the following application code:

```
void foo(sycl::device dev) {
  dev.ext_acme().fancy();
};
```

If the extension "fancy" is later promoted to the core SYCL specification, the application need only remove the call to `ext_acme()` in order to migrate the application.

Extensions may also add C++ attributes. The attribute namespace `sycl::` is reserved for attributes in the core SYCL specification and for Khronos ratified extensions. Vendor defined extensions should use a different attribute namespace.

Applications must include a special header file in order to get declarations for the types and free functions of an extension. Each Khronos ratified extension has an associated header named `"SYCL/khr/<extensionname>.hpp"`.

The include path `"SYCL/ext/<vendorname>"` is reserved for vendor extensions. Vendors can choose to provide a single header for all extensions or to provide separate headers for each extension. For example, the Acme vendor could provide the header `"SYCL/ext/acme/extensions.hpp"` for access to all of its extensions. As with namespaces, vendors are encouraged to define header files in `"SYCL/ext/<vendorname>"`, but a vendor may also define header files in another file system path that is outside of the `"SYCL"` directory. Vendors may not define header files in the `"SYCL"` path unless they are underneath `"SYCL/ext/<vendorname>"`.

## 6.2 Predefined macros

Each Khronos ratified extension has a corresponding feature test macro of the form `SYCL_KHR_<extensionname>` whose value follows the C++20 pattern for language feature test macros. The value is a number with 6 decimal digits in YYYYMM format identifying the year and month the extension was first adopted or the date the extension was last updated. An implementation must predefine this macro only if it implements the extension, so applications can use the macro in order to determine if the extension is available.

If an implementation provides a vendor specific extension, it should also predefine a feature test macro of the form `SYCL_EXT_<vendorname>_<extensionname>` (e.g. `SYCL_EXT_ACME_FANCY`). The value of the macro must be an integer that monotonically increases for each version of the extension, and vendors are encouraged to use the same YYYYMM format described above.

*[Note: The feature test macros are defined uniformly across all parts of a SYCL application, just like any macro. If an implementation uses SMCP, all compiler passes predefine a particular feature test macro the same way, regardless of whether that compiler pass's device supports the feature. Thus, the feature test macros cannot be used to determine whether any particular device supports a feature. If the feature is device-specific, the application must use `device::has()` or `platform::has()` to test the feature's aspect in order to determine whether a particular device supports the feature. – end note]*

Each vendor's implementation must also predefine a macro of the form `SYCL_VENDOR_<vendorname>` (e.g. `SYCL_VENDOR_ACME`), which applications can use to determine whether they are being compiled by that vendor's toolchain.

An implementation, of course, is allowed to predefine additional macros too. However, an implementation may not predefine a macro whose name starts with `SYCL` unless it starts with `SYCL_EXT_<vendorname>` or `SYCL_VENDOR_<vendorname>`.

## 6.3 Device aspects and conditional features

An extension may define additional device aspects and it may provide features which are only available on devices with certain aspects. If it does so, the extension documentation must describe which aspects enable these conditional features. If an extension provides a new enumerated `aspect` value, the type of the new value must be `::sycl::aspect` but the enumerated value must be in the extension's namespace scope. For example, a Khronos ratified extension could add a new `aspect` value like this:

```
namespace sycl {
namespace khr {
namespace aspect {

static constexpr auto foo = static_cast<sycl::aspect>(1000);

} // namespace aspect
} // namespace khr

template<> struct is_aspect_active<khr::aspect::foo> : std::true_type {};

} // namespace sycl
```

A vendor extension could add an `aspect` value in a similar way:

```
namespace sycl {
namespace ext {
namespace acme {
namespace aspect {

static constexpr auto bar = static_cast<sycl::aspect>(-1);

} // namespace aspect
} // namespace acme
} // namespace ext

```

```
template<> struct is_aspect_active<ext::acme::aspect::bar> : std::true_type {};

} // namespace sycl
```

In the examples above, the vendor has decided to implement aspects from Khronos ratified extensions starting at 1000 and to implement vendor specific aspects as negative integers. However, these are just example implementation details. The SYCL specification does not prescribe the numerical value of any `aspect`.

## 6.4 Backends

A vendor extension may define a new SYCL backend. If it does so, the enumerated value for the `backend` should be defined in the extension's namespace, similar to the way an extended `aspect` is defined:

```
namespace sycl {
namespace ext {
namespace acme {
namespace backend {

static constexpr auto foo = static_cast<sycl::backend>(-1);

} // namespace backend
} // namespace acme
} // namespace ext
} // namespace sycl
```

The backend's interoperability API should be made available through a header named `"SYCL/ext/<vendorname>/backend/<backendname>.hpp"` and it should be defined in the namespace `::sycl::ext::<vendorname>::<backendname>`. The implementation should also predefine a macro of the form `SYCL_EXT_<vendorname>_BACKEND_<backendname>` when the backend is active.

## 6.5 Conditional features and compilation errors

SYCL applications are allowed to contain kernels for heterogeneous devices and those kernels, of course, are allowed to use features that are available only on certain devices. Applications are responsible for ensuring that a kernel using such a feature is never submitted to a device that does not support the feature and is never compiled for a device that does not support the feature (e.g. via the `module` `build()` or `compile()` functions). If an application fails to adhere to this requirement, the implementation raises a `feature_not_supported` exception.

*[Note: If an implementation defines a compiler flag that causes some kernels to be pre-compiled for some devices, the vendor is responsible for defining the semantics about when errors are reported for kernels that use device specific extensions. – end note]*

An implementation may not raise a spurious error as a result of speculative compilation of a kernel for a device when the application did not specifically ask to submit the kernel to that device or to compile the kernel for that device. To clarify, consider the following example. An application with kernels K1 and K2 runs on devices D1 and D2. Kernel K1 uses extensions specific to D1, and kernel K2 uses extensions specific to D2. The application is coded to ensure that K1 is only submitted to D1 and that K2 is only submitted to D2. An implementation may not raise errors due to speculative compilation of K1 for device D2 or for compilation of K2 for device D1.

An implementation is required, however, to raise an error for a kernel that is not valid for any device. Therefore an implementation must raise an error for a kernel K that is invalid for all devices, even if the application is coded such that kernel K is never submitted to any device.

# A. Information descriptors

The purpose of this chapter is to include all the headers of the memory object descriptors, which are described in detail in Chapter 4, for platform, context, device, and queue.

## A.1 Platform information descriptors

The following interface includes all the information descriptors for the `platform` class as described in Table 4.12.

```
namespace sycl {
namespace info {
enum class platform : unsigned int {
  profile,
  version,
  name,
  vendor,
  extensions // Deprecated
};
}  // namespace info
}  // namespace sycl
```

## A.2 Context information descriptors

The following interface includes all the information descriptors for the `context` class as described in Table 4.15.

```
namespace sycl {
namespace info {
enum class context : int {
  reference_count,
  platform,
  devices,
  atomic_memory_order_capabilities,
  atomic_fence_order_capabilities
  atomic_memory_scope_capabilities,
  atomic_fence_scope_capabilities
};
}  // info
}  // sycl
```

## A.3 Device information descriptors

The following interface includes all the information descriptors for the `device` class as described in Table 4.19.

```
namespace sycl {
namespace info {

enum class device : int {
  device_type,
  vendor_id,
  max_compute_units,
  max_work_item_dimensions,
  max_work_item_sizes,
  max_work_group_size,
  preferred_vector_width_char,
  preferred_vector_width_short,
  preferred_vector_width_int,
  preferred_vector_width_long,
  preferred_vector_width_float,
  preferred_vector_width_double,
  preferred_vector_width_half,
  native_vector_width_char,
  native_vector_width_short,
  native_vector_width_int,
  native_vector_width_long,
  native_vector_width_float,
  native_vector_width_double,
  native_vector_width_half,
  max_clock_frequency,
  address_bits,
  max_mem_alloc_size,
  image_support, // Deprecated
  max_read_image_args,
  max_write_image_args,
  image2d_max_height,
  image2d_max_width,
  image3d_max_height,
  image3d_max_width,
  image3d_max_depth,
  image_max_buffer_size,
  image_max_array_size,
  max_samplers,
  max_parameter_size,
  mem_base_addr_align,
  half_fp_config,
  single_fp_config,
  double_fp_config,
  global_mem_cache_type,
  global_mem_cache_line_size,
  global_mem_cache_size,
  global_mem_size,
  max_constant_buffer_size,
  max_constant_args,
  local_mem_type,
  local_mem_size,
  error_correction_support,
  host_unified_memory,
  atomic_memory_order_capabilities,
  atomic_fence_order_capabilities,
```

```
  atomic_memory_scope_capabilities,
  atomic_fence_scope_capabilities,
  profiling_timer_resolution,
  is_endian_little,
  is_available,
  is_compiler_available, // Deprecated
  is_linker_available, // Deprecated
  execution_capabilities,
  queue_profiling, // Deprecated
  built_in_kernels,
  platform,
  name,
  vendor,
  driver_version,
  profile,
  version,
  backend_version,
  aspects,
  extensions, // Deprecated
  printf_buffer_size,
  preferred_interop_user_sync,
  parent_device,
  partition_max_sub_devices,
  partition_properties,
  partition_affinity_domains,
  partition_type_property,
  partition_type_affinity_domain,
  reference_count
};

enum class device_type : unsigned int {
  cpu,         // Maps to OpenCL CL_DEVICE_TYPE_CPU
  gpu,         // Maps to OpenCL CL_DEVICE_TYPE_GPU
  accelerator, // Maps to OpenCL CL_DEVICE_TYPE_ACCELERATOR
  custom,      // Maps to OpenCL CL_DEVICE_TYPE_CUSTOM
  automatic,   // Maps to OpenCL CL_DEVICE_TYPE_DEFAULT
  host,
  all          // Maps to OpenCL CL_DEVICE_TYPE_ALL
};

enum class partition_property : int {
  no_partition,
  partition_equally,
  partition_by_counts,
  partition_by_affinity_domain
};

enum class partition_affinity_domain : int {
  not_applicable,
  numa,
  L4_cache,
  L3_cache,
  L2_cache,
  L1_cache,
  next_partitionable
```

```
};

enum class local_mem_type : int { none, local, global };

enum class fp_config : int {
  denorm,
  inf_nan,
  round_to_nearest,
  round_to_zero,
  round_to_inf,
  fma,
  correctly_rounded_divide_sqrt,
  soft_float
};

enum class global_mem_cache_type : int { none, read_only, read_write };

enum class execution_capability : unsigned int {
  exec_kernel,
  exec_native_kernel
};

} // namespace info
} // namespace sycl
```

## A.4 Queue information descriptors

The following interface includes all the information descriptors for the `queue` class as described in Table 4.23.

```
namespace sycl {
namespace info {
enum class queue : int {
  context,
  device,
  reference_count
};
} // namespace info
} // namespace sycl
```

## A.5 Kernel information descriptors

The following interface includes all the information descriptors for the `kernel` class as described in Table 4.98.

```
namespace sycl {
namespace info {
enum class kernel: int {
  function_name,
  num_args,
  context,
  module,
  reference_count,
```

```
  attributes
};

enum class kernel_work_group: int {
 global_work_size,
 work_group_size,
 compile_work_group_size,
 preferred_work_group_size_multiple,
 private_mem_size
};

enum class kernel_device_specific: int {
 global_work_size,
 work_group_size,
 compile_work_group_size,
 preferred_work_group_size_multiple,
 private_mem_size,
 max_num_sub_groups,
 compile_num_sub_groups,
 max_sub_group_size,
 compile_sub_group_size
};

}  // namespace info
}  // namespace sycl
```

## A.6 Event information descriptors

The following interface includes all the information descriptors for the `event` class as described in Table 4.98 and Table 4.29.

```
namespace sycl {
namespace info {
enum class event: int {
  command_execution_status,
  reference_count
};

enum class event_command_status : int {
  submitted,
  running,
  complete
};

enum class event_profiling : int {
  command_submit,
  command_start,
  command_end
};
}  // namespace info
}  // namespace sycl
```

# B. Feature sets

As of SYCL 2020 there are now two distinct feature sets which a SYCL implementation can conform to, in order to better fit the requirements of different domains, such as embedded, mobile, and safety critical, which may have limitations because of the toolchains used.

A SYCL implementation can choose to conform to either the full feature set or the reduced feature set.

## B.1 Full feature set

The full feature set includes all features specified in the core SYCL specification with no exceptions.

## B.2 Reduced feature set

The reduced feature set marks certain features as optional or restricted to specific form.

The full feature set is a subsumption of the reduced feature set, such that any set of applications developed for the reduced feature would be compatible with a SYCL full implementation, but any set of applications developed for the full feature set would not necessarily be compatible with a SYCL full implementation.

The reduced feature set makes the following changes to the the requirements laid out in the core SYCL specification.

1. **Un-named SYCL kernel functions:** SYCL kernel functions which are are defined using a lambda expression and therefore have no standard name are required to be provided a name via the kernel name template parameter of kernel invocation functions such as `parallel_for`. This overrides the core SYCL specification rules for SYCL kernel function naming as specified in Section 4.10.7.

## B.3 Compatibility

In order to avoid introducing any kind of divergence the reduced and full feature sets are defined such that the full feature set is a subsumption of the reduced feature set. This means that any applications which are developed for the reduced feature will be compatible with both a SYCL reduced implementation and a SYCL full implementation.

## B.4 Conformance

One of the reasons for having this be defined in the specification is that hardware vendors which wish to support SYCL on their platform(s) want to be able to demonstrate their support for it by passing conformance. However, if passing conformance means adopting features which they do not believe to be necessary at an additional development effort then this may deter them.

Each feature set has its own route for passing conformance allowing adopters of SYCL to specify the feature set they wish to test conformance against. The conformance test suite would then alter or disable the tests within the test suite according to how the feature sets are differentiated above.

# C. Host backend specification

This chapter describes how the SYCL is mapped on the SYCL host backend. The SYCL host backend exposes the host where the SYCL application is executing as a platform to dispatch SYCL kernels. The SYCL host backend exposes at least one SYCL host device.

## C.1 Mapping of the SYCL programming model on the host

The SYCL host device implements all functionality required to execute the SYCL kernels directly on the host, without relying on a third party API. It has full SYCL capabilities and reports them through the SYCL information retrieval interface. At least one SYCL host device must be exposed in the SYCL host backend in all SYCL implementations, and it must always be available. Any C++ application debugger, if available on the system, can be used for debugging SYCL kernels executing on a SYCL host device.

When a SYCL implementation executes kernels on the host device, it is free to use whatever parallel execution facilities available on the host, as long as it executes within the semantics of the kernel execution model defined by the SYCL kernel execution model.

Kernel math library functions on the host must conform to OpenCL math precision requirements. The SYCL host device needs to be queried for the capabilities it provides. This ensures consistency when executing any SYCL general application.

The SYCL host device must report as supporting images and therefore support the minimum image formats.

The range of image formats supported by the host device is implementation-defined, but must match the minimum requirements of the OpenCL specification.

SYCL implementors can provide extensions on the host-device to match any other backend-specific extension. This allows developers to rely on the host device to execute their programs when said backend is not available.

### C.1.1 SYCL memory model on the host

All SYCL device memories are available on devices from the host backend.

| SYCL | Host |
|---|---|
| Global | System memory |
| Constant | System memory |
| Local | System memory |
| Private | Stack |

Table C.1: Mapping of SYCL memory regions into host memory regions.

## C.2 Interoperability with the host application

The host backend must ensure all functionality of the SYCL generic programming model is always available to developers. However, since there is no heterogeneous API behind the host backend (it directly targets the host platform), there is no native types for SYCL objects to map to in the SYCL application.

Inside SYCL kernels, the host backend must ensure interoperability with existing host code, so that existing host libraries can be used inside SYCL kernels executing on the host. In particular, when retrieving a raw pointer from a multi pointer object, the pointer returned must be usable by any library accessible by the SYCL application.

# D. OpenCL backend specification

This chapter describes how the SYCL general programming model is mapped on top of OpenCL, and how the SYCL generic interoperability interface must be implemented by vendors providing SYCL for OpenCL implementations to ensure SYCL applications written for the OpenCL backend are interoperable.

## D.1 SYCL for OpenCL framework

The SYCL framework allows applications to use a host and one or more OpenCL devices as a single heterogeneous parallel computer system. The framework contains the following components:

- SYCL C++ template library: The template library provides a set of C++ templates and classes which provide the programming model to the user. It enables the creation of runtime classes such as SYCL queues, buffers and images, as well as access to some underlying OpenCL runtime object, such as contexts, platforms, devices and program objects.

- SYCL runtime: The SYCL runtime interfaces with the underlying OpenCL implementations and handles scheduling of commands in queues, moving of data between host and devices, manages contexts, programs, kernel compilation and memory management.

- *OpenCL Implementation(s)*: The SYCL system assumes the existence of one or more OpenCL implementations available on the host machine. If no OpenCL implementation is available, then the SYCL implementation provides only the SYCL host device to run kernels on.

- SYCL device compilers: The SYCL device compilers compile SYCL C++ kernels into a format which can be executed on an OpenCL device at runtime. There may be more than one SYCL device compiler in a SYCL implementation. The format of the compiled SYCL kernels is not defined. A SYCL device compiler may, or may not, also compile the host parts of the program.

The OpenCL backend is enabled using the `sycl::backend::opencl` value of `enum class backend`. That means that when the OpenCL backend is active, the value of `sycl::is_backend_active<sycl::backend::opencl>::value` will be `true`.

## D.2 Mapping of SYCL programming model on top of OpenCL

The SYCL programming model was originally designed as a high-level model for the OpenCL API, hence the mapping of SYCL on the OpenCL API is mostly straightforward.

When the OpenCL backend is active on a SYCL Application, all visible OpenCL platforms are exported as SYCL platforms.

When a SYCL implementation executes kernels on an OpenCL device, it achieves this by enqueuing OpenCL **commands** to execute computations on the processing elements within a device. The processing elements within an OpenCL compute unit may execute a single stream of instructions as ALUs within a SIMD unit (which execute

| SYCL | OpenCL |
|---|---|
| Global | Global memory |
| Constant | Constant memory |
| Local | Local memory |
| Private | Private memory |

Table D.1: Mapping of SYCL memory regions into OpenCL memory regions.

in lockstep with a single stream of instructions), as independent SPMD units (where each PE maintains its own program counter) or as some combination of the two.

### D.2.1 Platform mixed version support

The SYCL system presents the user with a set of devices, grouped into some number of platforms. The device version is an indication of the device's capabilities, as represented by the device information returned by the `sycl::device::get_info()` method. Examples of attributes associated with the device version are resource limits and information about functionality beyond the core SYCL specification's requirements. The version returned corresponds to the highest version of the OpenCL specification for which the device is conformant, but is not higher than the version of the device's platform which bounds the overall capabilities of the runtime operating the device.

### D.2.2 OpenCL memory model

The memory model for SYCL Devices running on OpenCL platforms follows the memory model of the OpenCL version they conform to. Work-items executing in a kernel have access to four distinct memory regions, with the mapping between SYCL and OpenCL described in table D.1.

### D.2.3 OpenCL resources managed by SYCL application

In OpenCL, a developer must create a context to be able to execute commands on a device. Creating a context involves choosing a platform and a list of devices. In SYCL, contexts, platforms and devices all exist, but the user can choose whether to specify them or have the SYCL implementation create them automatically. The minimum required object for submitting work to devices in SYCL is the queue, which contains references to a platform, device and context internally.

The resources managed by SYCL are:

1. Platforms: all features of OpenCL are implemented by platforms. A platform can be viewed as a given hardware vendor's runtime and the devices accessible through it. Some devices will only be accessible to one vendor's runtime and hence multiple platforms may be present. SYCL manages the different platforms for the user. In SYCL, a platform resource is accessible through a `sycl::platform` object. SYCL also provides a host platform object, which only contains a single host device.

2. Contexts: any OpenCL resource that is acquired by the user is attached to a context. A context contains a collection of devices that the host can use and manages memory objects that can be shared between the devices. Data movement between devices within a context may be efficient and hidden by the underlying OpenCL runtime while data movement between contexts may involve the host. A given context can only wrap devices owned by a single platform. In SYCL, a context resource is accessible through a `sycl::context` object.

3. Devices: platforms provide one or more devices for executing kernels. In SYCL, a device is accessible through a `sycl::device` object.

4. Kernels: the SYCL functions that run on SYCL devices (i.e. either an OpenCL device, or the host device) are defined as C++ function objects (a named function object type or a lambda function).

5. Modules: OpenCL objects that store implementation data for the SYCL kernels. These objects are only required for advanced use in SYCL and are encapsulated in the `sycl::program` class.

6. Queues: SYCL kernels execute in command queues. The user must create a queue, which references an associated context, platform and device. The context, platform and device may be chosen automatically, or specified by the user. In SYCL, command queues are accessible through `sycl::queue` objects.

## D.3 Interoperability with the OpenCL API

The OpenCL backend for SYCL ensures maximum compatibility between SYCL and OpenCL kernels and API. This includes supporting devices with different capabilities and support for different versions of the OpenCL C language, in addition to supporting SYCL kernels written in C++.

SYCL runtime classes which encapsulate an OpenCL opaque type such as SYCL `context` or SYCL `queue` must provide an interoperability constructor taking an instance of the OpenCL opaque type. These constructors must retain that instance to increase the reference count of the OpenCL resource.

The destructor for the SYCL runtime classes which encapsulate an OpenCL opaque type must release that instance to decrease the reference count of the OpenCL resource.

Note that an instance of a SYCL runtime class which encapsulates an OpenCL opaque type can encapsulate any number of instances of the OpenCL type, unless it was constructed via the interoperability constructor, in which case it can encapsulate only a single instance of the OpenCL type.

The lifetime of a SYCL runtime class that encapsulates an OpenCL opaque type and the instance of that opaque type retrieved via the `get()` member function are not tied in either direction given correct usage of OpenCL reference counting. For example if a user were to retrieve a `cl_command_queue` instance from a SYCL `queue` instance and then immediately destroy the SYCL `queue` instance, the `cl_command_queue` instance is still valid. Or if a user were to construct a SYCL `queue` instance from a `cl_command_queue` instance and then immediately release the `cl_command_queue` instance, the SYCL `queue` instance is still valid.

Note that a SYCL runtime class that encapsulates an OpenCL opaque type is not responsible for any incorrect use of OpenCL reference counting outside of the SYCL runtime. For example if a user were to retrieve a `cl_command_queue` instance from a SYCL `queue` instance and then release the `cl_command_queue` instance more than once without any prior retain then the SYCL `queue` instance that the `cl_command_queue` instance was retrieved from is now undefined.

Note that an instance of the SYCL `buffer` or SYCL `image` class templates constructed via the interoperability constructor is free to copy from the `cl_mem` into another memory allocation within the SYCL runtime to achieve normal SYCL semantics, for as long as the SYCL `buffer` or SYCL `image` instance is alive.

Table D.2 relates SYCL objects to their OpenCL native type in the SYCL Application.

| OpenCL backend native types | Description |
|---|---|
| device<br>cl_device_id | |
| context<br>cl_context | A SYCL context object encapsulates an OpenCL context object. |
| program<br>cl_program | When a SYCL program is constructed for the OpenCL backend, this maps directly to an OpenCL program object. |
| kernel<br>cl_kernel | The SYCL implementation will produce OpenCL programs from the SYCL Device kernels. They are dispatched on the OpenCL interface as OpenCL kernel objects. This also apply to built-in kernels. |
| event<br>std::vector<cl_event> | A SYCL event can encapsulate one or multiple OpenCL events, representing a number of dependencies in the same or different contexts, that must be satisfied for the SYCL event to be complete. |
| buffer<br>std::vector<cl_mem> | SYCL buffers containing OpenCL memory objects can handle multiple cl_mem objects in the same or different context. The interoperability interface will return a list of active buffers in the SYCL runtime. |
| image<br>std::vector<cl_mem> | SYCL images containing OpenCL image objects can handle multiple underlying cl_mem objects at the same time in the same or different OpenCL contexts. The interoperability interface will return a list of active images in the SYCL runtime. |
| | End of table |

Table D.2: List of native types per SYCL object in the OpenCL backend.

The user can also extract OpenCL cl_kernel and cl_program objects for kernels by providing the type name of the kernel.

Inside the SYCL kernel, the SYCL API offers interoperability with OpenCL device types. The table D.3 describes the mapping of kernel types.

| SYCL kernel native types in OpenCL | Description |
|---|---|
| multi_ptr::get() | Returns a pointer in the OpenCL address space corresponding to the type of multi pointer object |
| device_event::get() | Returns an event_t object, which can be used to identify copies from global to local memory and vice-versa |
| . | |
| | End of table |

Table D.3: List of native types per SYCL object on Kernel code.

When a buffer or image is allocated on more than one OpenCL device, if these devices are on separate contexts then multiple `cl_mem` objects may be allocated for the memory object, depending on whether the object has actively been used on these devices yet or not.

Some types in SYCL vary according to pointer size or vary on the host according to the host ABI, such as `size_t` or `long`. In order for the the SYCL device compiler to ensure that the sizes of these types match the sizes on the host and to enable data of these types to be shared between host and device, the OpenCL interoperability types are defined, `sycl::cl_int` and `sycl::cl_size_t`.

The OpenCL C function qualifier `__kernel` and the access qualifiers: `__read_only`, `__write_only` and `__read_write` are not exposed in SYCL via keywords, but are instead encapsulated in SYCL's parameter passing system inside accessors. Users wishing to achieve the OpenCL equivalent of these qualifiers in SYCL should instead use SYCL accessors with equivalent semantics.

Any OpenCL C function included in a pre-built OpenCL library can be defined as an `extern "C"` function and the OpenCL program has to be linked against any SYCL program that contains kernels using the external function. In this case, the data types used have to comply with the interoperability aliases defined in D.7.

## D.4 Programming interface

The following section describes the OpenCL-specific API. All free functions are available in the `sycl::opencl` namespace.

| OpenCL interoperability functions | Description |
|---|---|
| `sycl::context make_context (cl_context, async_handler asyncHandler = {})` | Constructs a SYCL `context` instance from an OpenCL `cl_context` in accordance with the requirements described in 4.5.2. |
| `cl_context get_interop(sycl::context)` | Returns a valid `cl_context` instance in accordance with the requirements described in 4.5.2. |
| `sycl::event make_event (cl_event clEvent, const context& syclContext)` | Constructs a SYCL `event` instance from an OpenCL `cl_event` in accordance with the requirements described in 4.5.2. The `syclContext` must match the OpenCL context associated with the clEvent. |
| `cl_event get_interop(sycl::event)` | Returns a valid `cl_event` instance in accordance with the requirements described in 4.5.2. |
| `sycl::device make_device(cl_device_id deviceId)` | Constructs a SYCL `device` instance from an OpenCL `cl_device_id` in accordance with the requirements described in 4.5.2. |
| `cl_device_id get_interop(sycl::device)` | Returns a valid `cl_device_id` instance in accordance with the requirements described in 4.5.2. |
| `sycl::queue make_queue(cl_command_queue clQueue, const context &syclContext, const async_handler &asyncHandler = {})` | Constructs a SYCL `queue` instance with an optional `async_handler` from an OpenCL `cl_command_queue` in accordance with the requirements described in 4.5.2. |

| OpenCL interoperability functions | Description |
|---|---|
| `sycl::buffer make_buffer(cl_mem clMemObject, const context &syclContext, event availableEvent = {})` | Available only when: `dimensions == 1`. Constructs a SYCL `buffer` instance from an OpenCL `cl_mem` in accordance with the requirements described in 4.5.2. The instance of the SYCL `buffer` class template being constructed must wait for the SYCL `event` parameter, if one is provided, `availableEvent` to signal that the `cl_mem` instance is ready to be used. The SYCL `context` parameter `syclContext` is the context associated with the memory object. |
| `sycl::image make_image(cl_mem clMemObject, const context &syclContext, event availableEvent = {})` | Constructs a SYCL `image` instance from an OpenCL `cl_mem` in accordance with the requirements described in 4.5.2. The instance of the SYCL `image` class template being constructed must wait for the SYCL `event` parameter, if one is provided, `availableEvent` to signal that the `cl_mem` instance is ready to be used. The SYCL `context` parameter `syclContext` is the context associated with the memory object. |
| `sycl::sampler make_sampler(cl_sampler clSampler, const context &syclContext)` | Constructs a SYCL `sampler` instance from an OpenCL `cl_sampler` in accordance with the requirements described in 4.5.2. |
| `sycl::kernel kernel(cl_kernel clKernel, const context& syclContext)` | Constructs a SYCL `kernel` instance from an OpenCL `cl_kernel` in accordance with the requirements described in 4.5.2. The SYCL `context` must represent the same underlying OpenCL context associated with the OpenCL kernel object. |
| `cl_kernel get()const` | Returns a valid `cl_kernel` instance in accordance with the requirements described in 4.5.2. |

### D.4.1 Reference counting

All OpenCL objects are reference counted. The SYCL general programming model doesnt require that native objects are reference counted. However, for convenience, the following function is provided in the `sycl::opencl` namespace.

| Reference counting | Description |
|---|---|
| `template <typename openCLT> cl_uint get_reference_count(openCLT obj)` | Returns the reference counting of the given object |

### D.4.2 Errors and limitations

If there is an OpenCL error associated with an exception triggered, then the OpenCL error code can be obtained by the free function `cl_int sycl::opencl::get_error_code(sycl::exception&)`. In the case where there is no

OpenCL error associated with the exception triggered, the OpenCL error code will be CL_SUCCESS.

### D.4.3 Interoperability with modules

In OpenCL [1] any kernel function that is enqueued over an nd-range is represented by a cl_kernel and must be compiled and linked via a cl_program using clBuildProgram, clCompileProgram and clLinkProgram.

For OpenCL SYCL backend this detail is abstracted away by modules and a module object containing all SYCL kernel functions in a translation is retrieved by calling the free function this_module::get.

However, there are cases where it is useful to be able to manually create a module from an input specific to the OpenCL SYCL backend such as OpenCL C source, and intermediate representation/language such as SPIR-V. This can be useful for interoperability with existing OpenCL kernels or libraries or binaries generated by another tool which need to be linked at runtime.

The OpenCL SYCL backend specification provides additional free functions which provide the above functionality, each resulting in a input module which can then be built, compiled and linked as described in 4.13.6.

```
namespace sycl {
namespace opencl {

using binary_blob_t = std::pair<const char*, size_t>;

module<module_state::input> create_module_with_source (context ctx, std::string source);

module<module_state::input> create_module_with_binary(context ctx, binary_blob_t binary);

module<module_state::input> create_module_with_il (context ctx, binary_blob_t il);

module<module_state::input> create_module_with_builtin_kernels (context ctx,
  std::vector<std::string> kernelNames);

}  // namespace opencl
}  // namespace sycl
```

#### D.4.3.1 Free functions

```
module<module_state::input> create_module_with_source (context ctx, std::string source); // (1)
```

1. *Preconditions:* The context specified by ctx must be associated with the OpenCL SYCL backend. The OpenCL C source specified by source must not be an empty string.

   *Effects:* Constructs a module from the provided OpenCL C source specified by source and associated with the context specified by ctx by invoking the necessary OpenCL APIs.

   *Returns:* A module of module_state::input containing the kernels defined in the OpenCL C source specified by source.

   *Throws:* invalid_object_error if any error is produced by invoking the OpenCL APIs.

```
module<module_state::input> create_module_with_binary(context ctx, binary_blob_t binary); // (1)
```

1. *Preconditions:* The context specified by `ctx` must be associated with the OpenCL SYCL backend. The binary blob specified by `il` must not contain a null pointer or zero size.

   *Effects:* Constructs a module from the provided binary blob specified by `binary` and associated with the context specified by `ctx` by invoking the necessary OpenCL APIs.

   *Returns:* A module of `module_state::input` containing the kernels defined in the binary blob specified by `source`.

   *Throws:* `invalid_object_error` if any error is produced by invoking the OpenCL APIs.

```
module<module_state::input> create_module_with_il (context ctx, binary_blob_t il); // (1)
```

1. *Preconditions:* The context specified by `ctx` must be associated with the OpenCL SYCL backend. The intermediate language specified by `il` must not contain a null pointer or zero size.

   *Effects:* Constructs a module from the provided intermediate language specified by `il` and associated with the context specified by `ctx` by invoking the necessary OpenCL APIs.

   *Returns:* A module of `module_state::input` containing the kernels defined in the binary intermediate language by `il`.

   *Throws:* `invalid_object_error` if any error is produced by invoking the OpenCL APIs.

```
module<module_state::input> create_module_with_builtin_kernels (context ctx, //(1)
  std::vector<std::string> kernelNames);
```

1. *Preconditions:* The context specified by `ctx` must be associated with the OpenCL SYCL backend. The list of names specified by `kernelNames` must not be empty.

   *Effects:* Constructs a module from the provided builtin kernel names specified by `source` and associated with the context specified by `ctx` by invoking the necessary OpenCL APIs.

   *Returns:* A module of `module_state::input` containing the built-in kernels defined by the list of kernel names specified by `source`.

   *Throws:* `invalid_object_error` if any error is produced by invoking the OpenCL APIs.

### D.4.4 Interoperability with kernels

It is possible to construct a kernel from a previously created OpenCL `cl_kernel` by calling the interop free function `make_kernel` defined in 4.5.2.3.

This will create a `kernel` object which can be invoked by any of kernel invocation commands such as `parallel_for` which take a `kernel` but not SYCL kernel function.

Call `make_kernel` must trigger a call to `clRetainKernel` and the resulting `kernel` object must call `clReleaseKernel` on destruction.

The kernel arguments for the OpenCL C kernel kernel can either be set prior to creating the `kernel` object or by calling the `set_arg` member function of the `handler` class.

If kernel arguments are set prior to creating the kernel object the SYCL runtime is not responsible for managing the data of these arguments.

### D.4.5 OpenCL kernel conventions and SYCL

OpenCL and SYCL use opposite conventions for the unit stride dimension. SYCL aligns with C++ conventions, which is important to understand from a performance perspective when porting code to SYCL. The unit stride dimension, at least for data, is implicit in the linearization equations in SYCL (Equation 4.3) and OpenCL. SYCL aligns with C++ array subscript ordering `arr[a][b][c]`, in that range constructor dimension ordering used to launch a kernel (e.g. `range<3> R{a,b,c}`) and range and ID queries within a kernel, are ordered in the same way as the C++ multi-dimensional subscript operators (unit stride on the right).

When specifying a `range` as the global or local size in a `parallel_for` that invokes an OpenCL interop kernel (through `cl_kernel` interop or `compile_with_source`/ `build_with_source`), the highest dimension of the range in SYCL will map to the lowest dimension within the OpenCL kernel. That statement applies to both an underlying enqueue operation such as `clEnqueueNDRangeKernel` in OpenCL, and also ID and size queries within the OpenCL kernel. For example, a 3D global range specified in SYCL as:

```
range<3> R{r0,r1,r2};
```

maps to an `clEnqueueNDRangeKernel` `global_work_size` argument of:

```
size_t cl_interop_range[3] = {r2,r1,r0};
```

Likewise, a 2D global range specified in SYCL as:

```
range<2> R{r0,r1};
```

maps to an `clEnqueueNDRangeKernel` `global_work_size` argument of:

```
size_t cl_interop_range[2] = {r1,r0};
```

The mapping of highest dimension in SYCL to lowest dimension in OpenCL applies to all operations where a multi-dimensional construct must be mapped, such as when mapping SYCL explicit memory operations to OpenCL APIs like `clEnqueueCopyBufferRect`.

Work-item and work-group ID and range queries have the same reversed convention for unit stride dimension between SYCL and OpenCL. For example, with three, two, or one dimensional SYCL global ranges, OpenCL and SYCL kernel code queries relate to the range as shown in Table D.6. The “SYCL kernel query” column applies for SYCL-defined kernels, and the “OpenCL kernel query” column applies for kernels defined through OpenCL interop.

### D.4.6 Data types

The OpenCL C language standard [1, §6.11] defines its own built-in scalar data types, and these have additional requirements in terms of size and signedness on top of what is guaranteed by ISO C++. For the purpose of interoperability and portability, SYCL defines a set of aliases to C++ types within the `sycl::opencl` namespace using the `cl_` prefix. These aliases are described in Table D.7

| SYCL kernel query | OpenCL kernel query | Returned Value |
|---|---|---|
| With enqueued 3D SYCL global `range` of `range<3> R{r0,r1,r2}` | | |
| `nd_item::get_global_range(0)/ item::get_range(0)` | `get_global_size(2)` | `r0` |
| `nd_item::get_global_range(1)/ item::get_range(1)` | `get_global_size(1)` | `r1` |
| `nd_item::get_global_range(2)/ item::get_range(2)` | `get_global_size(0)` | `r2` |
| `nd_item::get_global_id(0)/ item::get_id(0)` | `get_global_id(2)` | Value in range 0..(`r0`-1) |
| `nd_item::get_global_id(1)/ item::get_id(1)` | `get_global_id(1)` | Value in range 0..(`r1`-1) |
| `nd_item::get_global_id(2)/ item::get_id(2)` | `get_global_id(0)` | Value in range 0..(`r2`-1) |
| With enqueued 2D SYCL global `range` of `range<2> R{r0,r1}` | | |
| `nd_item::get_global_range(0)/ item::get_range(0)` | `get_global_size(1)` | `r0` |
| `nd_item::get_global_range(1)/ item::get_range(1)` | `get_global_size(0)` | `r1` |
| `nd_item::get_global_id(0)/ item::get_id(0)` | `get_global_id(1)` | Value in range 0..(`r0`-1) |
| `nd_item::get_global_id(1)/ item::get_id(1)` | `get_global_id(0)` | Value in range 0..(`r1`-1) |
| With enqueued 1D SYCL global `range` of `range<1> R{r0}` | | |
| `nd_item::get_global_range(0)/ item::get_range(0)` | `get_global_size(0)` | `r0` |
| `nd_item::get_global_id(0)/ item::get_id(0)` | `get_global_id(0)` | Value in range 0..(`r0`-1) |

Table D.6: Example range mapping from SYCL enqueued three dimensional global `range` to OpenCL and SYCL queries.

| Scalar data type alias | Description |
|---|---|
| `cl_bool` | Alias to a conditional data type which can be either true or false. The value true expands to the integer constant 1 and the value false expands to the integer constant 0. |
| `cl_char` | Alias to a signed 8-bit integer, as defined by the C++ core language. |
| `cl_uchar` | Alias to an unsigned 8-bit integer, as defined by the C++ core language. |
| `cl_short` | Alias to a signed 16-bit integer, as defined by the C++ core language. |
| `cl_ushort` | Alias to an unsigned 16-bit integer, as defined by the C++ core language. |
| `cl_int` | Alias to a signed 32-bit integer, as defined by the C++ core language. |
| `cl_uint` | Alias to an unsigned 32-bit integer, as defined by the C++ core language. |
| `cl_long` | Alias to a signed 64-bit integer, as defined by the C++ core language. |
| `cl_ulong` | Alias to an unsigned 64-bit integer, as defined by the C++ core language. |

Continued on next page

Table D.7: Scalar data type aliases supported by SYCL OpenCL backend.

| Scalar data type alias | Description |
|---|---|
| cl_float | Alias to a 32-bit floating-point. The float data type must conform to the IEEE 754 single precision storage format. |
| cl_double | Alias to a 64-bit floating-point. The double data type must conform to the IEEE 754 double precision storage format. |
| cl_half | Alias to a 16-bit floating-point. The half data type must conform to the IEEE 754-2008 half precision storage format. An exception with the errc::feature_not_supported error code must be thrown if the half type is used in a SYCL kernel function which executes on a SYCL device that does not support the extension khr_fp16. |
| | End of table |

Table D.7: Scalar data type aliases supported by SYCL OpenCL backend.

## D.5 Preprocessor directives and macros

- SYCL_BACKEND_OPENCL substitutes to one if the OpenCL SYCL backend is active while building the SYCL application.

### D.5.1 Offline linking with OpenCL C libraries

SYCL supports linking SYCL kernel functions with OpenCL C libraries during offline compilation or during online compilation by the SYCL runtime within a SYCL application.

Linking with OpenCL C kernel functions offline is an optional feature and is unspecified. Linking with OpenCL C kernel functions online is performed by using the SYCL program class to compile and link an OpenCL C source; using the compile_with_source or build_with_source member functions.

OpenCL C functions that are linked with, using either offline or online compilation, must be declared as extern "C" function declarations. The function parameters of these function declarations must be defined as the OpenCL C interoperability aliases; pointer of the multi_ptr class template, vector_t of the vec class template and scalar data type aliases described in Table D.7.

For example:

```
extern "C" typename sycl::global_ptr<std::int32_t>::pointer my_func(
    sycl::float4::vector_t x, double y);
```

# D.6 SYCL support of non-core OpenCL features

In addition to the OpenCL core features, SYCL also provides support for OpenCL extensions which provide features in OpenCL via khr extensions.

Some extensions are natively supported within the SYCL interface, however some can only be used via the OpenCL interoperability interface. The SYCL interface required for native extensions must be available. However if the respective extension is not supported by the executing SYCL device, the SYCL runtime must throw an exception with the errc::feature_not_supported error code.

The OpenCL backend exposes khr extensions to SYCL applications through the sycl::aspect enumerated type. Therefore, applications can query for the existence of khr extensions by calling the device::has() or platform ::has() member functions.

All OpenCL extensions are available through the OpenCL interoperability interface, but some can also be used through core SYCL APIs. Table D.8 shows which these are. Table D.8 also shows the mapping from each OpenCL extension name to its associated SYCL device aspect. Note that some aspects are part of the core SYCL specification, and these are in namespace ::sycl::aspect. Other aspects are specific to the OpenCL backend, and these are in namespace ::sycl::opencl::aspect.

| SYCL Aspect | OpenCL Extension | Core SYCL API |
|---|---|---|
| aspect::int64_base_atomics | cl_khr_int64_base_atomics | Yes |
| aspect::int64_extended_atomics | cl_khr_int64_extended_atomics | Yes |
| aspect::fp16 | cl_khr_fp16 | Yes |
| opencl::aspect::3d_image_writes | cl_khr_3d_image_writes | Yes |
| opencl::aspect::khr_gl_sharing | cl_khr_gl_sharing | No |
| opencl::aspect::apple_gl_sharing | cl_apple_gl_sharing | No |
| opencl::aspect::d3d10_sharing | cl_khr_d3d10_sharing | No |
| opencl::aspect::d3d11_sharing | cl_khr_d3d11_sharing | No |
| opencl::aspect::dx9_media_sharing | cl_khr_dx9_media_sharing | No |
| | | End of table |

Table D.8: SYCL support for OpenCL 1.2 extensions.

## D.6.1 Half precision floating-point

The half scalar data type: half and the half vector data types: half1, half2, half3, half4, half8 and half16 must be available at compile-time. However if any of the above types are used in a SYCL kernel function, executing on a device which does not support the extension khr_fp16, the SYCL runtime must throw an exception with the errc::feature_not_supported error code.

The conversion rules for half precision types follow the same rules as in the OpenCL 1.2 extensions specification [5, par. 9.5.1].

The math functions for half precision types follow the same rules as in the OpenCL 1.2 extensions specification [5, par. 9.5.2, 9.5.3, 9.5.4, 9.5.5]. The allowed error in ULP(Unit in the Last Place) is less than 8192, corresponding to Table 6.9 of the OpenCL 1.2 specification [1].

### D.6.2 Writing to 3D image memory objects

The `accessor` class for target `access::target::image` in SYCL support member functions for writing 3D image memory objects, but this functionality is *only allowed* on a device if the extension `cl_khr_3d_image_writes` is supported on that device.

### D.6.3 Interoperability with OpenGL

Interoperability between SYCL and OpenGL is not directly provided by the SYCL interface, however can be achieved via the SYCL OpenCL interoperability interface.

# E. What changed from previous versions

## E.1 What changed from SYCL 1.2.1 to SYCL 2020

The SYCL runtime moved from namespace `cl::sycl` provided by `#include "CL/sycl.hpp"` to namespace `sycl` provided by `#include "SYCL/sycl.hpp"` as explained in §4.3. The old header file is still available for compatibility with SYCL 1.2.1 applications.

The list of built-in integer math functions was extended with `ctz()` in Tables 4.154. Specification of `clz()` was extended with the case of x is 0.

The classes `vector_class`, `string_class`, `function_class`, `mutex_class`, `shared_ptr_class`, `weak_ptr_class` , `hash_class` and `exception_ptr_class` have been removed from the API and the standard classes `std::vector`, `std::string`, `std::function`, `std::mutex`, `std::shared_ptr`, `std::weak_ptr`, `std::hash` and `std::exception_ptr` are used instead.

`operator[]` of SYCL `accessor` for SYCL `buffer` was changed to return const reference when `accessMode == access::mode::read`.

The specific `sycl::buffer` API taking `std::unique_ptr` has been removed. The behavior is the same as in SYCL 1.2.1 but with a simplified API. Since there is still the API taking `std::shared_ptr` and there is an implicit conversion from a `std::unique_ptr` prvalue to a `std::shared_ptr`, the API can still be used as before with a `std::unique_ptr` to give away memory ownership.

Unified Shared Memory (USM), in §4.8, has been added as a pointer-based strategy for data management. It defines several types of allocations with various accessibility rules for host and devices. USM is meant to complement buffers, not replace them.

The `queue` class received a new `property` that requires in-order semantics for a queue where operations are executed in the order in which they are submitted.

The `queue` class received several new methods to define kernels directly on a queue objects instead of inside a command group handler in the `submit` method.

The `program` class has been replaced with the `module` which provides a type-safe and thread-safe interface for compiling and linking SYCL kernel function. The previous member functions of the `program` class are now free functions. Modules are now retrieved from the `this_module::get` function which produces a `module` containing the SYCL kernel functions of the current translation unit.

The `module` class now supports specialization constants which allow SYCL kernel functions to define constant variables, whose value is not known until the containing module is compiled. The `specialization_constant` class has been introduced to represent a reference to a specialization constant both in the SYCL application for setting the value and in a SYCL kernel function for retrieving the value.

The `kernel_handler` class has been introduced as an optional parameter to kernel invocation commands to provide functionality and queries relating to a SYCL kernel function at the kernel scope, including setting the value of a specialization constant.

The constructors for SYCL `context` and `queue` are made `explicit` to prevent ambiguities in the selected constructor resulting from implicit type conversion.

The requirement for C++ standard layout for data shared between host and devices has been softened and now only C++ trivially copyable is required, as explained mainly in Section 4.14.4.

The concept of a group of work-items was generalized to include work-groups and sub-groups. A work-group is represented by the `sycl::group` class as in SYCL 1.2.1, and a sub-group is represented by the new `sycl::sub_group` class.

The `host_task` member function for the `queue` has been introduced for en-queueing host tasks on a queue to schedule the SYCL runtime to invoke native C++ functions, conforming to the SYCL memory model. Host tasks also support for interoperability with the native SYCL backend objects associated at that point in the DAG using the the optional `interop_handle` class.

A library of algorithms based on the C++17 algorithms library was introduced in §4.19.4. These algorithms provide a simple way for developers to apply common parallel algorithms using the work-items of a group.

The definition of the `sycl::group` class was modified to support the new group functions in §4.19.5. New member types and variables were added to enable generic programming, and member functions were updated to encapsulate all functionality tied to work-groups in the `sycl::group` class. See Table 4.83 for details.

Changes in the SYCL `vec` class described in §4.16.2:

- `operator[]` was added;
- unary `operator+()` and `operator-()` were added;
- `get_count()` and `get_size()` were made `static constexpr`.

Buffer and local accessors now meet the C++ requirement of `ContiguousContainer`. This includes (but is not limited to) returning `begin` and `end` iterators, specifying a default constructible accessor that can be passed to a kernel but not dereferenced, and making them equality comparable within kernel functions. `accessor::get_size()` has been removed to prevent confusion with `accessor::size()`, and replaced with `accessor::byte_size()`. All buffer and local accessor size and iterator queries have been marked `noexcept`.

The device selection now relies on a simpler API based on ranking functions used as device selectors described in Section 4.6.1.1.

A new reduction library consisting of the `reduction` function, `reducer` class and `parallel_reduce` was introduced to simplify the expression of variables with reduction semantics in SYCL kernels. See §4.10.2.

Global and constant accessor can now be constructed as placeholders without specifying the `access::placeholder` template parameter (which has been deprecated). It is allowed to call `handler::require` on both placeholder and non-placeholder global and constant buffer accessors, and it is allowed to call it multiple times. `accessor::is_placeholder` is not `constexpr` anymore.

The `image` class has been replaced with `unsampled_image` and `sampled_image` classes. The `sampler` class has been modified to support these changes.

Global and constant accessor can now be constructed as placeholders without specifying the `access::placeholder` template parameter (which has been deprecated). It is allowed to call `handler::require` on both

placeholder and non-placeholder global and constant buffer accessors, and it is allowed to call it multiple times. `accessor::is_placeholder` is not `constexpr` anymore.

Specified the constness semantics of accessors. `const dataT` and `access::mode::read` are semantically equivalent, and having at least one of those parameters part of the accessor type makes the accessor read-only. Defined implicit conversions based on these semantics. Specified default access mode to be `access::mode::read` for `const dataT` and `access::mode::read_write` otherwise. Specified default accessor dimensions to be 1. All of these rules enable most buffer accessor code to only need to use `accessor<T>` for mutable data and `accessor<const T>` for const data.

The `atomic` class from SYCL 1.2.1 and accessors using `access::mode::atomic` were deprecated in favor of a new `atomic_ref` interface.

The SYCL exception class hierarchy has been condensed into a single exception type: `exception`. The variety of errors are now provided via error codes, which aligns with the C++ error code mechanism.

The new error code mechanism now also generalizes the previous `get_cl_code` interface to provide a generic interface way for querying backend-specific error codes.

A new concept called device aspects has been added, which tells the set of optional features a device supports. This new mechanism replaces the `has_extension()` function and some uses of `get_info()`.

There is a new Chapter 6 which describes how extensions to the SYCL language can be added by vendors and by the Khronos SYCL group.

# References

[1] Khronos OpenCL Working Group, *The OpenCL Specification, version 1.2.19*, 2012. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf

[2] International Organization for Standardization (ISO), "Programming Languages - C++," Tech. Rep. ISO/IEC 14882:2017, 2017.

[3] ——, *Working Draft, Standard for Programming Language C++*, 2020. [Online]. Available: http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2020/n4849.pdf

[4] Khronos OpenCL Working Group, *The OpenCL Specification, Version 2.0*, 2015. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf

[5] ——, *The OpenCL Extension Specification, version 1.2.22*, 2012. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.2-extensions.pdf

# Glossary

**accessor** An accessor is a class which allows a SYCL kernel function to access data managed by a buffer or image class. Accessors are used to express the dependencies among the different command groups. For the full description please refer to section [4.7.6]. 26, 27, 33, 34, 104, 105, 120, 138, 139, 141, 253, 254, 256, 413

**application scope** The application scope starts with the construction first SYCL runtime class object and finishes with the destruction of the last one. Application refers to the C++ SYCL application and not the SYCL runtime. 26

**aspect** A characteristic of a device which determines whether it supports some optional feature. Aspects are always boolean, so a device either has or does not have an aspect. 66, 74, 84, 87, 379, 409

**`async_handler`** An asynchronous error handler object is a function class instance providing necessary code for handling all the asynchronous errors triggered from the execution of command groups on a queue, within a context or an associated event. For the full description please refer to section [4.15.2]. 7, 90, 96, 100, 102, 276, 277, 278

**asynchronous error** A SYCL asynchronous error is an error occurring after the host API call invoking the error causing action has returned, such that the error cannot be thrown as a typical C++ exception from a host API call. Such errors are typically generated from device kernel invocations which are executed when SYCL task graph dependencies are satisfied, which occur asynchronously from host code execution. For the full description and associated asynchronous error handling mechanisms, please refer to section [4.15]. 42, 90, 96, 100, 102, 276, 277, 278

**blocking accessor** A blocking accessor is an accessor which provides immediate access and continues to provide access until it is destroyed. For the full description please refer to section [4.7.6.4]. 142

**buffer** The buffer class manages data for the SYCL C++ host application and the SYCL device kernels. The buffer class may acquire ownership of some host pointers passed to its constructors according to the constructor kind.

The buffer class, together with the accessor class, is responsible for tracking memory transfers and guaranteeing data consistency among the different kernels. The SYCL runtime manages the memory allocations on both the host and the device within the lifetime of the buffer object. For the full description please refer to section [4.7.2]. 27, 33, 45, 104, 139, 155, 254, 255, 413

**command** A request to execute work that is submitted to a queue such as the invocation of a SYCL kernel function, the invocation of a host task or an asynchronous copy.. 253, 257, 415

**command group handler** The command group handler class provides the interface for the commands that can be executed inside the command group scope. It is provided as a scoped object to all of the data access requests within the command group scope. For the full description please refer to section [4.10.4]. 230, 235, 251, 257, 258, 414, 418

**command group function object** A type which is callable with `operator()` that takes a reference to a command group handler, that defines a command group which can be submitted by a queue. The function object can be a named type, lambda function or `std::function`. 27, 28, 42, 43, 44, 96, 234, 235, 277, 414

**command group scope** The command group scope is the function scope defined by the command group function object. The command group command group handler object lifetime is restricted to the command group scope. For more details please see [4.10.3]. 26, 27, 47, 53, 234, 235, 413

**command group** In SYCL, the operations required to process data on a device are represented using a command group function object. Each command group function object is given a unique command group handler object to perform all the necessary work required to correctly process data on a device using a kernel. In this way, the group of commands for transferring and processing data is enqueued as a command group on a device for execution. A command group is submitted atomically to a SYCL queue. 11, 29, 30, 31, 32, 33, 34, 35, 90, 100, 101, 103, 104, 141, 142, 156, 239, 251, 252, 253, 256, 257, 263, 413, 414, 415, 416

**constant memory** A region of global memory that remains constant during the execution of a kernel. The SYCL runtime allocates and initializes memory objects placed into constant memory. 36, 140, 144

**context** A context represents the runtime data structures and state required by a SYCL backend API to interact with a group of devices associated with a platform. The context is defined as the `sycl::context` class, for further details please see [4.6.3]. 27, 28, 30, 44, 68, 69, 70, 238, 254, 255, 257, 258, 262, 263, 267, 269, 270, 271, 272, 394, 399, 400, 414, 416

**device** A SYCL device encapsulates a number of heterogeneous devices exposed by a sycl-platform from a given SYCL backend. SYCL devices can execute SYCL kernel functions. 25, 26, 27, 28, 29, 30, 44, 67, 69, 70, 75, 76, 84, 87, 88, 89, 90, 99, 254, 255, 257, 262, 266, 267, 270, 271, 394, 395, 405, 413, 414, 415, 416, 417, 418, 419

**device image selection function** A callable object which takes the begin and end iterators of a module pointing to a sequence of device image and returns an iterator to a chosen device image. 257

**device image** A SYCL device image represents an implementation defined file format that encapsulates the relevant functions, symbols and meta-data to represent the SYCL kernel function set of a module. 238, 257, 258, 262, 263, 265, 267, 268, 269, 270, 271, 414, 416

**device selector** A way to select a device used in various places. This is a callable object taking a device reference and returning an integer rank. One of the device with the highest positive value is selected. See Section 4.6.1.1 for more details. 44, 62, 63, 64, 65, 66, 71, 73, 93, 94, 408

**device compiler** A SYCL device compiler is a compiler that produces OpenCL device binaries from a valid SYCL application. For the full description please refer to section [5]. 26, 38, 47, 226, 269, 367, 393

**event** A SYCL object that represents the status of an operation that is being executed by the SYCL runtime. 100, 104

**executable** A state which a module can be in, representing SYCL kernel functions as an executable. 265, 267, 414

**executable module** A module that is of module state executable. 263, 267, 268, 269, 416

**generic memory** Generic memory is a virtual memory region which can represent global, local and private memory region. 37

**global memory** Global memory is a memory region accessible to all work-items executing on a device. 36, 140, 144, 184

**global id** As in OpenCL, a global ID is used to uniquely identify a work-item and is derived from the number of global work-items specified when executing a kernel. A global ID is a one, two or three-dimensional value that starts at 0 per dimension. 32, 216, 217, 415, 416

**group** A group of work-items within the index space of a SYCL kernel execution, such as a work-group or sub-group. 38, 316, 344, 408, 416

**group mem-fence** A group mem-fence guarantees that any access on the corresponding memory address space before a group barrier must complete before continuing to process any data from that memory space after the barrier. 350, 415, 416

**group barrier** A synchronization function within a group of work-items. All the work-items of a group must execute the barrier construct for any work-item continues execution beyond the barrier. Additionally a group barrier performs a group mem-fence. 38, 42, 316, 348, 350, 415, 418, 419

**host** Host is the system that executes the C++ application including the SYCL API. 25, 29, 46, 142, 156, 164, 170, 418

**host task command** A type of command that can be used inside a command group in order to schedule a native C++ function.. 253

**host task** A command which invokes a native C++ callable, scheduled conforming to SYCL dependency rules.. 253, 254, 255, 256, 257, 408, 413

**host pointer** A pointer to memory on the host. Cannot be accessed directly from a device. 137

**id** It is a unique identifier of an item in an index space. It can be one, two or three dimensional index space, since the SYCL Kernel execution model is an nd-range. It is one of the index space classes. For the full description please refer to section 4.10.1.3. 209, 214, 216, 217, 222, 226, 415, 417, 418

**image** Images in SYCL, like buffers, are abstractions of multidimensional structured arrays. Image can refer to `unsampled_image` and `sampled_image`. For the full description please refer to section [4.7.3]. 27, 33, 45, 104, 139, 254, 255, 413

**index space classes** The OpenCL Kernel Execution Model defines an nd-range index space. The SYCL runtime class that defines an nd-range is the `sycl::nd_range`, which takes as input the sizes of global and local work-items, represented using the `sycl::range` class. The kernel library classes for indexing in the defined nd-range are the following classes:

- `sycl::id` : The basic index class representing a id.
- `sycl::item` : The index class that contains the global id and local id.
- `sycl::nd_item` : The index class that contains the global id, local id and the work-group id.
- `sycl::group` : The group class that contains the work-group id and the methods on a work-group.

244

**input** A state which a module can be in, representing SYCL kernel functions as a source or intermediate representation. 265, 266, 416

**input module** A module that is of module state input. 263, 266, 269, 270, 399, 416, 418

**item** An item id is an interface used to retrieve the global id, work-group id and local id. For further details see [4.10.1.4]. 212, 218, 219, 418

**kernel** A SYCL kernel which can be executed on a device, including the SYCL host device. Is created implicitly when defining a SYCL kernel function (see 4.10) but can also be created manually in order to pre-compile SYCL kernel functions. 29, 30, 46, 47, 71, 105, 239, 243, 244, 251, 258, 260, 261, 262, 272, 395, 400, 418

**kernel invocation command** A type of command that can be used inside a command group in order to schedule a SYCL kernel function, includes `single_task`, all variants of `parallel_for` and `parallel_for_workgroup`. 253, 257, 262, 263, 400, 407, 419

**kernel handler** A representation of a SYCL kernel function being invoked that is available to the kernel scope. . 238

**kernel name** A kernel name is a class type that is used to assign a name to the kernel function, used to link the host system with the kernel object output by the device compiler. For details on naming kernels please see [5.2]. 46, 239, 240, 241, 242, 243, 274, 419

**kernel scope** The function scope of the `operator()` on a SYCL kernel function. Note that any function or method called from the kernel is also compiled in kernel scope. The kernel scope allows C++ language extensions as well as restrictions to reflect the capabilities of OpenCL devices. The extensions and restrictions are defined in the SYCL device compiler specification. 26, 238, 407, 416

**local memory** Local memory is a memory region associated with a work-group and accessible only by work-items in that work-group. 37, 139, 140, 164, 248

**local id** A unique identifier of a work-item among other work-items of a work-group. 32, 216, 223, 415, 416

**mem-fence** A memory fence guarantees that any memory access must complete before continuing to process any data after the fence. The `sycl::atomic_fence` function acts as a fence across all work-items and devices specified by a `memory_scope` argument, and a group mem-fence acts as a fence across all work-items in a specific group. 38, 316

**module** A SYCL module represents a set of SYCL kernel functions which can be executed on a number of devices associated with a context. 30, 238, 257, 258, 262, 263, 265, 266, 267, 268, 269, 270, 271, 272, 395, 399, 400, 407, 414, 416, 417

**module state** A SYCL module state represents an the state abstract state of a module and therefore it's capabilities in the SYCL programming API. Can be either a input module, object module or an executable module. 265, 266, 267, 269, 272, 414, 416, 417

**native-specialization constant** A specialization constant which is natively supported by the device image which contains it. . 263, 268

**native backend object** An opaque object defined by a specific backend that represents a high-level SYCL object on said backend. There is no guarantee of having native backend objects for all SYCL types. 44, 53, 54, 55, 56, 57, 64, 71, 90, 187, 254, 255, 256, 257

**nd-item** A unique identifier representing a single work-item and work-group within the index space of a SYCL kernel execution. Can be one, two or three dimensional. In the SYCL interface a nd-item is represented by the `nd_item` class (see Section 4.10.1.5). 99, 241, 243, 417, 418

**nd-range** A representation of the index space of a SYCL kernel execution, the distribution of work-items within into work-groups. Contains a range specifying the number of global work-items, a range specifying the number of local work-items and a id specifying the global offset. Can be one, two or three dimensional. The minimum size of each range within the nd-range is 1 per dimension. In the SYCL interface an nd-range is represented by the `nd_range` class (see Section 4.10.1.2). 32, 99, 216, 222, 239, 241, 243, 247, 248, 417

**object** A state which a module can be in, representing SYCL kernel functions as a non-executable object. 265, 266, 417

**object module** A module that is of module state object. 263, 266, 416

**platform** The host together or a collection of devices managed by the OpenCL framework that allow an application to share resources and execute kernels on devices in the platform. A SYCL application can target one or multiple OpenCL platforms provided by OpenCL device vendors [1]. 28, 29, 30, 44, 67, 70, 394, 414

**private memory** A region of memory private to a work-item. Variables defined in one work-item's private memory are not visible to another work-item. [1]. The `sycl::private_memory` class provides access to the work-item's private memory for the hierarchical API as it is described at [4.10.7.3]. 37, 249

**queue** A SYCL command queue is an object that holds command groups to be executed on a SYCL device. SYCL provides a heterogeneous platform integration using device queue, which is the minimum requirement for a SYCL application to run on a SYCL device. For the full description please refer to section [4.6.5]. 27, 28, 29, 30, 43, 44, 99, 253, 254, 255, 256, 257, 258, 394, 395, 408, 413, 414

**range** A representation of a number of work-items or work-group within the index space of a SYCL kernel execution. Can be one, two or three dimensional. In the SYCL interface a work-group is represented by the `group` class (see Section 4.10.1.7 ). 417

**reduction** An operation that produces a single value by combining multiple values in an unspecified order using a binary operator. If the operator is non-associative or non-commutative, the behavior of a reduction may be non-deterministic.. 40, 227, 228, 408

**rule of zero** For a given class, if the copy constructor, move constructor, copy assignment operator, move assignment operator and destructor would all be inlined, public and defaulted, none of them should be explicity declared . 58, 59

**rule of five** For a given class, if at least one of the copy constructor, move constructor, copy assignment operator, move assignment operator or destructor is explicitly declared, all of them should be explicity declared . 58, 59

**SMCP** The single-source multiple compiler-passes (SMCP) technique allows a single source file to be parsed by multiple compilers for building native programs per compilation target. For example, a standard C++ CPU compiler for targeting host will parse the SYCL file to create the C++ SYCL application which offloads parts of the computation to other devices. A SYCL device compiler will parse the same source file and target only SYCL kernels. 23, 46, 88, 367, 379

**specialization id** An identifier which represents a reference to a specialization constant both in the SYCL application for setting the value prior to the compilation of an input module and in a SYCL kernel function for retrieving the value during invocation. . 263, 266

**specialization constant** A constant variable where the value is not known until compilation of the SYCL kernel function. . 238, 258, 263, 266, 268, 269, 274, 407, 408, 416, 418

**string kernel name** The name of a SYCL kernel function in string form, this can be the name of a kernel function created via interop or a string form of a type kernel name. 268, 271

**sub-group barrier** A group barrier for all work-items in a sub-group. 42

**sub-group** The SYCL sub-group (`sycl::sub_group` class) is a representation of a collection of related work-items within a work-group that execute concurrently, and which may make independent forward progress with respect to other sub-groups in the same work-group. For further details for the `sycl::sub_group` class see [4.10.1.8]. 38, 216, 225, 226, 408, 415, 418

**SYCL file** A SYCL C++ source file that contains SYCL API calls. 418

**SYCL C++ template library** The template library is a set of C++ templated classes which provide the programming interface to the SYCL developer. 393

**SYCL host device** See SYCL host backend. 391, 393, 416

**SYCL host backend** The SYCL host device is a native C++ implementation of a device. It does not have an native handle. It has full SYCL capabilities and reports them through the SYCL information retrieval interface. The SYCL host device is mandatory for every SYCL implementation and is always available, but may not achieve the same performance as a different backend running on the CPU. Any C++ application debugger can be used for debugging SYCL kernels executing on a SYCL host device. 29, 51, 62, 63, 65, 67, 69, 75, 95, 391, 418

**SYCL runtime** A SYCL runtime is an implementation of the SYCL API specification. The SYCL runtime manages the different OpenCL platforms, devices, contexts as well as memory handling of data between host and OpenCL contexts to enable semantically correct execution of SYCL programs. 28, 35, 41, 43, 44, 45, 46, 51, 52, 53, 54, 55, 56, 58, 59, 60, 62, 90, 100, 105, 106, 114, 117, 119, 121, 123, 124, 125, 126, 127, 128, 129, 132, 133, 135, 136, 137, 138, 139, 144, 155, 156, 164, 170, 222, 235, 238, 240, 241, 242, 243, 245, 253, 254, 255, 256, 257, 258, 263, 267, 275, 276, 280, 282, 317, 367, 369, 393, 395, 401, 403, 404, 408, 413, 414, 415

**SYCL application** A SYCL application is a C++ application which uses the SYCL programming model in order to execute kernels on devices. 26, 29, 53, 54, 55, 256, 257, 263, 316, 407, 413, 414, 418

**SYCL kernel function** A type which is callable with `operator()` that takes a id, item, nd-item or work-group which can be passed to kernel enqueue member functions of the command group handler. A SYCL kernel function defines an entry point to a kernel. The function object can be a named trivially copyable type or lambda function. 25, 26, 27, 28, 29, 39, 52, 53, 97, 98, 99, 235, 238, 239, 240, 241, 242, 243, 244, 247, 250, 253, 257, 258, 262, 263, 265, 266, 267, 268, 269, 270, 271, 272, 274, 389, 399, 400, 403, 407, 413, 414, 416, 417, 418, 419

**SYCL backend API** The exposed API for writing SYCL code against a given SYCL backend. 3, 25, 30, 31, 44, 46, 53

**SYCL backend** An implementation of the SYCL programming model using an heterogeneous programming API. A SYCL backend exposes one or multiple SYCL platforms. For example, the OpenCL backend, via the ICD loader, can expose multiple OpenCL platforms. 3, 19, 25, 27, 28, 29, 30, 31, 33, 34, 41, 42, 43, 44, 45, 47, 51, 52, 53, 54, 57, 60, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 84, 88, 89, 95, 97, 99, 101, 102, 103, 104, 105, 106, 120, 175, 256, 257, 259, 260, 261, 262, 263, 269, 278, 280, 284, 339, 361, 367, 371, 373, 374, 380, 399, 400, 403, 408, 414, 419

**type kernel name** The name of a SYCL kernel function in type form, this can be either a kernel name provided to akernel invocation command or the type of a function object use as a SYCL kernel function. 271, 418

**work-group barrier** A group barrier for all work-items in a work-group. 42, 247, 249, 316

**work-group id** As in OpenCL, SYCL kernels execute in work-groups. The group ID is the ID of the work-group that a work-item is executing within. A group ID is an one, two or three dimensional value that starts at 0 per dimension. 32, 214, 222, 415, 416

**work-group** The SYCL work-group (`sycl::group` class) is a representation of a collection of related work-items that execute on a single compute unit. The work-items in the group execute the same kernel-instance and share local memory and work-group functions [1]. For further details for the `sycl::group` class see [4.10.1.7]. 32, 38, 156, 164, 216, 220, 222, 223, 226, 239, 247, 248, 408, 415, 417, 418, 419

**work-item** The SYCL work-item (`sycl::nd_item` class) is a representation of an OpenCL work-item. One of a collection of parallel executions of a kernel invoked on a device by a command. A work-item is executed by one or more processing elements as part of a work-group executing on a compute unit. A work-item is distinguished from other work-items by its global ID or the combination of its work-group ID and its local ID within a work-group [1]. 32, 164, 214, 217, 218, 247, 248, 249, 408, 415, 417, 418, 419