SYCL™ Specification

SYCL integrates OpenCL devices with modern C++

Version 1.2

Revision Date: 2014-09-16

Khronos OpenCL Working Group — SYCL subgroup

Editors: Lee Howes and Maria Rovatsou

Contents

List of Tables

Acknowledgements

Editors: Lee Howes, Qualcomm; Maria Rovatsou, Codeplay

Contributors:

- Eric Berdahl, Adobe
- Shivani Gupta, Adobe
- David Neto, Altera
- Ronan Keryell, AMD
- Brian Sumner, AMD
- Balázs Keszthelyi, Broadcom
- Gordon Brown, Codeplay
- Paul Keir, Codeplay
- Ralph Potter, Codeplay
- Ruyman Reyes, Codeplay
- Andrew Richards, Codeplay
- Maria Rovatsou, Codeplay
- Allen Hux, Intel
- Matt Newport, EA
- Lee Howes, Qualcomm
- Chu-Cheow Lim, Qualcomm
- Jack Liu, Qualcomm

1. Introduction

SYCL is a C++ programming model for OpenCL. SYCL builds on the underlying concepts, portability and efficiency of OpenCL while adding much of the ease of use and flexibility of C++. Developers using SYCL are able to write standard 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 OpenCL both through the features of the SYCL libraries and, where necessary, through interoperation with code written directly to the OpenCL APIs.

SYCL implements a shared source design which offers the power of source integration while allowing toolchains to remain flexible. The shared source design supports embedding of code intended to be compiled for an OpenCL device, for example a GPU, inline with host code. This embedding of code offers three primary benefits:

Simplicity For novice programmers to use OpenCL the separation of host and device code can become complicated to deal with, particularly when similar kernel code is used for multiple different operations. 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 shared source 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 optimised 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 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 preprocessor 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 of the underlying OpenCL standard. The OpenCL C specification 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.

The use of C++ features such as templates and inheritance on top of the OpenCL 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 libraries that build simple, yet efficient, interfaces to offer more developers access to OpenCL capabilities and devices. SYCL is intended to serve as a foundation for innovation in programming models for heterogeneous systems, that builds on an open and widely implemented standard foundation in the form of OpenCL.

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

- The hierarchical parallelism syntax offers a way of expressing the data-parallel OpenCL execution model in an easy-to-understand 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 are already part of the OpenCL execution model to be built up easily and safely by SYCL programmers.

To summarize, SYCL enables OpenCL kernels to be written inside C++ source files. 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 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 heterogeneous processing innovation.

2. SYCL Architecture

This chapter builds on the structure of the OpenCL specification's architecture chapter to explain how SYCL overlays the OpenCL specification and inherits its capabilities and restrictions as well as the additional features it provides on top of OpenCL 1.2.

2.1 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 OpenCL device or on the host CPU.

The terminology used for SYCL inherits that of OpenCL with some SYCL-specific additions. The code that can run on either an OpenCL device or host CPU is called a *kernel*. To ensure maximum backward-compatibility, a software developer can produce a program that mixes standard OpenCL C kernels and OpenCL API code with SYCL code and expect fully compatible interoperation.

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 OpenCL standard. 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. OpenCL developers can produce templated algorithms that are easily usable by developers in other fields.

2.2 The SYCL Platform Model

The SYCL platform model is based on the OpenCL platform model, but there are a few additional abstractions available to programmers.

The model consists of a host connected to one or more OpenCL devices. An OpenCL 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. A SYCL application runs on a host according to the standard C++ CPU execution model. The SYCL application submits **command groups** to **queues**, which execute either on OpenCL devices, or on the host CPU.

When a SYCL implementation executes command groups on an OpenCL device, it achieves this by enqueuing OpenCL **commands** to execute computations on the processing elements within a device. The processing ele-

ments within an OpenCL compute unit may execute a single stream of instructions as ALUs within a SIMD unit (which execute 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.

When a SYCL implementation executes command groups on the host, it is free to use whatever parallel execution facilities are available on the host, as long as it executes within the semantics of the kernel execution model defined by OpenCL.

2.2.1 Platform Mixed Version Support

OpenCL is designed to support devices with different capabilities under a single platform. This includes devices which conform to different versions of the OpenCL specification and devices which support different extensions to the OpenCL specification. There are three important sets of capabilities to consider for a SYCL device: the platform version, the version of a device and the extensions supported.

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 `cl::sycl::device::get_info()` method. Examples of attributes associated with the device version are resource limits and information about functionality beyond the core OpenCL 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.

In OpenCL, a device has a *language version*. 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: `CL_SYCL_LANGUAGE_VERSION`.

2.3 SYCL Execution Model

Execution of a SYCL program occurs in two parts: *kernels* that execute on either the host CPU, or one or more *OpenCL devices*, and a *host program* that executes on the host CPU. The host program defines the context for the kernels and manages their execution. Like OpenCL, SYCL is capable of running kernels on multiple device types. However, SYCL builds on top of OpenCL due to the integration into a host toolchain by providing an ability to run kernel code directly on the CPU without interacting with an OpenCL runtime. This is distinct from running on the CPU via an OpenCL device and can act as a fall-back when no OpenCL platform is available on the machine.

In OpenCL, *queues* contain *commands*, which can include data transfer operations, synchronization commands, or *kernels* submitted for execution. In SYCL, the commands are grouped together into aggregates called *command groups*. Command groups associate sets of data movement operations with kernels that will be enqueued together on an underlying OpenCL queue. These data transfer operations may be needed to make available data that the kernel needs or to return its results to other devices.

When a kernel is submitted for execution by the host, 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 `range<N>` and `id<N>` are each N-dimensional vectors of integers. The iteration space defined via an `nd_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 workgroup size is left undefined. A kernel invoked over a `range<N>`, instead of an `nd_range<N>` is executed within an iteration space of undefined workgroup size. In this case, less information is available to each work-item through the simpler `item<N>` class.

2.3.1 Execution Model: Queues, Command Groups and Contexts

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 `cl::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 runtime while data movement between contexts must involve the host. A given context can only wrap devices owned by a single platform. In SYCL, a context resource is accessible through by a `cl::sycl::context` object.

3. *Devices*: Platforms provide one or more devices for executing kernels. In SYCL, a device is accessible through a `cl::sycl::device` object. SYCL provides the abstract `cl::sycl::device_selector` class which the user can subclass to define how the runtime should select the best device from all available platforms for the user to use. For ease of use, SYCL provides a set of predefined concrete `device_selector` instances that select devices based on common criteria, such as type of device. SYCL, unlike OpenCL, defines a host device, which means any work that uses the host device will execute on the host and not on any OpenCL device.

4. *Command groups*: SYCL groups lists of OpenCL *commands* into a group to perform all the necessary work required to correctly process host data on a device using a kernel. In this way, they group the commands of transferring and processing these data in order to enqueue them on a device for execution. Command groups are defined using the `cl::sycl::command_group` class.

5. *Kernels*: The SYCL functions that run on SYCL devices (i.e. either an OpenCL device, or the host) are defined as C++ functors or lambda functions. In SYCL, all kernels must have a *name*, which must be a globally-accessible C++ typename. This is required to enable kernels compiled with one compiler to be linked to host code compiled with a different compiler. For functors, the typename of the functor is sufficient as the kernel *name*, but for C++11 lambda functions, the user must provide a user- defined *name*. All instantiations of templated kernels must have distinct names to avoid linking conflicts. The easiest way to achieve this is to ensure that any template parameters for any kernel defined via a lambda function are also included as template parameters in the kernel name.

6. *Program Objects*: 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 `cl::sycl::program` class.

7. *Command-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 `cl::sycl::queue` objects.

The command-queue schedules commands (from command groups) for execution on a device. Commands launched by the host execute asynchronously with respect to the host thread, and not necessarily ordered with respect to each other. It is the responsibility of the SYCL implementation to ensure that the different commands execute in an order which preserves SYCL semantics. This means that a SYCL implementation must map, move or copy data between host and device memory, execute kernels and perform synchronization between different queues, devices and the host in a way that matches the semantics defined in this specification. If a command group runs on an OpenCL device, then this is expected to be achieved by enqueuing the right memory and synchronization commands to the queue to ensure correct execution. If a command group runs on the host, then this is expected to be achieved by host-specific synchronization as well as by ensuring that no OpenCL device is simultaneously using any required data.

In OpenCL, queues can operate using in-order execution or out-of-order execution. In SYCL, the implementation must add synchronization commands to queues to ensure execution ordering is defined, regardless of whether the underlying OpenCL queue is in-order or out-of-order.

2.4 Memory Model

Work-items executing in a kernel have access to four distinct memory regions:

- *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 a distinct memory region shared between 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 that work-group. Work-group-level visibility allows local memory to be implemented as dedicated regions of memory on an OpenCL device 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.

The application running on the host uses SYCL buffer objects using instances of the `cl::sycl::buffer` class to allocate memory in the global address space, or can allocate specialized image memory using the `cl::sycl::image` class. In OpenCL, a memory object is attached to a specific context. In SYCL, a `cl::sycl::buffer` or `cl::sycl::image` object can encapsulate multiple underlying OpenCL memory objects and host memory allocations to enable the same buffer or image to be shared between multiple devices in different contexts, and hence different platforms. It is the responsibility of the SYCL implementation to ensure that a buffer or image object shared between multiple OpenCL contexts is moved between contexts using the correct synchronization and copy commands to preserve SYCL memory ordering semantics.

2.4.1 Access to memory

To access global memory inside a kernel, the user must create a `cl::sycl::accessor` object which parameterizes the type of access the kernel requires. 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, write-only or read-write. 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. Atomic access can also be requested on an accessor which allows `cl::sycl::atomic` classes to be used via the accessor.

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 `cl::sycl::local_accessor` object to the kernel as a parameter, or can define a variable in workgroup scope inside `cl::sycl::parallel_for_workgroup`.

Any variable defined inside a `cl::sycl::parallel_for` scope or `cl::sycl::parallel_for_workitem` scope will be allocated in private memory. Variables defined in functions called from workgroup scope (i.e. `cl::sycl::parallel_for_workgroup` will also be local, while variables defined in functions called from workitem scope (i.e. `cl::sycl::parallel_for` or `cl::sycl::parallel_for_workitem`) will be allocated in private memory.

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

Within kernels, accessors can be implicitly cast to C++ pointer types. The pointer types will contain a compile-time deduced address space. So, for example, if an accessor to global memory is cast to a C++ pointer, 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 memory 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: `cl::sycl::local_ptr`, `cl::sycl::global_ptr`, `cl::sycl::private_ptr`, or `cl::sycl::constant_ptr`. An accessor declared with one address space can be implicitly cast to an explicit pointer class for the same address space. 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 `cl::sycl::multi_ptr` class is defined which is templated via a compile-time constant enumerator value to specify the address space.

2.4.2 Memory consistency

OpenCL uses a relaxed memory consistency model, i.e. the state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times. This also applies to SYCL kernels.

As in OpenCL, within a work-item memory has load/store consistency. Both local memory and global memory may be made consistent across work-items in a single work-group through use of a work-group barrier operation with appropriate flags. There are no guarantees of memory consistency between different work-groups executing a kernel or between different kernels during their execution.

Memory consistency for `cl::sycl::buffer` and `cl::sycl::image` objects shared between enqueued commands is enforced at synchronization points derived from completion of enqueued commands. Consistency of such data between the OpenCL runtime and the host program is ensured via copy commands or map and unmap operations.

2.4.3 Atomic operations

Atomic operations can be performed on memory in buffers. The range of atomic operations available on a specific OpenCL device is limited by the atomic capabilities of that device. The `cl::sycl::atomic<T>` must be used for elements of a buffer to provide safe atomic access to the buffer from device code.

2.5 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 C++ features used in SYCL are a subset of the C++11 standard features. Users will need to compile SYCL source code with C++ compilers which support the following C++ features:

- All C++03 features, apart from Run Time Type Information
- Exception handling
- C++11 lambda functions

- C++11 variadic templates
- C++11 template aliases
- C++11 rvalue references
- C++11 std::function, std::string and std::vector, although users can optionally define and use their own versions of these classes, which SYCL can use via template aliases

SYCL programs are explicitly parallel and expose the full heterogeneous parallelism of the underlying machine model of OpenCL. This includes exposing the data-parallelism, multiple execution devices and multiple memory storage spaces of OpenCL. However, SYCL adds on top of OpenCL a higher level of abstraction allowing developers to hide much of the complexity from the source code, when a developer so chooses.

A SYCL program is logically split into host code and kernels. Host code is standard C++ code, as provided by whatever C++ compiler the developer chooses to use for the host code. The kernels are C++ *functors* (see C++ documentation for an explanation of functors) or C++11 lambda functions which have been designated to be compiled as SYCL kernels. SYCL will also accept OpenCL `cl_kernel` objects.

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

In SYCL, kernels are contained within command groups, which include all of the data movement/mapping/copying required to correctly execute the kernel. A command group is instantiated from the `cl::sycl::command_group` class, which takes a functor parameter and a queue. The functor is executed on the host to add all the specified commands to the specified queue. Kernels can only access shared data via accessor objects constructed within the command group.

2.5.1 Basic data parallel kernels

Data-parallel kernels that execute as multiple work-items and where no local synchronization is required are enqueued with the `cl::sycl::parallel_for` function parameterized by a `cl::sycl::range` parameter. These kernels will execute the kernel function body once for each work-item in the range. The range passed to `cl::sycl::parallel_for` represents the global size of an OpenCL 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.

2.5.2 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 `cl::sycl::parallel_for` function with a `cl::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 `barrier` function on an `nd_item` object. 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.

2.5.3 Hierarchical data parallel kernels

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 `cl::sycl::parallel_for` the user calls `cl::sycl::parallel_for_workgroup` with a `cl::sycl::nd_range` value. All code within the `parallel_for_workgroup` scope effectively executes once per work-group. Within the `parallel_for_workgroup` scope, it is possible to call `parallel_for_workitem` 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_workgroup` scope are allocated in workgroup local memory, whereas all variables declared inside the `parallel_for_workitem` scope are declared in private memory. All `parallel_for_workitem` calls within a given `parallel_for_workgroup` execution must have the same dimensions.

2.5.4 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 `cl::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.

2.5.5 Synchronization

In SYCL, synchronization can be either global or local within a work-group. The SYCL implementation may need to provide extra synchronization commands and host-side synchronization in order to enable synchronization across OpenCL contexts, but this is handled internally within the SYCL host runtime.

Synchronization between work-items in a single work-group is achieved using a work-group barrier. This matches the OpenCL C behaviour. All the work-items of a work-group must execute the barrier before any are allowed to continue execution beyond the barrier. Note that the work-group barrier must be encountered by all work-items of a work-group executing the kernel or by none at all. There is no mechanism for synchronization between work-groups. In SYCL, workgroup barriers are exposed through a method on the `cl::sycl::nd_item` class, which is only available inside kernels that are executed over workgroups. This ensures that developers can only use workgroup barriers inside workgroups.

Synchronization points in SYCL are exposed through the following operations:

- *Buffer destruction*: The destructors for `cl::sycl::buffer` and `cl::sycl::image` objects wait for all enqueued work on those objects to complete. If the objects were constructed with attached host memory, then the destructor copies the data back to host memory, if necessary, before returning. The programmer can change the behaviour of the destructor to use custom synchronization by using *storage objects*.

- *Accessor construction*: 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 enqueued to any queue will wait for the accessor to be destroyed.

- *Command group enqueue*: The SYCL scheduler 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.

- *Interaction with OpenCL synchronization operations*: The user can obtain OpenCL events from command groups, images and buffers which will enable the user to add barrier packets to their own queues to correctly synchronize for buffer or image data dependencies.

- *Queue destruction*: The destructor for `cl::sycl::queue` objects waits for all commands executing on the queue to complete before the destructor returns.

- *Context destruction*: The destructor for `cl::sycl::context` objects waits for all commands executing on any queues in the context to complete before the destructor returns. *SYCL event objects*: SYCL provides `cl::sycl::event` objects which can be used for user synchronization. If synchronization is required between two different OpenCL contexts, then the SYCL runtime ensures that any extra host-based synchronization is added to enable the SYCL event objects to operate between contexts correctly.

2.5.6 Error handling

In SYCL, there are two types of error: synchronous errors that can be detected immediately, and asynchronous errors that can only be detected later. 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 user-supplied asynchronous error-handlers.

A `cl::sycl::context` can be constructed with a user-supplied asynchronous error handler. If a `cl::sycl::queue` is constructed without a user-supplied context, then the user can supply an asynchronous error handler for the queue, otherwise errors on that queue will be reported to its context error handler.

Asynchronous errors are not reported immediately as they occur. The asynchronous error handler for a context or queue is called with a `cl::sycl::exception_list` object, which contains a list of asynchronously-generated exception objects, either on destruction of the context or queue that the error handler is associated with, or via an explicit `wait_and_throw` method call on an associated queue. This style of asynchronous error handling is similar to that proposed for an upcoming revision of the C++ standard.

2.5.7 Scheduling of kernels and data movement

Within `command_group` scope, accessor objects specify what data the command group reads and what data it will write. When enqueuing a command group, the runtime ensures that synchronization operations are also enqueued to ensure that the reads and writes are semantically equivalent to an in-order execution. Different command groups may execute out-of-order relative to each other, as long as read and write dependencies are enforced.

A `command_group` can be given either a single queue to be executed on, or two queues: a primary queue and a secondary queue. If a `command_group` fails to be enqueued to the primary queue, then the system will attempt to enqueue it to the secondary queue, if supplied. If the `command_group` fails to be queued to all of the supplied 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` 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.

A `command_group`, when it is constructed, takes a functor or C++11 lambda as a parameter. The functor or lambda is called immediately and any function calls within this section are said to be within the `command_group`. The intention is that a user will perform calls to SYCL functions, methods, destructors and constructors inside the `command_group`. These calls will be non-blocking on the host, but enqueue operations to the `command_group`'s associated queue. All user functions within the `command_group` will be called on the host as the `command_group` functor is executed, but SYCL operations will be queued.

The scheduler must treat command groups atomically. So if two threads simultaneously enqueue two command groups onto the same queue, then each command group must be added to the queue as an atomic operation. The order of two simultaneously enqueued command groups relative to each other is undefined but the constituent commands must not interleave.

Command groups are scheduled to enforce the ordering semantics of operations on memory objects (both buffers and images). These ordering rules apply regardless of whether the command groups are enqueued in the same context, queue, device or platform. Therefore, a SYCL implementation may need to produce extra synchroniza-

tion operations between contexts, platforms, devices and queues using OpenCL constructs such as user events. How this is achieved is implementation defined. An implementation is free to re-order or parallelize command groups in queues as long as the ordering semantics on memory objects are not violated.

The ordering semantics on memory objects are:

1. The ordering rules apply based on the totality of accessors constructed in the command group. The order in which accessors are constructed within the command group is not relevant. If multiple accessors in the same command group operate on the same memory object, then the command group's access to that memory object is the union of the access permissions of the accessors.

2. Accessors can be created to operate on *sub-buffers*. A buffer may be overlaid with any number of sub-buffers. 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.

3. If a command group has any accessor with *discard* access to a memory object, then the scheduler does not need to preserve the previous contents of the memory object when scheduling the command group.

4. All other accessors must preserve normal read-write ordering and data access. This means the scheduler must ensure that a command group that reads a memory object must first copy or map onto the device the data that might be read. Reads must follow writes to memory objects or overlapping sub-buffers.

5. 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 scheduler to maintain multiple read-only copies of the data on multiple devices.

In OpenCL, there are in-order queues and out-of-order queues. In SYCL, the default type of queue is implementation-defined. An OpenCL implementation can require different queues for different devices and contexts. The synchronization required to ensure order between commands in different queues varies according to whether the queues have shared contexts. A SYCL implementation must determine the required synchronization to ensure the above ordering rules above are enforced.

SYCL provides *host accessors*. These accessors give temporary access to data in buffers on the host, outside command groups. Host accessors are the only kinds of accessors that can be created outside command groups. Creation of a host accessor is a blocking operation: all command groups that read or write data in the buffer or image that the host accessor targets must have completed before the host thread will continue. All data being written in an enqueued command group to the buffer or image must be completed and written to the associated host memory before the host accessor constructor returns. Any subsequently enqueued command group that accesses overlapping data in the buffer or image of the host accessor will block and not start execution until the host accessor (and any copies) has been destroyed. This approach guarantees that there is no concurrent access to a memory object between the host thread and any SYCL device.

If a user creates a SYCL buffer, image or accessor from an OpenCL object, then the SYCL runtime will correctly manage synchronization and copying of data between the OpenCL memory object for the lifetime of the SYCL buffer, image or accessor constructed from it. If a user makes use of the underlying OpenCL memory object at the same time as a SYCL buffer, image or accessor is live, then the behaviour is undefined.

2.5.8 Managing object lifetimes

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

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

When an OpenCL object that is encapsulated in a SYCL object is copied in C++, then the underlying OpenCL object is not duplicated, but its OpenCL reference count is incremented. When the original or copied SYCL object is destroyed, then the OpenCL reference count is decremented.

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 can optionally provide a *storage object* which specifies the behaviour on construction and destruction (see Storage section 2.7.1 below). If host memory is attached and the user does not supply a storage object, then the default behaviour 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.

The only blocking operations in SYCL (apart from explicit wait operations) are: host accessor construction, memory object destruction (only if host memory attached and not overridden by a storage object), queue destruction, context destruction, device destruction and platform destruction.

2.5.9 Device discovery and selection

A user specifies which queue a command group must run 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 selector causes the SYCL runtime to perform device discovery. No device discovery is performed until a SYCL 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 both OpenCL devices and platforms as well as a host platform and host device. The host device allows queue creation and running of kernels, but does not support OpenCL-specific features. It is an error for a user to request an underlying OpenCL device for the SYCL host device.

2.5.10 Interfacing with OpenCL

All SYCL objects which encapsulate an OpenCL object (such as contexts or queues) can be constructed from the OpenCL object. The constructor takes one argument, the OpenCL object, and performs an OpenCL `retain`

operation on the OpenCL object to increase its reference count. The destructor for the SYCL object performs an OpenCL `release` operation on the OpenCL object. The copy construction semantics of the SYCL object ensure that each new SYCL copy of the object also does an OpenCL retain on the underlying object.

To obtain the underlying OpenCL object from a SYCL object, there is a `get` method on all relevant SYCL objects. The `get` method returns the underlying OpenCL object and also performs a `retain` operation on the object. It is the user's responsibility to `release` the OpenCL object when the user has finished with it.

SYCL images and buffers are treated differently in that SYCL image and buffer objects do not refer to an OpenCL context and may reference multiple underlying OpenCL image or buffer objects as well as host allocations. It is the accessors to the image and buffer objects that refer to an actual OpenCL context. Accessors provide synchronization in place of the events that the OpenCL runtime would use directly. Therefore, obtaining OpenCL `cl_mem` objects from SYCL is achieved via special accessor classes which can return OpenCL `cl_mem` and `cl_event` objects. SYCL memory objects can be constructed from `cl_mem` objects, but the SYCL system is free to copy from the OpenCL memory object into another memory object or host memory, to achieve normal SYCL semantics, for as long as the SYCL memory object is live.

2.6 Anatomy of a SYCL application

Below is an example of a typical SYCL application which schedules a job to run in parallel on any OpenCL GPU.

```
1  #include <CL/sycl.hpp>
2
3  int main()
4  {
5    using namespace cl::sycl;
6
7    int data[1024]; // initialize data to be worked on
8
9    // By including all the SYCL work in a {} block, we ensure
10   // all SYCL tasks must complete before exiting the block
11   {
12     //  create a queue to enqueue work to
13     queue myQueue;
14
15     // wrap our data variable in a buffer
16     buffer<int, 1> resultBuf(data, 1024);
17
18     // create a command_group to issue commands to the queue
19     command_group(myQueue, [&]()
20     {
21       // request access to the buffer
22       auto writeResult = resultBuf.get_access<access::write>();
23
24       // enqueue a prallel_for task
25       parallel_for<class simple_test>(range<1>(1024), [=](id<1> idx)
26       {
27         writeResult[idx] = idx;
28       }); // end of the kernel function
29     }); // end of our commands for this queue
30   } // end of scope, so we wait for the queued work to complete
31
32   // print result
33   for (int i = 0; i < 1024; i++)
34     printf("data[%d] = %d\n", i, data[i]);
35
36   return 0;
37 }
```

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

A SYCL application has 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 28. The *command group scope* specifies a unit of work which will comprise of a *kernel function* and *accessors*. In this example *command group scope* is defined by lines 19 to 29. 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.

A *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 or by the binary `cl_kernel` entity generated from an OpenCL C string. Each instance of the *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 only as a name to enable the kernel (compiled with a device compiler) and the host code (possibly compiled with a different, host compiler) to be linked. This is required because C++ lambda functions have no name that a linker could use to link the kernel to the host code.

The `parallel_for` wrapper 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 *kernel function* will be executed over the given range from 0 to 1023.

A *kernel function* can only be defined within a *command group scope*. Command group scope is the syntactic scope wrapped by the construction of a `command_group` object as seen on line 19. The *command group* describes a sequence of enqueue operations to a given queue, which is passed into the constructor in this case as `myQueue`. In this case the constructor used for `myQueue` on line 13 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 *kernel function* must be contained within a *buffer* or *image*. We construct a buffer on line 16. Access to the buffer is controlled via an *accessor* which is constructed on line 22. 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 *kernel functions*. In this example the *accessor* is used to write to the data buffer on line 27. All *buffers* must be constructed in the *application scope*, whereas all *accessors* must be constructed in the *command group scope*.

2.7 Memory objects

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 (`cl::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 OpenCL implementation. Images are encapsulated in the `cl::sycl::image` type, which is 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 can be converted within a kernel into C++ pointer types, or the `cl::sycl::global_ptr`, `cl::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 stored in the same format as it is accessed by the kernel, but in the case of an image object the data format used to store the image elements may not be the same as the data format used inside the kernel. 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 it is stored into 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.

Memory objects, both buffers and images, may have one or more underlying OpenCL `cl_mem` objects. 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.

2.7.1 Storage objects

Users may want fine-grained control of the synchronization 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. This fine-grained control is provided by user-defined *storage objects*.

For a user to make use of the storage object feature, they must first define their own storage object class, which is derived from the `cl::sycl::storage` abstract class. They then construct an object of this class and pass it into the constructor for the image or buffer instead of passing in any associated host memory. The storage object will be called by the system on key synchronization operations, such as the destructor on the memory object being called, or the underlying SYCL system asynchronously no longer requiring access to the data. The user's storage object can implement these methods to provide the destruction, synchronization and memory management features that the user requires.

The storage object is not responsible for copying or mapping data. However, it will be queried by the SYCL runtime to determine if and where data copying or mapping may be required.

2.8 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 layer provides a set of C++ templates and classes which provide the programming model to the user. It enables the creation of queues, buffers and images, as well as access to some underlying OpenCL features 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 a SYCL-specific host device to run kernels on.

- *SYCL Device Compiler(s)*: 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.

2.9 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.

2.9.1 Building a SYCL program

A SYCL program runs on a *host* and one or more OpenCL 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 host.

The design of SYCL enables a single SYCL source file to be passed to multiple, different compilers. 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 OpenCL 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 be 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 compiler and the 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 (potentially a macro) 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.

2.9.2 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 runtime to load the kernel and pass it to an OpenCL runtime. This requires the kernel to have a globally-visible name to enable an association between the kernel invocation and the kernel itself. The association is achieved using a *kernel name*, which is a C++ typename. For a functor, the kernel name can be the same type as the functor itself, as long as the functor type is globally accessible. For a C++11 lambda function, there is no globally-visible name, so the user must provide one. In SYCL, the name is provided as a template parameter to the kernel invocation, e.g. `parallel_for<kernelname>`.

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 user can also extract OpenCL `cl_kernel` and `cl_program` objects for kernels by providing the typename of the kernel.

2.10 Language restrictions in kernels

The following restrictions are applied to device functions and kernels:

- Any class shared between host and SYCL devices must be C++11 *standard layout*
- Pointer data should not be shared between host and SYCL devices. 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.
- No virtual methods or function pointers are allowed to be called on device.
- No class with a vtable can be used in kernel code.
- RTTI and exception-handling cannot be used on in kernel code.
- Recursion is not allowed in kernel code.
- Global or static variables are not allowed to be used in kernel code.
- Static member variables are not allowed to be used in kernel code.

There are also restrictions on the kinds of data that may be passed to a kernel from the host as parameters:

- It is not possible to pass non-standard-layout parameters from host to device.
- No parameter to a kernel can be a pointer or reference type.
- The following SYCL classes can be passed as parameters to SYCL kernels, or within struct parameters to SYCL kernels: `accessor`, `sampler`, `vec<T,dim>` and the POD types such as `id`, `range`, `nd_range`.
- If any parameter is a struct or class, all of its members must also follow the rules above. The one exception to this rule is that structs and classes can contain pointers and be passed as parameters to kernels, however those pointer values are not defined to be usable inside a kernel.

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`. It is the responsibility of the SYCL device compiler to ensure that the sizes of these types match the sizes on the host, to enable data of these types to be shared between host and device.

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 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.

2.10.1 Functions and datatypes available in kernels

Inside kernels, the functions and datatypes available are restricted by the underlying capabilities of OpenCL devices. All OpenCL C features are provided by C++ classes and functions, which are available on host and device.

2.11 Execution of kernels on the SYCL host device

SYCL enables kernels to run on either the host device or on OpenCL devices. When kernels run on an OpenCL device, then the features and behaviour of that execution follows the OpenCL specification, otherwise they follow the behaviour specified for the SYCL host device.

Any kernel enqueued to a host queue (either by a command group's primary or secondary queue) executes on host according to host execution rules.

Kernel math library functions on the host must conform to OpenCL math precision requirements.

The range of image formats supported by the host device is implementation- defined, but must match the minimum requirements of the OpenCL specification.

Some of the OpenCL extensions and optional features may be available on a SYCL host device, but since these are optional features and vendor specific extensions, the user must query the host device to determine availability. A SYCL implementer must state what OpenCL device features are available on their host device implementation.

The synchronization and data movement that occurs when a kernel is executed on the host may be implemented in a variety of ways on top of OpenCL. The actual mechanism is implementation-defined.

2.12 Example SYCL application

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

```
1  #include <CL/sycl.hpp>
2  #include <iostream>
3
4  using namespace cl::sycl;
5
```

```
6  // Size of the matrices
7  const size_t N = 2000;
8  const size_t M = 3000;
9
10 int main() {
11   { // By including all the SYCL work in a {} block, we ensure
12     // all SYCL tasks must complete before exiting the block
13
14     // Create a queue to work on
15     queue myQueue;
16
17     // Create some 2D buffers of float for our matrices
18     buffer<float, 2> a({ N, M });
19     buffer<float, 2> b({ N, M });
20     buffer<float, 2> c({ N, M });
21
22     // Launch a first asynchronous kernel to initialize a
23     command_group (myQueue, [&] () {
24         // The kernel write a, so get a write accessor on it
25         auto A = a.get_access<access::write>();
26
27         // Enqueue a parallel kernel iterating on a N*M 2D iteration space
28         parallel_for<class init_a>(range<2>( N, M ),
29                         [=] (id<2> index) {
30                           A[index] = index[0]*2 + index[1];
31                         });
32       });
33
34     // Launch an asynchronous kernel to initialize b
35     command_group (myQueue, [&] () {
36         // The kernel write b, so get a write accessor on it
37         auto B = b.get_access<access::write>();
38         /* From the access pattern above, the SYCL runtime detect this
39            command_group is independant from the first one and can be
40            scheduled independently */
41
42         // Enqueue a parallel kernel iterating on a N*M 2D iteration space
43         parallel_for<class init_b>(range<2>( N, M ),
44                         [=] (id<2> index) {
45                           B[index] = index[0]*2014 + index[1]*42;
46                         });
47     });
48
49     // Launch an asynchronous kernel to compute matrix addition c = a + b
50     command_group (myQueue, [&] () {
51         // In the kernel a and b are read, but c is written
52         auto A = a.get_access<access::read>();
53         auto B = b.get_access<access::read>();
54         auto C = c.get_access<access::write>();
55         // From these accessors, the SYCL runtime will ensure that when
56         // this kernel is run, the kernels computing a and b completed
57
58         // Enqueue a parallel kernel iterating on a N*M 2D iteration space
59         parallel_for<class matrix_add>(range<2>( N, M ),
60                         [=] (id<2> index) {
```

```
61                          C[index] = A[index] + B[index];
62                        });
63       });
64
65     /* Ask an access to read c from the host-side. The SYCL runtime
66        ensures that c is ready when the accessor is returned */
67     auto C = c.get_access<access::read, access::host_buffer>();
68     std::cout << std::endl << "Result:" << std::endl;
69     for(size_t i = 0; i < N; i++)
70       for(size_t j = 0; j < M; j++)
71         // Compare the result to the analytic value
72         if (C[i][j] != i*(2 + 2014) + j*(1 + 42)) {
73           std::cout << "Wrong value " << C[i][j] << " on element "
74                     << i << " " << j << std::endl;
75           exit(-1);
76         }
77
78   } /* End scope of myQueue, this wait for any remaining operations on the
79        queue to complete */
80
81   std::cout << "Good computation!" << std::endl;
82   return 0;
83 }
```

3. SYCL Programming Interface

The SYCL programming interface provides a C++ abstraction to OpenCL 1.2 functionality and feature set. This section describes all the available classes and interfaces of SYCL, focusing on the C++ interface to the underlying runtime. In this section, we are defining all the classes and methods for the SYCL API, which are available for host and device code. This section also describes the synchronization rules and OpenCL API interoperability rules which guarantee that all the methods, including constructors, of the SYCL classes are thread safe.

It is assumed that the OpenCL API is also available to the developer at the same time as SYCL.

3.1 Header files and namespaces

SYCL provides one standard header file: `"CL/sycl.hpp"`, which needs to be included in every SYCL program.

All SYCL classes, constants, types and functions are defined within the `cl::sycl` namespace.

3.2 Error handling

Error handling in SYCL uses exceptions. If an error can be propagated at the point of calling a function, an exception will be thrown and may be caught by the user using standard C++ exception handling mechanisms.

Due to the asynchronous execution of SYCL programs, some errors cannot be propagated directly from the call site because they will not be detected until the error-causing task executes, or tries to execute. These errors are called asynchronous errors.

For asynchronous errors in queues, the queue constructor can be provided with a functor, or lambda function, which receives a list of C++ exception objects. These errors will be passed to the asynchronous error handler at appropriate wait points on queues and events.

If an asynchronous error occurs in a context that has no user-supplied asynchronous error handler, then no exception is thrown and the error is not available to the user in any specified way. Implementations may provide extra debugging information to users to trap and handle asynchronous errors.

If an 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. If the command group is successfully enqueued on a secondary queue instead of the primary queue, then no exceptions are thrown and no asynchronous errors are reported via the asynchronous error handler. In the case of failure to enqueue a command group on its secondary queue, for whatever reason, then errors are reported.

```
namespace cl{
  namespace sycl{
```

```
    struct exception
    {
      string_class get_description ();

      // returns associated context. nullptr if none
      context *get_context();
    };

    struct cl_exception: exception
    {
      // thrown as a result of an OpenCL API error code
      cl_int get_cl_code() const;
    };

    struct async_exception: exception
    {
      // stored in an exception_list for asynchronous errors
    };

    class exception_list : public exception
    {
      // Used as a container for a list of asynchronous exceptions
    public:
      typedef exception_ptr value_type;
      typedef const value_type& reference;
      typedef const value_type& const_reference;
      typedef size_t size_type;
      typedef /*unspecified*/ iterator;
      typedef /*unspecified*/ const_iterator;

      size_t size() const;
      iterator begin() const; // first asynchronous exception
      iterator end() const;   // lasst asynchronous exception
    };

    typedef /*unspecified*/ exception_ptr;
  }  // namespace sycl
}  // namespace cl
```

The `cl_exception` class is the exception thrown when the OpenCL API returns an error code. The OpenCL error code can be queried with the `get_cl_code` method. The `async_exception` is stored in `exception_list` objects and is generated when an asynchronous error occurs on a SYCL-managed context. The `exception_ptr` class is used to store `cl::sycl::exception` objects and allows exception objects to be transferred between threads. It is equivalent to the `std::exception_ptr` class.

Methods	Description
`string_class get_description()`	Returns a descriptive string for the error, if available.
`context *get_context()`	Returns the context that caused the error. Returns null if not a buffer error. The pointer is to an object that is only valid as long as the exception object is valid
`cl_int get_cl_code()`	Returns the OpenCL error code encapsulated in the exception. Only valid for the `cl_exception`<s>ubclass.

Table 3.1: Methods for the `exception` class and its subclasses.

3.3 Vector, string and function classes in interfaces

The SYCL programming interfaces make extensive use of vectors, strings and functions objects to carry information. SYCL will default to using the STL string, vector and function classes, unless told otherwise. These types are exposed internally as `cl::sycl::vector_class`, `cl::sycl::string_class` and `cl::sycl::function_class`.

It is possible to disable the STL versions of these classes when required. A common reason for doing this is to specify a custom allocator to move memory management under the control of the SYCL user. This is achieved by defining `CL_SYCL_NO_STD_VECTOR`, `CL_SYCL_NO_STD_STRING`, `CL_SYCL_NO_STD_FUNCTION` respectively, before including `"CL/sycl.hpp"`, and by replacing the template aliases in the `cl::sycl` namespace as necessary.

```
namespace cl {
  namespace sycl {{
    #define CL_SYCL_NO_STD_VECTOR
    #include <vector>
    template < class T, class Alloc = std::allocator<T> >
    using vector_class = std::vector<T, Alloc>;

    #define CL_SYCL_NO_STD_STRING
    #include <string>
    using string_class = std::string;

    #define CL_SYCL_NO_STD_FUNCTION
    #include <functional>
    using function_class = std::function

    #include <cl/sycl.hpp>
  }
}
```

3.4 Synchronization

The SYCL specification offers the same set of synchronization operations that are available to OpenCL C programs, for compatibility and portability across OpenCL devices. 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: OpenCL 1.2 devices only support the equivalent of relaxed C++ atomics and SYCL uses the C++11 library syntax to make this available. This is provided for forward compatibility with future SYCL versions.
- Barriers: Barrier primitives are made available to synchronize sets of work-items within individual work-groups. They are exposed through the `nd_item` class that abstracts the current point in the overall iteration space.
- 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_workgroup` function call, rather than through the use of explicit barrier operations.

Barriers may provide ordering semantics over the local address space, global address space or both. All memory operations initiated before the barrier in the specified address space(s) will be completed before any memory operation after the barrier. Address spaces are described using the `fence_space` enum class:

```
namespace cl {
  namespace sycl {
    namespace access {
      enum class fence_space :char {
        local,
        global,
        global_and_local
      } // enum class address_space
    } // namepaces access
  } // namespace sycl
} // namespace cl
```

The SYCL specification provides atomic operations based on the C++11 library syntax. The only available ordering, due to constraints of the OpenCL 1.2 memory model, is `memory_order_relaxed`. No default order is supported because a default order would imply sequential consistency. The SYCL atomic library may map directly to the underlying C++11 library in host code, and must interact safely with the host C++11 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 1.2 device compilers should give a compilation error on use of the `std::atomic` classes and functions in device code. Only `atomic<int>`, `atomic<unsigned int>` and `atomic<float>` types are available in SYCL 1.2. Only the `exchange` operation is available for float atomics.

No construction of atomic objects is possible in SYCL 1.2. All atomic objects must be obtained by-reference from an accessor (see Section 3.6.4.3).

The atomic types are defined as follows, and methods are listed in Table 3.2:

```
namespace cl {
```

```
namespace sycl {
    template<typename T>
    class atomic<T> {
        public:
        // Constructors
        atomic() = delete;

        // Methods
        // Only memory_order_relaxed is supported in SYCL 1.2
        void store( T, std::memory_order );
        void store( T, std::memory_order ) volatile;
        T load( memory_order ) const;
        T load( memory_order ) const volatile;

        T exchange( T, std::memory_order );
        T exchange( T, std::memory_order ) volatile;

        T compare_exchange_strong(
            T*, T, std::memory_order success, std::memory_order fail );
        T compare_exchange_strong(
            T*, T, std::memory_order success, std::memory_order fail ) volatile;

        T fetch_add( T, std::memory_order );
        T fetch_add( T, std::memory_order ) volatile;

        T fetch_sub( T, std::memory_order );
        T fetch_sub( T, std::memory_order ) volatile;

        T fetch_and( T, std::memory_order );
        T fetch_and( T, std::memory_order ) volatile;

        T fetch_or( T, std::memory_order );
        T fetch_or( T, std::memory_order ) volatile;

        T fetch_xor( T, std::memory_order );
        T fetch_xor( T, std::memory_order ) volatile;

        // Additional functionality provided beyond that of C++11
        T fetch_min( T, std::memory_order );
        T fetch_min( T, std::memory_order ) volatile;

        T fetch_max( T, std::memory_order );
        T fetch_max( T, std::memory_order ) volatile;
    };

    typedef atomic<int> atomic_int;
    typedef atomic<unsigned int> atomic_uint;
    typedef atomic<float> atomic_float;
  } // namespace sycl
} // namespace cl
```

As well as the methods, a matching set of operations on atomic types is provided by the SYCL library. As in the previous case, the only available memory order in SYCL 1.2 is `memory_order_relaxed`. The global functions are as follows and described in Table 3.3.

```
namespace cl {
  namespace sycl {
      template<class T> T atomic_load_explicit( atomic<T>*, std::memory_order );
      template<class T> T atomic_load_explicit( volatile atomic<T>*, std::memory_order );

      template<class T> void atomic_store_explicit( atomic<T>*, T, std::memory_order );
      template<class T> void atomic_store_explicit( volatile atomic<T>*, T, std::memory_order );

      template<class T> T atomic_exchange_explicit( atomic<T>*, T, std::memory_order );
      template<class T> T atomic_exchange_explicit( volatile atomic<T>*, T, std::memory_order );

      template<class T> bool atomic_compare_exchange_strong_explicit(
          atomic<T>*, T*, T, std::memory_order success, std::memory_order fail);
      template<class T> bool atomic_compare_exchange_strong_explicit(
          volatile atomic<T>*, T*, T, std::memory_order success, std::memory_order fail);

      template<class T> T atomic_fetch_add_explicit( atomic<T>*, T, std::memory_order );
      template<class T> T atomic_fetch_add_explicit( volatile atomic<T>*, T, std::memory_order );

      template<class T> T atomic_fetch_sub_explicit( atomic<T>*, T, std::memory_order );
      template<class T> T atomic_fetch_sub_explicit( volatile atomic<T>*, T, std::memory_order );

      template<class T> T atomic_fetch_and_explicit( atomic<T>*, T, std::memory_order );
      template<class T> T atomic_fetch_and_explicit( volatile atomic<T>*, T, std::memory_order );

      template<class T> T atomic_fetch_or_explicit( atomic<T>*, T, std::memory_order );
      template<class T> T atomic_fetch_or_explicit( volatile atomic<T>*, T, std::memory_order );

      template<class T> T atomic_fetch_xor_explicit( atomic<T>*, T, std::memory_order );
      template<class T> T atomic_fetch_xor_explicit( volatile atomic<T>*, T, std::memory_order );

      // Additional functionality beyond that provided by C++11
      template<class T> T atomic_fetch_min_explicit( atomic<T>*, T, std::memory_order );
      template<class T> T atomic_fetch_min_explicit( volatile atomic<T>*, T, std::memory_order );

      template<class T> T atomic_fetch_max_explicit( atomic<T>*, T, std::memory_order );
      template<class T> T atomic_fetch_max_explicit( volatile atomic<T>*, T, std::memory_order );
  } // namespace sycl
} // namespace cl
```

The atomic operations and methods behave as described in the C++11 specification, barring the restrictions discussed above. Note that car must be taken when using `compare_exchange_strong` to perform many of the operations that would be expected of it in standard CPU code 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.

Methods	Description
`T load( memory_order )const;`	Atomically load the current value of `*this` and return that value. T may be int, unsigned int or float. `order` must be `memory_order_relaxed`.
`void store( T operand, std::memory_order order );`	Atomically store `operand` in `*this`. T may be int, unsigned int or float. `order` must be `memory_order_relaxed`.
`T exchange( T operand, std::memory_order order );`	Atomically replace `*this` with `operand`. Return the original value of object. T may be int, unsigned int or float. `order` must be `memory_order_relaxed`.

`T compare_exchange_strong(`
`  T* expected,`
`  T desired,`
`  std::memory_order success,`
`  std::memory_order fail );`
	Atomically compare the value of `*this` against `*expected`. If equal replace `*this` with `desired` otherwise store the original value of `*this` in `*expected`. Returns true if the comparison succeeded.
	Both memory orders must be `memory_order_relaxed`. T must be int or unsigned int.

`T fetch_add( T operand, std::memory_order order );`	Atomically add `operand` to `*this`. Store the result in `*this`. `order` must be `memory_order_relaxed`. T must be int or unsigned int.
`T fetch_sub( T operand, std::memory_order order );`	Atomically subtract `operand` from `*this`. Store the result in `*this`. `order` must be `memory_order_relaxed`. T must be int or unsigned int.
`T fetch_and( T operand, std::memory_order order );`	Atomically perform a bitwise and of `operand` and `*this`. Store the result in `*this`. `order` must be `memory_order_relaxed`. T must be int or unsigned int.
`T fetch_or( T operand, std::memory_order order );`	Atomically perform a bitwise or of `operand` and `*this`. Store the result in `*this`. `order` must be `memory_order_relaxed`. T must be int or unsigned int.
`T fetch_xor( T operand, std::memory_order order );`	Atomically perform a bitwise exclusive-or of `operand` and `*this`. Store the result in `*this`. `order` must be `memory_order_relaxed`. T must be int or unsigned int.
`T fetch_min( T operand, std::memory_order order );`	Atomically compute the minimum of `operand` and `*this`. Store the result in `*this`. `order` must be `memory_order_relaxed`. T must be int or unsigned int.
`T fetch_max( T operand, std::memory_order order );`	Atomically compute the maximum of `operand` and `*this`. Store the result in `*this`. `order` must be `memory_order_relaxed`. T must be int or unsigned int.

Table 3.2: Methods available on an object of type `atomic<T>`.

Functions	Description
`template<class T> T atomic_load_explicit( atomic<T>* object, std::memory_order order);`	Atomically load the current value of `object` and return that value. T may be int, unsigned int or float. `order` must be `memory_order_relaxed`.
`template<class T> void atomic_store_explicit( atomic<T>* object, T operand, std::memory_order order);`	Atomically store `operand` in `object`. T may be int, unsigned int or float. `order` must be `memory_order_relaxed`.
`template<class T> T atomic_exchange_explicit( atomic<T>* object, T operand, std::memory_order order);`	Atomically replace `object` with `operand`. Return the original value of object. T may be int, unsigned int or float. `order` must be `memory_order_relaxed`.
`template<class T> bool atomic_compare_exchange_strong_explicit( atomic<T>* object, T* expected, T desired, std::memory_order successes, std::memory_order fail);`	Atomically compare the value of `object` against `expected`. If equal replace `object` with `desired`. Otherwise store the original value of `object` in `expected`. Returns true if the comparison succeeded. Both memory orders must be `memory_order_relaxed`. T must be int or unsigned int.
`template<class T> T atomic_fetch_add_explicit( atomic<T>* object, T operand, std::memory_order order);`	Atomically add `operand` to `object`. Store the result in `object`. `order` must be `memory_order_relaxed`. T must be int or unsigned int.
`template<class T> T atomic_fetch_sub_explicit( atomic<T>* object, T operand, std::memory_order order);`	Atomically subtract `operand` from `object`. Store the result in `object`. `order` must be `memory_order_relaxed`. T must be int or unsigned int.
`template<class T> T atomic_fetch_and_explicit( atomic<T>* operand, T object, std::memory_order order);`	Atomically perform a bitwise and of `operand` and `object`. Store the result in `object`. `order` must be `memory_order_relaxed`. T must be int or unsigned int.
`template<class T> T atomic_fetch_or_explicit( atomic<T>* object, T operand, std::memory_order order);`	Atomically perform a bitwise or of `operand` and `object`. Store the result in `object`. `order` must be `memory_order_relaxed`. T must be int or unsigned int.
`template<class T> T atomic_fetch_xor_explicit( atomic<T>* object, T operand, std::memory_order order);`	Atomically perform a bitwise exclusive-or of `operand` and `object`. Store the result in `object` . `order` must be `memory_order_relaxed`. T must be int or unsigned int.
`template<class T> T atomic_fetch_min_explicit( atomic<T>* object, T operand, std::memory_order order);`	Atomically compute the minimum of `operand` and `object`. Store the result in `object`. `order` must be `memory_order_relaxed`. T must be int or unsigned int.
`template<class T> T atomic_fetch_max_explicit( atomic<T>* object, T operand, std::memory_order order);`	Atomically compute the maximum of `operand` and `object`. Store the result in `object`. `order` must be `memory_order_relaxed`. T must be int or unsigned int.

Table 3.3: Global functions available on atomic types.

3.5 Platforms, contexts, devices and queues

3.5.1 Device selection class

The class `device_selector` is a functor which enables the SYCL runtime to choose the best device based on heuristics specified by the user, or by one of the built-in device selectors. The built-in device selectors are listed in Table 3.4.

```
namespace cl {
  namespace sycl {
    class device_selector {
     public:
      explicit device_selector();

      virtual ~device_selector();

      device select_device() const;

      virtual int operator()(const device &device) const = 0;
    };
  }  // namespace sycl
}  // namespace cl
```

`operator()` is an abstract method which returns a "score" per-device. At the stage where the SYCL runtime selects a device, the system will go through all the available devices in the system and choose the one with the highest score as computed by the current device selection class. If a device has a negative score it will never be chosen. While OpenCL devices may or may not be available, the SYCL host device is always available, so the developer is able to choose the SYCL host device as a fall-back. Selection of the SYCL host device will allow execution of CPU-compiled versions of kernels through all queues created against that device.

The system also provides built-in device selectors, including selectors which choose a device based on the default behavior of the system. An important note is that the system has no global state and its behavior is defined by the platforms the developer chooses to target.

The *default_selector* is the selector that incorporates the *default* behavior of the system, and it is implicitly used by the system for the creation of any SYCL objects when no other *device_selector* or underlying OpenCL identifier is provided. The method the default selector uses to rank and select devices is implementation-defined. The *default_selector* will have the SYCL Host Mode enabled and if there are no OpenCL devices available, it will select the SYCL Host Platform instead.

SYCL Device Selectors	Description
default_selector	Devices selected by heuristics of the system. If no OpenCL device is found then the execution is executed on the SYCL Host Mode.
gpu_selector	Select devices according to device type CL_DEVICE_TYPE_GPU from all the available OpenCL devices. If no OpenCL GPU device is found the selector fails.
cpu_selector	Select devices according to device type CL_DEVICE_TYPE_CPU from all the available devices and heuristics. If no OpenCL CPU device is found the selector fails.
host_selector	Selects the SYCL host CPU device that does not require an OpenCL runtime.

Table 3.4: Standard device selectors included with all SYCL implementations.

3.5.2 Platform class

The platform class represents a SYCL platform: a collection of related SYCL supported devices. Each platform may be either an OpenCL platform, and thus have a cl_platform_id queryable through the get() method, or it may be the SYCL Host platform, containing only the SYCL Host. The host platform may be useful when no OpenCL platform is available or when the host compiler offers better performance on the CPU. The platform class offers a selection of static methods to obtain information about the platforms available at runtime.

The constructors and methods of the Platform class are listed in Tables 3.5 and 3.6.

```
namespace cl {
  namespace sycl {
    class platform {
    public:
      platform();
      explicit platform(cl_platform_id platformID);
      platform(const platform &rhs);
      platform &operator=(const platform &rhs);

      ~platform();

      cl_platform_id get() const;

      bool is_host() const;

      template <cl_int param>
      typename param_traits<detail::cl_platform_info, param>::type
        get_info() const;

      bool has_extension(string_class extension) const;

      vector_class<device> get_devices(
        cl_device_type deviceType = CL_DEVICE_TYPE_ALL) const;
```

```
    static vector_class<platform> get_platforms();
  };
 }  // namespace sycl
}  // namespace cl
```

The SYCL Host Mode platform is not an OpenCL platform. Consequently, the *get()* method when on SYCL Host Mode will not return a valid *cl_platform_id* and it will not be included in the output of by the static function *get_platforms*.

Constructors	**Description**
`platform()`	Default constructor for platform. It constructs a platform object to encapsulate the device returned by the default *device_selector*. Returns errors via C++ exception class.
`platform(cl_platform_id platform_id)`	Construct a platform object from an OpenCL platform id. Returns errors via SYCL C++ exception class.
`platform(device_selector &dev_selector)`	Construct a platform object from the device returned by a device selector of the user's choice. Returns errors via C++ exception class.
`platform(const platform &rhs)`	Copy construct a platform from another.

Table 3.5: Constructors of platform class

The default constructor will create an instance of the platform class where the underlying platform will be chosen according to the *default_selector*, please see 3.5.1 for the set of rules that apply on chosing a valid SYCL platform.

Methods	Description
`cl_platform_id get ()const`	Returns the *cl_platform_id* of the underlying OpenCL platform. If the platform is not a valid OpenCL platform, for example it is the SYCL host, a null *cl_platform_id* will be returned.
`static vector_class<platform> get_platforms ()const`	Returns all available platforms in the system.
`static vector_class<device> get_devices( cl_device_type device_type=CL_DEVICE_TYPE_ALL) const`	Returns all the available devices for all platforms.
`vector_class<device> get_devices( cl_device_type device_type=CL_DEVICE_TYPE_ALL) const`	Returns all the available devices for this platform.
`static vector_class<device> get_devices ( cl_device_type device_type)const`	Returns all the available devices of type *cl_device_type* for all platforms.
`vector_class<device> get_devices ( cl_device_type device_type)const`	Returns all the available devices of type *cl_device_type* for this platform.
`template<cl_int name> typename param_traits <cl_platform_info,name>::param_type get_info ()const`	Queries the platform for *cl_platform_info*. See the *cl_platform_info* table in the OpenCL specification for a full list of options to pass to the template parameter. The return value will be correctly typed as described in the table.
`bool has_extension (string_class extension)const`	Specifies whether a specific extension is supported on the platform.
`bool is_host ()const`	Returns true if this is a SYCL host platform.

Table 3.6: Methods of platform class

3.5.3 Device class

The SYCL `device` class encapsulates a particular SYCL device against which the runtime may create queues and on which the runtime may execute kernels. The SYCL device may be an OpenCL device, in which case it should have valid `cl_device_id` and `cl_platform_id` available, or it may be a SYCL host device representing the host CPU. If a SYCL device is constructed from an existing `cl_device_id` the system will call `clRetainDevice`. On destruction the runtime will call `clReleaseDevice`.

The constructors and methods of the Device class are listed in Tables 3.7 and 3.8.

```
namespace cl {
  namespace sycl {
    class device {
    public:
      device();

      explicit device(cl_device_id deviceID);

      device(const device &rhs);

      device &operator=(const device &rhs);

      ~device();

      cl_device_id get() const;

      bool is_host() const;

      template <cl_int param>
      typename param_traits<cl_device_info, param>::type get_info() const;

      bool has_extension(string_class extension) const;

      platform get_platform() const;

      vector_class<device> create_sub_devices(
          cl_device_partition_property *properties, cl_int devices,
          cl_uint *numDevices) const;

      static vector_class<device> get_devices(cl_device_type deviceType =
                                              CL_DEVICE_TYPE_ALL);

      detail::device *get_impl();
    };
  }  // namespace sycl
}  // namespace cl
```

The default constructor will create an instance of the device class where the underlying platform will be chosen according to the *default_selector*, please see 3.5.1 for the set of rules that apply on chosing a valid SYCL device.

The developer can partition existing devices through the `create_sub_devices` API. More documentation on this is in the OpenCL 1.2 specification [1, sec. 4.3]. It is valid to construct a SYCL device directly from an OpenCL sub-device.

Information about the SYCL device may be queried through the `get_info` method. In the case of a SYCL Host device this method is invalid, as there is no OpenCL info available for it.

The developer can also query the device instance for the `cl_device_id` and its corresponding `cl_platform_id`. However, in the case of the *SYCL Host Mode device*, there is no platform associated OpenCL device id or platform id, so the outcome of the query is undefined.

To facilitate the different options for SYCL devices, there are methods that check the type of device. The method `is_host()` returns true if the device is actually the host. In the case where an OpenCL device has been initialized through this API, the methods `is_cpu()` and `is_gpu()` return true if the OpenCL device is either CPU or GPU.

Constructors	**Description**
`device ()`	Default constructor for the device. It choses a device using *default_selector*. Returns errors via C++ exception class.
`device (device_selector &selector)`	Constructs a device class instance using the *device_selector* provided. Returns errors via C++ exception class.
`device (cl_device_id device_id)`	Constructs a device class instance using *cl_device_id* of the OpenCL device. Returns errors via C++ exception class.
`device (const device &rhs)`	Copy constructor. Returns errors via C++ exception class.

Table 3.7: Constructors for device class

Methods	**Description**
`cl_device_id get ()const`	Returns the *cl_device_id* of the underlying OpenCL platform. Returns errors via C++ exception class.
`platform get_platform ()const`	Returns the platform of device.
`bool is_host ()const`	Returns true if the device is a SYCL host device.
`template <cl_int param> typename param_traits <cl_device_info, param>::type get_info ()const`	Queries the device for OpenCL *cl_device_info*.
`bool has_extension (string_class extension)const`	Specifies whether a specific extension is supported on the device.
`vector_class<device> create_sub_devices ( cl_device_partition_property *properties, cl_int devices, cl_uint *numDevices)const`	Partitions the device into sub devices based upon the properties provided.
`static vector_class<device> get_devices ( cl_device_type deviceType = CL_DEVICE_TYPE_ALL)`	Returns a list of all available devices. Returns errors via C++ exception class.

Table 3.8: Methods of the device class

3.5.4 Context class

The class `context` encapsulates an OpenCL context, which is implicitly created and the lifetime of the context instance defines the lifetime of the underlying OpenCL context instance. On destruction `clReleaseContext` is called.

The constructors and methods of the Context class are listed in Tables 3.9 and 3.10.

```
namespace cl {
  namespace sycl {
    class context {
    public:
      context();

      explicit context(cl_context context);

      context(const device_selector &deviceSelector,
              cl_context_properties *properties = nullptr);

      context(const device &dev, cl_context_properties *properties = nullptr);

      context(const platform &plt, cl_context_properties *properties = nullptr);

      context(vector_class<device> deviceList,
              cl_context_properties *properties = nullptr);

      /* constructors with asynchronous error handler supplied */
      context (function_class &async_handler);
      context(const device_selector &deviceSelector,
              cl_context_properties *properties = nullptr,
              function_class &async_handler);
      context(const device &dev, cl_context_properties *properties = nullptr,
              function_class &async_handler);
      context(const platform &plt, cl_context_properties *properties = nullptr,
              function_class &async_handler);
      context(vector_class<device> deviceList,
              cl_context_properties *properties = nullptr,
              function_class &async_handler);

      context(const context &rhs);

      context &operator=(const context &rhs);

      virtual ~context();

      cl_context get() const;

      bool is_host() const;

      template <cl_int param>
      typename param_traits<detail::cl_context_info, param>::type get_info() const;

      vector_class<cl::sycl::device> get_devices() const;
    };
```

```
  }  // namespace sycl
}  // namespace cl
```

Constructors	Description
`context ()`	Default constructor that chooses the context ~~according the~~ heuristics of the *default_selector*. Returns errors via C++ exception class.
`context (cl_context context)`	Context constructor, where the underlying OpenCL context is given as a parameter. The constructor executes a retain on the *cl_context*. Returns errors via C++ exception class.
`context (const device_selector &deviceSelector, cl_context_properties *properties = nullptr)`	Constructs a context object using a device_selector object. The context is constructed with a single device retrieved from the device_selector object provided. Returns errors via C++ exception class.
`context (device &dev, cl_context_properties *properties = nullptr)`	Constructs a context object using a device object.
`context(platform &plt, cl_context_properties *properties = nullptr)`	Constructs a context object using a platform object.
`context(vector_class<device> deviceList, cl_context_properties *properties = nullptr)`	Constructs a context object using a vector_class of device objects.
`context(function_class &async_handler)`	Constructs a context object for SYCL host using an async_handler for handling asynchronous errors.
`context(const device_selector &deviceSelector, cl_context_properties *properties = nullptr, function_class &async_handler)`	Constructs a context object using a device_selector object. The context is constructed with a single device retrieved from the device_selector object provided. Returns errors via C++ exception class and asynchronous errors are handled via the async_handler.
`context(const device &dev, cl_context_properties *properties = nullptr, function_class &async_handler)`	Constructs a context object using a device object. Returns errors via C++ exception class and asynchronous errors are handled via the async_handler.
`context(const platform &plt, cl_context_properties *properties = nullptr, function_class &async_handler)`	Constructs a context object using a platform object. Returns errors via C++ exception class and asynchronous errors are handled via the async_handler.
`context(vector_class<device> deviceList, cl_context_properties *properties = nullptr, function_class &async_handler)`	Constructs a context object using a vector_class of device objects. Returns errors via C++ exception class and asynchronous errors are handled via the async_handler.
`context (cl_context context, function_class &async_handler)`	Context constructor, where the underlying OpenCL context is given as a parameter. The constructor executes a retain on the *cl_context*. Returns errors via C++ exception class and the asynchronous errors are handled via the async_hander.
`context(const context &rhs)`	Constructs a context object from another context object and retains the cl_context object if the context is not SYCL host.

Table 3.9: Constructors for context class

Methods	Description
`cl_context get ()const`	Returns the underlying *cl_context* object, after retaining the cl_context.
`bool is_host ()const`	Specifies whether the context is in SYCL Host Execution Mode.

`template <cl_int param>`
`typename param_traits`
`<cl_context_info, param>::type`
`get_info ()const`
	Queries OpenCL information for the underlying *cl_context*.

Table 3.10: Methods of context class

3.5.4.1 Context error handling

On construction of a context, it is possible to supply an asynchronous error handler function object. If supplied, then asynchronous errors can be reported to the error handler. Asynchronous errors are only reported to the user when a queue attached to the context is destroyed or as its `wait_and_throw` method called.

3.5.5 Queue class

The class `queue` is a SYCL's encapsulation of an OpenCL `cl_command_queue`. Can be constructed from a `cl_command_queue`. The destructor waits for all execution on the queue to end and then passes any exceptions that occurred asynchronously on the queue to the asynchronous error handler, if provided to the constructor, before calling `clReleaseCommandQueue`.

The constructors and methods of the Queue class are listed in Tables 3.11, 3.12 and 3.13.

```
namespace cl {
  namespace sycl {
    class queue {
    public:
      queue();

      queue(function_class &async_handler);

      explicit queue(const device_selector &deviceSelector);

      queue(const device_selector &deviceSelector,
            function_class &async_handler);

      queue(const context &syclContext, const device_selector &selector);

      queue(const context &syclContext, const device_selector &selector,
            function_class &async_handler);

      queue(const context &syclContext, const device &device);

      queue(const context &syclContext, const device &device,
           function_class &async_handler);

      queue (const context &dev_context, const device &dev_device,
             cl_command_queue_properties * properties = nullptr);

      queue (const context &dev_context, const device &dev_device,
             cl_command_queue_properties * properties = nullptr,
             function_class &async_handler);

      explicit queue(const device &device);

      queue(const device &device, function_class &async_handler);

      queue(cl_command_queue cl_queue);

      queue(cl_command_queue cl_queue, function_class &async_handler);

      ~queue();

      bool is_host();

      context get_context();

      cl_command_queue get();
```

```
      device get_device();

      context get_context();

      template <cl_int param>
      typename param_traits<cl_command_queue_info, param>::type
        get_info() const;

      void wait();

      void wait_and_throw();

      void throw_asynchronous();
    };
  } // namespace sycl
} // namespace cl
```

Constructors (Part I)	Description
`queue ()`	Default constructor that creates a queue for a device it chooses ~~according the~~ heuristics of the *default_selector*. The OpenCL context object is created ~~implicitely.~~
`queue (function_class &async_handler)`	Creates a queue for a device it chooses according the heuristics of the*default_selector*. The OpenCL context object is created implicitely. Reports asynchronous errors using the *async_handler* callback function.
`explicit queue (const device_selector &selector)`	Creates a queue for the device provided by the device_selector. If no device is selected, an error is reported via a C++ exception.
`explicit queue (const device_selector &selector, function_class &async_handler)`	Creates a queue for the device provided by the device_selector. If no device is selected, an error is reported. Reports asynchronous errors using the *async_handler*.
`explicit queue (const device &queue_device)`	A queue is created for queue_device. Any error is reported via C++ exceptions.
`queue (const device &queue_device, function_class &async_handler)`	A queue is created for queue_device. Returns asynchronous errors via the *async_handler* callback function.
`queue (context & dev_context, device_selector &selector)`	This constructor chooses a device based on the provided device_selector in the given context. If no device is selected, an error is reported via C++ exceptions.

Table 3.11: Constructors for the `queue` class. (Part I)

Constructors (Part II)	Description
`queue (~~context &~~ dev_context, device_selector &selector function_class &async_handler)`	This constructor chooses a device based on the provided device_selector, which needs to be in the given context. If no device is selected, an error is reported. Returns asynchronous errors via the *async_handler* callback function.
`queue (const context &dev_context, const device &dev_device)`	Creates a command queue using `clCreateCommandQueue` from a context and a device. Returns errors via C++ exceptions.
`queue (const context &dev_context, const device &dev_device function_class &async_handler)`	Creates a command queue using `clCreateCommandQueue` from a context and a device. Returns errors via the *async_handler* callback function.
`queue (const context &dev_context, const device &dev_device, cl_command_queue_properties *properties = nullptr)`	Creates a command queue using `clCreateCommandQueue` from a context and a device ~~give~~ the cl_command_queue_properties. Returns errors via C++ exceptions.
`queue (const context &dev_context, const device &dev_device, cl_command_queue_properties *properties = nullptr, function_class &async_handler)`	Creates a command queue using `clCreateCommandQueue` from a context and a device. Returns asynchronous errors via the *async_handler* callback function.
`queue (cl_command_queue cl_queue)`	This constructor creates a SYCL queue from an OpenCL queue. At construction it does a retain on the queue memory object. Returns errors via C++ exceptions.
`queue (cl_command_queue opencl_queue, function_class &async_handler)`	This constructor creates a SYCL queue from an OpenCL queue. At construction it does a retain on the queue memory object. Reports errors via the *async_handler* callback function.
`queue (queue &dev_queue)`	Copy constructor

Table 3.12: Constructors for the `queue` class. (Part II)

Methods	Description
`cl_command_queue get()`	Returns the underlying OpenCL command queue after doing a retain. This memory object is expected to be released by the developer.
`context get_context ()`	Returns the SYCL context the queue is using. Reports errors using C++ exceptions.
`device get_device ()`	Returns the SYCL device the queue is associated with. Reports errors using C++ exceptions.
`void wait ()`	Performs a blocking wait for the completion all enqueued tasks in the queue. Synchronous errors will be reported through C++ exceptions.
`void wait_and_throw ()`	Performs a blocking wait for the ~~completion all~~ enqueued tasks in the queue. Synchronous errors will be reported via C++ exceptions. Asynchronous errors will be passed to the `async_handler` passed to the queue on construction. If no `async_handler` was provided then asynchronous exceptions will be lost.
`void throw_asynchronous ()`	Checks to see if any asynchronous errors have been produced by the queue and if so reports them by passing them to the `async_handler` passed to the queue on construction. If no `async_handler` was provided then asynchronous exceptions will be lost.

`template<cl_int name>`
`typename param_traits`
`<cl_command_queue_info,name>::param_type`
`get_info ()const`
	Queries the platform for *cl_command_queue_info*

Table 3.13: Methods for class queue

3.5.5.1 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 returned through exceptions.

- **Asynchronous errors** are those that are produced through callback functions only. These will be stored within the queue's context until they are dispatched to the context's asynchronous error handler. If a queue is constructed with a user-supplied context, then it is this context's asynchronous error handler to which asynchronous errors are reported. If a queue is constructed without a user-supplied context, then the queue's constructor can be supplied with a queue-specific asynchronous error handler which will be used to construct the queue's context. To ensure that such errors are processed predictably in a known host thread these errors are only passed to the asynchronous error handler on request when either `wait_and_throw` is called or when `throw_asynchronous` is called. If no asynchronous error handler is passed to the queue or its context on construction, then such errors go unnoticed, much as they would if no callback were passed to an OpenCL context.

3.5.6 Command group class

A *command group* in SYCL as it is defined in Section 2.3.1 consists of a kernel and all the commands for queued data transfers in order for the kernel's execution to be successful. In SYCL, as it is in OpenCL, a kernel needs to be enqueued and there are a lot of commands that need to be enqueued as well for making the data available to the kernel. The commands that enqueue a kernel and all the relevant data transfers to be enqueued for it, form the `command_group`. 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` class serves as interface for the encapsulation of *command groups*. A kernel is defined in a command group either as a functor object or as a lambda function. All the device data accesses are defined inside this group and any transfers are managed by the system. The rules for the data transfers regarding device and host data accesses are better described in the data management section (3.6), where data storage (3.6.3), buffers (3.6.1) and accessor (3.6.4) classes are described.

It is possible to obtain events for the start of the command group, 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 due to the fact that the command group does not rigidly specify which memory data are stored on kernel completion.

It is possible for a `command_group` to fail to enqueue to a queue, or for it to fail to execute correctly. A user can therefore supply a secondary queue when creating a command group. 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 undefined in the specification.

The constructors and methods of the `command_group` class are listed in Tables 3.14 and 3.15.

The `command_group` is templated with a functor type representing the body of the command group in which *accessor* objects are declared and kernels enqueued.

```
namespace cl {
  namespace sycl {
    class command_group {
    public:
      template <typename functorT>
      command_group(queue &primaryQueue,
                    const functorT &lambda);

      template <typename functorT>
      command_group(queue &primaryQueue,
                    queue &secondaryQueue,
                    const functorT &lambda);

      ~command_group();

      event start_event();

      event kernel_event();

      event complete_event();
```

```
    };
  }  // namespace sycl
}  // namespace cl
```

Constructors	Description

`template<typename FunctorT>`
`command_group(queue &primaryQueue,`
`const FunctorT &lambda)`
	Construct a *command_group* with the queue the group will enqueue its commands to and a lambda function or function object containing the body of commands to enqueue.

`template<typename FunctorT>`
`command_group(queue &primaryQueue,`
`queue &secondaryQueue,`
`const FunctorT &lambda)`
	Construct a *command_group* with a primaryQueue that is the primary queue to be used in order to enqueue its commands to and a lambda function or function object containing the body of commands to enqueue. If the command_group execution fails in the *primaryQueue* the SYCL runtime will try to re-schedule the whole command_group to the *secondaryQueue*.

Table 3.14: Constructors for the `command_group` class

Methods	Description
`event start_event()`	Return the event object that the command group waits on to begin execution.
`event kernel_event()`	Return the event representing completion of the *command_group*'s kernel.
`event complete_event()`	Return the event representing completion of the entire command group including any required data movement commands.

Table 3.15: Methods for the `command_group` class

3.5.7 Event class

An *event* in SYCL abstracts the `cl_event` objects in OpenCL. In OpenCL events' mechanism is comprised of low-level event objects that require from the developer to use them in order to synchronize memory transfers, enqueuing kernels and signaling barriers.

In SYCL, events are an abstraction of the OpenCL event objects, but they retain the features and functionality of the OpenCL event mechanism. They accommodate synchronization between different contexts, devices and platforms. It is the responsibility of the SYCL implementation to ensure that when SYCL events are used in OpenCL queues, the correct synchronization points are created to allow cross-platform or cross-device synchronization.

An SYCL event can be constructed from an OpenCL event or can return an OpenCL event. The constructors and methods of the Event class are listed in Tables 3.16 and 3.17.

```
namespace cl {
  namespace sycl {
    class event {
    public:
      event();
      ~event();
      cl_event get();
      vector_class<event> get_wait_list();
      void wait();
      static void wait(const vector_class<event> &event_list);
      void wait_and_throw();
      static void wait_and_throw(const vector_class<event> &event_list);
    };
  }  // namespace sycl
}  // namespace cl
```

Constructors	Description
`event (event)`	Construct a copy sharing the same underlying event. The underlying event will be reference counted.

Table 3.16: Constructors for the `event` class

Methods	Description
`cl_event get()`	Return the underlying OpenCL event reference.
`vector_class<event> get_wait_list()`	Return the list of events that this event waits for in the dependence graph.
`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. If any uncaught asynchronous errors occurred on the context (or contexts) that the event is waiting on executions from, then will also call that context's asynchronous error handler with those errors.
`static void wait(` `  const vector_class<event> &event_list)`	Synchronously wait on a list of events.
`static void wait_and_throw(` `  const vector_class<event> &event_list)`	Synchronously wait on a list of events. If any uncaught asynchronous errors occurred on the context (or contexts) that the events are waiting on executions from, then will also call those contexts' asynchronous error handlers with those errors.

Table 3.17: Methods for the `event` class

3.6 Data access and storage in SYCL

In SYCL, data storage and access are handled by separate classes. *Buffers*, *images* and *storage* objects handle storage and ownership of the data, whereas *accessors* handle access to the data. Buffers and images in SYCL are different to OpenCL buffers and images in that they can be bound to more than one device or context and they get destroyed when they go out-of-scope. Storage classes handle ownership of the data in the buffers and images, while allowing exception handling for blocking and non-blocking data transfers. Accessors manage data transfers between host and all the devices in the system, as well as tracking data dependencies.

3.6.1 Buffers

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

Buffer constructors are listed in Table 3.18 and methods in Table 3.19.

```
namespace cl{
  namespace sycl {
    template <typename T,
              int dimensions = 1>
    class buffer {
    public:
      buffer(const range<dimensions> &r);

      buffer(T * host_data, range<dimensions> r);

      buffer(const T * host_data, range<dimensions> r);

      buffer(storage<T> &store, range<dimensions> r);

      buffer(buffer<T, dimensions> &b);

      buffer(buffer<T, dimensions> b,
             id<dimensions> base_index,
             range<dimensions> sub_range) ;

      template <class InputIterator>
      buffer<T, 1> (InputIterator first, InputIterator last);

      buffer(cl_mem mem_object,
             queue &from_queue, event available_event);

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

      size_t get_size();
      size_t get_count();
```

```
    };
  }  // namespace sycl
}  // namespace cl
```

Constructors	Description
`buffer<T, dimensions> (range<dimensions>)`	Create a new buffer of the given size with storage managed by SYCL
`buffer<T, dimensions> (` `  const T* host_data, range<dimensions>)`	Create a new buffer with associated host memory. `host_data` points to the storage and values used by the buffer and `range<dimensions>` defines the size. Since the host memory is marked as `const`, the buffer is also read-only and only read-only accessors can be used
`buffer<T, dimensions> (` `  T* host_data, range<dimensions>)`	Create a new buffer with associated host memory. `host_data` points to the storage and values used by the buffer and `range<dimensions>` defines the size
`buffer<T, dimensions> (` `  storage<T> &store, range<dimensions>)`	Create a new buffer from a `storage<T>` abstraction provided by the user and with the required size. The storage object has to exist during all the life of the buffer object.
`template <class InputIterator> buffer<T, 1> (` `  InputIterator first, InputIterator last)`	Create a new allocated 1D buffer initialized from the given elements ranging from `first` up to one before `last`
`buffer<T, dimensions>(buffer<T, dimensions> &b)`	Create a new buffer copy that shares the data with the origin buffer. The system use reference counting to deal with data lifetime

`buffer<T, dimensions>(`
`  buffer<T, dimensions> &b,`
`  index<dimensions> base_index,`
`  range<dimensions> sub_range)`
	Create a new sub-buffer without allocation to have separate accessors later. `b` is the buffer with the real data. `base_index` specifies the origin of the sub-buffer inside the buffer `b`. `sub_range` specifies the size of the sub-buffer.

`buffer<T, dimensions>(`
`  cl_mem mem_object,`
`  queue from_queue,`
`  event available_event)`
	Create a buffer from an existing OpenCL memory object associated to a context after waiting for an event signaling the availability of the OpenCL data. `mem_object` is the OpenCL memory object to use. `from_queue` is the queue associated to the memory object. `available_event` specifies the event to wait for if non null

Table 3.18: Constructors for the `buffer` class

There are 6 ways of constructing a `buffer`. These 6 ways control the underlying storage for the buffer and what happens when the buffer object is destroyed.

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

Methods	Description
`range<dimensions> get_range()`	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()`	Returns the total number of elements in the buffer. Equal to get_range()[0] * ... * get_range()[dimensions-1].
`size_t get_size()`	Returns the size of the buffer storage in bytes. Equal to get_count()*sizeof(T).
`template< access::mode mode, access::target target=access::global_buffer> accessor<T, dimensions, mode, target> get_access()`	Returns a valid accessor to the buffer with the specified access mode and target. The value of target can be `access::global_buffer`, `access::constant_buffer` or `access::host_buffer`.

Table 3.19: Methods for the `buffer` class.

destroyed and the buffer destruction behaviour defined below is followed.

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

1. A buffer can be constructed with just a size and no associated storage. The storage for this type of buffer is entirely handled by the SYCL system. The destructor for this type of buffer never blocks, 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 undefined.

2. A buffer can be constructed with associated host memory. The buffer will use this host memory for its full lifetime, but the contents of this host memory are undefined 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.

3. If the pointer to host memory 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).

4. A buffer can be constructed with an associated `storage` object (see 3.6.3). The `storage` object must not be destroyed by the user until after the buffer has been destroyed. The synchronization and copying behaviour of the storage is determined by the storage object. The behaviour on destruction of a buffer attached to a storage object is defined by the user-defined storage object.

5. 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 will not block.

6. A buffer constructed from a `cl_mem` object creates a SYCL buffer that is initialized from a `cl_mem` object and may use the `cl_mem` object for the lifetime of the buffer. The destructor for this type of buffer will

block until all operations on the buffer have completed and then will (if necessary) copy all modified data back into the associated `cl_mem` object.

As a convenience for the user, any constructor that takes a range argument can instead be passed range values as 1, 2 or 3 arguments of type `size_t`.

A buffer object can also be copied, which just copies a reference to the buffer. The buffer objects use reference counting, so copying a buffer object increments a reference count on the underlying buffer. If after destruction, the reference count for the buffer is non-zero, then no further action is taken.

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, but which only have access to the range specified at time of construction of the sub-buffer.

If a buffer object is constructed from a `cl_mem` object, then the buffer is created and initialized from the OpenCL memory object. The SYCL system may copy the data to another device and/or context, but must copy it back (if modified) at the point of destruction of the buffer. The user must provide a `queue` and `event`. The memory object is assumed to only be available to the SYCL scheduler after the event has signaled and is assumed to be currently resident on the context and device signified by the `queue`.

3.6.2 Images

The class image<int dimensions> (Table 3.20) defines shared image data of one, two or three dimensions, that can be used by kernels in queues and has to be accessed using accessor classes with image accessor modes.

Image constructors are listed in Table 3.20 and methods in Table 3.21. Where relevant, it is the responsibility of the user to ensure that the format of the data contained within storage matches the format described by order and type.

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 undefined 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 behaviour occurs on image object destruction.

```
namespace cl {
  namespace sycl {
    template<int dimensions>
    class image {
    public:
      image(void *hostPointer, cl_channel_order order, cl_channel_type type,
            range<dimensions> size);
      image(void *hostPointer, cl_channel_order order, cl_channel_type type,
            range<dimensions> size, range<dimensions-1> pitch);
      image(storage &store, cl_channel_order order, cl_channel_type type,
            range<dimensions> size);
      image(storage &store, cl_channel_order order, cl_channel_type type,
            range<dimensions> size, range<dimensions-1> pitch);
      image(image<dimensions> &);
      image(cl_mem mem_object, queue &from_queue, event available_event);
      ~image();

      range<dimensions> get_range();
    range<dimensions-1> get_pitch();

      size_t get_size();
      size_t get_count();

      template <access::mode accessMode,
        access::target accessTarget = access::global_buffer>
        accessor<T, dimensions, accessMode, accessTarget>
        get_access();
    };
  }  // namespace sycl
}  // namespace cl
```

Constructors	Description

`image<dimensions>(`
`  void *hostPointer,`
`  cl_channel_order order,`
`  cl_channel_type type,`
`  range<dimensions> size)`
	Construct an image. The associated host memory is in `hostPointer`. The type of the image data is defined by `order` and `type`. The size of the image in pixels is defined by `size`.

`image<dimensions>(`
`  void *hostPointer,`
`  cl_channel_order order,`
`  cl_channel_type type,`
`  range<dimensions> size,`
`  range<dimensions-1> pitch)`
	Construct an image. The associated host memory is in `hostPointer`. The type of the image data is defined by `order` and `type`. The size of the image in pixels is defined by `size`. The pitch of the image data, in bytes, is defined by `pitch`.

`image<dimensions>(`
`  storage &store,`
`  cl_channel_order order,`
`  cl_channel_type type,`
`  range<dimensions> size)`
	Construct an image. The associated host memory and synchronization behaviour is defined by `store`. The type of the image data is defined by `order` and `type`. The size of the image in pixels is defined by `size`.

`image<dimensions>(`
`  storage &store,`
`  cl_channel_order order,`
`  cl_channel_type type,`
`  range<dimensions> size,`
`  range<dimensions-1> pitch)`
	Construct an image. The associated host memory and synchronization behaviour is defined by `store`. The type of the image data is defined by `order` and `type`. The size of the image in pixels is defined by `size`. The pitch of the image data, in bytes, is defined by `pitch`.

`image<dimensions>(image<dimensions> &)`	Copy construct an image as a reference to another image. The runtime reference counts such that all copies share the same underlying storage and the last one destroyed free any associated memory.

Table 3.20: Constructors for the `image` class.

Methods	Description
`range<dimensions> get_range()`	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()`	Return a range object representing the pitch of the image in bytes.
`size_t get_count()`	Returns the total number of elements in the image. Equal to get_range()[0] * ... * get_range()[dimensions-1].
`size_t get_size()`	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<`
`  access::mode mode,`
`  access::target target=access::image>`
`  accessor<T, dimensions, mode, target>`
`  get_access()`
	Returns a valid accessor to the image with the specified access mode and target. The value of target can be `access::image` or `access::host_image`.

Table 3.21: Methods for the `image` class.

3.6.3 Storage classes

The `storage` class is an abstract superclass that allows users to define ownership of shared data. The `storage` class may be implemented by users to create custom data management classes that change the behaviour of the default data management. As it is abstract, the storage class has no public constructors, however methods are defined in Table 3.22.

Users can create buffers and images with their own storage objects. One storage object can only be attached to one memory object. The SYCL system calls the storage object asynchronously to notify it of events relevant to the ownership and use of the data. The `get_size` method is called by the SYCL system in order to get the number of elements of type `T` of the underlying data.

The SYCL system is responsible for copying data into and out of OpenCL buffers as required. The user implements methods in their storage class to tell the SYCL system where data should be copied (or mapped) to and from. If the user returns nullptr from one of these methods, then the SYCL system will assume that no data copying is required.

The user is responsible for ensuring that their `storage` class implementation is thread-safe.

A storage class needs to ensure that it maintains whether it is in use or not. Its `in_use` method will be called when the runtime starts using the data in the storage object, while its `completed` method will be called when the runtime is no longer using data in the storage object. After `completed` is called, there may be further `in_use` calls when further command groups are enqueued to operate on the data.

The destruction of the storage object may happen while the data in the storage object is still in use. If the storage object's `destroy` method returns while the data is no longer in use, then the SYCL runtime will call the `get_final_data` and, if the result is non-null, copy back any contents of any buffer to the targeted data, if required. Therefore, if the `destroy` method waits for the `completed` method to be called, the storage object can ensure that data is copied back. If the `destroy` method returns while the storage object's data is still in use, then the SYCL runtime will not copy back data until the `completed` method is called, at which point it will then call the `get_final_data` method to determine if copy-back is required.

```
namespace cl {
  namespace sycl {
    template <typename T>
    struct storage {
      virtual size_t get_size() = 0;
      virtual T* get_host_data() = 0;
      virtual const T* get_initial_data() = 0;
      virtual T* get_final_data() = 0;
      virtual void destroy() = 0;
      virtual void in_use() = 0;
      virtual void completed() = 0;
    };

  }  // namespace sycl
}  // namespace cl
```

Methods	Description
`virtual size_t get_size()`	Returns the size of the storage object's memory in terms of number of elements.
`virtual T* get_host_data()`	Returns 0 if no host memory is attached. Otherwise returns a pointer to the host memory backing the storage object. The attached host memory is assumed to be available to the SYCL runtime for the lifetime of the storage object. The SYCL runtime can use this host memory directly during execution of kernels (for host execution of kernels, for example) or it can map OpenCL buffers into this host memory, or it can use this host memory as temporary storage. If the storage object returns 0, then the SYCL runtime must ensure that there is memory available somewhere for the data to be stored while kernels are executing. This runtime-allocated data might be only on device and not on host.
`virtual const T* get_initial_data()`	Called at the point of construction to request the initial contents of the buffer. If non-null the buffer will be initialized to the contents of the attached memory. If this method returns a pointer to the same data as `get_host_data` then the SYCL runtime may not have to perform any copying at all.
`virtual T* get_final_data()`	Called at the point of construction to request the location to where data should be finally stored to. If non-null then the contents of the buffer will be written to that location. If the pointer is the same as `get_initial_data` or `get_host_data` then the system shall determine whether copying is necessary. The SYCL runtime calls `get_final_data` after both the `destroy` method and the `completed` method have been called. The order in which `destroy` and `completed` have been called is dependent on the execution order of operations on the data.
`virtual void destroy()`	Called when the last copy of the associated memory object is destroyed. This method will only be called once. Exceptions thrown by the destroy method will be caught and ignored.
`virtual void in_use()`	Called when the first command group that requires the data is added to a queue. If the execution of the queued operations on that data complete, then `completed` is called. After a call to `completed`, any further operations on the same data will cause `in_use` to be called again. The `in_use` method will not be called twice without a call of `completed` in between.
`virtual void completed()`	Called when the final enqueued command has completed. After `completed` is called there may be further calls of `in_use` if new work is enqueued that operates on this data. The `completed` method will be called once for each call of the `in_use` method.

Table 3.22: Methods for the `storage` class.

3.6.3.1 async_storage

The SYCL platform provides `async_storage` as a built-in data storage manager that enables the developer to allocate memory elsewhere and decouple destruction of a buffer from waiting on completion of operations on that buffer. This is an implementation of `storage` which provides easy asynchronous memory management. The advantage of this approach is that it makes it easy for software to quickly respond to errors or events, while leaving the system to continue asynchronous execution on data that will be freed from memory once execution has completed.

As its name suggests, the `async_storage` class allows a buffer to be created and destroyed while its storage lifetime is decoupled from the lifetime of the buffer. By default, this storage object will block on destruction of the storage object until any kernels operating on the associated data have completed. However, after the `release` method has been called on the storage object, on destruction of the storage object (but not of the associated buffer, which never blocks) the `async_storage` class will not block, but instead free the associated memory once all operations on the data have completed.

Interface methods as for the abstract superclass, `storage`, are listed in Table 3.22. Constructors for the concrete `async_storage` are listed in Table 3.23 and its user-callable methods in Table 3.24.

```
namespace cl{
  namespace sycl{
    template<typename T>
    class async_storage :
        public storage<T>
    {
    public:
      // Construct a storage class with mapped data
      async_storage(T *mapped_data, size_t size);
      // Construct a storage class which will manage its own memory
      async_storage(size_t size);

      // After called, the storage object will no longer block on destroy
      void release();

      // Return a pointer to the mapped data
      T *data();
    }
  }
}
```

Constructors	Description
`async_storage(T *mapped_data, size_t size)`	Constructs an async storage object from mapped host data. Any buffer constructed with this storage object will use the mapped host memory for the lifetime of the storage object. If the storage object is destroyed, then it will block for any command groups processing the data to release, unless or until the `release` method is called, in which case the storage object's destructor will not block. When the host memory is no longer in use, the storage object will call `delete[]` on the mapped data.
`async_storage(size_t size)`	Constructs an async storage object with no mapped host memory. This storage object, or any buffer constructed with this storage object will never block on destruction. The storage object will allocate and free host memory using `new[]` and `delete[]`.

Table 3.23: Constructors for the `async_storage` class.

Methods	Description
`void release()`	After the `release` method is called on the storage object, any buffer constructed from this storage object will not block on destruction.
`T *data()`	Returns the address of the mapped host data for this storage object, whether passed in by the user on construction, or allocated by the storage object itself. Returns 0 if there is no mapped host memory currently attached.

Table 3.24: Methods for the `async_storage` class.

3.6.4 Accessors

Accessors manage the access to data in buffers and images. The user specifies the type of access to the data and the SYCL implementation ensures that the data is accessible in the right way on the right device in a queue. This separation allows an SYCL implementation to choose an efficient way to provide access to the data within an execution schedule. Common ways of allowing data access to shared data in a heterogeneous system include copying between different memory systems, mapping memory into different device address spaces, or direct sharing of data in memory.

Accessors are *device accessors* by default, but can optionally be specified as being host accessors. Device accessors can only be constructed within command groups and provide access to the underlying data in a queue. Only a kernel can access data using a device accessor. Constructing a device accessor is a non-blocking operation: the synchronization is added to the queue, not the host.

Host accessors can be created outside command groups and give immediate access to data on the host. Construction of host accessors is blocking, waiting for all previous operations on the underlying buffer or image to complete, including copying from device memory to host memory. Any subsequent device accessors need to block until the processing of the host accessor is done and the data are copied to the device.

Accessors always have an *element data type*. When accessing a buffer, the accessor's element data type must match the same data type as the buffer. An image accessor may have an element data type of either an integer vector or a floating-point vector. The image accessor data type provides the number and type of components of the pixel read. The actual format of the underlying image data is not encoded in the accessor, but in the image object itself.

When integrating OpenCL C code with SYCL code, it may be necessary to make use of OpenCL `cl_event` and `cl_mem` objects. To obtain OpenCL objects for an SYCL buffer, it is necessary to query the corresponding OpenCL memory object inside a *command group* and create an accessor with the corresponding access mode of the query.

There are two enumeration types inside `namespace cl::sycl::access`, `access::mode` and `access::target`. These two enumerations define both the access mode and the data that the accessor is targeting.

3.6.4.1 Access modes

The `mode` enumeration, shown in Table 3.25, has a base value, which must be provided.

The user must provide the *access mode* when defining an accessor. This information is used by the scheduler to ensure that any data dependencies are resolved by enqueuing any data transfers before or after the execution of a kernel. If a command group contains only *write mode* accesses to a buffer, then the previous contents of the buffer (or sub-range of the buffer, if provided) are not preserved. 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. A command-group's access to a specific buffer is the union of all access modes to that buffer in the command group, regardless of construction order.

3.6.4.2 Access targets

The `target` enumeration, shown in Table 3.26, describes the type of object to be accessed via the accessor. The different values of the `target` enumeration require different constructors for the accessors.

access::mode	Description
`read`	read-only access
`write`	write-only access. Previous contents not discarded.
`read_write`	read and write access
`discard_write`	write-only access. Previous contents discarded.
`discard_read_write`	read and write access. Previous contents discarded.

Table 3.25: Enumeration of access modes available to accessors.

access::target	Description
`global_buffer`	Access `buffer` via global memory.
`constant_buffer`	Access `buffer` via constant memory.
`local`	Access work-group-local memory.
`image`	Access an `image`.
`host_buffer`	Access a `buffer` immediately in host code.
`host_image`	Access an `image` immediately in host code.
`image_array`	Access an array of `image`s on a device.
`cl_buffer`	Access an OpenCL `cl_mem` buffer for use directly with the OpenCL C API.
`cl_image`	Access an OpenCL `cl_mem` image for use directly with the OpenCL C API.

Table 3.26: Enumeration of access modes available to accessors.

3.6.4.3 Accessor class

The `accessor` makes a data available to host code, or to a specific kernel. The `accessor` is parameterized with the type and number of dimensions of the data. An accessor also has a `mode`, which defines the operations possible on the underlying data (see Table 3.25) and a `target` (see Table 3.26, which defines the type of data object to be modified. The constructors and methods available on an accessor depend on the `mode` and `target`.

The generic methods for the `accessor` class are defined in Table 3.28. Available methods are limited by the access mode and target provided as template parameters to the accessor object.

```
namespace cl {
  namespace sycl {
    namespace access {
      enum class mode {
        read = 1,
        write,
        read_write,
        discard_write,
        discard_read_write
      };

      enum class target {
        global_buffer = 2014,
        constant_buffer,
        local,
        image,
        host_buffer,
        host_image,
        image_array,
        cl_buffer,
        cl_image
      };
    } // namespace access

    template <typename elementType, int dimensions,
              access::mode accessMode, access::target accessTarget>
    class accessor{
      public:
      // Available only for: access::global_buffer, access::host_buffer,
      // access::constant_buffer, and access::cl_buffer
      explicit accessor(buffer<elementType, dimensions> &bufferRef);

      // Available only for: access::image, access::host_image,
      // and access::cl_image
      explicit accessor(image<dimensions> &imageRef);

      // Available only for: access::local
      accessor(range<dimensions> allocationSize);

      size_t get_size();
      cl_mem get_cl_mem_object();
      cl_event get_cl_event_object();

      // Methods available to buffer targets
```

```
      // Available when access_mode includes write permissions
      elementType &operator[](id<dimensions>);
      // Available when access_mode is read-only
      const elementType &operator[](id<dimensions>);

      // Methods available for image targets
      __sampler_ref<elementType> &operator()(sampler sample);
      __image_ref<elementType> &operator[](id<dimensions>);

      // Available when the accessor is to an image array
      // Returns an accessor to a particular slice
      accessor<elementType, 2, mode, image> operator[](
          size_t index)
    };

  }  // namespace sycl
}  // namespace cl
```

Constructors	Description
`accessor(buffer<elementType, dimensions> &bufferRef)`	Construct a buffer accessor from a buffer. Constructor only available for access modes `global_buffer`, `host_buffer`, `constant_buffer`, or `cl_buffer` see Table 3.25. `access_target` defines the form of access being obtained. See Table 3.26.

`accessor(`
`  buffer<elementType, dimensions> &bufferRef,`
`  range<dimensions> offset,`
`  range<dimensions> range)`
	Construct a buffer accessor from a buffer given a specific range for access permissions and an offset that provides the starting point for the access range. This accessor limits the processing of the buffer to the [offset, offset+range] for every dimension. Any other parts of the buffer will be unaffected. Constructor only available for access modes `global_buffer`, `host_buffer` , `constant_buffer`, or `cl_buffer` see Table 3.25. `access_target` defines the form of access being obtained. See Table 3.26. This accessor is recommended for *discard_write* and *discard_read_write* access modes, when the unaffected parts of the processing should be retained.

`accessor(image<dimensions> &imageRef)`	Construct an image accessor from an image. Constructor only available if `accessMode` is `image`, `host_image` or `cl_image`, see Table 3.25. `access_target` defines the form of access being obtained. See Table 3.26. The `elementType` for image accessors must be defined by the user and is the type returned by any sampler or accessor read operation, as well as the value accepted by any write operation. It must be an `int`, `unsigned int` or `float` vector of 4 dimensions.
`accessor(range<dimensions> allocationSize)`	Construct an accessor of `dimensions` dimensions with elements of type `elementType` using the passed range to specify the size in each dimension. Constructor only available if `accessMode` is `local`, see Table 3.25.

Table 3.27: Accessor constructors.

Methods	Description
`size_t get_size()`	Returns the size of the underlying buffer in number of elements.
`cl_mem get_cl_mem_object()`	Returns the `cl_mem` object corresponding to the access and allowing for interoperation with the OpenCL API. Only available when `target` is `cl_image` or `cl_buffer`.
`cl_event get_cl_event_object()`	Returns the `cl_event` object corresponding to the last command to access the memory object and allowing for interoperation with the OpenCL API. Only available when `target` is `cl_image` or `cl_buffer`.
`dataType &operator[](id<dimensions>)`	Return a writeable reference to an element in the buffer. Available when `mode` includes write permissions
`const dataType &operator[](id<dimensions>)`	Return the value of an element in the buffer. Available when `mode` is read-only
`accessor<dataType, 2, mode, image> operator[](size_t index)`	Returns an accessor to a particular plane of an image array. Available when accessor acts on an image array.

Table 3.28: Methods for the `accessor` class.

3.6.4.4 Buffer accessors

Accessors to buffers are constructed from a buffer with the same element data type and dimensionality as the accessor. A buffer accessor uses *global* memory by default, but can optionally be set to use *constant* memory. Accessors that use constant memory are restricted by the underlying OpenCL restrictions on device constant memory, i.e. there is a maximum total constant memory usable by a kernel and that maximum is specified by the OpenCL device. Only certain methods and constructors are available for buffer accessors.

The array operator `[id<dimensions>]` provides access to the elements of the buffer. The user can provide an index as an `id` parameter of the same dimensionality of the buffer, or just like C++ arrays, can provide one array operator per dimension, with individual indices of type `size_t` (e.g. `myAccessor[i][j][k]`).

The address space for the index operator matches that of the accessor target. For an `access::global_buffer`, the address space is `global`. For an `access::constant_buffer`, the address space is `global`.

Accessors to buffers can be constructed to only access a *sub-range* of the buffer. The sub-range restricts access to just that range of the buffer, which the scheduler can use as extra information to extract more parallelism from queues as well as restrict the amount of information copied between devices.

3.6.4.5 Image accessors

Accessors that target images must be constructed from images of the same dimensionality as the accessor. The `target` parameter must be either `image`, `host_image` or `cl_image`. The `dataType` parameter must be a 4 dimension vector of `unsigned int`, `int` or `float`. The array operator `[id<dimensions>]` provides samplerless reading and writing of the image. The user can provide an index as an `id` parameter of the same dimensionality of the image, or just like C++ arrays, can provide one array operator per dimension, with individual indices of type `size_t` (e.g. `myAccessor[i][j][k]`). The bracket operator takes a `sampler` (see 3.6.6) parameter, which then allows floating-point sampler-based reading using the array operator (e.g. `myAccessor(mySampler)[my2dFloatVector]`).

To enable the reading and writing of pixels with and without samplers, using standard C++ operators, there are two internal classes: `__image_ref` and `_ _sampler`. These classes only exist to ensure that assignment to pixels uses image write functions and reading the value of pixels uses image read functions.

3.6.4.6 Local accessors

Accessors can also be created for *local* memory, to enable pre-allocation of local buffers used inside a kernel. These accessors are constructed using `cl::sycl::range`, which defines the size of the memory to be allocated on a per work-group basis and must be constructed with an access target of `local`. Local memory is only shared across a work-group. A local accessor can provide a `local_ptr` to the underlying data within a kernel and is only usable within a kernel. The host has no access to the data of the local buffer and cannot read or write to the data, so the accessor cannot read or write data back to the host. There can be no associated host pointer for a local buffer or data transfers.

Local accessors are not valid for single-task or basic `parallel_for` invocations.

3.6.4.7 Host accessors

Host accessors have a target of `access::host_buffer` or `access::host_image`. Unlike other accessors, host accessors should be constructed outside of any `command_group`. The constructor will block until the data is ready for the host to access, while the destructor will block any further operations on the data in any SYCL queue. There are no special constructor or method signatures for host accessors, so there are is no table for special host accessors here (see buffer and image accessors above).

Host accessors are constructed outside command groups and not associated with any queue, so any error reporting is synchronous. By default, error reporting is via exceptions.

3.6.4.8 OpenCL interoperability accessors

SYCL provides two `access::target` values for interoperability with OpenCL C software. One is constructed from an SYCL buffer and the other from an SYCL image. Both are constructed from an SYCL queue. The interoperability accessors provide the same lifetime acquire-release semantics as the normal SYCL accessors. For the lifetime of the interoperability accessor, the SYCL system assumes that the user is enqueueing OpenCL commands to modify the OpenCL memory objects. Like device accessors, the synchronization semantics are to enqueue work, so none of the synchronization operations are blocking: instead, they operate via synchronization events added to queues. The user is responsible for ensuring that their own OpenCL C code uses the correct event-based synchronization.

The interoperability accessors cannot be used inside SYCL kernels or constructed inside command groups.

The queue provided at construction time defines the device and context that the SYCL system will ensure the contents of the buffer are available on. The `get` method returns the `cl_mem` object that the user can modify using OpenCL. The user must also ensure that any operations on the `cl_mem` object are synchronized to occur after the event returned by `get_event` has signaled.

The user must provide an event to signal to the SYCL system when the OpenCL code has finished operating on the `cl_mem` object. This event is set using the `release_event` method.

3.6.4.9 Accessor capabilities and restrictions

Accessors provide access on the device or on the host to a buffer or image. The access modes allowed depend on the accessor type and target. A device accessor grants access to a kernel inside a `command_group`, and depending on the access target, there are different accesses allowed. A host accessor grants access to the host program to the access target. Tables 3.29, 3.30 and 3.31 show all the permitted access modes depending on target.

Accessor Type	Access Target	Access mode	Data Type	Description

Device	`global_buffer`	`read`
		`write`
		`read_write`
		
		`discard_write`
		`discard_read_write`
			All available data types supported in SYCL.	Access a buffer allocated in global memory on the device.

Device	`constant_buffer`	`read`	All available data types supported in SYCL.	Access a buffer allocated in constant memory on the device.

Host	`host_buffer`	`read`
		`write`
		`read_write`
		`discard_write`
		`discard_read_write`
			All available data types supported in SYCL.	Access a host allocated buffer on host.

Device or host	`cl_buffer`	`read`
		`write`
		`read_write`
		
		`discard_write`
		`discard_read_write`
			All available data types supported in SYCL.	Allow access to a buffer from OpenCL code using an OpenCL `cl_mem` buffer object which resides on device.
				This accessor cannot be used inside SYCL kernels.

Device	`local`	`read`
		`write`
		`read_write`
			All supported data types in local memory	Access work-group local buffer, which is not associated with a host buffer.
				This is only accessible on device.

Table 3.29: Description of all the `accessor` types and modes with their valid combinations for buffers and local memory

Accessor Type	Access Target	Access mode	Data Type	Description
Device	image	read write read_write discard_- writediscard_read_- write	uint4, int4, float4, half4	Access an image on device.
Host	host_image	read write read_write discard_- writediscard_read_- write	uint4, int4, float4, half4	Access an image on the host.
Device	image_array	read write read_write discard_- writediscard_read_- write	uint4, int4, float4, half4	Access an array of images on device.
Device	cl_image	read write read_write discard_- writediscard_read_- write	uint4, int4, float4, half4	Allow access to an SYCL image object using an OpenCL cl_mem image object from OpenCL C code. Cannot be used inside SYCL kernels.

Table 3.30: Description of all the accessor types and modes with their valid combinations for images

Rules for casting apply to the accessors, as there is only a specific set of permitted conversions.

Accessor Types	Original Accessor Target	Original Access Mode	Converted Accessor Target	Converted Access Mode
Device	global_buffer	read_write	global_buffer	read write discard_read_write
Device	local_buffer	read_write	local_buffer	read write
Device	cl_buffer	read_write	cl_buffer	read write discard_read_write
Host	host_buffer	read_write	host_buffer	read write discard_read_write

Table 3.31: Description of the accessor to accessor conversions allowed

3.6.5 Explicit pointer classes

In OpenCL, there are four different address spaces. These are: global, local, constant and private. In OpenCL C, these address spaces are manually specified using OpenCL-specific keywords. In SYCL, the device compiler is expected to auto-deduce the address space for pointers in common situations of pointer usage. However, there are situations where auto-deduction is not possible. Here are the most common situations:

- When linking SYCL kernels with OpenCL C functions. In this case, it is necessary to specify the address space for any pointer parameters when declaring an `extern "C"` function.
- When declaring data structures with pointers inside, it is not possible for the SYCL compiler to deduce at the time of declaration of the data structure what address space pointer values assigned to members of the structure will be. So, in this case, the address spaces will have to be explicitly declared by the developer.
- When a pointer is declared as a variable, but not initialized, then address space deduction is not automatic and so an explicit pointer class should be used, or the pointer should be initialized at declaration.

Explicit pointer classes are just like pointers: they can be converted to and from pointers with compatible address spaces, qualifiers and types. Assignment between explicit pointer types of incompatible address spaces is illegal. In SYCL 1.2, all address spaces are incompatible with all other address spaces. For a future SYCL 2.0, a generic address space will be compatible with all other address spaces. Conversion from an explicit pointer to a C++ pointer preserves the address space.

In `extern ''C''` declarations, an explicit pointer is compatible with an OpenCL C pointer of the same address space.

```
namespace cl {
  namespace sycl {

    template <typename ElementType>
    class global_ptr {
    public:
      global_ptr(ElementType *); // global pointer

      template<enum access::mode Mode>
      global_ptr(accessor<ElementType, 1, Mode, global_buffer>);

      ElementType &operator*();
      ElementType &operator[](size_t i);

      operator ElementType *();
    };

    template <typename ElementType>
    class constant_ptr {
    public:
      constant_ptr(ElementType *); // constant pointer

      template<enum access::mode Mode>
      global_ptr(accessor<ElementType, 1, Mode, constant_buffer>);

      ElementType &operator*();
      ElementType &operator[](size_t i);
```

```
    operator ElementType *();
  };

  template <typename ElementType>
  class local_ptr {
  public:
    local_ptr(ElementType *); //local pointer

    template<enum access::mode Mode>
    global_ptr(accessor<ElementType, 1, Mode, local_buffer>);

    ElementType &operator*();
    ElementType &operator[](size_t i);

    operator ElementType *();
  };

  template <typename ElementType>
  class private_ptr {
  public:
    private_ptr(ElementType *); // private pointer

    ElementType &operator*();
    ElementType &operator[](size_t i);

    operator ElementType *();
  };
 } // namespace sycl
} // namespace cl
```

Explicit Pointer Classes	Original OpenCL Address Space	Compatible Accessor Target
global_ptr	__global	global_buffer
constant_ptr	__constant	constant_buffer
local_ptr	__local	local
private_ptr	__private	none

Table 3.32: Description of explicit pointer classes

3.6.5.1 Multi-pointers

There are situations where a user may want to template a datastructure by an address space. Or, a user may want to write templates that adapt to the address space of a pointer. 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.

```
namespace cl {
  namespace sycl {
    enum address_space {
        global_space,
        local_space,
```

```
      constant_space,
      private_space
  };

  template <typename ElementType, enum address_space Space>
  class multi_ptr {
  public:
    const address_space space;

    ElementType &operator*();
    ElementType &operator[](size_t i);

    operator ElementType *();

    // Only if Space == global_space
    operator global_ptr<ElementType>();
    global_ptr<ElementType> pointer();

    // Only if Space == local_space
    operator local_ptr<ElementType>();
    local_ptr<ElementType> pointer();

    // Only if Space == constant_space
    operator constant_ptr<ElementType>();
    constant_ptr<ElementType> pointer();

    // Only if Space == private_space
    operator private_ptr<ElementType>();
    private_ptr<ElementType> pointer();
  };

  template<typename ElementType, enum address_space Space>
  multi_ptr<ElementType, Space> make_ptr(ElementType *);
 }  // namespace sycl
}  // namespace cl
```

3.6.6 Samplers

Samplers use the `cl::sycl::sampler` type which is equivalent to the OpenCL C `cl_sampler` and `sampler_t` types. Constructors for the `sampler` class are listed in Table 3.33.

```
namespace cl {
  namespace sycl {
    class sampler {
    public:
      enum class sampler_addressing_mode {
        SYCL_SAMPLER_ADDRESS_MIRRORED_REPEAT,
        SYCL_SAMPLER_ADDRESS_REPEAT,
        SYCL_SAMPLER_ADDRESS_CLAMP_TO_EDGE,
        SYCL_SAMPLER_ADDRESS_CLAMP,
        SYCL_SAMPLER_ADDRESS_NONE
      };
      enum class sampler_filter_mode {
        SYCL_SAMPLER_FILTER_NEAREST,
        SYCL_SAMPLER_FILTER_LINEAR
      };


      sampler(
        bool normalized_coords,
        sampler_addressing_mode addressing_mode,
        sampler_filter_mode filter_mode);

      sampler(cl_sampler);

      ~sampler() {}

      cl_addressing_mode get_address() const;

      cl_filter_mode get_filter() const;

      cl_sampler get_opencl_sampler_object() const;
    };
  }  // namespace sycl
}  // namespace cl
```

Constructors	Description

`sampler(`
`  bool normalized_coords,`
`  sampler_addressing_mode addressing_mode,`
`  sampler_filter_mode filter_mode)`
	`normalized_coords` selects whether normalized or unnormalized coordinates are used for accessing image data.
	`addressing_mode` specifies how out-of-range image coordinates are handled.
	`filter_mode` specifies the type of filter that must be applied when reading an image.

`sampler(cl_sampler)`	Construct a sampler from an OpenCL sampler object. Calls `clRetainSampler` on construction and `clReleaseSampler` on destruction. Available in host code only.

Table 3.33: Constructors for the `sampler` class.

3.7 Expressing parallelism through kernels

3.7.1 Ranges and identifiers

The data parallelism of the OpenCL execution model and its exposure through SYCL requires instantiation of a parallel execution over a range of iteration space coordinates. To achieve this we expose types to define the range of execution and to identify a given execution instance's point in the iteration space.

To achieve this we expose five types: `range`, `nd_range`, `id`, `item`, `nd_item` and `group`.

When constructing ids or ranges from integers, the elements are written in row-major format.

3.7.1.1 Range class

`range<int dims>` 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. Constructors for the range class are described in Table 3.34, methods in Table 3.35 and global operators on ranges in Table 3.36.

```
namespace cl {
  namespace sycl {
    template <int dims = 1>
    struct range {
      range(const range<dims> &r);

      // Valid only for when dims==1
      range(size_t x);
      // Valid only for when dims==2
      range(size_t x, size_t y);
      // Valid only for when dims==3
      range(size_t x, size_t y, size_t z);

      size_t get(int index) const;
      size_t &operator[](int index);
      // Valid only for when dims==1
    };

    template <int dims>
    range<dims> operator *(range<dims> a,
                           range<dims> b);

    template <int dims>
    range<dims> operator /(range<dims> dividend,
                           range<dims> divisor);

    template <int dims>
    range<dims> operator +(range<dims> a,
                           range<dims> b);

    template <int dims>
    range<dims> operator -(range<dims> a,
```

```
                        range<dims> b);

  }  // sycl
}  // cl
```

Constructors	Description
`range<dimensions>(size_t,...)`	Construct a range from a list of `dimensions` size_t values representing each dimension. The values are ordered as row-major.
`range<dimensions>(const range<dimensions>&)`	Construct a range by deep copy from another range.

Table 3.34: Constructors for the `range` class.

Methods	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`.

Table 3.35: Methods for the `range` class.

Global operators	Description

`template <int dimensions>`
`  range<dimensions> operator *(`
`  range<dimensions> a,`
`  range<dimensions> b)`
	Multiply each element of `a` by its respective element of `b` and return a range constructed from the resulting values.

`template <int dimensions>`
`  range<dimensions> operator /(`
`  range<dimensions> dividend,`
`  range<dimensions> divisor)`
	Divide each element of `dividend` by its respective element element in `divisor` and return a range constructed of the resulting value.

`template <int dimensions>`
`  range<dimensions> operator +(`
`  range<dimensions> a,`
`  range<dimensions> b)`
	Add each element of `a` to its respective element of `b` and return a range constructed from the resulting values.

`template <int dimensions>`
`  range<dimensions> operator -(`
`  range<dimensions> a,`
`  range<dimensions> b)`
	Subtract each element of `b` from its respective element of `a` and return a range constructed from the resulting values.

Table 3.36: Global operators for the `range` class.

3.7.1.2 nd_range class

```
namespace cl {
  namespace sycl {
    template <int dims = 1>
    struct nd_range {
      static_assert(1 <= dims && dims <= 3,
                    "Dimensions are between 1 and 3");

      nd_range(const nd_range<dimensions> &nd);

      nd_range(range<dims> global_size,
               range<dims> local_size,
               id<dims> offset = id<dims>()) ;

      range<dims> get_global_range() const;
      range<dims> get_local_range() const;
      range<dims> get_group_range() const;
      id<dims> get_offset() const;
    };
  } // namespace sycl
} // namespace cl
```

nd_range<int dims> 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. Constructors for the nd_range class are described in Table 3.37 and methods in Table 3.38.

Constructors	Description
nd_range<dimensions>(range<dimensions> global_size, range<dimensions> local_size) 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.
nd_range<dimensions>(const nd_range<dimensions> &)	Construct an nd_range by deep copy from another ndrange.

Table 3.37: Constructors for the nd_range class.

Methods	Description
range<dimensions> get_global_range()const	Return the constituent global range.
range<dimensions> get_local_range()const	Return the constituent local range.
id<dimensions> get_offset()const	Return the constituent offset.
range<dimensions> get_group_range()const	Return a range representing the number of groups in each dimension. This range would result from global_size/local_size as provided on construction.

Table 3.38: Methods for the nd_range class.

3.7.1.3 ID class

`id<int dims>` is a vector of dimensions that is used to represent an *index* 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`. Constructors for the id class are described in Table 3.39, methods in Table 3.40 and global operators on ids in Table 3.41

```
namespace cl {
  namespace sycl {
    template <int dims>
    struct id {
      id();
      id(const id<dims> &init);
      id(const range<dims> &r);
      id(const item<dims> &it);

      // Valid only for when dims==1
      id(size_t x);
      // Valid only for when dims==2
      id(size_t x, size_t y);
      // Valid only for when dims==3
      id(size_t x, size_t y, size_t z);

      size_t get(unsigned int dimension) const;
      size_t &operator[](unsigned int dimension);

      id &operator=(const id &rhs);

      bool operator==(const id_impl &rhs) const;
    };

    template <size_t Dimensions>
    id<dims> operator *(id<dims> a,
                        id<dims> b);

    template <size_t dims>
    id<dims> operator /(id<dims> dividend,
                        id<dims> divisor);

    template <size_t Dimensions>
    id<dims> operator +(id<dims> a,
                        id<dims> b);

    template <size_t Dimensions>
    id<dims> operator -(id<dims> a,
                        id<dims> b);
  }  // namespace sycl
}  // namespace cl
```

Constructors	Description
`id<dimensions>()`	Construct an identity id with 0 in each dimension.
`id<dimensions>(const id<dimensions> &)`	Construct an id by deep copy.
`id<dimensions>(const range<dimensions> &r)`	Construct an id from the dimensions of a range.
`id<dimensions>(const item<dimensions> &it)`	Construct an id from it.get_global_id().
`id<dimensions>(size_t,...)`	Construct an id from a list of `dimensions` size_t values representing each dimension.

Table 3.39: Constructors for the `id` class.

Methods	Description
`size_t get(unsigned int dimension)const`	Return the value of the `id` for dimension `dimension`.
`size_t &operator[](unsigned int dimension)`	Return the value of the `id` for dimension `dimension`.

Table 3.40: Methods for the `id` class.

Global operators	Description

`template <int dimensions>`
`  id<dimensions> operator *(`
`  id<dimensions> a,`
`  id<dimensions> b)`
	Multiply each element of `a` by its respective element of `b` and return an id constructed from the resulting values.

`template <int dimensions>`
`  id<dimensions> operator /(`
`  id<dimensions> dividend,`
`  id<dimensions> divisor)`
	Divide each element of `dividend` by its respective element element in `divisor` and return an id constructed of the resulting value.

`template <int dimensions>`
`  id<dimensions> operator +(`
`  id<dimensions> a,`
`  id<dimensions> b)`
	Add each element of `a` to its respective element of `b` and return an id constructed from the resulting values.

`template <int dimensions>`
`  id<dimensions> operator -(`
`  id<dimensions> a,`
`  id<dimensions> b)`
	Subtract each element of `b` from its respective element of `a` and return an id constructed from the resulting values.

Table 3.41: Global operators for the `id` class.

3.7.1.4 Item class

`item<int dims>` identifies an instance of the functor executing at each point in a `range<>` passed to a `parallel_for` call. It encapsulates enough information to identify the work-item's global ID, the range over which the kernel is executing, and 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 functor. Methods for the `item<>` class are described in Table 3.42.

```
namespace cl {
  namespace sycl {
    template <int dims = 1>
    struct item {
      item() = delete;

      id<dims> get_global_id() const;

      size_t get(int dimension) const;

      size_t &operator[](int dimension);

      range<dims> get_global_range() const;

      id<dims> get_offset() const;
    };
  }  // namespace sycl
}  // namespace cl
```

Methods	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(int dimension)const`	Return the constituent global `id<>` representing the work-item's position in the global iteration space in the given `dimension`.
`size_t &operator[](int dimension)`	Return the constituent global `id<>` l-value representing the work-item's position in the global iteration space in the given `dimension`.
`range<dimensions> get_global_range()const`	Returns a `range<>` representing the dimensions of the `nd_range<>`
`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.

Table 3.42: Methods for the `item` class.

3.7.1.5 nd_item class

nd_item<int dims> identifies an instance of the functor executing at each point in an nd_range<int dims> passed to a parallel_for_ndrange call. It encapsulates enough information to identify the work-item's local and global IDs, the work-groups ID and also provides barrier functionality to synchronize work-items. Instances of the nd_item<int dims> class are not user-constructible and are passed by the runtime to each instance of the functor. Methods for the nd_item<int dims> class are described in Table 3.43.

```
namespace cl {
  namespace sycl {
    template <int dims = 1>
    struct nd_item {
      nd_item() = delete;

      id<dims> get_global_id() const;

      size_t get_global_id(int dimension) const;

      id<dims> get_local_id() const;

      size_t get_local_id(int dimension) const;

      id<dims> get_group_id() const;

      size_t get_group_id(int dimension) const;

      range<dims> get_global_range() const;

      range<dims> get_local_range() const;

      id<dims> get_offset() const;

      nd_range<dims> get_nd_range() const;

      void barrier(access::fence_space flag = access::global_and_local) const;
    };
  }  // namespace sycl
}  // namespace cl
```

Methods	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 global iteration space in the given dimension .
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.
id<dimensions> get_group_id()const	Return the constituent group id<> representing the work-group's position within the overall nd_range<>.
size_t get_group_id(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.
range<dimensions> get_global_range()const	Returns a range<> representing the dimensions of the nd_range<>
range<dimensions> get_local_range()const	Returns a range<> representing the dimensions of the current work-group.
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.
void barrier(access::fence_space flag=access::global_and_local)const	Executes a barrier with memory ordering on the local address space, global address space or both based on the value of flag. The current work-item will wait at the barrier until all work-items in the current work-group have reached the barrier. In addition the barrier performs a fence operation ensuring that all memory accesses in the specified address space issued before the barrier complete before those issued after the barrier.

Table 3.43: Methods for the nd_item class.

3.7.1.6 Group class

The group<int dims> is passed to each instance of the functor execution a parallel_for_workgroup in a hierarchical parallel execution. The group encapsulates all functionality required to represent a particular group within a parallel execution. It is not user-constructable. Methods for the group<> class are described in Table 3.44.

```
namespace cl {
  namespace sycl {
    template <int dims = 1>
    struct group {
      id<dims> get_group_id() const;

      range<dims> get_local_range() const;

      range<dims> get_global_range() const;

      id<dims> get_offset() const;

      nd_range<dims> get_nd_range() const;

      size_t get(int dimension) const;

      size_t operator[](int dimension) const;

    };
  } // sycl
} // cl
```

Methods	Description
range<dimensions> get_global_range()	Return the constituent global range.
range<dimensions> get_local_range()	Return a range<> representing the dimensions of the current group.
id<dimensions> get_offset()	Return the offset of the nd_range.
nd_range<dimensions> get_nd_range()	Return the nd_range wrapping the global range, group range and offset.
id<dimensions> get_group_id()	Return an id<> representing the index of the group within the nd_range<>.
size_t get(int dimension)const	Return the index of the group in the given dimension within the nd_range<>.
size_t operator[](int dimension)const	Return the index of the group in the given dimension within the nd_range<>.

Table 3.44: Methods for the group class.

3.7.2 Defining kernels

In SYCL functions that are executed in parallel on a SYCL device are referred to as *kernel methods*. A *kernel* containing such a *kernel method* is enqueued on a device queue in order to be executed on that particular device. The return type of the *kernel method* is `void`, and all kernel accesses between host and device are defined using the accessor class 3.6.4.

There are three ways of defining kernels, defining them as functors, as C++11 lambda functions or as OpenCL `cl_kernel` objects. However, in the case of OpenCL kernels, the developer is expected to have created the kernel and set the kernel arguments.

3.7.2.1 Defining kernels as functors

A kernel can be defined as a C++ functor. In this case, the *kernel method* is the method defined as `operator()` in the normal C++ functor style. These functors provide the same functionality as any C++ functors, with the restriction that they need to follow C++11 standard layout rules. The kernel method can be templated via templating the kernel functor class. The *operator()* function may take different parameters depending on the data accesses that defined for the specific kernel.

In the following example we define a trivial functor with no outputs and no accesses of host or pre-allocated device data. The kernel is executed on a unary index space for the specific example, since its using single_task at its invocation.

```
class MyFunctor
{
    float m_parameter;

public:
    MyFunctor(float parameter):
        m_parameter(parameter)
    {
    }

    void operator() ()
    {
        // [kernel code]
    }
};

void workFunction(float scalarValue)
{
    MyFunctor myKernel(scalarValue);

    command_group(queue, [&] () {
        single_task(myKernel);
    });
}
```

3.7.2.2 Defining kernels as lambda functions

In C++11, functors can be defined using lambda functions. We allow lambda functions to define kernels in SYCL, but we have an extra requirement to *name lambda functions* in order to enable the linking of the SYCL device kernels with the host code to invoke them. The name of a lambda function in SYCL is a C++ class. If the lambda function relies on template arguments, then 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 implemented.

To invoke a C++11 lambda, the kernel name must be included explicitly by the user as a template parameter to the kernel invoke function.

The kernel method for the lambda function is the lambda function method itself. The kernel lambda must use copy for all of its captures (i.e. `[=]`).

```
class MyKernel;

command_group(command_queue, [&] () {
        single_task<class MyKernel>([=] () {
                    // [kernel code]
                });
    });
```

3.7.2.3 Defining kernels using program objects

In case the developer needs to specify compiler flags or special linkage options for a kernel, then a program object can be used, as described in 3.7.2.6. The kernel is defined as a functor 3.7.2.1 or lambda function 3.7.2.2. The user can obtain a program 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 program they wish to obtain.

Once the user has obtained a program object for a kernel, they can compile and/or link the program object using methods on the `program` class. To force a kernel invoke to use the version of a kernel that is within a specific program object, they provide the program object before the kernel functor (or lambda) in the kernel invoke call.

In the following example, the kernel is defined as a lambda function. The example obtains the program object for the lambda function kernel and then passes it to the `parallel_for`.

```
class MyKernel; //Forward declaration of the name of the lambda functor

cl::sycl::queue myQueue;
cl::sycl::program MyProgram(myQueue.get_context());

/* use the name of the kernel to obtain the associated program */
MyProgram.build_kernel_from_name<MyKernel>();

cl::sycl::command_group(myQueue, [&] () {
    cl::sycl::parallel_for<class MyKernel>(cl::sycl::nd_range<2>(4,4),
                MyProgram, // execute the kernel as compiled in MyProgram
                ([=] (cl::sycl::item index) {
                  //[kernel code]
```

```
            }));
  });
```

In the above example, the *kernel method* is defined in the codelineparallel_for invocation as part of a lambda functor which is named using the type of the forward declared class "myKernel". The type of the functor and the program object enable the compilation and linking of the kernel in the program 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 next example, a SYCL kernel is linked with an existing pre-compiled OpenCL C program object to created a combined program object, which is then called in a `parallel_for`.

```
class MyKernel; //Forward declaration of the name of the lambda functor

cl::sycl::queue myQueue;

// obtain an existing OpenCL C program object
cl_program myClProgram = ...;

// Create a SYCL program object from a cl_program object
cl::sycl::program myExternProgram(myQueue.get_context(), myClProgram);

// Add in the SYCL program object for our kernel
cl::sycl::program mySyclProgram (myQueue.get_context ());
mySyclProgram.compile_from_kernel_name<MyKernel>("-my-compile-options");

// Link myClProgram with the SYCL program object
cl::sycl::program myLinkedProgram ({myExternProgram,
                   mySyclProgram}, "-my-link-options");

cl::sycl::command_group(myQueue, [&] () {
    cl::sycl::parallel_for<class MyKernel>(cl::sycl::nd_range<2>(4,4),
               myLinkedProgram, // execute the kernel as compiled in MyProgram
               ([=] (cl::sycl::item index) {
                 //[kernel code]
               }));
  });
```

3.7.2.4 Defining kernels using OpenCL C kernel objects

In OpenCL C [1] program and kernel objects can be created using the OpenCL C API, which is available in the SYCL system. Interoperability of OpenCL C kernels and the SYCL system is achieved by allowing the creation of a *SYCL kernel* object from an *OpenCL kernel* object.

The constructor using kernel objects from 3.45:

```
kernel::kernel(cl_kernel kernel)
```

creates a `cl::sycl::kernel` which can be enqueued using all the `parallel_for` functions which can enqueue a kernel object as described in the next section. This way of defining kernels assumes the developer is using OpenCL C to create the kernel and set the kernel arguments. The system assumes that the developer has already

called set kernel args when they are trying to enqueue the kernel. Buffers do give ownership to their accessors on specific contexts and the developer can enqueue OpenCL kernels in the same way as enqueuing SYCL kernels. However, the system is not responsible for data management at this point.

3.7.2.5 The kernel class

The *kernel class* is an abstraction of a kernel object in SYCL. At 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 3.7.3. 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 program class, as defined in 3.7.2.6. An both the above cases, the developer cannot instantiate a kernel object but can instantiate an object a functor class that he could use or create a functor from a kernel method using C++11 features. The kernel class object needs a parallel_for invocation or an explicit compile_and_link() call through the program class, for this compilation of the kernel to be triggered.

Finally, a kernel class instance may encapsulate an OpenCL *kernel* object that was created using the OpenCL C interface and the arguments of the kernel already set. In this case since the developer is providing the cl_kernel object, this constructor is allowed to be used by the developer.

The kernel class also provides the interface for getting information from a kernel object.

```
namespace cl {
  namespace sycl {
    class kernel {
    private:
      friend class program;

      // The default object is not valid because there is no
      // program or cl_kernel associated with it
      kernel();

    public:
      kernel (const kernel& rhs);

      kernel (cl_kernel openclKernelObejct);

      cl_kernel get() const;

      context get_context() const;

      program get_program() const;

      string_class get_kernel_attributes() const;

      template<cl_int name> typename
      detail::param_traits<detail::cl_kernel_info, name>::param_type get_info() const;
    };
  }  // namespace sycl
}  // namespace cl
```

Constructor	Description
`kernel (cl_kernel openclKernelObj)`	Constructor for SYCL kernel class given an OpenCL kernel object with set arguments, valid for enqueuing.
`kernel (const kernel& rhs)`	Copy constructor for kernel class.

Table 3.45: `kernel` class constructors

Methods	Description
`cl_kernel get()`	Return the OpenCL kernel object for this kernel.
`context get_context()`	Return the context that this kernel is defined for.
`program get_program()`	Return the program that this kernel is part of.
`string_class get_kernel_attributes()`	
`string_class get_function_name()`	Return the name of the kernel function.

Table 3.46: Methods for the `kernel` class.

3.7.2.6 Program class

A *program* contains one or more kernels and any functions or libraries necessary for the program's execution. A program will be enqueued inside a context and each of the kernels will be enqueued on a corresponding device. Program class can be really useful for pre-compiling kernels and enqueuing them on multiple command_groups. It also allows usage of functions define in OpenCL kernels from SYCL kernels via compiling and linking them in the same program object.

```
namespace cl {
  namespace sycl {
    class program {
    public:
    // Create an empty program object
      program(const context& context);

    // Create an empty program object
      program(const context& context, vector_class<device> device_list);

    // Create a program object from a cl_program object
      program(const context& context, cl_program clProgram);

    // Create a program by linking a list of other programs
      program(vector_class<program> program_list, string_class link_options="");

      ~program();

      program(const program& rhs);

    /* This obtains a SYCL program object from a SYCL kernel name
       and compiles it ready to link */
      template<typename kernelT>
      void compile_from_kernel_name(string_class compile_options = "");

    /* This obtains a SYCL program object from a SYCL kernel name
       and builds it ready-to-run */
      template<typename kernelT>
      void build_from_kernel_name(string_class compile_options = "");

      // Get a kernel from a given Name (Functor)
      template<typename kernelT>
      kernel get_kernel<kernelT>() const;

      template<cl_int name> typename
      detail::param_traits<detail::cl_program_info,name>::param_type
        get_info() const;

      vector_class<vector_class<char>> get_binaries() const;

      vector_class<::size_t> get_binary_sizes() const;

      vector_class<device> get_devices() const;

      string_class get_build_options() const;
```

```
    cl_program get() const;
  };
 } // namespace sycl
} // namespace cl
```

Constructors	Description
`program (` `  const context & context)`	Constructs an empty program object for context for all associated devices with context.

`program (`
`  const context & context,`
`  vector_class<device> device_list)`
	Constructs an empty program object for all the devices of *device_list* associated with the *context*.

`program (`
`  vector_class<program> program_list,`
`  string_class link_options="")`
	Constructs a program object for a list of programs and links them together using the `link_options`

`program (`
`  const context & context,`
`  cl_program program)`
	Constructs a program object for an OpenCL program object.

`program(const program& rhs);`	Copy constructor for the program class.

Table 3.47: Constructors for the `program` class

Methods	Description

`template<typename kernelT>`
`  void compile_from_kernel_name(`
`  string_class compile_options="")`
	Compile the kernel defined to be of type *kerneT* into the program, with compile options given by compile_options". The kernel can be defined either as a functor of type *kerneT* or as a lambda function which is named with the class name *kernelT*. The program object will need to be linked later.

`template<typename kernelT>`
`  void build_from_kernel_name(`
`  string_class compile_options="")`
	Build the kernel defined to be of type *kerneT* into the program, with compile options given by compile_options". The kernel can be defined either as a functor of type *kerneT* or as a lambda function which is named with the class name *kernelT*.

`template<cl_int name>`
`  typename detail::param_traits<detail::`
`cl_program_info,name>::param_type`
`  get_info()`
	Retrieve information of the built OpenCL program object.

`vector_class<char*> get_binaries()`	Return the array of compiled binaries associated with the program, as compiled for each device.
`vector_class<device> get_devices()`	Return the list of devices this program was constructed against
`string_class get_build_options()`	Retrieve the set of build options of the program. A program is created with one set of build options.

Table 3.48: Methods for the `program` class

Programs allow the developers to provide their own compilation and linking options and also compile and link on demand one or multiple kernels. Compiler options allowed are described in the OpenCL specification [1, p. 145, § 5.6.4] and the linker options are described in [1, p. 148, § 5.6.5].

3.7.3 Invoking kernels

Kernels can be invoked as *single tasks*, basic *data-parallel kernels*, OpenCL-style *NDRanges* in *work-groups*, or SYCL *hierarchical parallelism*. There are five functions that enable these for types of invocation of kernels.

Each function takes a kernel name template parameter. The kernel name must be a datatype that is unique for each kernel invocation. If a kernel is a functor, then the kernel's functor type will be automatically used as the kernel name and so the user does not need to supply a name. If the kernel function is a C++11 lambda function, then the user must manually provide a kernel name to enable linking between host and device code to occur.

```
template<typename KernelName, class KernelType>
void single_task(KernelType);

template<typename KernelName, class KernelType, int dimensions>
void parallel_for(
    range<dimensions> num_work_items,
    KernelType);

template<typename KernelName, class KernelType, int dimensions>
void parallel_for(
    range<dimensions> num_work_items,
    id<dimensions> work_item_offset,
    KernelType);

template<typename KernelName, class KernelType, int dimensions>
void parallel_for(
    nd_range<dimensions> execution_range,
    KernelType);

template<class KernelName, class WorkgroupFunctionType, int dimensions>
void parallel_for_work_group(
    range<dimensions> num_work_groups,
    WorkgroupFunctionType);

template<class KernelType, int dimensions>
void parallel_for_work_item(
    group num_work_items,
    KernelType);
```

3.7.3.1 Single Task invoke

SYCL provides a simple interface to enqueue a kernel that will be sequentially executed on an OpenCL device. Only one instance of the kernel will be executed. This interface is useful a primitive for more complicated parallel algorithms, as it can easily create a chain of sequential tasks on an OpenCL device with each of them managing its own data transfers.

This function can only be called inside a command group and any accessors that are used within it should be defined inside the same command group.

Local accessors are disallowed for single task invocations.

```
single_task<class kernel_name>(
    [=] () {
      // [kernel code]
    }));
```

For single tasks, the kernel method takes no parameters, as there is no need for indexing classes in a unary index space.

3.7.3.2 Parallel For invoke

The `parallel_for` interface offers the ability to the SYCL users to declare a kernel and enqueue it as a parallel execution over a range of instances. There are three variations to the parallel_for interface and they depend on the index space that the developer would like the kernels to be executed on and the feature set available in those kernels.

In the simplest case, the developer only needs to provide the number of work-items the kernel will use in total and the system will use the best range available to enqueue it on a device. In this case the developer, may only need to know the index over the total *range* that he has provided, by providing the number of work-items that will be executing on. This type of kernels will be using the parallel_for invocation with a `range` type to provide the range of the execution and an `id` to provide the index within that range. Whether it is a lambda function or a kernel functor the parameter to the invocation function need to be `id`.

An example of a parallel_for using a lambda function for a kernel invocation in this case of parallel_for is the following.

```
class MyKernel;

command_group(myQueue, [&] ()
{
    auto acc=myBuffer.get_access<read_write>();

    parallel_for<class MyKernel>(range<1>(workItemNo),
                                 [=] (id index)
    {
        acc[index] = 42.0f;
    });
});
```

Another case, which is based on this very basic parallel_for, is the case where the developer would like to let the runtime choose the index space that is matching best the *range* provided but would like to use information given the scheduled interface instead of the general interface. This is enabled by using the class `item` as an indexing class in the kernel, and of course that would mean that the kernel invocation would match the `range` with the `item` parameter to the kernel.

```
class MyKernel;

command_group(myQueue, [&] ()
{
    auto acc=myBuffer.get_access<read_write>();
```

```
    parallel_for<class MyKernel>(range<1>(workItemNo),
                                 [=] (item myItem)
    {
        size_t index = item.get_global();
        acc[index] = 42.0f;
    });
});
```

Local accessors are disallowed for the basic parallel_for invocations described above.

The following two examples show how a kernel functor can be launched over a 3D grid, 3 elements in each dimension. In the first case work-item IDs range from 0 to 2 inclusive, in the second case work-item IDs they run from 1 to 3.

```
parallel_for<class example_kernel1>(
   range<3>(3,3,3), // global range
     [=] (item<3> it) {
       //[kernel code]
     });

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 gives to the developer the low-level functionality for 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, which provides all the functionality of OpenCL for an NDRange. In this case, the developer needs to defined 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 functor or lambda is passed an nd_item<i>nstance making all the information available as well as barrier primitives to synchronize the work-items in the 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 sixteen work-groups. Each group of work-items synchronizes with a barrier.

```
parallel_for<class example_kernel>(
    nd_range(range(4, 4, 4), range(2, 2, 2)),
      [=] (nd_item<3> item) {
        //[kernel code]
        // Internal synchronization
        item.barrier(access::fence_space::global);
        //[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<c>lass added to the identifier for each point in the range.

In all of these case the underlying *nd_range* will be created and the kernel defined as a lambda or as a kernel functor will be created and enqueued as part of the command_group.

3.7.3.3 Parallel For hierarchical invoke

The hierarchical parallel kernel execution interface provides the same functionality as is available from the *NDRange* 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 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 it is more efficient to do so this code will be executed for each work-item.

Also within this 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 barrier in the earlier example.

```
parallel_for_work_group<class example_kernel>(
    nd_range<3>(range<3>(10, 10, 10), range<3>(2, 2, 2))
    [=] (group<3> myGroup)
      {
        //[workgroup code]
        int myLocal; // this variable shared between workitems
        parallel_for_work_item(
          myGroup,
          [=] (id<3> myItem)
            {
                //[work-item code]
            });
        parallel_for_work_item(
          myGroup,
          [=] (id<3> myItem)
            {
                //[work-item code]
            });
      //[workgroup code]
    });
```

This interface offers a more intuitive way to 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_workgroup` and the nested `parallel_for_workitem` functions. It also provides this visibility to the compiler without the need for difficult loop fission such that a host execution may be more efficient.

3.7.4 Rules for parameter passing to kernels

In a case where a kernel is a C++ functor or C++11 lambda object, any values in the functor or captured in the C++11 lambda object 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. For OpenCL 1.0–1.2 class devices, this means that the argument must be passed via `clSetKernelArg` and be compiled as a kernel parameter of the valid reference type. For global shared data access, the parameter must be an OpenCL `global` pointer. For an accessor that specifies OpenCL `constant` access, the parameter must be an OpenCL `constant` pointer. For images, the accessor must be passed as an `image_t` and/or sampler.

- 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.

- 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++ standard layout values must be passed by value to the kernel.

- C++ non-standard layout values 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 (`cl::sycl::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 OpenCL kernel parameters if all members of the aggregate type are unaffected by the rules above.

3.8 Data Types

3.8.1 Vector types

```
namespace cl {
  namespace sycl {
    template <typename dataT, int numElements>
    class vec {
    public:
      vec();

      explicit vec(const dataT &arg);

      vec(const T0 &arg0...args);

      vec(const vec<dataT, numElements> &rhs);

      vec<dataT, numElements> operator+(const vec<dataT, numElements> &rhs) const;
      vec<dataT, numElements> operator-(const vec<dataT, numElements> &rhs) const;
      vec<dataT, numElements> operator*(const vec<dataT, numElements> &rhs) const;
      vec<dataT, numElements> operator/(const vec<dataT, numElements> &rhs) const;
      vec<dataT, numElements> operator+=(const vec<dataT, numElements> &rhs);
      vec<dataT, numElements> operator-=(const vec<dataT, numElements> &rhs);
      vec<dataT, numElements> operator*=(const vec<dataT, numElements> &rhs);
      vec<dataT, numElements> operator/=(const vec<dataT, numElements> &rhs);
      vec<dataT, numElements> operator+(const dataT &rhs) const;
      vec<dataT, numElements> operator-(const dataT &rhs) const;
      vec<dataT, numElements> operator*(const dataT &rhs) const;
      vec<dataT, numElements> operator/(const dataT &rhs) const;
      vec<dataT, numElements> operator+=(const dataT &rhs);
      vec<dataT, numElements> operator-=(const dataT &rhs);
      vec<dataT, numElements> operator*=(const dataT &rhs);
      vec<dataT, numElements> operator/=(const dataT &rhs);

      vec<dataT, numElements> &operator=(const vec<dataT, numElements> &rhs);
      vec<dataT, numElements> &operator=(const dataT &rhs);

      bool operator==(const vec<dataT, numElements> &rhs) const;
      bool operator!=(const vec<dataT, numElements> &rhs) const;

      // Swizzle methods (see notes)
      swizzled_vec<T, out_dims> swizzle<int s1, ...>();
      #ifdef SYCL_SIMPLE_SWIZZLES
      swizzled_vec<T, 4> xyzw();
      ...
      #endif // #ifdef SYCL_SIMPLE_SWIZZLES
    };
  }  // namespace sycl
}  // namespace cl
```

SYCL provides a templated cross-platform vector type 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 swizzeled set of component elements. The vector are defined in Table 3.49 and Table 3.50

`vec<typename T, int dims>` is a vector type that compiles down to the OpenCL built-in vector types on OpenCL devices where possible and provides compatible support on the host. The `vec` class is templated on its number of dimensions and its element type. The dimensions parameter, `dims`, can be one of: 1, 2, 3, 4, 8 or 16. Any other value should produce a compilation failure. The element type parameter, `T`, must be one of the basic scalar types supported in device code.

The SYCL library provides typedefs for: `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float` and `double` in all valid sizes. These vector typedefs are named `TypenameSize`, for example: `int2` is a vector of two integer elements, mapping to `vec<int, 2>`.

`swizzled_vec<T, out_dims> vec<T, in_dims>::swizzle<int s1, int s2...> ()` returns a temporary object representing a swizzled set of the original vector's member elements. The number of `s1`, `s2` parameters is the same as `out_dims`. All `s1`, `s2` parameters must be integer constants from zero to `in_dims-1`. The swizzled vector may be used as a source (r-value) and destination (l-value). In order to enable the r-value and l-value swizzling to work, this returns an intermediate swizzled-vector class, which can be implicitly converted to a vector (r-value evaluation) or assigned to.

If the user `#defines` the macro `SYCL_SIMPLE_SWIZZLES` before `#include <cl/sycl.hpp>`, then swizzle functions are defined for every combination of swizzles for 2D, 3D and 4D vectors only. The swizzle functions take the form:

```
swizzled_vec<T, out_dims> vec<T, in_dims>::xyzw();
swizzled_vec<T, out_dims> vec<T, in_dims>::rgba();
```

where, as above, the number of `x`, `y`, `z`, `w` or `r`, `g`, `b`, `a` letters is the same as `out_dims`. All `x`, `y`, `z`, `w` or `r`, `g`, `b`, `a` parameters must be letters from the sets first `in_dims` letters in "`xyzw`" or "`rgba`".

Swizzle letters may be repeated or re-ordered. For example, from a vector containins integers [0, 1, 2, 3], `vec.xxzy()` would return a vector of [0, 0, 2, 1].

Constructors	Description
`vec<T, dims>()`	Default construct a vector with element type `T` and with `dims` dimensions by default construction of each of its elements.
`explicit vec<T, dims>(const T &arg)`	Construct a vector of element type `T` and `dims` dimensions by setting each value to arg by assignment.

`vec<T, dims> (`
`  const T &element_0,`
`  const T &element_1,`
`  . . . ,`
`  const T &element_dims-1)`
	Construct a vector with element type `T` and with `dims` dimensions out of `dims` initial values.

`vec<T, dims>(const &vec<T, dims>)`	Construct a vector of element type `T` and `dims` dimensions by copy from another similar vector.

Table 3.49: Constructors for the `vec` class

Methods	Description
`vec<dataT, numElements> operator+(` `  const vec<dataT, numElements> &rhs)`	Construct vector from the sum of the respective elements of the current vector and rhs.
`vec<dataT, numElements> operator-(` `  const vec<dataT, numElements> &rhs)`	Construct vector by subtracting the elements of rhs from the respective elements of the current vector.
`vec<dataT, numElements> operator*(` `  const vec<dataT, numElements> &rhs)`	Construct vector from the product of the elements of the current vector by the respective elements of rhs.
`vec<dataT, numElements> operator/(` `  const vec<dataT, numElements> &rhs)`	Construct vector from the division of the elements of the current vector by the elements of rhs.
`vec<dataT, numElements> operator+=(` `  const vec<dataT, numElements> &rhs)`	Add each element of rhs to the respective element of the current vector in-place.
`vec<dataT, numElements> operator-=(` `  const vec<dataT, numElements> &rhs)`	Subtract each element of rhs from the respective element of the current vector in-place.
`vec<dataT, numElements> operator*=(` `  const vec<dataT, numElements> &rhs)`	Multiple each element of the current vector by the respective element of rhs in-place.
`vec<dataT, numElements> operator/=(` `  const vec<dataT, numElements> &rhs)`	Divide each element of the current vector in-place by the respective element of rhs.
`vec<dataT, numElements> operator+(` `  const dataT &rhs)`	Construct vector by adding rhs to each element of the current vector.
`vec<dataT, numElements> operator-(` `  const dataT &rhs)`	Construct vector by subtracting rhs from each element of the current vector.
`vec<dataT, numElements> operator*(` `  const dataT &rhs)`	Construct vector by multiplying each element of the current vector by rhs.
`vec<dataT, numElements> operator/(` `  const dataT &rhs)`	Construct vector by dividing each element of the current vector by rhs.
`vec<dataT, numElements> operator+=(` `  const dataT &rhs)`	Add rhs in-place to each element of the current vector.
`vec<dataT, numElements> operator-=(` `  const dataT &rhs)`	Subtract rhs in-place from each element of the current vector.
`vec<dataT, numElements> operator+*=(` `  const dataT &rhs)`	Multiple in-place each element of the current vector by rhs.
`vec<dataT, numElements> operator/=(` `  const dataT &rhs)`	Divide in-place each element of the current vector by rhs.
`vec<dataT, numElements> &operator=(` `  const vec<dataT, numElements> &rhs)`	Update each element of the current vector with the respective element of rhs and return a reference to the current vector.
`vec<dataT, numElements> &operator=(` `  const dataT &rhs)`	Update each element of the current vector with rhs and return a reference to the current vector.
`bool operator==(const vec<dataT, numElements> &rhs) const`	Return true if all elements of rhs compare equal to the respective element of the current vector.
`bool operator!=(const vec<dataT, numElements> &rhs) const`	Return true if any one element of rhs does not compare equal to the respective element of the current vector.

Table 3.50: Methods for the vec class

3.9 Kernel Functions for SYCL Host and Device

SYCL kernels may execute on an OpenCL device or on SYCL host, which requires that the functions used in the kernels can be compiled and linked for both device and host. In the SYCL system the OpenCL built-ins are available for the SYCL host and device within the `cl::sycl` namespace, although, their semantics may be different. This section follows the OpenCL 1.2 specification document [1, ch. 6.12] and describes the behaviour of these functions for SYCL Host and Device.

3.9.1 Description of the built-in types available for SYCL host and device

All the OpenCL built-in types are available in the namespace *cl::sycl*. For the purposes of this document we use type names for describing sets of SYCL valid types. The type names themselves are not valid SYCL types, but they represent a set of valid types, as defined in tables 3.9.1 and 3.9.1.

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 3.7.1.4, 3.7.1.5 and 3.7.1.6.

Generic type name	Description
floatn	cl::sycl::float2, cl::sycl::float3, cl::sycl::float4, cl::sycl::float8, cl::sycl::float16
genfloatf	float, floatn
doublen	cl::sycl::double2, cl::sycl::double3, cl::sycl::double4, cl::sycl::double8, cl::sycl::double16
genfloatd	double, doublen
genfloat	float, floatn double, doublen
sgenfloat	float, double It is a scalar type that matches the corresponding vector type *floatn* or *doublen*.
charn	cl::sycl::char2, cl::sycl::char3, cl::sycl::char4, cl::sycl::char8, cl::sycl::char16
ucharn	cl::sycl::uchar2, cl::sycl::uchar3, cl::sycl::uchar4, cl::sycl::uchar8, cl::sycl::uchar16
genchar	char, charn
ugenchar	unsigned char, ugenchar

Table 3.51: Generic type name description, which serves as a description for all valid types of parameters to kernel functions. [1] (Part I)

Generic type name	Description
shortn	cl::sycl::short2, cl::sycl::short3, cl::sycl::short4, cl::sycl::short8, cl::sycl::shor16
genshort	short, shortn
ushortn	cl::sycl::ushort2, cl::sycl::ushort3, cl::sycl::ushort4, cl::sycl::ushort8, cl::sycl::ushor16
ugenshort	unsigned short, ugenshort
uintn	cl::sycl::uint2, cl::sycl::uint3, cl::sycl::uint4, cl::sycl::uint8, cl::sycl::uint16
ugenint	int, uintn
intn	cl::sycl::int2, cl::sycl::int3, cl::sycl::int4, cl::sycl::int8, cl::sycl::int16
genint	int, intn
ulongn	cl::sycl::ulong2, cl::sycl::ulong3, cl::sycl::ulong4, cl::sycl::ulong8, cl::sycl::ulong16
ugenlong	unsigned long, ulongn
longn	cl::sycl::long2, cl::sycl::long3, cl::sycl::long4, cl::sycl::long8, cl::sycl::long16
genlong	long, longn
geninteger	genchar, ugenchar, genshort, ugenshort, genint, ugenint, genlong, ugenlong
sgeninteger	char,short, int, long, unsigned char, unsigned short, unsigned int, unsigned long
igeninteger	uchar, ucharn, ushort, ushortn, uint, uintn, ulong, ulongn
gentype	char, charn, uchar, ucharn, short, shortn, ushort, ushortn, int, intn, uint, uintn, long, longn, ulong, ulongn, float, floatn, double, doublen.

Table 3.52: Generic type name description, which serves as a description for all valid types of parameters to kernel functions. [1] (Part II)

3.9.2 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 3.7.1.5 and 3.7.1.6.

3.9.3 Math Functions

In SYCL the OpenCL math functions are available in the namespace *cl::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* counterparts, for dimensions 2,3,4,8, and 16. On the host the vector types are going to be using the *vec* class and on an OpenCL device are going to be using the corresponding OpenCL vector types.

The built-in functions available for SYCL host and device with the same precision requirements for both host and device, are described in tables 3.53, 3.54, 3.55 and 3.56.

Math Function (Part I)	**Description**
`genfloat acos ( genfloat x )`	Inverse cosine function.
`genfloat acosh ( genfloat x )`	Inverse hyperbolic cosine.
`genfloat acospi ( genfloat x )`	Compute $acosx/\pi$
`genfloat asin ( genfloat x )`	Inverse sine function.
`genfloat asinh ( genfloat x )`	Inverse hyperbolic sine.
`genfloat asinpi ( genfloat x)`	Compute $asinx/\pi$
`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) / π.
`genfloat atan2pi ( genfloat y, genfloat x )`	Compute atan2 (y, x) / π.
`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.

Table 3.53: Math functions which work on SYCL Host and device. They correspond to Table 6.7 of the OpenCL 1.2 specification [1](Part I)

Math Function (Part II)	Description
`genfloat cos ( genfloat x )`	Compute cosine.
`genfloat cosh ( genfloat x )`	Compute hyperbolic cosine.
`genfloat cospi ( genfloat x )`	Compute cos (πx).
`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 )` `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 )` `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 $xy * trunc(x/y)$.
`floatn fract ( floatn x, intn *iptr )` `float fract ( float x, int * iptr )`	Returns fmin(x − floor (x), 0x1.fffffep-1f). floor(x) is returned in iptr.
`doublen frexp ( doublen x, intn *exp )` `double frexp ( double x, int * 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.

`int logb ( float x )`
`intn ilogb ( genfloat x )`
`int logb ( double x )`
`intn logb ( doublen x )`
	Return the exponent as an integer value.

`genfloat ldexp (genfloat x, genint k)`
`floatn ldexp (floatn x, int k)`
`doublen ldexp (doublen x, int k)`
	Multiply x by 2 to the power k.

Table 3.54: Math functions which work on SYCL Host and device. They correspond to Table 6.7 of the OpenCL 1.2 specification [1] (Part II)

Math Function (Part III)	Description
`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, genint *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)`	Compute natural logarithm.
`genfloat log2 (genfloat)`	Compute a base 2 logarithm.
`genfloat log10 (genfloat)`	Compute a base 10 logarithm.
`genfloat log1p (genfloat x)`	Compute $log_e(1.0 + x)$.
`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, genfloat *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.

`floatn nan ( unintn nancode )`
`float nan (unsigned int nancode )`
`doublen nan (ulongn nancode )`
`double nan (unsigned long nancode )`
	Returns a quiet NaN. The nancode may be placed in the significand of the resulting NaN.

`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.

Table 3.55: Math functions which work on SYCL Host and device. They correspond to Table 6.7 of the OpenCL 1.2 specification [1] (Part III)

Math Function (Part IV)	Description
`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, genint *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)`	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)`	Compute inverse square root.
`genfloat sin (genfloat)`	Compute sine.
`genfloat sincos (genfloat x, genfloat *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 (π 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 (π x).
`genfloat tgamma ( genfloat x )`	Compute the gamma function.
`genfloat trunc ( genfloat x )`	Round to integral value using the round to zero rounding mode.

Table 3.56: Math functions which work on SYCL Host and device. They correspond to Table 6.7 of the OpenCL 1.2 specification [1] (Part IV)

In SYCL the implementation defined precision math functions are defined in the namespace *cl::sycl::native*. The functions that are available within this namespace are specified in tables 3.57 3.58.

Native Math Function (Part I)	Description
`genfloat cos (genfloat x)`	Compute cosine over an implementation-defined range. The maximum error is implementation-defined.
`genfloat divide (genfloat x, genfloat y)`	Compute x / y over an implementation-defined range. The maximum error is implementation-defined.
`genfloat exp (genfloat x)`	Compute the base- e exponential of x over an implementation-defined range. The maximum error is implementation-defined.
`genfloat exp2 (genfloat x)`	Compute the base- 2 exponential of x over an implementation-defined range. The maximum error is implementation-defined.
`genfloat exp10 (genfloat x)`	Compute the base- 10 exponential of x over an implementation-defined range. The maximum error is implementation-defined.
`genfloat log (genfloat x)`	Compute natural logarithm over an implementation defined range. The maximum error is implementationdefined.
`genfloat log2 (genfloat x)`	Compute a base 2 logarithm over an implementationdefined range. The maximum error is implementation-defined.
`genfloat log10 (genfloat x)`	Compute a base 10 logarithm over an implementationdefined range. The maximum error is implementation-defined.
`genfloat powr (genfloat x, genfloat y)`	Compute x to the power y, where x is ¿= 0. The range of x and y are implementation-defined. The maximum error is implementation-defined.
`genfloat recip (genfloat x)`	Compute reciprocal over an implementation-defined range. The maximum error is implementation-defined.

Table 3.57: Native functions which work on SYCL Host and device, are available in the *cl::sycl::native* namespace. They correspond to Table 6.9 of the OpenCL 1.2 specification [1] (Part I)

Native Math Function (Part II)	Description
`genfloat rsqrt (genfloat x)`	Compute inverse square root over an implementationdefined range. The maximum error is implementationdefined.
`genfloat sin ( genfloat x)`	Compute sine over an implementation-defined range. The maximum error is implementation-defined.
`genfloat sqrt ( genfloat x )`	Compute square root over an implementation-defined range. The maximum error is implementation-defined.
`genfloat tan ( genfloat x)`	Compute tangent over an implementation-defined range. The maximum error is implementation-defined.

Table 3.58: Native functions which work on SYCL Host and device, are available in the *cl::sycl::native* namespace. They correspond to Table 6.9 of the OpenCL 1.2 specification [1] (Part II)

3.9.4 Integer Functions

In SYCL the OpenCL integer math functions are available in the namespace *cl::sycl* on host and device as defined in the OpenCL 1.2 specification document [1, par. 6.12.3]. The built-in functions can take as input char, unsigned char, short, unsigned short, int, unsigned int, long unsigned long and their *vec* counterparts, for dimensions 2,3,4,8, and 16. On the host the vector types are going to be using the *vec* class and on an OpenCL device are going to be using the corresponding OpenCL vector types. The supported integer math functions are described in tables 3.59 3.60.

Integer Function (Part I)	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, 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.
`geninteger clamp (geninteger x, geninteger minval, geninteger maxval)`	Returns min(max(x, minval), maxval). Results are undefined if minval > maxval.

Table 3.59: Integer functions which work on SYCL Host and device, are available in the *cl::sycl* namespace. They correspond to Table 6.10 of the OpenCL 1.2 specification [1] (Part I)

Integer Function (Part II)	Description
`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, geninteger y)`	Returns y if $x < y$, otherwise it returns x.
`gentype min (genint x, genint y)` `genint min (genint x, sgenint y)`	Returns y if $y < x$, otherwise it returns x.
`genint mul_hi (genint x, genint y)`	Computes x * y and returns the high half of the product of x and y.
`genint rotate (genint v, genint 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.
`genint sub_sat (genint x, genint y)`	Returns $x - y$ and saturates the result.
`shortn upsample (charn hi, ucharn lo)`	$result[i] = ((short)hi[i] << 8)\|lo[i]$
`ushortn upsample (ucharn hi, ucharn lo)`	$result[i] = ((ushort)hi[i] << 8)\|lo[i]$
`intn upsample (shortn hi, ushortn lo)`	$result[i] = ((int)hi[i] << 16)\|lo[i]$
`uintn upsample (ushortn hi, ushortn lo)`	$result[i] = ((uint)hi[i] << 16)\|lo[i]$
`longn upsample (intn hi, uintn lo)`	$result[i] = ((long)hi[i] << 32)\|lo[i]$
`ulongn upsample (uintn hi, uintn lo)`	$result[i] = ((ulong)hi[i] << 32)\|lo[i]$
`genint popcount (genint x)`	Returns the number of non-zero bits in x.
`intn mad24 (intn x, intn y, intn z)` `uintn mad24 (uintn x, uintn y, uintn z)`	Multipy 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.
`intn mul24 ( intn x, intn y )` `uintn mul24 ( uintn x, uintn 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.

Table 3.60: Integer functions which work on SYCL Host and device, are available in the *cl::sycl* namespace. They correspond to Table 6.10 of the OpenCL 1.2 specification [1] (Part II)

3.9.5 Common Functions

In SYCL the OpenCL *common* functions are available in the namespace *cl::sycl* on host and device as defined in the OpenCL 1.2 specification document [1, par. 6.12.4].. Description is in table 3.61 3.62. The built-in functions can take as input float or optionally double and their *vec* counterparts, for dimensions 2,3,4,8, and 16. On the host the vector types are going to be using the *vec* class and on an OpenCL device are going to be using the corresponding OpenCL vector types.

Common Function (Part I)	Description

`genfloat clamp ( genfloat x, genfloat minval, genfloat maxval )`
`floatn clamp ( floatn x, float minval, float maxval )`
`doublen clamp ( doublen 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)`
`genfloatf max (genfloatf x, float y)`
`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 )`
`genfloatf min ( genfloatf x, float y )`
`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 )`
`genfloatf mix ( genfloatf x, genfloatf y, float a )`
`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.

`genfloat radians (genfloat degrees)`	Converts degrees to radians, i.e. $(\pi/180) * degrees$.

`genfloat step (genfloat edge, genfloat x)`
`genfloatf step (float edge, genfloatf x)`
`genfloatd step (double edge, genfloatd x)`
	Returns 0.0 if $x < edge$, otherwise it returns 1.0.

Table 3.61: Common functions which work on SYCL Host and device, are available in the *cl::sycl* namespace. They correspond to Table 6.12 of the OpenCL 1.2 specification [1] (Part I)

Common Function (Part II)	Description

`genfloat smoothstep (genfloat edge0, genfloat edge1, genfloat x)`
`genfloatf smoothstep (float edge0, float edge1, genfloatf x)`
`genfloatd smoothstep (double edge0, double edge1, genfloatd x)`
	Returns 0.0 if x <= *edge*0 and 1.0 if x >= *edge*1 and performs smooth Hermite interpolation between 0 and 1 when *edge*0 < x < *edge*1. This is useful in cases where you would want a threshold function with a smooth transition.
	This is equivalent to:
	`gentype t;`
	`t = clamp ((x <= edge0)/ (edge1 >= edge0), 0, 1);`
	`return t * t * (3 - 2 * t);`
	
	Results are undefined if *edge*0 >= *edge*1 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.

Table 3.62: Common functions which work on SYCL Host and device, are available in the *cl::sycl* namespace. They correspond to Table 6.12 of the OpenCL 1.2 specification [1]()Part II

3.9.6 Geometric Functions

In SYCL the OpenCL *geometrics* functions are available in the namespace *cl::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* counterparts, for dimensions 2,3,4,8, and 16. On the host the vector types are going to be using the *vec* class and on an OpenCL device are going to be using the corresponding OpenCL vector types. All of the geometric functions are using round-to-nearest-even rounding mode. Tables 3.63 3.64 contain all the definitions of supported geometric functions.

Geometric Function (Part I)	Description

`float4 cross (float4 p0, float4 p1)`
`float3 cross (float3 p0, float3 p1)`
`double4 cross (double4 p0, double4 p1)`
`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.

`float dot (floatn p0, floatn p1)` `double dot (doublen p0, doublen p1)`	Compute dot product.
`float distance (floatn p0, floatn p1)` `double distance (doublen p0, doublen p1)`	Returns the distance between p0 and p1. This is calculated as $length(p0 - p1)$.
`float length (floatn p)` `double length (doublen p)`	Return the length of vector p, i.e., $\sqrt{p.x^2 + p.y^2 + ...}$
`floatn normalize (floatn p)` `doublen normalize (doublen p)`	Returns a vector in the same direction as p but with a length of 1.

Table 3.63: Geometric functions which work on SYCL Host and device, are available in the *cl::sycl* namespace. They correspond to Table 6.13 of the OpenCL 1.2 specification [1] (Part I)

Geometric Function (Part II)	Description
`float fast_distance (floatn p0, floatn p1)`	Returns $fast_length(p0 - p1)$.
`float fast_length (floatn p)`	Returns the length of vector p computed as: `sqrt ((half)(p.x2 + p.y2 + ?.))`
`floatn fast_normalize (floatn p)`	Returns a vector in the same direction as p but with a length of 1. fast_normalize is computed as: `p * rsqrt((half)(p.x2 + p.y2 + ? ))` The result shall be within 8192 ulps error from the infinitely precise result of `if (all(p == 0.0f)) result = p; else result = p / sqrt (p.x2 + p.y2 + ... );` with the following exceptions: 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. 2. If the sum of squares is less than FLT_MIN then the implementation may return back p. 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.

Table 3.64: Geometric functions which work on SYCL Host and device, are available in the *cl::sycl* namespace. They correspond to Table 6.13 of the OpenCL 1.2 specification [1]

3.9.7 Relational Functions

In SYCL the OpenCL *relational* functions are available in the namespace *cl::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* counterparts, for dimensions 2,3,4,8, and 16. On the host the vector types are going to be using the *vec* class and on an OpenCL device are going to be using the corresponding OpenCL vector types. The relational operators are available in both host and device, these relational functions are provided in addition to the the operators and will return 0 if the conditional is *false* and 1 otherwise. The available built-in functions are described in tables 3.65 3.66

Relational Function (Part I)	Description

`int isequal (float x, float y)`
`intn isequal (floatn x, floatn y)`
`int isequal (double x, double y)`
`longn isequal (doublen x, doublen y)`
	Returns the component-wise compare of $x == y$.

`int isnotequal (float x, float y)`
`intn isnotequal (floatn x, floatn y)`
`int isnotequal (double x, double y)`
`longn isnotequal (doublen x, doublen y)`
	Returns the component-wise compare of $x! = y$.

`int isgreater (float x, float y)`
`intn isgreater (floatn x, floatn y)`
`int isgreater (double x, double y)`
`longn isgreater (doublen x, doublen y)`
	Returns the component-wise compare of $x > y$.

`int isgreaterequal (float x, float y)`
`intn isgreaterequal (floatn x, floatn y)`
`int isgreaterequal (double x, double y)`
`longn isgreaterequal (doublen x, doublen y)`
	Returns the component-wise compare of $x >= y$.

`int isless (float x, float y)`
`intn isless (floatn x, floatn y)`
`int isless (double x, double y)`
`longn isless (doublen x, doublen y)`
	Returns the component-wise compare of $x < y$.

`int islessequal (float x, float y)`
`intn islessequal (floatn x, floatn y)`
`int islessequal (double x, double y)`
`longn islessequal (doublen x, doublen y)`
	Returns the component-wise compare of $x <= y$.

`int islessgreater (float x, float y)`
`intn islessgreater (floatn x, floatn y)`
`int islessgreater (double x, double y)`
`longn islessgreater (doublen x, doublen y)`
	Returns the component-wise compare of $(x < y)\|\|(x > y)$.

Table 3.65: Relational functions which work on SYCL Host and device, are available in the *cl::sycl* namespace. They correspond to Table 6.14 of the OpenCL 1.2 specification [1] (Part I)

Relational Function (Part II)	Description

`int isfinite (float)`
`intn isfinite (floatn)`
`int isfinite (double)`
`longn isfinite (doublen)`
	Test for finite value.

`int isinf (float)`
`intn isinf (floatn)`
`int isinf (double)`
`longn isinf (doublen)`
	Test for infinity value (positive or negative) .

`int isnan (float)`
`intn isnan (floatn)`
`int isnan (double)`
`longn isnan (doublen)`
	Test for a NaN.

`int isnormal (float)`
`intn isnormal (floatn)`
`int isnormal (double)`
`longn isnormal (doublen)`
	Test for a normal value.

`int isordered (float x, float y)`
`intn isordered (floatn x, floatn y)`
`int isordered (double x, double y)`
`longn isordered (doublen x, doublen y)`
	Test if arguments are ordered. isordered() takes arguments x and y, and returns the result isequal(x, x) && isequal(y, y).

`int isunordered (float x, float y)`
`intn isunordered (floatn x, floatn y)`
`int isunordered (double x, double y)`
`longn isunordered (doublen x, doublen y)`
	Test if arguments are unordered. isunordered() takes arguments x and y, returning non-zero if x or y is NaN, and zero otherwise.

`int signbit (float)`
`intn signbit (floatn)`
`int signbit (double)`
`longn signbit (doublen)`
	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.
	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.

`int any (igentype 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.

`gentype select (gentype a, gentype b, igeninteger c)`
`gentype select (gentype a, gentype b, ugeninteger c)`
	For each component of a vector type:
	`result[i] = (MSB of c[i] is set)? b[i] : a[i].`
	For a scalar type:
	`result = c ? b : a.`
	`igeninteger` and `ugeninteger` must have the same number of elements and bits as `gentype`.

Table 3.66: Relational functions which work on SYCL Host and device, are available in the *cl::sycl* namespace. They correspond to Table 6.14 of the OpenCL 1.2 specification [1] (Part II)

3.9.8 Vector Data and Store Functions

The functionality from the OpenCL functions as defined in in the OpenCL 1.2 specification document [1, par. 6.12.7] is available in SYCL through the *vec* class in section 3.8.1.

3.9.9 Synchronization Functions

In SYCL the OpenCL *synchronization* functions are available through the item class 3.7.1.4, as they are applied to work-item for local or global address spaces. Please see 3.42.

3.9.10 printf function

The printf function is available for SYCL host and device side. On host it follows the Standard C Library headers and on the device side it follows the OpenCL 1.2 specification document [1, par. 6.12.13]. For the full description of the printf format string see: paragraph 6.12.13.2 of the OpenCL 1.2 specification document [1]

Function	Description
`int printf(constant char * restrict format,...)`	The printf built-in function writes output to an implementation-defined stream such as stdout under control of the string pointed to by format that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The printf function returns when the end of the format string is encountered. printf returns 0 if it was executed successfully and -1 otherwise.

Table 3.67: printf function definition

3.9.10.1 printf output synchronization

As described in paragraph 6.12.13.1 of the OpenCL 1.2 specification document [1] the synchronization rules for printf are the following.

- The output of all the printf() calls in a kernel invocation are flushed to an implementation-defined output stream, when the associated event of the kernel invocation reaches the state *completed*.
- If clFinish is called on an OpenCL queue, all pending output of any printf calls which were enqueued on that queue and have already completed are flushed to an implementation-defined output. In the case of SYCL queues, the output could be flushed to the implementation-defined output the later possible on the destruction of the SYCL queue.

- There are no guarantees that printf() output ordering follows the ordering of the data in each work-item, when printf() is executed per item in an nd_range.

4. SYCL support of non-core OpenCL features

OpenCL apart from *core* features that are supported in *every* platform, has *optional* features as well as *extensions* that are only supported in some platforms. The *optional* features, as described in the specification [1], and the OpenCL "khr" *extensions*, as described in the extension specification [2], are supported by the SYCL framework, but the ability to use them is completely dependent on the underlying OpenCL platforms. A SYCL implementation may support some vendor extensions in order to enable optimizations on certain platforms.

All OpenCL extensions are available through SYCL interoperability with OpenCL C, so all the extensions can be used through the OpenCL API as described in the extensions specification [2].

When running command groups on the host device, not all extensions are required to be available. The extensions available for the *host* are available to query in the same way as for SYCL devices, see Table 3.6.

4.1 Enable extensions in a SYCL kernel

In order to enable extensions in an OpenCL kernel the following compiler directive is used:

```
#pragma OPENCL EXTENSION <extension_name> : <behaviour>
```

The keyword *extension_name* can be:

- **`all`**, which refers to all the extensions available on a platform
- an **`extension name`** from the available extensions on a platform.

They keyword *behaviour* can be:

- **`enable`**: it will enable the extension specified by *extension_name* if it is available on the platform or otherwise it triggers a compiler warning. If *all* is specified in the *extension_name* then it will enable all extensions available.
- **`disable`**: it will disable all or any extension provided in the *extension_name*.

4.2 Half Precision Floating-Point

The half precision floating-point data scalar and vector types are supported in the SYCL system, if they are supported on the OpenCL platform in use as well.

The extension name is **`cl_khr_fp16`** and it needs to be used in order to enable the usage of the half data type on the device.

Extension	Support using OpenCL/SYCL API	Support using SYCL API
cl_khr_int64_base_atomics	Yes	Yes
cl_khr_int64_base_atomics	Yes	Yes
cl_khr_fp16	Yes	Yes
cl_khr_3d_image_writes	Yes	Yes
cl_khr_gl_sharing	Yes	Yes
cl_apple_gl_sharing	Yes	Yes
cl_khr_d3d10_sharing	Yes	No
cl_khr_d3d11_sharing	Yes	No
cl_khr_dx9_media_sharing	Yes	No

Table 4.1: SYCL support for OpenCL 1.2 API extensions. This table summarizes the levels of SYCL support to the API extensions for OpenCL 1.2. These extensions can be supported and/or using OpenCL/SYCL interoperability or by the extended SYCL API calls. This only applies for using them in the framework and only for devices that are supporting these extensions

The half type class, along with any OpenCL macros and definitions, is defined in the namespace `cl::sycl` as `half`. The vector type of half is supported sizes 2, 3, 4, 8 and 16 using the SYCL vectors (§ 3.8.1) along with all the methods supported for vectors.

The conversion rules follows the same rules as in the OpenCL 1.2 extensions specification [2, par. 9.5.1].

The math, common, geometric and relational functions can take `cl::SYCL ::opencl::half` as a type as they are defined in [2, par. 9.5.2, 9.5.3, 9.5.4, 9.5.5]. The valid type for the functions defined for half is described by the generic type name *genhalf* is described in table 4.2.

Generic type name	Description
genhalf	cl::sycl::half, cl::sycl::half2, cl ::sycl::half3, cl::sycl::half4, cl:: sycl::half8, cl::sycl::half16

Table 4.2: Generic type name description, which serves as a description for all valid types of parameters to kernel functions. [1]

4.3 Writing to 3D image memory objects

The `image` and `accessor` classes in SYCL support methods 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*.

4.4 Interoperability with OpenGL

OpenCL has a standard extension that allows interoperability with OpenGL objects. The features described in this section are only defined within SYCL if the underlying OpenCL implementation supports the OpenCL/OpenGL interoperability extension (`cl_khr_gl_sharing`).

Math function	Description
`genhalf cos (genhalf x)`	Compute cosine. x must be in the range -216 +216.
`genhalf divide (genhalf x, genhalf y)`	Compute x / y.
`genhalf exp (genhalf x)`	Compute the base- e exponential of x.
`genhalf exp2 (genhalf x)`	Compute the base- 2 exponential of x.
`genhalf exp10 (genhalf x)`	Compute the base- 10 exponential of x.
`genhalf log (genhalf x)`	Compute natural logarithm.
`genhalf log2 (genhalf x)`	Compute a base 2 logarithm.
`genhalf log10 (genhalf x)`	Compute a base 10 logarithm.
`genhalf powr (genhalf x,genhalf y)`	Compute x to the power y, where $x >= 0$.
`genhalf recip (genhalf x)`	Compute reciprocal.
`genhalf rsqrt (genhalf x)`	Compute inverse square root.
`genhalf sin (genhalf x)`	Compute sine. x must be in the range -216 +216.
`genhalf sqrt (genhalf x)`	Compute square root.
`genhalf tan (genhalf x)`	Compute tangent. x must be in the range -216 +216.

Table 4.3: Math functions which work on SYCL Host and device. If the half type is given as a parameter then the allowed ULP is less than 8192. They correspond to Table 6.9 of the OpenCL 1.2 specification [1]

4.4.1 OpenCL/OpenGL extensions to the context class

If the `cl_khr_gl_sharing` extension is present then the developer can create an OpenCL context from an OpenGL context by providing the corresponding attribute names and values to *properties* for the devices chosen by device selector. Table 3.10 has the additions shown on Table 4.5.

cl_context_properties flag	Description
`CL_GL_CONTEXT_KHR`	OpenGL context handle (default: 0)
`CL_EGL_DISPLAY_KHR`	CGL share group handle (default: 0)
`CL_GLX_DISPLAY_KHR`	EGLDisplay handle (default: `EGL_NO_DISPLAY`)
`CL_WGL_HDC_KHR`	X handle (default: None)
`CL_CGL_SHAREGROUP_KHR`	HDC handle (default: 0)

Table 4.4: Additional optional properties for creating context for SYCL/OpenGL sharing.

The SYCL extension for creating OpenCL context from an OpenGL context is based on the OpenCL extension specification and all the capabilities and restrictions are based on it and developers and implementers are advised to refer to [2, sec. 9.6].

Methods	Description
`device get_gl_current_device ()`	Returns the OpenGL enabled device in the current context.
`vector_class<device> get_gl_context_devices ()`	Returns the OpenGL supported devices in this context.

Table 4.5: Additional methods of the context class which are defined for the OpenGL extensions. If the OpenGL extensions are not available then their behavior is implementation defined.

4.4.2 Sharing OpenCL/OpenGL memory objects

It is possible to share objects between OpenCL and OpenGL, if the corresponding platform extensions for these are available on available platforms. OpenCL memory objects based on OpenGL objects can only be created only if the OpenCL context is created from an OpenGL share group object or context. As the latter are OS specific, the OpenCL extensions are platform specific as well. In MacOS X the extension `cl_apple_gl_sharing` needs to be available for this functionality. If it is Windows/Linux/Unix, then the extension `cl_khr_gl_sharing` needs to be available. All the OpenGL objects within the shared group used for the creation of the context can be used apart from the default OpenGL objects.

Any of the buffers or images created through SYCL using the shared group objects for OpenGL are invalid if the corresponding OpenGL context is destroyed through usage of the OpenGL API. If buffers or images are used after the destruction of the corresponding OpenGL context then the behaviour of the system is undefined.

4.4.2.1 OpenCL/OpenGL extensions to SYCL buffer

A SYCL *buffer* can be created from an OpenGL buffer object but the lifetime of the SYCL buffer is bound to the lifetime of the OpenCL context given in order to create the buffer. The GL buffer has to be created a priori in using the OpenGL API, although it doesn't need to be initialized. If the OpenGL buffer object is destroyed or otherwise manipulated through the OpenGL API, before its usage through SYCL is completed, then the behaviour is undefined.

The functionality of the buffer and the accessor class is retained as for any other OpenCL buffer defined in this system.

Constructor	Description
`template <typename T, int dimensions = 1>` `buffer(context &cl_gl_context, GLuint gl_buffer_obj)`	Constructs a buffer from a OpenCL//OpenGL interop context and a gl_buffer object.

Table 4.6: Additional optional constructor of the buffer class which have the defined behavior when the OpenGL extensions are available on device, otherwise their behavior is implementation-defined.

Method	Description
`cl_gl_object_type get_gl_info ( GLuint gl_buffer_obj )`	Returns the cl_gl_object_type of the underlying OpenGL buffer.

Table 4.7: Additional optional methods of the buffer class which have the defined behavior when the OpenGL extensions are available on device, otherwise their behavior is implementation-defined.

4.4.2.2 OpenCL/OpenGL extensions to SYCL image

A SYCL *image* can be created from an OpenGL buffer, from an OpenGL texture or from an OpenGL renderbuffer. However, the lifetime of the SYCL image is bound to the lifetime of the OpenCL context given in order to create the image and the OpenGL object's lifetime. The GL buffer, texture or renderbuffer has to be created a priori via the OpenGL API, although it doesn't need to be initialized. If the OpenGL object is destroyed or otherwise manipulated through the OpenGL API before its usage through SYCL is completed, then the behaviour is undefined.

Constructor (Part I)	Description
`template<int dimensions = 1> image(context &cl_gl_context, GLuint gl_buffer_obj)`	Creates an 1-D Image from an OpenGL buffer object.
`template<int dimensions = 2 image( context &cl_gl_context, GLuint gl_renderbuffer_obj)`	Create a 2-D image from an OpenGL renderbuffer object.
`template<int dimensions = 1 image(context &cl_gl_context, GLenum texture_target, GLuint texture, GLint miplevel`	Creates a 1-D image from an OpenGL texture object with given texture_target and mipmap level. The texture_target can be one of the following: • GL_TEXTURE_1D • GL_TEXTURE_1D_ARRAY • GL_TEXTURE_BUFFER
`template<int dimensions = 2 image(context &cl_gl_context, GLenum texture_target, GLuint texture, GLint miplevel`	Creates a 2-D image from an OpenGL texture object with given texture_target and mipmap level. The texture_target can be one of the following: • GL_TEXTURE_2D • GL_TEXTURE_2D_ARRAY • GL_TEXTURE_CUBE_MAP_POSITIVE_X • GL_TEXTURE_CUBE_MAP_POSITIVE_Y • GL_TEXTURE_CUBE_MAP_POSITIVE_Z • GL_TEXTURE_CUBE_MAP_NEGATIVE_X • GL_TEXTURE_CUBE_MAP_NEGATIVE_Y • GL_TEXTURE_CUBE_MAP_NEGATIVE_Z • GL_TEXTURE_RECTANGLE

Table 4.8: Additional optional *image* class constructors. (Part I)

Constructor (Part II)	Description

`template<int dimensions = 3`
`image(context &cl_gl_context, GLenum texture_target,`
`GLuint texture, GLint miplevel`
	Creates a 3-D image from an OpenGL texture object with given texture_target and mipmap level. The texture_target can be one of the following:
	• GL_TEXTURE_3D

Table 4.9: Additional optional *image* class constructors. (Part II)

Method	Description
`GLenum get_gl_texture_target ()`	Returns the OpenGL texture_target corresponding to the underlying texture which the context was created with
`GLint get_gl_mipmap_level ()`	Returns the mipmap level of the underlying texture.

Table 4.10: Additional optional *image* class method.

The *texture* provided has to be an OpenGL texture created through the OpenGL API and has to be a valid 1D, 2D, 3D texture or 1D array, 2D array texture or a cubemap, rectangle or buffer texture object. The format and the dimensions provided for the miplevel of the texture are used to create the OpenCL image object. The format of the OpenGL texture or renderbuffer object needs to match the format of the OpenCL image format. The compatible formats are specified in Table 9.4 of the OpenCL 1.2 extensions document [2, par. 9.7.3.1] and are also included in Table 4.11. If the texture or renderbuffer has a different format than the ones specified in 4.11, it is not guaranteed that the image created will be mapped to the the original texture.

OpenGL internal format	Corresponding OpenCL image format (channel order, channel data type)
GL_RGBA8	CL_RGBA, CL_UNORM_INT8, CL_BGRA, CL_UNORM_INT8
GL_RGBA, GL_UNSIGNED_INT_8_8_8_8_REV	CL_RGBA, CL_UNORM_INT8
GL_BGRA, GL_UNSIGNED_INT_8_8_8_8_REV	CL_BGRA, CL_UNORM_INT8
GL_RGBA16	CL_RGBA, CL_UNORM_INT16
GL_RGBA8I, GL_RGBA8I_EXT	CL_RGBA, CL_SIGNED_INT8
GL_RGBA16I, GL_RGBA16I_EXT	CL_RGBA, CL_SIGNED_INT16
GL_RGBA32I, GL_RGBA32I_EXT	CL_RGBA, CL_SIGNED_INT32
GL_RGBA8UI, GL_RGBA8UI_EXT	CL_RGBA, CL_UNSIGNED_INT8
GL_RGBA16UI, GL_RGBA16UI_EXT	CL_RGBA, CL_UNSIGNED_INT16
GL_RGBA32UI, GL_RGBA32UI_EXT	CL_RGBA, CL_UNSIGNED_INT32
GL_RGBA16F, GL_RGBA16F_ARB	CL_RGBA, CL_HALF_FLOAT
GL_RGBA32F, GL_RGBA32F_ARB	CL_RGBA, CL_FLOAT

Table 4.11: Mapping of GL internal format to CL image format (reference: [2, table 9.4])

Enumerator name:	`access::target`
Values:	`cl_gl_buffer`: access `buffer` which is created from an OpenGL buffer `cl_gl_image`: access an `image` or `image_array` that is created from an OpenGL shared object

Table 4.12: Enumerator description for `access::target`

4.4.2.3 OpenCL/OpenGL extensions to SYCL accessors

In order for SYCL to support the OpenCL/OpenGL interoperability, the classes for buffers and images have to be extended, in order for OpenCL objects to be created from OpenGL objects. This extension, apart from restrictions on the creation and the life-time of the OpenCL objects, also requires that before the usage of any of these objects in an OpenCL *command_queue* an `acquire` command has to be enqueued first. In SYCL, the `command_group` and the `accessor` classes make sure that the data are made available on a device. For this extension the `command_group` will acquire any of the accessors whose targets are declared as interoperability targets.

The required extension for the accessor class are shown on Table 4.12.

The SYCL system is responsible for synchronizing the OpenCL and OpenGL objects in use inside a `command_group` when the SYCL API is used and given that all the accessors for the buffers and images are marked as the interoperability targets.

4.4.2.4 OpenCL/OpenGL extensions to SYCL events

In the case where the extension `cl_khr_gl_event` is available on a platform, the functionality for creating synchronizing OpenCL events with OpenGL events is available.

A SYCL event can be constructed from an OpenGL sync object with the extensions to the `event` class shown on Table 4.13.

Constructor	Description
`event ( context cl_gl_context, GL_sync sync_obj`	Creates an event which enables waiting on events to also include synchronization with OpenGL objects that are shared using the OpenCL/OpenGL context

Table 4.13: Additional optional class constructors for `event` class.

Method	Description
`GL_sync get_gl_info ()`	Returns GL_sync object.

Table 4.14: Additional optional class method for `event` class.

The specification of the underlying OpenCL/OpenGL interoperability system for synchronizing OpenCL event with OpenGL sync objects is in the OpenCL extensions specification [2, sec. 9.8].

4.4.2.5 Extension for depth and depth-stencil images

The extension `cl_khr_depth_images` adds support for depth images and the extension `cl_khr_gl_depth_images` allows sharing between OpenCL depth images and OpenGL depth or depth-stencil textures. The SYCL system doesn't add any additional functionality towards this extension and follows the OpenCL 1.2 Specification [2, sec. 9.12] for depth and depth-Stencil images extension. All the image class constructors and methods of the SYCL API as described in Table 3.20 and 3.21 on page 65 are extended to enable the use of the same API when this extension is present. The API is able to support the type `image2d_depth_t` and `image2d_array_depth_t`. The OpenCL C API defined in [2, sec. 9.12] can be used as well with all the rules that apply for SYCL/OpenCL C interoperability.

5. SYCL Device Compiler

This section specifies the requirements of the SYCL device compiler. Most features described in this section relate to underlying OpenCL 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 *shared source device compiler* and a *single-source device compiler*.

A SYCL shared source 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, such as a macro definition, to the host compiler 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 if an implementer so chooses.

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 parses and outputs device code for kernels, but does not specify the host compilation.

5.2 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` or `single_task`). The kernel invoke functions are templated by their kernel parameter, which is a function object (either a functor or a lambda). The code inside the function object that is invoked as a kernel is called the "kernel function". 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 is a function that matches the C++11 specification, plus any extensions to the C++11 specification defined in this SYCL specification.

5.3 Supported data types

5.3.1 Built-in scalar data types

In a SYCL device compiler, the standard C++ scalar types, including `int`, `short` and `long` need to be set so that the device definitions of those types match the host definitions of those types. A device compiler may have this configured, 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.

To ensure that pointer types and `size_t` use the same amount of storage on host and device when inside structures, SYCL also requires that device compilers use the host sizes for pointers or `size_t`. Devices may be using a smaller size internally for pointers and `size_t`, but this should not impact the programming model for users.

Scalar datatypes for a SYCL device compiler:

- `bool`: a boolean value;
- `char`: a signed 8-bit integer;
- `unsigned char`: an unsigned 8-bit integer;
- `short int`: a signed integer whose size must match the definition on the host;
- `unsigned short int`: an unsigned integer whose size must match the definition on the host;
- `int`: a signed integer whose size must match the definition on the host;

- `unsigned int`: an unsigned integer whose size must match the definition on the host;
- `long int`: a signed integer whose size must match the definition on the host;
- `unsigned long int`; an unsigned integer whose size must match the definition on the host;
- `float`: a 32-bit IEEE 754 floating-point value;
- `double`: a 64-bit IEEE 754 floating-point value;
- `half`: a 16-bit IEEE 754-2008 half-precision floating-point value;
- `size_t`: the unsigned integer type of the result of the `sizeof` operator on host.

For users who want to use the OpenCL built-in scalar types and those who want to interop with OpenCL C, the types are defined in the standard OpenCL header files, such as `cl_int` [1, Sec 6.1]. For example, if an OpenCL C kernel has the "short" datatype in a function definition, when declaring that function's prototype in SYCL, a user should use the `cl_short` datatype from the OpenCL header file.

5.4 Preprocessor directives and macros

The standard C++ preprocessing directives and macros are supported.

- `CL_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 `CL_SYCL_LANGUAGE_VERSION` substitute the integer 120;
- `__FAST_RELAXED_MATH__` is used to determine of the `-cl-fast-relaxed-math` optimization option is specified in the build options given to the SYCL device compiler. This is an integer constant of 1 if the option is specified and undefined otherwise;
- `__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_TARGET_SPIR__` is defined to 1 if the source file is being compiled with a SYCL compiler which is producing OpenCL SPIR binary.

5.4.1 Attributes

The `attribute` syntax defined in the OpenCL C specification is supported in SYCL.

The `vec_type_hint`, `work_group_size_hint` and `reqd_work_group_size` kernel attributes in OpenCL C apply to kernel functions, but this is not syntactically possible in SYCL. In SYCL, these attributes are legal on device functions and their specification is propagated down to any caller of those device functions, such that the kernel attributes are the sum of all the kernel attributes of all device functions called. If there are any conflicts between different kernel attributes, then the behaviour is undefined.

5.5 Address-space deduction

In SYCL, there are several different types of pointer, or reference:

- Accessors 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 OpenCL address space based on their access mode.
- Explicit pointer classes (e.g. `global_ptr`) contain an OpenCL address space. This allows the compiler to determine whether the pointer references global, local, constant or private memory.
- C++ pointer and reference types (e.g. `int*`) are allowed within SYCL kernels. They can be constructed from the address of local variables, from explicit pointer classes, or from accessors. In all cases, a SYCL device compiler will need to auto-deduce the address space.

Inside kernels, conversions between accessors to buffers, explicit pointer classes and C++ pointers are allowed as long as they reference the same datatype and have compatible qualifiers and address spaces.

If a kernel function or device function contains a pointer or reference type, then address-space deduction must be attempted using the following rules:

- If a an explicit pointer class is converted into a C++ pointer value, then the C++ pointer value will have the address space of 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.
- 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. This default assumption is expected to change to be the `generic` address space for OpenCL versions that support the generic address space.

It is illegal to assign a pointer value of one address space to a pointer variable of a different address space, unless the variable has already been deduced to have the same address space.

A. Glossary

The purpose of this glossary is to define the key concepts involved in specifying OpenCL SYCL. This section includes definitions of terminology used throughout the specification document.

Accessor: An accessor is an interface which allows a kernel function to access data maintained by a buffer.

Application scope: The application scope is the normal C++ source code in the application, outside of command groups and kernels.

Buffer: A buffer is an interface which maintains an area of memory which is to be accessed by a kernel function. It represents storage of data only, with access to that data achieved via accessors. The storage is managed by the SYCL runtime, but may involve OpenCL buffers.

Barrier: SYCL barriers are the same as OpenCL barriers. In SYCL, OpenCL's command-queue-barriers are created automatically on demand by the SYCL runtime to ensure kernels are executed in a semantically-correct order across multiple OpenCL contexts and queues. OpenCL's work-group barriers are available as an intrinsic function (same as in OpenCL) or generated automatically by SYCL's hierarchical parallel-for loops.

Command Group: All of the OpenCL commands, memory object creation, copying, mapping and synchronization operations to correctly execute a kernel on a device are collected together in SYCL and called a command group. Command groups executed in different threads are added to queues atomically, so it is safe to issue command-groups operating on shared queues, buffers and images.

Command Group Scope: A command group is created inside a command group functor or lambda. The user's code inside the functor or lambda is called command group scope.

Command Queue: SYCL's command queues are either a simple wrapper for an OpenCL command queue, or a SYCL-specific host command queue, which executes SYCL kernels on the host.

Constant Memory: " 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." As defined in [1, p.15]

Device: SYCL's devices are the same as OpenCL devices.

Device Compiler: A SYCL device compiler is a compiler that is capable of taking in C++ source code containing SYCL kernels and outputting a binary object suitable for executing on an OpenCL device.

Functor: Functors are a concept from C++. An alternative name for functions in C++ is "function object". A functor is a C++ class with an **operator()** method that enables the object to be executed in a way that looks like a function call, but where the object itself is also passed in as a parameter.

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 an N-dimensional value that starts at (0, 0, ...0).

Global Memory: As in OpenCL, global memory is a memory region accessible to all work-items executing in a context. Buffers are mapped or copied into global memory for individual contexts inside the SYCL runtime in order to enable accessors within command groups to give access to buffers from kernels.

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.

Group Range: A group range is the range specifying the size of the work group.

Host: As in OpenCL, the host is the system that executes the SYCL API and the rest of the application.

Host pointer: A pointer to memory that is in the virtual address space on the host.

ID: An id is a one, two or three dimensional vector of integers. There are several different types of ID in SYCL: global ID, local ID, group ID. These different IDs are used to define work items

Image: Images in SYCL, like buffers, are abstractions of the OpenCL equivalent. As in OpenCL, an image stores a two- or three-dimensional structured array. The SYCL runtime will map or copy images to OpenCL images in OpenCL contexts in order to semantically correctly execute kernels in different OpenCL contexts. Images are also accessible on the host via the various SYCL accessors available.

Implementation defined: Behavior that is explicitly allowed to vary between conforming implementations of SYCL. A SYCL implementer is required to document the implementation defined behavior.

Item ID: An item id is an interface used to retrieve the global id, group id and local id of a work item.

Kernel: A SYCL kernel is a C++ functor or lambda function that is compiled to execute on a device. There are several ways to define SYCL kernels defined in the SYCL specification. It is also possible in SYCL to use OpenCL kernels as specified in the OpenCL specification. Kernels can execute on either an OpenCL device or on the host.

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.

Kernel Scope: The scope inside the kernel functor or lambda is called kernel scope. Also, 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.

Local ID: A local id is an id which specifies a work items location within a group range.

Local Memory: As in OpenCL, local memory is a memory region associated with a work-group and accessible only by work-items in that work-group.

NDRange: An NDRange consists of two vectors of integers of one, two or three-dimensions that define the total number of work items to execute as well as the size of the work groups that the work items are to be executed within.

Platform: A platform in SYCL is an OpenCL platform as defined in the OpenCL specification.

Private Memory: As in OpenCL, private memory is a region of memory private to a work-item. Variables defined in one work-items private memory are not visible to another work-item.

Program Object: A program object in SYCL is an OpenCL program object encapsulated in A SYCL class. It

contains OpenCL kernels and functions compiled to execute on OpenCL devices. A program object can be generated from SYCL C++ kernels by the SYCL runtime, or obtained from an OpenCL implementation.

Shared Source Build System: A shared source build system means that a single source file passed through both a host compiler and one or more device compilers. This enables multiple devices, instruction sets and binary formats to be produced from the same source code and integrated into the same piece of software.

SYCL Device: A SYCL device is either an OpenCL device wrapped in a SYCL device object, or it is the host. Therefore SYCL devices are either OpenCL devices or an abstraction for executing SYCL kernels on the host.

SYCL Runtime: A SYCL runtime is an implementation of the SYCL runtime specification. The SYCL runtime manages the different OpenCL platforms, devices, contexts as well as the mapping or copying of data between host and OpenCL contexts to enable semantically correct execution of SYCL kernels.

Work-Group: A work group is an OpenCL work group, defined in OpenCL as a collection of related work-items that execute on a single compute unit. The work-items in the group execute the same kernel and share local memory and work-group barriers.

Work-Item: A work item is an OpenCL work item, defined in OpenCL as one of a collection of parallel executions of a kernel invoked on a device by a command. A work-item is execute by one or more processing elements as part of a work-group executing on a compute unit. A work-item is distinguished from other executions within the collection by its global ID and local ID.

References

[1] Khronos OpenCL Working Group, *The OpenCL Specification, version 1.2.19*, 2012. [Online]. Available: https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf

[2] ——, *The OpenCL Extension Specification, version 1.2.19*, 2012. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.2-extensions.pdf