SYCL^{TM} Provisional Specification

SYCL integrates OpenCL devices with modern C++

Version 1.2

Revision Date: 2014-03-09

Khronos OpenCL Working Group — HLM subgroup

Editor: Lee Howes

Contents

List of Tables

Acknowledgements

Editor: Lee Howes, Qualcomm

Contributors:

- Shivani Gupta, Adobe
- David Neto, Altera
- Ronan Keryell, AMD
- Brian Sumner, AMD
- Balázs Keszthelyi, Broadcom
- Gordon Brown, Codeplay
- Paul Keir, 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 complicated and ugly passing of string specializations to the device compiler.

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 advantage 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 specialized low-level OpenCL features that only a SYCL device compiler knows how to consume are not used, a standard

C++ compiler can compile the SYCL programs and they will run correctly. Any use of specialized low-level features can be masked using the C pre-processor in the same way that compiler-specific intrinsics may be hidden to ensure portability between different host compilers.

SYCL retains the execution model, runtime feature set and device capabilities 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 assumes that there is an OpenCL API and one or more OpenCL devices provided in the system that a SYCL program runs on. The terminology used for SYCL inherits that of OpenCL with some SYCL-specific additions. 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 of SYCL is 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 the same as the core 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** made up of OpenCL **commands** from the host to execute computations on the processing elements within a device. The processing elements within a compute unit execute a single stream of instructions as SIMD units (which execute in lockstep with a single stream of instructions), as SPMD units (where each PE maintains its own program counter) or as some combination of the two.

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 off-line, 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`.

SYCL for OpenCL is designed to be backwards compatible, so a SYCL device compiler is not required to support more than a single language version to be considered conformant.

2.3 SYCL Execution Model

As in OpenCL, execution of a SYCL program occurs in two parts: *kernels* that execute on 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.

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 syntactic structures 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 OpenCL is called an NDRange. An NDRange is an N-dimensional index space, where N is one, two or three. In SYCL, the NDRange 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 nd_range<N> and a global offset, represented via a value of type id<N>. The types nd_range<N> and id<N> are each *N*-dimensional vectors of integers. The iteration space defined via an nd_range<N> is an *N*-dimensional index space starting at the NDRange'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 NDRange is identified by a value of type item<N>. The type item<N> consists of a global ID and a local ID, both of type item<N>. The global ID components are values in the range from the NDRange's global offset to the global offset plus the NDRange's global range minus one. Work-groups are assigned IDs using a similar approach to that used for work-item global IDs. The class size<*N*> which is an abstraction of an array of length *N* defines the number of work-groups in each dimension. 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. This combination of work-group ID and local ID is stored in the class item<N> from which a global ID, local ID and work-group ID can be obtained.

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.

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).

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 currently using 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 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. Conceptually, private memory is an extension of a work-item's register set.

The application running on the host uses SYCL buffer objects using the `cl::sycl::buffer` class to create memory objects in global memory. 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 OpenCL memory objects to enable the same buffer or image to be shared between multiple devices in different platforms. It is the responsibility of the SYCL implementation to ensure that a `cl::sycl::buffer` or image object shared between multiple OpenCL devices is enqueued to SYCL queues with the correct synchronization and copy commands to ensure that the SYCL ordering semantics are preserved.

For a kernel to access global memory on a device, the user must create a `cl::sycl::accessor` object which defines the type of access the kernel requires. The `cl::sycl::accessor` object defines whether the access is via global pointer access, constant pointer access or image sampler access, as well as whether the access is read-only, write-only or read-write. It is not possible to pass a global pointer as a parameter to a kernel from the host. However, it is possible on the device to obtain the global or constant pointer value of an accessor once it has been passed as a parameter to the kernel.

For a kernel to access local memory on a device, the user can either create a dynamically-sized local accessor object to the kernel as a parameter.

2.4.1 Memory consistency

OpenCL uses a relaxed 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. Local memory is consistent across work-items in a single work-group at a work-group barrier. Global memory is consistent across work-items in a single work-group at a work-group barrier, but there are no guarantees of memory consistency between different work-groups executing a kernel.

Memory consistency for `cl::sycl::buffer` and `cl::sycl::image` objects shared between enqueued commands is enforced at a synchronization point derived from completion of an enqueued command.

2.5 The SYCL programming model

A SYCL program is written in standard C++. Developers can also choose to use OpenCL-specific C++ extensions. The new C++11 features are optional in SYCL, but are commonly used and improve readability of code so most samples in this specification will use C++11 features.

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 normal 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. Because SYCL programs target heterogeneous systems, the kernels may be compiled for multiple different processor cores.

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 defined with 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 Kernels that are not launched in parallel

Simple non-parallel kernels are enqueued with the `cl::sycl::single_task` function. This function enqueues a kernel that takes no `cl::sycl::item_id` parameter and only executes once. It is logically equivalent to executing a kernel on a compute unit with a work-group containing a single work-item. Such kernels may be enqueued on multiple queues and multiple devices and may be executed, like any other OpenCL entity, as parallel tasks.

2.5.2 Basic data parallel kernels

Basic data-parallel kernels execute the code of the kernel across an entire NDRange with each execution being instantiated with a different "work-item id" parameter. Basic data parallel kernels are enqueued with the `cl↩ ::sycl::parallel_for` function supplied with a `cl::sycl::range` parameter to define the range. The execution of a basic data-parallel kernel is divided into work-groups whose size is chosen by the SYCL runtime. Barrier synchronization can not be used within these work-groups.

2.5.3 Work-group data parallel kernels

Data parallel kernels can also execute in a mode where the set of work-items is divided into user-defined work-groups. The user specifies the range and work-group size as parameters to the `cl::sycl::parallel_for↩` function with a `cl::sycl::ndrange` parameter. In this mode of execution, kernels execute over the NDRange 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 using `cl::sycl::barrier` function calls. All work-groups in a given `parallel_for` will be the same size and the global size defined in the NDRange must be a multiple of the work-group size in each dimension.

2.5.4 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::ndrange` 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 shareable among the work items in the work group, whereas all variables declared inside the `parallel_for_workitem` scope are not. All `parallel_for_workitem` calls within a given `parallel_for_workgroup` execution must have the same dimensions.

2.5.5 Synchronization

In OpenCL, synchronization is limited to a single OpenCL *context* or, at a finer granularity, within a single work-group. In SYCL, synchronization can similarly be either global, or 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 implementation.

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.

The synchronization operations in a SYCL application are:

- The destructor 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*.

- 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.

- The SYCL scheduler internally ensures that any command groups added to queues have the correct barrier packets added to those queues to ensure correct operation. Adding command groups to queues never blocks. Instead any required synchronization is added to the queue.

- 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.

2.5.6 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 using namespace cl::sycl;
 4
 5 int main ()
 6 {
 7     int result; // this is where we will write our result
 8
 9     {  // by sticking 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 work on
13         queue myQueue;
14
15          // wrap our result variable in a buffer
16         buffer<int> resultBuf (&result, 1);
17
18          // create some  c o m m a n d s  for our  q u e u e
19         command_group (myQueue, [&] ()
20         {
21             // request access to our buffer
22             auto writeResult = resultBuf.get_access<access::write> ();
23
24             // enqueue a single, simple task
25             single_task(kernel_lambda<class simple_test>([=] ()
26             {
27                 writeResult [0] = 1234;
28             }))
29         }); // end of our commands for this queue
30
31     } // end scope, so we wait for the queue to complete
32
33     printf ("Result = %d\n", result);
34
35     return result;
36 }
```

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 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 defined as a scoped block of code that is wrapped by one of a set of *task APIs*. In this example on line 25 we wrap a lambda function around the function body and annotate that with the `single_task` class which constructs a single instance of the lambda which will be executed with a single flow of execution.

In between the lambda function and the `single_task` object declared on line 25 is a *kernel_lambda* object. This is a template function that is parametrized with a type name. This type name is used to name the kernel function and to provide a concrete mapping between the host compiler and device compiler in a split compilation flow.

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*. We construct the 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 26. All *buffers* must be constructed in the *application scope*, whereas all *accessors* must be constructed in the *command group scope*.

2.5.7 Error handling

The default error reporting mechanism in SYCL is via C++ exceptions. This is the simplest form of error handling for the user. If a queue generates exceptions asynchronously, during execution, then exceptions are stored in a list and (optionally) passed to a user-defined asynchronous exception-handling functor once the queue is destroyed, or when the user specifically requests asynchronous exceptions to be thrown.

Some users may not want to use C++ exception handling for error reporting. All SYCL functions that can cause errors (including constructors) have an optional reference `cl::sycl::error_handler` parameter and the functions with this parameter report errors via the user-implemented error class and not via exceptions.

If an SYCL program is compiled with exceptions disabled in the compiler, or with a predefined macro (`SYCL_NO_EXCEPTIONS`) then SYCL functions that could throw exceptions will not be available to the user.

2.5.8 Scheduling of kernels and data movement

A command group will include accessor objects that specify what data the command group reads and writes. When enqueuing a command group, synchronization operations will be entered into the queue 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/write dependencies are enforced.

If a `command_group` fails to be enqueued correctly into the specified OpenCL queue, then execution will fall back to host. This fall-back may require extra data copying to occur to ensure that data is correctly available on the host for processing. This behaviour is the default as it can be critical for the application to always execute the kernels. However, it can be disabled by the developer.

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 synchronization 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 two 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 two set of accessors.

2. Accessors can be created to operate on *sub-buffers*, which is a subsection of a buffer. 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 contains *only* write access to a memory object, or sub-buffer, then the scheduler does not need to preserve the previous contents of the memory object or sub-buffer when scheduling the command group. Therefore, if a user wishes to only modify a part of a memory object and preserve the rest of the memory object, they must use a read/write accessor, or an accessor to just the sub-buffer they wish to modify.

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 or sub-buffer, must first copy the data to be read onto the device. 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 also provides *host accessors*. These accessors give temporary access to data in buffers on the host, outside command groups. These are the only kinds of accessors that can be created outside command groups. Creation of host accessors is blocking: all command groups that read or write data in the buffer or image that the host accessor targets must have completed. 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.

If a user creates an SYCL buffer, image or accessor from an OpenCL object, then the SYCL runtime will correctly manage synchronization if all access to the object is performed in host access scope following the same correctness rules as for host pointers. SYCL provides operations that create accessors from OpenCL buffers (`cl_mem` objects) and events to enable users to write their own synchronization. If custom synchronization is used the user is responsible for synchronization between the user's access of that data and SYCLs access of that data.

2.5.9 Managing object lifetimes

SYCL does not initialize any OpenCL features until an `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 an `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 will be reference-counted and 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 within SYCL implementations, apart from the possibility of caching.

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. If host memory is attached at the point of construction, then the user can optionally provide a *storage object* which specifies the behaviour on destruction (see Storage Objects section 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 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.

2.5.10 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.

2.5.11 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 does an OpenCL "retain" on the OpenCL object. The destructor for the SYCL object does an OpenCL "release" on the OpenCL object. Copy constructors for 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" on the object. It is the user's responsibility to do a "release" on 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. It is the accessors to the image and buffer objects that refer to an actual OpenCL context. Also, accessors provide synchronization which in OpenCL requires events. Therefore, interfacing OpenCL `cl_mem` objects to SYCL is achieved via special accessor classes which can return or be constructed from OpenCL `cl_mem` and `cl_event` objects.

2.6 Memory objects

Memory objects are categorized into two types: *buffer* objects and *image* objects. A buffer object stores a one, two or three-dimensional collection of elements whereas an image object is used to store a two- or three- dimensional texture, frame-buffer or image.

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 used to represent an OpenCL buffer that can be used as a texture or a frame-buffer. 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 `__global` or `__constant` pointers. 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.

2.6.1 Storage objects

Users may want fine-grained control of the synchronization and storage semantics of SYCL memory objects (images or buffers). 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 storage object can implement these methods to provide the destruction, synchronization and memory management features that they require.

2.7 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, kernels will fall back onto the host where possible, while any OpenCL-dependent features of SYCL will fail.

- *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.

2.8 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.

2.8.1 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.

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 one or more 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 `#include`d 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.

3. *Extendible single-source compiler*: Options 1 and 2 above can be combined so that there is a single-source compiler that compiles for host and one or more devices, with the ability to add further SYCL device compilers using stub header files as well.

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

2.8.2 Functions and datatypes available in kernels

Inside kernels, when compiling for a SYCL device, all OpenCL C features are available. All OpenCL C datatypes and kernel functions are available in the sycl::openclc namespace. The features of the OpenCL-defined types, such as vector types are the same in SYCL as in OpenCL. So, when using a built-in OpenCL vector type in SYCL, the syntax for swizzles is the same as for OpenCL C. SYCL also provides an alternative C++ vector library with swizzle support that is portable to non-OpenCL-kernel code.

It is the responsibility of the user to ensure that when compiling the same source file using a host compiler, that the source file is legal for the host compiler. For example, when using an OpenCL-specific feature such as swizzles or kernel library functions, the user must ensure that the host compiler can compile swizzles or kernel library functions (see the below section on fall-back to host).

2.9 Fall-back execution of kernels on the host

SYCL enables kernels to run on either the host or OpenCL devices. When kernels run on an OpenCL device, then the features and behaviour of that execution follows the OpenCL specification. When running kernels on host, the "fall-back" mode is used.

If a kernel fails to run on an OpenCL device and the fall-back to host option is enabled, or if a kernel is specified by the user to only run on the host, then the "fall-back" mode of kernel execution is used.

Kernel math library functions in fall-back mode may use the standard C library on the host platform and are not required conform to OpenCL math precision. If OpenCL math precision is required on the host, then the user must use an OpenCL conformant math library on the host, which is not a mandated requirement for SYCL. Alternatively, users should ensure that their kernels run on an OpenCL CPU device, which is guaranteed to provide conformant math precision.

The range of image formats supported by the host fall-back will match those available on an OpenCL device.

Some of the OpenCL extensions and optional features may be available in a SYCL implementation, but since these are optional features and vendor specific extensions, their use is only recommended to the devices in which these become available. An implementation will need to supply the list of non-core and non-portable features that is allowing on the host.

The synchronization and data movement that occurs when a kernel is executed on the host using the fall-back mechanism may be implemented in a variety of ways on top of OpenCL. The actual mechanism is implementation-defined, but may, for example, be implemented as native kernel.

A user may specify that a kernel must be executed on the host if they desire, by creating a host queue and issuing command groups to that queue.

Some features of OpenCL kernels are not specified to be supported (i.e. undefined) on the host:

- If a user specifies that a variable must reside in `__local` memory in the variable's declaration, then this is undefined on the host. Hierarchical parallelism will work on the host and this provides a mechanism for work-group-local variables.
- Functions overloaded by memory space are undefined on the host. The user must ensure their host compiler uses the necessary `#ifdef`s to ensure that the host compiler sees only functions that it can correctly compile.
- Swizzles in the OpenCL format are undefined on the host.

3. SYCL Programming Interface

The SYCL programming interface provides a C++ abstraction to OpenCL 1.2 functionality and feature set. In this section all the available classes and interfaces of SYCL are described, with main focus on the C++ interface. The functions that are using the OpenCL C API are still available in SYCL. 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.

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 are defined within the `cl::sycl` namespace and all the OpenCL builtins are included in the namespace `cl::sycl::openclc`.

3.2 Platforms, contexts, devices and queues

This part of the API requires the use of STL-like vectors and strings. We use the same approach as the standard OpenCL header files. In this specification, we refer to `VECTOR_CLASS` and `STRING_CLASS`. These macros, by default, are `std::vector` and `std:string`.

A user can choose to use a very basic built-in `cl::sycl::vector` and `cl::sycl::string` class. This is achieved by defining `__NO_STD_VECTOR` and `__NO_STD_STRING`, respectively, before including `"CL/sycl.hpp"`.

Alternatively, a user can define `__USE_DEV_VECTOR` and `__USE_DEV_STRING` to be able to supply custom `VECTOR_CLASS` and `STRING_CLASS`. See section 5 of the OpenCL C++ Wrapper API for more details.

Vector Class Macro	Vector Library Used
`VECTOR_CLASS`	`std::vector`
`__NO_STD_VECTOR`	`cl::sycl::vector`
`__USE_DEV_VECTOR`	custom vector library provided by the developer

Table 3.1: Definition of `VECTOR_CLASS`

String Class Macro	String Library Used
`STRING_CLASS`	`std::string`
`__NO_STD_STRING`	`cl::sycl::string`
`__USE_DEV_STRING`	custom string library provided by the developer

Table 3.2: Definition of `STRING_CLASS`

3.2.1 Platform class

The `platform` class encapsulates a `cl_platform_id`. It can be constructed from a `cl_platform_id`.

The `get_info` method is a direct equivalent of the OpenCL C API.

The `get_platforms` method returns a vector of platform and errors can be returned via C++ exceptions or via a reference to an `error_code`.

There are two variants of the `get_devices` method. SYCL supports the OpenCL C API variant, but also supports variants that need only the `device_type` to return a vector of corresponding devices. All the variants support error handling via a reference to `error_code`, but on the SYCL variants C++ exception handling is supported as well.

Class name:	**platform**
Constructors:	**platform**() **platform**(cl_platform_id platform_id) Optional constructor parameter (one of the following): **platform**(..., error_handler &handler) — C++ exceptions handling or **platform**(..., int &error_code) —OpenCL C error code reference
Destructors:	~ **platform**()
Methods:	cl_platform_id **get**() static VECTOR_CLASS<platform> **get_platforms**() VECTOR_CLASS<device> **get_devices**(cl_device_type device_type) static VECTOR_CLASS<device> **get_devices**(cl_device_type device_type=CL_DEVICE_TYPE_ALL) template<cl_int name> typename param_traits<cl_platform_info, name>::param_type **get_info**() bool **is_host**() bool **has_extension**(const STRING_CLASS extension_name)

Table 3.3: Class description for platform

3.2.2 Device class

The `device` class encapsulates a `cl_device_id` and a `cl_platform_id`. This information fully specifies an OpenCL device across multiple platforms. In the case of constructing a device instance from an existing `cl_device_id` the system triggers a `clRetainDevice`. On destruction a call to `clReleaseDevice` is triggered. The "default" device constructed corresponds to the host. This is also the device that the system will "fall-back" to, if there are no existing or valid OpenCL devices associated with the system.

The developer can partition existing devices through the `create_sub_devices` API. More documentation on this is in the OpenCL 1.2 specification [**?**, sec. 4.3].

Querying information is done through the OpenCL C equivalent, `get_info`.

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 *default device*, which is the host, 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.

Class name:	**device**
Constructors:	**device**() **device**(`cl_device_id` device_id) Optional constructor parameter (one of the following): **device**(..., `error_handler` &handler) — C++ exceptions handling or **device**(..., `int` &error_code) —OpenCL C error code reference
Destructors:	~ **device**()

Methods:	`cl_device_id` **get**()
	
	`platform` **get_platforms**()
	
	`VECTOR_CLASS`<device> **get_devices**(`cl_device_type` device_type)
	
	`VECTOR_CLASS`<device>
	get_devices(`cl_device_type` device_type=`CL_DEVICE_TYPE_ALL`)
	
	`template`<`cl_int` name> `typename`
	`param_traits`<`cl_device_info`, name>`::param_type`
	get_info()
	
	`bool` **has_extension**(`const STRING_CLASS` extension_name)
	
	`bool` **is_host**()
	
	`bool` **is_cpu**()
	
	`bool` **is_gpu**()
	
	`VECTOR_CLASS`<device>
	create_sub_devices(`const cl_device_partition_property` *properties,
	`int` devices,
	`unsigned int` *num_devices)

Table 3.4: Class description for `device`

3.2.3 Context class

The class `context` encapsulates a `cl_context` and allows support of all the OpenCL functionalities. The SYCL system is able to provide mechanisms that make the creation and the construction of contexts easier for the user. The lifetime of the SYCL context object defines the lifetime of the underlying OpenCL `cl_context`.

The constructor creates a context and in the case of copying it calls a `clRetainContext`. On destruction `clReleaseContext` is called. The available constructors reflect these mechanisms.

In order to provide an easier API for the user to make use of the context notification call-back, we also provide the `context_notify` class. The `context_notify` class is an abstract base class, which users implement with `operator()` to be used as the notification function for contexts, instead of the C-style notification functions for contexts in OpenCL.

The `get_info` method is defined using a template to return the correct datatype for the item of information requested from the context. This method uses the same technique as already used in the standard OpenCL C++ wrapper classes.

Class name:	**`context`**
Constructors:	**`context`**() **`context`**(`device_selector` dev_sel) **`context`**(`const cl_context_properties` *properties, `device_selector` dev_sel) **`context`**(`const cl_context_properties` *properties, `VECTOR_CLASS<devices>` target_devices) **`context`**(`const cl_context_properties` *properties, `device` target_device) **`context`**(`cl_context` c) Optional constructor parameter (one of the following): **`context`**(..., `error_handler` &handler) — C++ exceptions handling or **`context`**(..., `context_notify` con_ntf) — OpenCL C error handling
Destructors:	~ **`context`**()
Methods:	`cl_context` **`get`**() `template<cl_int` name> `typename` `param_traits<cl_context_info`, name>`::param_type` **`get_info`**()

Table 3.5: Class description for `context`

Class name:	**context_notify**
Constructors:	**context_notify**()
Destructors:	~ **context_notify**()
Methods:	**operator**(const STRING_CLASS errinfo, const void *private_info, size_t cb) **operator**(program t_program)

Table 3.6: Class description for `context_notify`

3.2.4 Device selection class

The class `device_selector` is an abstract class 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.

```
virtual int device_selector::operator() (device dev) = 0
```

This is the abstract method which returns a "scoring" per-device. When auto-choosing a device, using a selector, the auto-chooser will choose the device with the highest score. Devices with a negative score will never be chosen.

Built-in device selectors (these are all built-in global variables which store no state):

- `gpu_selector`: A built-in selector object which chooses the fastest GPU in the system based on heuristics. If no GPU is available, this selector will choose a CPU device or the host. The details of the heuristics are undefined.
- `cpu_selector`: A built-in selector object which chooses the fastest CPU in the system based on heuristics. If no CPU device is available, this will choose the host. The details of the heuristics are undefined.
- `host_selector`: Forces the select to choose the host platform.

Class name:	**device_selector**
Constructors:	**device_selector**()
Destructors:	~ **device_selector**()
Methods:	`virtual int` **`operator`**()(`device dev`) = `0`
Built-in device selectors:	Class name: `gpu_selector` Class name: `cpu_selector` Class name: `host_selector`

Table 3.7: Class description for `device_selector`

3.2.5 Queue class

Class name:	`queue`
Constructors:	**queue**() **queue**(`cl_command_queue cmd_queue`) **queue**(`device_selector &selector`) **queue**(`context`, `device_selector &selector`, `cl_command_queue_properties = 0`) **queue**(`device queue_device`) **queue**(`queue &cmd_queue`) Optional parameters: **queue**(`..., error_handler &sync_handler`) **queue**(`..., std::function &async_handler`)
Destructors:	~ **queue**()
Methods:	`cl_command_queue` **get**() `context` **get_context**() `device` **get_device**() `cl_int` **get_error**() `template<cl_int` name> typename `param_traits<cl_command_queue_info`, name>::param_type **get_info**() `void` **disable_exceptions**() `void` **throw_asynchronous**() `void` **wait**() `void` **wait_and_throw**()

Table 3.8: Class description for `queue`

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 (if exceptions are enabled on the queue) throws any exceptions that occurred asynchronously on the queue before calling `clReleaseCommandQueue`.

```
queue::queue(context, device, cl_command_queue_properties = 0)
```

Creates a command queue using `clCreateCommandQueue` from a context and a device. Returns errors via C++ exceptions.

```
queue::queue()
```

This chooses a device to run the `command_groups` on. If no device is selected, runs on the host. The method for choosing the "best" device is undefined. This constructor cannot report an error, as any error during queue creation enforces queue creation on the host.

```
queue::queue(device_selector &selector)
```

This chooses a device to to run the `command_groups` on based on the provided selector. If no device is selected, runs on the host. This constructor cannot report an error, as any error during queue creation enforces queue creation on the host.

```
queue::queue(context, device_selector &selector)
```

This chooses a device to run the `command_groups` based on the provided selector, but must be within the provided context. If no device is selected, an error is reported via a C++ exception.

```
queue::queue(device queue_device)
```

This creates a queue on the given device. Any error is reported via C++ exceptions.

```
void queue::wait()
```

This a blocking wait for all enqueued tasks in the queue to complete.

3.2.5.1 Queue error handling

There are two optional parameters at the end of every queue constructor. If the `sync_handler` is present, then all synchronous errors are passed to it (see 3.34). If the `async_handler` is present, then all asynchronous errors (i.e. errors in queues) are passed to it at the point of the call to `wait_and_throw`, `throw_asynchronous` or destruction. The `async_handler` is passed a single parameter: a `std::exception_list &errors` value. If no `async_handler` is provided, then all asynchronous errors have no impact on program execution.

The error handling behaviour applies to all operations on queues, such as kernel invocation and accessor construction.

```
void queue::throw_asynchronous()
```

This function checks to see if any asynchronous errors have been thrown in the queue and if so reports them via exceptions, or via the supplied `async_handler`.

```
void queue::wait_and_throw()
```

This function waits for work on the queue to complete and then checks to see if any asynchronous errors have been thrown in the queue and if so reports them via exceptions, or via the supplied `async_handler`.

3.2.6 Command group class

A *command group* in SYCL as it is defined in 2.3.1 includes a kernel to be enqueued along with all the commands for queued data transfers that it needs in order for its 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 class `command_group` on Table 3.9 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 3.3, where data storage 3.3.3, buffers 3.3.1 and accessor 3.3.4 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.

Class name	**`command_group`**`<typename` functorT> Template parameters: `typename functorT:` kernel functor or lambda function
Constructors:	**`command_group`**`<typename` functorT>(`queue`, `functorT`)
Destructors:	~ **`command_group`**`<typename` functorT>()
Methods :	`event` **`kernel_event`**() `event` **`start_event`**() `event` **`complete_event`**()

Table 3.9: Class description for `command_group<typename` functorT>

3.2.7 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.

Class name	**`event`**
Constructors:	**`event`**()
	`event`(`cl_event` from_cl_event)
Destructors:	~ **`event`**()
Methods :	`cl_event` **`get`**(`cl_context` context)
	`static void` **`wait`**(`VECTOR_CLASS<event>` event_list)
	`VECTOR_CLASS<event>` **`get_wait_list`**()

Table 3.10: Class description for `event`

3.3 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.

3.3.1 Buffers

The class `cl::sycl::buffer`<typename T, int dimensions> (Table 3.11) 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.

There are 5 ways of constructing a `buffer`. These 5 ways control the underlying storage for the buffer and what happens when the buffer object is destroyed.

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 has 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 3.3.3). 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.

5. A buffer can be constructed by using a copy constructor from any structure that has contiguous storage in which case the original structure or class does not get updated by the buffer after the processing. In this case the developer needs to provide the *startIterator* and the *endIterator* addresses and the buffer allocates a new host storage object where the values of the the copy constructor are used as an initialization.

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.

Class name:	`buffer<typename T, int dimensions>` Template parameters: `typename T`: The type of the elements of the buffer `int dimensions`: number of dimensions of the buffer: 1, 2 or 3

Constructors:	`buffer<T, dimensions>(range<dimensions>)`
	
	`buffer<T, dimensions>(T *host_data, range<dimensions>)`
	
	`buffer<T, dimensions>(storage<T> &store, range<dimensions>)`
	
	`buffer<T,1>(T * startIterator, T * endIterator )`
	
	`buffer<T, dimensions>(buffer<T, dimensions>)`
	
	`buffer<T, dimensions>(buffer<T, dimensions>, index<dimensions> base_index, range<dimensions> sub_range)`
	
	`buffer<T, dimensions>(cl_mem mem_object, queue from_queue, event available_event)`

Destructors:	~ **buffer**()
Methods:	`range<dimensions>` **get_range**() `size_t` **get_count**() `size_t` **get_size**() `template<access::mode mode, access::target target=access::global_buffer>` `accessor<T, dimensions, mode, target>` **get_access**()

Table 3.11: Class description for `buffer<typename T, int dimensions>`

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`.

The `get_range` method returns the *n*-dimensional range of the buffer (i.e. the sizes of the 1, 2 or 3 dimensions created on construction). The `get_count` method returns the total number of elements of type `T` in the buffer. The `get_size` method returns the total number of bytes in the buffer.

The `get_access` method returns an accessor to the buffer of the requested access mode and (if specified) target, with the right base type and dimensions. The target value can be `access::global_buffer`, `access::constant_buffer` or `access::host_buffer`.

3.3.2 Images

Class name:	`image<int dimensions>` Template parameters: `int dimensions`: number of dimensions of the image: 1, 2 or 3

Constructors:	`image<dimensions>(const cl_image_desc &desc, const cl_image_format &format, range<dimensions> size)`
	
	`image<dimensions>(const cl_image_desc &desc, const cl_image_format &format, void *host_data, range<dimensions> size)`
	
	`image<dimensions>(const cl_image_desc &desc, const cl_image_format &format, storage &store, range<dimensions> size)`
	
	`image<dimensions>(image<dimensions>)`

Destructors:	`~ image<dimensions>()`

Methods:	`range<dimensions> get_range()`
	`size_t get_count()`
	`size_t get_size()`
	
	`template<typename T, access::mode mode, access::target target=access::image>`
	`accessor<T, dimensions, mode, target> get_access()`

Table 3.12: Class description for `image<int dimensions>`

The class `image<typename T, int dimensions>` (Table 3.12) 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 take a `desc` argument and `format` argument. These describe the format of the data storage for the image. It is the responsibility of the user to ensure that any provided host memory or storage object matches the storage format supplied.

There are 4 ways of constructing an `image`. These 4 ways control the underlying storage for the image and what happens when the image object is destroyed.

1. An image can be constructed with just a size and no associated storage. The storage for this type of

image is entirely handled by the SYCL system. The destructor for this type of image never blocks, even if work on the image has not completed. Instead, the SYCL system frees any storage required for the image asynchronously when it is no longer in use in queues. The initial contents of the image are undefined.

2. An image can be constructed with associated host memory. The image will use this host memory for its full lifetime, but the contents of this host memory are undefined for the lifetime of the image. If the host memory is modified by the host, or mapped to another buffer or image during the lifetime of this image, then the results are undefined. The initial contents of the image will be the contents of the host memory at the time of construction. When the image is destroyed, the destructor will block until all work in queues on the image has completed, then copy the contents of the image back to the host memory (if required) and then return.

3. If the pointer to host memory is `const`, then the image is read-only, only read accessors are allowed on the image and no-copy-back to host memory is performed (although the host memory must still be kept available for use by SYCL).

4. An image can be constructed with an associated `storage` object (see 3.3.3). The `storage` object must not be destroyed by the user until after the image has been destroyed. The synchronization and copying behaviour of the storage is determined by the storage 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`.

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

The `get_range` method returns the *n*-dimensional range of the image (i.e. the sizes of the 1, 2 or 3 dimensions created on construction). The `get_count` method returns the total number of pixels in the buffer. The `get_size` method returns the total number of bytes in the image.

The `get_access` method returns an accessor to the image of the requested sample type, access mode and (if specified) target, with the right dimensions. The target value can be `access::image` or `access::host_image`.

3.3.3 Storage classes

The `storage` class (Table 3.13) is an abstract superclass that allows users to define ownership of data. The `storage` class may be implemented by users to create custom data management classes that change the behaviour of the default data management.

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 `get_data` method is called by the SYCL system to know where that data is held in host memory. It should return 0 if no host memory is attached to this data and therefore the SYCL system will manage temporary storage of the data.

The `get_initial_data` method is called at the point of construction to request the initial contents of the buffer. If it returns a non-null pointer, then the buffer will be initialized to the contents of that memory.

Class name:	`storage<typename T>` Template parameters: `typename T:` The type of the elements of the underlying data

Methods:	`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`

Table 3.13: Class description for `storage`

The `get_final_data` method is called at the point of construction to request where the contents of the buffer should be finally stored to. If `get_final_data` returns a non-null pointer, then the contents of the buffer will be written to that location in host memory. If `get_host_data` returns the same pointer as `get_initial_data` and/or `get_final_data` then the SYCL system should determine whether copying is actually necessary or not.

The `destroy` method is called when the associated memory object is destroyed. This method is only called once, so if a memory object is copied multiple times, only when the last copy of the memory object is destroyed is the `destroy` method called. Exceptions thrown by the `destroy` method will be caught and ignored.

The `in_use` method is called when a command group which accesses the data is added to a queue. The `completed` method is 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 the memory object.

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

The SYCL platform provides `async_storage` (Table 3.14) 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. 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. However, on destruction the `async_storage` class will block until no buffers are using it and no tasks are executing that might read or write the data wrapped by the `async_storage` object. By this means the developer can ensure that it is safe to free the underlying memory.

Class name:
Superclass:
	`async_storage<typename T>`
	`storage<typename T>`
	
	Template parameters:
	`typename T:` The type of the elements of the underlying data

Constructors:	**`async_storage`**`(void *mappedData, size_t size)`
	
	`async_storage``(size_t size)`

Destructors:	`virtual ~` **`async_storage`**`()`

Table 3.14: Class description for `async_storage`

3.3.4 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.

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. A global accessor can provide a `__global` pointer to the underlying data within a kernel, while a constant accessor can provide a `__constant` pointer to the underlying data.

Accessors to images are constructed from an image with the same dimensionality as the accessor. Image accessors can be indexed by floating-point indices and use user-supplied *samplers*. An image can only be allocated in global memory and any address space qualifiers are not allowed.

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. Local memory is only shared across a work-group. A local accessor can provide a `__local` pointer 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.

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.

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.

An accessor to a global or local buffer can have *atomic* access, which means all data accesses via this accessor are atomic operations.

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.

3.3.4.1 Access modes

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.

The `mode` enumeration has a base value, which must be provided.

Enumerator name:	`access::mode`
Values:	`read`: read-only access `write`: write-only access. Previous target object contents discarded `atomic`: atomic read and write access `read_write`: read/write access `discard_read_write`: read/write access. Previous target object contents discarded

Table 3.15: Enumerator description for `access::mode`

3.3.4.2 Access targets

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

Enumerator name:	`access::target`
Values:	`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 `buffer` immediately on the host `host_image`: access `image` immediately on the host `image_array`: access an array of `image`s on device `cl_buffer`: access an OpenCL `cl_mem` buffer on device `cl_image`: access an OpenCL `cl_mem` image on device

Table 3.16: Enumerator description for `access::target`

3.3.4.3 Core accessors class

The `accessor` (see Table 3.17) makes a buffer's data available to user code. An accessor has a `mode`, which defines the operations possible on the underlying data and a `target`, which defines the type of data object to be modified. The constructors and methods available on an accessor depend on the `mode` and `target`.

Class name:	**accessor**`<typename dataType, int dimensions, access::mode mode, access::target target=access::global_buffer>` Template parameters: `typename dataType`: The type of the data the accessor accesses `int dimensions`: the number of dimensions of the accessor, can be 1,2, or 3 `access::mode mode` defines mode of access. See 3.15 `access::target target` defines type of data to access. See 3.16
Constructors:	Depends on `target`. See below
Destructor:	Destructor behaviour depends on `target`
Methods:	`int` **`get_size`**() `cl_mem` **`get_cl_mem_object`**() `cl_event` **`get_cl_event_object`**() Other methods depend on `mode` and `target`. See below

Table 3.17: Base class description for `accessor<typename` T, `int` dimensions, `access::mode mode`, `access::target target>`

3.3.4.4 Buffer accessors

Accessors that target buffers (see 3.18) must be constructed from buffers of the same element type and dimensionality of the accessor. The target parameter must be either `global_buffer`, `constant_buffer` or `host_buffer`. 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`.

Class name:	**`accessor`**`<typename dataType, int dimensions, access::mode mode, access::target target>` Template parameters: `typename dataType:` Must be same as base type of buffer `int dimensions:` Must be same as dimensionality of buffer `access::mode mode:` Defines mode of access. See 3.15 `access::target target:` `global_buffer`, `constant_buffer` or `host_buffer`. See 3.16
Constructors:	**`accessor`**(`buffer`<dataType,dimensions> &target)
Methods:	`dataType` **`&operator[]`**(`id`<dimensions>) • Reference to target element. • Only if `mode` contains write access `const dataType` **`&operator[]`**(`id`<dimensions>) • Read element from target data. • Only if `mode` is read-only `__atomic_ref`<dataType> **`operator[]`**(`id`<dimensions>) • Atomic reference to element from target data. • Only if `mode` is `atomic`.

Table 3.18: Class constructor and methods description for `accessor` that targets buffers

To provide atomic access to data when the access mode is `access::atomic`, the index operator returns an `__atomic_ref` internal class which allows the main read and write operators to perform atomic operations. The operators supported for atomic operations by the `__atomic_ref` class are: =, +=, -=, --, ++, &=, |=, ^=.

3.3.4.5 Image accessors

Accessors that target images (see 3.19) must be constructed from images of the same dimensionality as the accessor. The `target` parameter must be either `image` or `host_image`. The `dataType` parameter must be a 1 to 4 dimension vector of either `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 3.3.4.6) parameter, which then allows floating-point sampler-based reading using the array operator (e.g. `myAccessor(mySampler)[my2dFloatVector]`).

Class name:	`accessor<typename T, int dims, access::mode mode, access::target target>` Template parameters: `typename T`: `vec<float` or `int`, 1 2 3 or 4> `int` dims: Dimensions match the dimensionality of the image `access::mode mode`: defines mode of access. See 3.15 `access::target target`: either `image` or `host_image`. See 3.16
Constructors:	`accessor`<T,dims,mode,target>(`image`<dimensions> &target)
Methods:	`T` **`operator[]`**(`id`<dimensions>) • Read pixel from target image. • Only if `mode` does not contain write access `__image_ref`<dataType> **`operator[]`**(`id`<dimensions>) • Reference to target pixel (via internal intermediate object). • Only if `mode` contains write access. • Writing will use OpenCL image writing operations `__sampler`<dataType> **`operator()`**(`sampler` smpl) • Allow reading via sampler `dataType __sampler`<dataType>::**`operator[]`**(`vec`<float, dimensions>) • Performs reading via sampler

Table 3.19: Class constructor and methods description for `accessor` that targets images

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.

3.3.4.6 Samplers

Samplers use the `cl::sycl::sampler` type which is equivalent to the OpenCL C `cl_sampler` and `sampler_t` types.

Class name:	**`sampler`**
Host constructors:	**`sampler`**(`bool` normalized_coords, `cl_addressing_mode` addressing_mode, `cl_filter_mode` filter_mode) **`sampler`**(`cl_sampler`)
Kernel constructors:	**`sampler`**(`unsigned int` samplerflags)

Table 3.20: Class constructor for samplers

The host constructors for samplers are only available outside kernels, whereas the kernel constructors are only available inside kernels. The kernel constructor takes the same parameters as the constant constructors defined in the OpenCL C language spec.

3.3.4.7 Image array accessors

An image array accessor provides access to an array of 2D images.

Class name:	**accessor**<typename dataType, int dimensions, access::mode mode, access::image_array> Template parameters: typename dataType: vec<float or int, 1 2 3 or 4> int dimensions: must be 2 access::mode mode: defines mode of access. See 3.15 access::target target: image_array. See 3.16
Constructors:	**accessor**(image<2> *target, size_t num_images)
Methods:	**accessor**<dataType, 2, mode, image> **operator[]**(size_t index) • Returns an image accessor from the array

Table 3.21: Class constructor and methods description for accessor that targets arrays of 2D images

3.3.4.8 Local accessors

Local accessors have a target of `access::local`. The data they access is only available within the same work-group. When constructing a local accessor, the user just provides a `range`, which is the number of elements of type `dataType` in the underlying work-group-local array.

The user can provide an index as an `id` parameter of the same dimensionality of the accessor, 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 size parameter can be provided as a list of scalar arguments of `size_t` for each dimension of the accessor, instead of a `range`.

Class name:	**`accessor`**`<typename dataType, int dimensions, access::mode mode, access::target target>` Template parameters: `typename dataType`: must be same as base type of buffer `int dimensions`: must be same as dimensionality of buffer `access::mode mode`: defines mode of access. See 3.15 `access::target target`: must be `local`. See 3.16
Constructors:	**`accessor`**`(range<dimensions> size)`

Methods:	`__local dataType` **`&operator[]`**`(id<dimensions>)`
	• Reference to target element if has write access
	
	`const __local dataType` **`&operator[]`**`(id<dimensions>)`
	• Read element from target data. Only if `mode` is read-only
	
	`__atomic_l_ref<dataType>` **`operator[]`**`(id<dimensions>)`
	• Atomic reference to element from target data.
	• Only if `mode` contains `atomic` flag.

Table 3.22: Class constructor and methods description for `accessor` that targets work-group local memory

3.3.4.9 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. The constructors for host accessors have an optional `error_handler&` parameter, which if present receives any errors.

3.3.4.10 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.

Class name:	**`accessor`**`<typename dataType, int dimensions,`
	`access::mode mode,`
	`access::target target>`
	
	Template parameters:
	`typename dataType`: must be same as base type of buffer
	`int dimensions`: must be same as dimensionality of buffer
	`access::mode mode`: defines mode of access. See 3.15
	`access::target target`: must be `cl_buffer`. See 3.16

Constructors:	**`accessor`**(`buffer`<dataType, dimensions> source, `queue` in_queue)
Methods:	`cl_mem` **`get`**() `cl_mem` **`get_event`**() `void` **`release_event`**(`cl_event` event)

Table 3.23: Class constructor and methods description for `accessor` that allows access to SYCL buffers from OpenCL C

Class name:	`accessor<int dimensions,`
	`         access::mode mode,`
	`         access::target target>`
	
	Template parameters:
	`int dimensions`: must be same as dimensionality of image
	`access::mode mode`: defines mode of access. See 3.15
	`access::target target`: must be `cl_image`. See 3.16

Constructors:	**`accessor`**(`image`<dimensions> source, `queue` in_queue)
Methods:	`cl_mem` **`get`**() `cl_mem` **`get_event`**() `void` **`release_event`**(`cl_event` event)

Table 3.24: Class constructor and methods description for `accessor` that allows access to SYCL images from OpenCL C

3.3.4.11 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 3.25, 3.26 and 3.27 show all the permitted access modes depending on target.

Accessor Type	Access Target	Access mode	Data Type	Description
Device Accessor	`global_buffer`	`read` `write` `atomic` `read_write` `discard_read_write`	All available data types supported in SYCL.	Access a buffer allocated in global memory on the device.
Device Accessor	`constant_buffer`	`read`	All available data types supported in SYCL.	Access a buffer allocated in constant memory on the device.
Host Accessor	`host_buffer`	`read` `write` `read_write` `discard_read_write`	All available data types supported in SYCL.	Access a host allocated buffer on host.
Device or host access Accessor	`cl_buffer`	`read` `write` `atomic` `read_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 Accessor	`local`	`read` `write` `atomic` `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 3.25: Description of all the `accessor` types and modes with their valid combinations for buffers and local memory

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

Accessor Type	**Access Target**	**Access mode**	**Data Type**	**Description**

Device Accessor	`image`	`read`
		`write`
		`read_write`
		`discard_-read_write`
			`uint4, int4, float4, half4`	Access an image on device.

Host Accessor	`host_image`	`read`
		`write`
		`read_write`
		`discard_-read_write`
			`uint4, int4, float4, half4`	Access an image on the host.

Device Accessor	`image_array`	`read`
		`write`
		`read_write`
		`discard_-read_write`
			`uint4, int4, float4, half4`	Access an array of images on device.

Device Accessor	`cl_image`	`read`
		`write`
		`read_write`
		`discard_-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 3.26: Description of all the `accessor` types and modes with their valid combinations for images

Accessor Types	**Original Accessor Target**	**Original Access Mode**	**Converted Accessor Target**	**Converted Access Mode**

Device Accessor	`global_buffer`	`read_write`	`global_buffer`	`read`
				`write`
				`discard_-read_write`

Device Accessor	`local_buffer`	`read_write`	`local_buffer`	`read` `write`

Device Accessor	`cl_buffer`	`read_write`	`cl_buffer`	`read`
				`write`
				`discard_-read_write`

Host Accessor	`host_buffer`	`read_write`	`host_buffer`	`read`
				`write`
				`discard_-read_write`

Table 3.27: Description of the `accessor` to `accessor` conversions allowed

3.4 Expressing parallelism through kernels

3.4.1 Ranges and identifiers

Class name	**range**<int dims> Template parameters: int dims: range dimensions
Constructors:	**range**<dims>(sizeof_dimension_1,...) **range**<dims>(range<dims>)
Destructors:	~ **range**<dims>()

Class name	**nd_range**<int dims> Template parameters: int dims: nd-range dimensions
Constructors:	**nd_range**<dims>(range<dims> global_size, range<dims> local_size) **nd_range**<dims>(nd_range<dims>) Optional constructor parameters: **nd_range**<dims>(..., id<dims> offset)
Destructors:	~ **nd_range**<dims>()
Methods:	range **get_global_range**() range **get_local_range**() range **get_group_range**()

Table 3.28: Class description for range and nd_range

The data parallelism of OpenCL and SYCL requires ranges to define the range of kernel execution and the range of individual work groups. Each work-group and work item also needs to have an identifier for the item or group.

range<int dims> is a 1D, 2D or 3D vector that defines a range. It can be constructed from integers, or from an int vector of the same rank.

nd_range<int dims> contains 2 ranges: the whole range over which the kernel is to be executed, and the range of each work group. It is constructed from 2 ranges.

id<int dims> is a vector of dimensions that is used to represent an *ND index* into the *NDRange* or work-group. It can be implicitly converted to/from an int vector of the same rank. It can also be used as an index in an accessor of the same rank. The [n] operator returns the component n as an int.

item<int dims> contains 2 vectors to identify a work item: the id of the work group ("group id" in OpenCL)

Class name	**id**<int dims> Template parameters: int dims: id dimensions
Constructors:	**id**<int dims>(range<dims> global_size, range<dims> local_size) **id**<dims>(id<int dims>)
Destructors:	~ **id**<int dims>()
Methods:	int **get**(int dimension)

Table 3.29: Class description for `id`

Class name	**item**<int dims> Template parameters: int dims: id dimensions
Constructors:	**item**<dims>(range<dims> global_size, range<dims> local_size) **item**<dims>(item<dims>)
Destructors:	~ **item**<dims>()
Methods:	int **get_global**(int dimension) int **get_local**(int dimension) id<dims> **get_global**() id<dims> **get_local**() range<dims> **get_local_range**() range<dims> **get_global_range**()

Table 3.30: Class description for `item`

and the id of the work item ("global id") in OpenCL). The method `get_global` returns the global id (of type `id`). The method `get_local` returns the local id (of type `id`). The method `get_group_id` returns the group id (of type `id`). It cannot be constructed by the user, only passed in as a parameter from the runtime.

`group<int` dims> contains 2 vectors to identify the group id, as well as the group range. This is used in hierarchical parallelism to pass into the inner `parallel_for` loop. It cannot be constructed by the user, only passed in as a parameter from the runtime.

Class name	`group`<`int` dims>
Template parameters:	`int` dims: number of dimensions
Constructors:	`group`<dims>()
Destructors:	~ `group`<dims>()
Methods:	`id`<dims> **`get_group_id`**() `id`<dims> **`get_local_range`**() `id`<dims> **`get_global_range`**()

Table 3.31: Class description for `group`

3.4.2 Defining kernels

In SYCL, users define kernels which get enqueued to command queues. The user calls an enqueue function (see 3.4.3). A kernel has a *kernel method* which is the actual method called on the device with parameters which allow the kernel to determine its work-item id and other useful identification information. The signature of this method must match the requirements of the enqueue function that invokes the kernel. The kernel method must always return `void`.

The `kernel` class allows interoperability with OpenCL C kernels. There are three ways of defining kernels, defining them as functors, as lambda functions or as C strings.

The constructor `kernel<typename T>(context target_cont, device target_dev)` will construct a default program and a kernel from the functor type `T`. In the case of creating a kernel from a C string then the constructor `kernel(context target_cont, device target_dev, std::string string_kernel, std::string ↩ string_name)` will create a default program for the context and the device specified and will define the kernel from the source C string and the name C string.

The function `void set_arg(int index, accessor acc_obj)` needs to be called inside a `command_group` in order to set the kernel arguments when defining a kernel using OpenCL C string instead of a kernel functor or lambda. This function should be called for every accessor in the same order with the corresponding kernel parameters.

Class name	**kernel**<typename T> Template parameters: typename T: kernel functor type
Constructors:	**kernel**<typename T>(context target_cont, device target_dev) **kernel**(context target_cont, device target_dev, STRING_CLASS string_kernel, STRING_CLASS string_name)
Destructors:	~ **kernel**<T>()
Methods:	cl_kernel **get**() context **get_context**() program **get_program**() STRING_CLASS **get_kernel_attributes**() STRING_CLASS **get_function_name**() void **set_arg**(int arg_index, accessor acc_obj) template<typename T> void **set_arg**(int arg_index, T scalar_value)

Table 3.32: Class description for kernel

3.4.2.1 Defining kernels as functors

A kernel can be defined as a functor. In the case of a C++ functor, the *kernel method* is the method defined as `operator()` in the normal C++ functor style. Kernels defined as functors may be templated.

```
class MyFunctor
{
    float m_parameter;

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

    void operator() (cl::sycl::id<2> myId)
    {
       // do work on myId
    }
}

    MyFunctor myKernel(3.2f);

    command_group(&queue, [&] ()
        {
            parallel_for(nd_range<2>(4,4), myKernel);
        });
```

3.4.2.2 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 convert a C++ 11 lambda function into an SYCL kernel lambda, pass the lambda function to the `kernel_lambda` function with the name of the kernel lambda as its template argument.

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(kernel_lambda<class MyKernel>([=] ()
            {
                // [kernel code]
            }));
    });
```

3.4.2.3 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 3.6. The kernel is defined as a functor 3.4.2.1 or lambda function 3.4.2.2 and and the corresponding program built will be called from the program.

For example, if the kernel is defined as a lambda function the definition of the kernel will be as follows:

```
program MyProgram<MyKernel>(context, device);

command_group(&queue, [&] ()
    {
        parallel_for(nd_range<2>(4,4),
                              MyProgram,
                              kernel_lambda<class MyKernel>([=] ()
                                  {
                                      //[kernel code]
                                  }));
    }
```

In the above example the code for creating the command queue is not included, for more detailed information on the queue please refer to 3.2.5.

3.4.2.4 Defining kernels using OpenCL C strings

OpenCL C strings can be used to enqueue kernels in SYCL utilizing the `parallel_for` function in a *command group*.

In this case a program object and a kernel object have to be created, as these are required by the OpenCL C runtime. Program objects are described in 3.6

A kernel object can be created in SYCL with the use of a *default program* object, which the runtime produces for every kernel invocation. The constructor from the kernel class on Table 3.32

```
kernel::kernel(context con, device target_dev, char * kernel_string, char *  ↩
    kernel_name)
```

will create a *default program* for the corresponding context and will match the kernel string with the target device. The kernel name is used in order to extract the kernel from the program.

It is possible to create a program object from a string using the methods in 3.35:

```
program::program(context con, device target_dev, char* kernel_string)
```

and extract the kernel object using the method:

```
program::get_kernel(char *kernel_name)
```

In either case, as the kernel is defined using a C string the system cannot deduce the accessors for the kernel, so the developer needs to set the accessors inside a command group in the same order as they are expected from the kernel arguments. The kernel method as described in 3.32:

```
kernel::set_arg(int index, accessor defined_accessor)
```

sets the kernel arguments using the underlying OpenCL C calls for setting kernel arguments and uses the context and the device of the queue of the corresponding *command group* the accessors are defined.

In this sense the kernel is still enqueued using a *command group* and data management is the same as with the SYCL kernels. Of course the kernel which is constructed using a kernel object can be used in every command group which corresponds to the same context and target device.

3.4.2.5 Defining kernels using OpenCL C kernel objects

In OpenCL C [**?**] 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. In that way all the functionality which is available form *SYCL kernel* enqueuing and data management is available for kernels constructed using OpenCL C API calls.

The constructor using kernel objects from 3.32:

```
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 also requires from the developer to set the accessors as kernel arguments inside a *command group*. The function

```
kernel::set_arg(int index, accessor defined_accessor)
```

needs to be called for every accessor which is needed by the kernel in the same order as the kernel arguments are defined. The system in this case is not able to deduce the kernel arguments, as the kernel is defined using the OpenCL C API, so the way the accessors should be defined and their order has to be handled by the developer. The advantage of this method is that the OpenCL C kernels can use the same data management and synchronization as the *SYCL kernels*.

3.4.3 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 3 functions that enable these for types of invocation of kernels. Each function can take any of the kernel specifications described in section 3.4.3.

3.4.3.1 Single Task invoke

SYCL provides a simple interface to enqueue a kernel on an OpenCL device which is going to be sequentially executed, since it only enqueues an *NDRange* of (1, 1), which is essentially one thread for one work-group. 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.

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

For single tasks, the kernel method takes no parameters.

3.4.3.2 Parallel For invoke

The `parallel_for` interface offers the ability to the SYCL users to declare a kernel and enqueue it in a command queue using using the accessors defined inside a command group. In the basic case the developer only needs to provide the number of work-item the kernel will use in total and the system will use the best range available to enqueue it on a device. The kernel defined as a lambda or as a kernel functor will be enqueued along with the rest of the commands of the command group. The developer can define the kernel as functor object and use this interface for enqueuing it.

```
parallel_for(total_number_of_work_items,
             kernel_functor<class example_kernel>([=] (id t_id)
                 {
                     //[kernel code]
                 }));
```

Kernels which are defined using the `cl::sycl::kernel` class can be enqueued using the `parallel_for` interface. In this case, the kernel is already defined, so the function call will require a kernel object created as described in sections 3.4.2.4 and 3.4.2.5. The kernel arguments cannot be captured by the lambda function or the kernel functor, so in this case the kernel arguments have to set explicitly in the `command_group`.

```
kernel sycl_kernel(target_context, target_device, kernel_string, kernel_name);
command_group(&myQueue, [&] ()
    {
        accessor<float, 2, read_write, global_buffer> ptr(buf);

        kernel.set_arg(0, ptr);

        parallel_for(nd_range(range(4, 4, 4), range(2, 2, 2)), sycl_kernel);
    });
```

In the case where specifying the work-group and work-item sizes is the preferred way of enqueueing a kernel, the developers need to specify an `nd_range`, which has the same semantics as the OpenCL *NDRange*. Inside a `parallel_for` with an `nd_range` parameter, it is possible to call the `cl::sycl::barrier` function which provides a barrier between work items in a work-group. The semantics and single parameter of the barrier function are exactly the same as those for OpenCL.

```
parallel_for(nd_range(range(4, 4, 4), range(2, 2, 2)),
             kernel_functor<class example_kernel>([=] (item t_item)
                 {
                     // [kernel code]
                     barrier(CL_LOCAL_MEM_FENCE);
                 }));
```

Kernels which are defined using the `cl::sycl::kernel` class can be enqueued using the `parallel_for` *NDRange* interface in the same way as with the *basic* `parallel_for` interface.

```
kernel sycl_kernel(target_context, target_device, kernel_string, kernel_name);
command_group(&myQueue, [&] ()
    {
        accessor<float, 2, read_write, global_buffer> ptr(buf);

        kernel.set_arg(0, ptr);

        parallel_for(nd_range(range(4, 4, 4), range(2,2,2)), sycl_kernel);
    });
```

3.4.3.3 Parallel For hierarchical invoke

The *parallel for hierarchical* interface offers a different way of structuring an OpenCL kernel, although it will be enqueued as such by the system. This interface offers a more intuitive way to tiling parallel programming paradigms. The kernel in this case has two levels, the work-group level and the work-item level of execution. In the work-group level (defined in `parallel_for_workgroup`), a coarse-grain parallelism is available, where the execution of all these commands follows single-threaded approach (only one execution flow is followed). The inner level, which is defined in the `parallel_for_workitem` function, allows execution across all the work-items inside a work-group. 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.

```
parallel_for_workgroup(nd_range(size(global_sz, global_sz, global_sz),
                                size(local_sz, local_sz, local_sz)),
                       kernel_functor<class hierarchical_kernel_name>(
                           [=] (group group)
                               {
                                   //[kernel code for work-group level]

                                   parallel_for_workitem(group, [=] (item t_item)
                                       {
                                           // [kernel code for work-item level]
                                       });

                                   // [kernel code for work-group level]
                               }));
```

3.4.4 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++ POD values must be passed by value to the kernel.

- C++ non-POD values passed as arguments to a kernel that is compiled for a device are illegal in SYCL when targeting OpenCL 1.*x* generation devices.

- 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 an OpenCL 1.*x* generation device. 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.

3.5 Error handling

There are two alternatives for handling errors in SYCL. The first and recommended alternative for platforms that support it, is C++ exception handling. In that case the user can use the C++ mechanisms for handling errors. When users want errors not reported via exception handling, they can add an optional `error_handler&` parameter which receives errors instead of throwing exceptions.

Error handling in SYCL uses exceptions for synchronous errors, by default. 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. Functions and methods which can generate synchronous errors can also be passed in an error-handling object, which receives the exception objects as parameters instead of being thrown as exceptions.

Any SYCL call defined above which has a reference parameter (`error_handler &handler`) calls the user-supplied `error_handler` function instead of reporting an exception. Any SYCL call which operates on a queue (such as accessor construction or `parallel_for` inside a `command_group`) will report the error to the queue, using the queue's error-handling function.

Class name:	**`exception`**
Methods:	`cl_int` **`get_cl_code`**(): returns the OpenCL error code. Returns 0 if not an OpenCL error `cl_int` **`get_sycl_code`**(): returns the SYCL-specific error code. Returns 0 if not a SYCL-specific error `queue *`**`get_queue`**(): returns the queue that caused the error. Returns 0 if not a queue error `buffer *`**`get_buffer`**(): returns the buffer that caused the error. Returns 0 if not a buffer error `image *`**`get_image`**(): returns the image that caused the error. Returns 0 if not a image error

Table 3.33: Class description for `exception`

Class name:	**`error_handler`**
Methods:	**`report_error`**(`exception` &error) `= 0`: called on error

Table 3.34: Class description for `error_handler`

3.6 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.

Class name	program
Constructors:	**program**<typename T>(context dev_context, device target_dev)
	program(context dev_context, device target_dev, STRING_CLASS kernel_-string)
Destructors:	~ **program**()
Methods:	void **build**(const STRING_CLASS options, context_notify &notify)
	void **compile**(const STRING_CLASS options, context_notify &notify)
	static void **link**(STRING_CLASS *options, VECTOR_CLASS<input_programs>, context_notify & notify)
	static const program<&> **get_default_program**(context dev_context, device target_dev)
	template<cl_int name> typename detail::param_traits<detail::cl_program_info, name>::param_type **get_info**()
	kernel **get_kernel**<typename T>()
	kernel **get_kernel**(STRING_CLASS opencl_c_name)
	VECTOR_CLASS<STRING_CLASS> **get_binaries**()
	VECTOR_CLASS<::size_t> **get_binary_sizes**()
	VECTOR_CLASS<device> **get_devices**()
	STRING_CLASS **get_kernel_name**()
	STRING_CLASS **get_build_options**()

Table 3.35: Class description for program

The are two ways to specify a kernel to a program. The first one is by providing the kernel type in the case of defining the kernel as a functor and the second one is to provide the OpenCL C string.

The constructor for providing creating a program from a kernel functor is

```
cl::sycl::program<typename T>(context dev_context, device target_dev)
```

and creates a program targeting `device` `target_dev` from `dev_context`. In order to get the kernel object from this program, the function

```
cl::sycl::program::get_info<typename T>()
```

will return a `kernel<typename T>` object.

The constructor

```
cl::sycl::program(context dev_context, device target_dev, char *kernel_string)
```

creates a program from the `kernel_string` targeting the `device` `target_dev`, which has to be included in the `context` given.

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

3.7 Data Types

3.7.1 Scalar types

SYCL defines a set of scalar data types which are guaranteed to be the same between host and device. These types are defined in the `cl::sycl` namespace:

- `cl_char`: an 8-bit signed character
- `cl_uchar`: an 8-bit unsigned character
- `cl_short`: a 16-bit signed integer
- `cl_ushort`: a 16-bit unsigned integer
- `cl_int`: a 32-bit signed integer
- `cl_uint`: a 32-bit unsigned integer
- `cl_long`: a 64-bit signed integer
- `cl_ulong`: a 64-bit unsigned integer
- `cl_float`: a 32-bit IEEE 754 floating-point value
- `cl_double`: a 64-bit IEEE 754 floating-point value
- `cl_half`: a 16-bit IEEE 754-2008 half-precision floating-point value.

3.7.2 Vector types

SYCL provides a templated cross-platform vector type that works efficiently on device as well as host. This type allows sharing of vectors between host and device and can be trivially converted to and from the OpenCL built-in vector types. The full swizzle feature-set is provided, but in a way that doesn't require special compiler support for swizzles.

`vec<typename` T, `int` dims> is a vector type that compiles down to the OpenCL built-in vector types on OpenCL devices and provides compatible support on the host. The `dims` parameter can be: 1, 2, 3, 4, 8 or 16. The `T` parameter must be one of the basic OpenCL types.

Vector types can be constructed from lists of `dims` values of type `T`. Or, vector types can be constructed from built-in vector types of the same base type and width.

On device, the "get" method returns the base value as the relevant OpenCL built-in vector type (e.g. `float4`). The return type on host is undefined. The SYCL device compiler must support OpenCL-style swizzles on the base vector types. On device, there is a constructor which creates a `vec` type from the relevant built-in vector type.

`swizzled_vec<T, out_dims> vec<T, in_dims>::swizzle<int s1, int s2...> ()` returns the vector swizzled. The number of `s1`, `s2` parameters is the same as `out_dims`. All `s1`, `s2` parameters must be integer

Class name:	`vec`<`typename` T, `int` dims> Template parameters: `typename T`: OpenCL data type `int` dims: vector width as 1, 2, 3, 4, 8 or 16
Constructors:	`vec`<T, dims> (value_1, value_2, . . . , value_dims) `vec`<T, dims>(&`vec`<T, dims>) `vec`<T, dims>(builtin_vec_type)
Destructors:	~ `vec`<T, dims>()
Methods:	`builtin_vec_type get()`: see in text By default swizzles are supported in this format: `swizzled_vec`<T, out_dims> `swizzle`<int s1, ...>() If `SYCL_SIMPLE_SWIZZLES` is defined for 2D, 3D or 4D `swizzled_vec<T, out_dims> vec<T, in_dims>::xyzw()`

Table 3.36: Class description for `vec<typename` T, `int` dims>

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`”.

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 [**?**], and the OpenCL "khr" *extensions*, as described in the extension specification [**?**], 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 [**?**].

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 3.3.

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

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 device.

The half type class, along with any OpenCL macros and definitions, is defined in the namespace `cl::sycl::opencl::` as `half`. The vector type of half is supported sizes 2, 3, 4, 8 and 16 using the SYCL vectors (§ 3.7.2) along with all the methods supported for vectors.

The conversion rules follows the same rules as in the OpenCL 1.2 extensions specification [**?**, 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 [**?**, par. 9.5.2, 9.5.3, 9.5.4, 9.5.5].

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`).

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 cho-

Class name:	**context**
Constructors	**context**(const cl_context_properties * properties, device_selector dev_sel) Properties: 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)
Methods	device **get_gl_current_device** () VECTOR_CLASS<device> **get_gl_context_devices** ()

Table 4.2: Extensions to context for creating an OpenCL context from an OpenGL context

sen by device selector. Table 3.5 has the additions shown on Table 4.2.

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 [**?**, sec. 9.6].

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.

Class name:	`buffer<typename T, int dimensions>` Template parameters: `typename` T: The type of the elements of the buffer `int` dimensions: number of dimensions of the buffer: 1,2, or 3
Constructors:	`buffer<T, 1>(context cl_gl_context, GLuint gl_buffer_obj)`

Methods:	`cl_gl_object_type get_gl_info(GLunit & gl_object_name)`
	
	`GLenum get_gl_texture_target()`
	
	`GLint get_gl_mipmap_level()`

Table 4.3: Class description for `buffer<typename T, int dimensions>`

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.

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 [**?**, par. 9.7.3.1] and are also included in Table 4.5. If the texture or renderbuffer has a different format than the ones specified in 4.5, it is not guaranteed that the image created will be mapped to the the original texture.

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 4.6.

The SYCL system is responsible for synchronizing the OpenCL and OpenGL objects in use inside a `command_-`

Class name:	`image`<`int` dimensions> Template parameters: `int` dimensions: number of dimensions of the image: 1, 2, or 3

Constructors:	`image`<1>(`context` cl_gl_context, `GLuint` gl_buffer_obj)
	
	`image`<2>(`context` cl_gl_context, `GLuint` gl_renderbuffer_obj)
	
	`image`<1>(`context` cl_gl_context, `GLenum` texture_target, `GLuint` texture, `GLint` miplevel),
	where the *texture_target* can only be one of the following:
	`GL_TEXTURE_1D`
	`GL_TEXTURE_1D_ARRAY`
	`GL_TEXTURE_BUFFER`
	
	`image`<2>(`context` cl_gl_context, `GLenum` texture_target, `GLuint` texture, `GLint` miplevel),
	where the *texture_target* can only 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`
	
	`image`<3> (`context` cl_gl_context, `GLenum` texture_target, `GLuint` texture, `GLint` miplevel),
	where the *texture_target* can only be one of the following:
	`GL_TEXTURE_3D`

Table 4.4: Class description for `image<int` dimensions>

`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 4.7.

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 4.5: Mapping of GL internal format to CL image format (reference: [**?**, 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 4.6: Enumerator description for access::target

Class name	**event**
Constructors:	**event**(context context, GL_sync sync_obj)
Methods :	GL_sync **get_gl_info**()

Table 4.7: Class description for event

The specification of the underlying OpenCL/OpenGL interoperability system for synchronizing OpenCL event with OpenGL sync objects is in the OpenCL extensions specification [**?**, 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 [**?**, 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 3.12 page 41 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 [**?**, 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.

OpenCL C features available in SYCL compilers are only defined for devices (not the host) and are within the `cl::sycl::openclc` namespace.

The host fallback behaviour of kernels that use the features described here is undefined. Users that wish to write code that works identically on host and device should only use features specified in Chapter 3, "SYCL Programming Interface".

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 only 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. `single_task` or `parallel_for`). 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(kernel_functor<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 Language restrictions in kernels

There are no restrictions in this specification to the language features provided by the host compiler, or by host code supported by the SYCL compiler. Host code passed to the SYCL device compiler for the purposes of kernel extraction supports all standard C++ 11 features.

The following restrictions are applied to device functions and kernels:

- It is not possible to share non-POD memory objects between host and device.
- Pointers should not be shared between host and device: the behaviour for this is undefined.
- No virtual methods or function pointers are allowed to be called on device.
- No class with a vtable can be used on device.
- RTTI and exception-handling cannot be used on device.
- Recursion is not allowed on device.

- Global and static variables are not allowed in kernels.

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-POD 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.

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`. These types are not guaranteed to be the same between host and device and so should not be shared or passed as parameters.

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.

5.4 Supported data types

5.4.1 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. Developers wanting to guarantee that scalar data types match between OpenCL C code, host and device should use the `cl::sycl::cl_` scalar data types.

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 pad out structures containing pointer or `size_t` fields to match the pointer size of the host.

Scalar datatypes for a SYCL device compiler:

- `bool`: a 1-byte 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 device. This is a 32-bit unsigned integer if compiling for a 32-bit device and a 64-bit unsigned integer if compiling for a 64-bit device.

5.4.1.1 The half data type

The restrictions on the `half` datatype are defined by the underlying OpenCL implementation for the device that the kernel runs on. The `half` data type is not supported on the host and may optionally be supported on device.

5.4.2 Built-in vector types

The OpenCL C built-in vector types (e.g. `float4`) are available within the namespace `cl::sycl::openclc`. The OpenCL C `swizzle` operations are available on these types.

SYCL also provides C++ vector classes, which can be converted to and from the built-in vector types. They may be implemented on top of the built-in vector types, but they are not the same as the built-in vector types. Users wishing to have vector types that interoperate between host and device should use the vector classes defined in programming interface chapter and not the built-in ones.

5.4.3 Built-in opaque data types

The built-in opaque data types are also available within the namespace `cl::sycl::openclc`. These data types should only be used within kernels and are undefined outside of kernels. Inside kernels, these datatypes are the underlying representation of the `accessor`, `event` and `sampler` classes.

- `image2d_t`: a 2D image;
- `image3d_t`: a 3D image;
- `image2d_array_t`: a 2D image array;
- `image1d_t`: a 1D image;
- `image1d_buffer_t`: a 1D image created from a buffer object. Refer to Section 6.12.14 for a detailed description of the built-in functions that use this type;
- `image1d_array_t`: a 1D image array;

- `sampler_t`: a sampler type;
- `event_t`: an event. This can be used to identify asynchronous copies from global to local memory and vice-versa.

5.4.4 Reserved data types

The reserved data types from the OpenCL C specification are not reserved in SYCL.

5.4.5 Alignment of types

The alignment of the built-in OpenCL C data types is exactly as in OpenCL C.

A data item declared to be a data type in memory is always aligned to the size of the data type in bytes. For example, a `cl::sycl::openclc::float4` variable will be aligned to a 16-byte boundary, a `cl::sycl::openclc↩ ::char2` variable will be aligned to a 2-byte boundary.

For 3-component vector data types, the size of the data type is 4 * `sizeof`(component). This means that a 3-component vector data type will be aligned to a 4 * `sizeof`(component) boundary. The `vload3` and `vstore3` built-in functions can be used to read and write, respectively, 3-component vector data types from an array of packed scalar data type.

5.4.6 Vector literals

The vector literals from OpenCL C are supported in the SYCL device compiler for the built-in vector types.

A vector literal is written as a parenthesized vector type followed by a parenthesized comma delimited list of parameters. A vector literal operates as an overloaded function. The forms of the function that are available is the set of possible argument lists for which all arguments have the same element type as the result vector, and the total number of elements is equal to the number of elements in the result vector. In addition, a form with a single scalar of the same type as the element type of the vector is available. For example, the following forms are available for `cl::sycl::openclc::float4`:

```
(cl::sycl::openclc::float4)(float, float, float, float)

(cl::sycl::openclc::float4)(float2, float, float)

(cl::sycl::openclc::float4)(float, float2, float)

(cl::sycl::openclc::float4)(float, float, float2)

(cl::sycl::openclc::float4)(float2, float2)

(cl::sycl::openclc::float4)(float3, float)

(cl::sycl::openclc::float4)(float, float3)
```

```
(cl::sycl::openclc::float4)(float)
```

5.4.7 Vector components

The OpenCL features for assigning to and reading from vector components are available in SYCL for the built-in vector types.

Components of vectors with 1 to 4 components can be addressed with the letters `x`, `y`, `z` and `w`. Vectors with 1 to 16 components can be addressed in hexadecimal by using the letter `s`, followed by the numbers `0` to `9` and the letters `a` to `f`.

The component selection syntax allows multiple components to be selected by appending their names after the period. When applied to an lvalue, there must be no duplicate components. For an rvalue, components can be duplicated, as for example:

```
float4 test;
test.xz = test.yy;
```

Vector data types can use the `.lo` (or `.even`) and `.hi` (or `.odd`) suffixes to get smaller vector types or to combine smaller vector types to a larger vector type. Multiple levels of `.lo` (or `.even`) and `.hi` (or `.odd`) suffixes can be used until they refer to a scalar term. The `.lo` suffix refers to the lower half of a given vector. The `.hi` suffix refers to the upper half of a given vector. The `.even` suffix refers to the even elements of a vector. The `.odd` suffix refers to the odd elements of a vector.

5.5 Keywords

The only OpenCL C keywords reserved in SYCL are: `__global`, `__local`, `__constant` and `__private`. The OpenCL C `kernel` keyword is not available as kernels are defined by functors or lambdas. The OpenCL C datatypes are not keywords in SYCL, but instead defined within namespaces. The access qualifiers `__read_only`, `__write_only` and `__read_write` from OpenCL C are not available in SYCL, but are implemented using accessors instead.

The behaviour of address spaces is defined in the address spaces section 5.8 of this chapter.

SYCL also adds two new address-space keywords allowing address spaces to be deduced from pointers and applied to other pointers. The two keywords are: `__set_address_space` and `__get_address_space`.

5.6 Conversions and type casting

5.6.1 Implicit conversions

Implicit conversions between scalar built-in types (except `void` and `half`) are supported. When an implicit conversion is done, it is not just a re-interpretation of the expression's value but a conversion of that value to an equivalent value in the new type. For example, the integer value 5 will be converted to the floating-point value 5.0.

Implicit conversions between built-in vector data types are disallowed.

Implicit conversions for pointer types follow the rules described in the C++ specification, but are impacted by address space deduction (see below).

5.6.2 Explicit casts

Explicit casts between the built-in vector types are not legal and will generate a compilation error, the same rule as OpenCL C.

Scalar to vector conversions may be performed by casting the scalar to the desired built-in vector data type. Type casting will also perform appropriate arithmetic conversion. The round to zero rounding mode will be used for conversions to built-in integer vector types. The default rounding mode will be used for conversions to built-in floating-point vector types. When casting a bool to a built-in vector integer data type, the built-in vector components will be set to -1 (i.e. all bits set) if the bool value is true and 0 otherwise.

5.6.3 Explicit conversion intrinsic functions

The OpenCL C explicit conversion intrinsic functions are available within the `cl::sycl::openclc` namespace.

5.6.4 Explicit rounding mode conversions

The OpenCL C explicit rounding mode conversion intrinsic functions are available within the `cl::sycl::`↩
`openclc` namespace.

5.6.5 Saturated conversion functions

The OpenCL C saturated conversion intrinsic functions are available within the `cl::sycl::openclc` namespace.

5.6.6 Reinterpret operations with unions

The OpenCL C reinterpret operations using unions are available in SYCL.

5.6.7 Reinterpret intrinsic functions

The OpenCL C reinterpret intrinsic functions (“`as_type`”) are available within the `cl::sycl::openclc` namespace.

5.6.8 Operations and default conversions

The OpenCL C operations and default conversions are also used in SYCL for the built-in datatypes.

5.7 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.

5.7.1 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.

5.8 Address spaces in device functions

OpenCL allows pointers have address spaces. SYCL also allows address spaces on pointers as well as on C++ references. In OpenCL C, the default address space is private, but in SYCL, the default is to auto-deduce the address space in the compiler using pre-defined deduction rules specified below.

SYCL provides no guarantees of correct behaviour of code with address spaces when compiled for host. It is up to the user to ensure that device functions with address spaces compile correctly on host.

5.8.1 Defining address spaces for pointers and references

Address spaces can be defined in SYCL device functions using the keywords: `__global`, `__constant`, `__local` and `__private`. The OpenCL C address space keywords without underscores (`__global`, `__constant`, `__local` and `__private`) are not available in SYCL.

As in OpenCL C, an assignment from a value with type pointer to one address space to a variable or parameter with a pointer to a different address space is a compile-time error.

Any address space can be assigned to any pointer or reference type. Some address spaces can also be applied to some variables. Local variables in the kernel function can be declared in the `__local` address space or `__private` address space. Variables allocated in the `__local` address space inside a kernel function cannot be initialized.

All string literal storage shall be in the `__constant` address space.

5.8.2 Overloading by address space

In SYCL, two pointers to different address spaces are different datatypes. It is legal to overload function and method definitions by the address space on pointer or reference arguments. The compiler will call the correct overloaded function for the address space of the given argument.

```
void g (__local int *p)
{
    *p = 3;
}

void g (__private int *p)
{
    *p = 2;
}

void f ()
{
    single_task(kernel_functor<class my_kernel_name>([=] ()
        {
            int a; // this variable is __private
            __local int b; // this variable is __local
            g (&a);  // assigns 2 to a
            g (&b);  // assigns 3 to b
        }));
}
```

5.8.3 Address-space deduction

If a kernel function or device function contains a pointer or reference type that does not have a address space defined, then address-space deduction must be attempted using the following rules:

- If a variable is declared as a pointer type without an address-space, but initialized in its declaration to a pointer value with an address space, then that variable will have the same address space as its initializer.
- If a function parameter is declared as a pointer type without an address-space, and the argument is a pointer value with an 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 behaviour.
- The rules for pointer types also apply to reference types. i.e. a reference variable without an address space takes its address space from its initializer. A function with a reference parameter without an address space 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 `__private`. This assumption may change in future versions of the SYCL standard as new address spaces become available.

It is illegal to assign a pointer value of one address space to a pointer variable of a different address space (even if no address space was specified for that variable at declaration), unless the variable has already been deduced to have the same address space.

5.8.4 Using compile-time-variable address spaces

SYCL allows the address space of a pointer value to be converted into a compile-time constant integer using the `__get_address_space` built-in compile-time function. The actual integer values for each address space are undefined in the SYCL specification.

The integer returned from `__get_address_space` can be used in `__set_address_space` to define address spaces instead of using the explicit address-space qualifiers.

For example:

```
__global int *gpi;

__get_address_space(gp) float *gpf; // this is equivalent to __global float *gpf;
```

The integer address space specifiers can be used in templates to enable templated types to be defined for multiple different address spaces at compile time.

A. Glossary

Before describing the programming model and the implementation in detail it is important to define the key concepts involved in specifying OpenCL SYCL. This section will list of key words and how they are defined.

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 [**?**, 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.

Fall Back Host: In SYCL, kernels are compiled for both host and device. Execution of a kernel on the host is called "fall-back-host" execution. This mode of execution is intended for debugging on the host or when there are no suitable devices on a system for executing a particular kernel. Fall-back execution on the host follows the execution model of the host compiler and platform and not the OpenCL model. For users that want to execute on a CPU, but using the OpenCL execution model, an OpenCL CPU device should be chosen for execution and not the fall-back mode.

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 : 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.

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.

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 : 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.

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.