WHAT IS SYCL?

LEARNING OBJECTIVES

- Heterogeneous parallel programming
- Learn about the SYCL specification and its implementations
- Learn about the components of a SYCL implementation
- Learn about how a SYCL source file is compiled
- Learn where to find useful resources for SYCL

SUPERCOMPUTING LANDSCAPE AT EXASCALE

- Need for high levels of performance driving use of accelerators
- Many, but not all, large supercomputers using GPUs:
    - LUMI at EuroHPC JU: AMD Trento CPU and AMD MI250X GPUs (4 per node)
    - Perlmutter at NERSC: AMD EPYC Milan CPUs and NVIDIA A100 GPUs
    - Frontier at ORNL: AMD EPYC custom CPUs and Radeon Instinct GPUs (4 per node)
    - Aurora at ALCF: Intel Xeon Sapphire Rapids CPUs and Xe Ponte Vecchio GPUs (6 per node)
    - El Capitan at LLNL: AMD EPYC Genoa CPUs and Radeon Instinct GPUs (4 per node)

Multiple vendor solutions to get to Exascale

PERFORMANCE PORTABLE HETEROGENEOUS PROGRAMMING

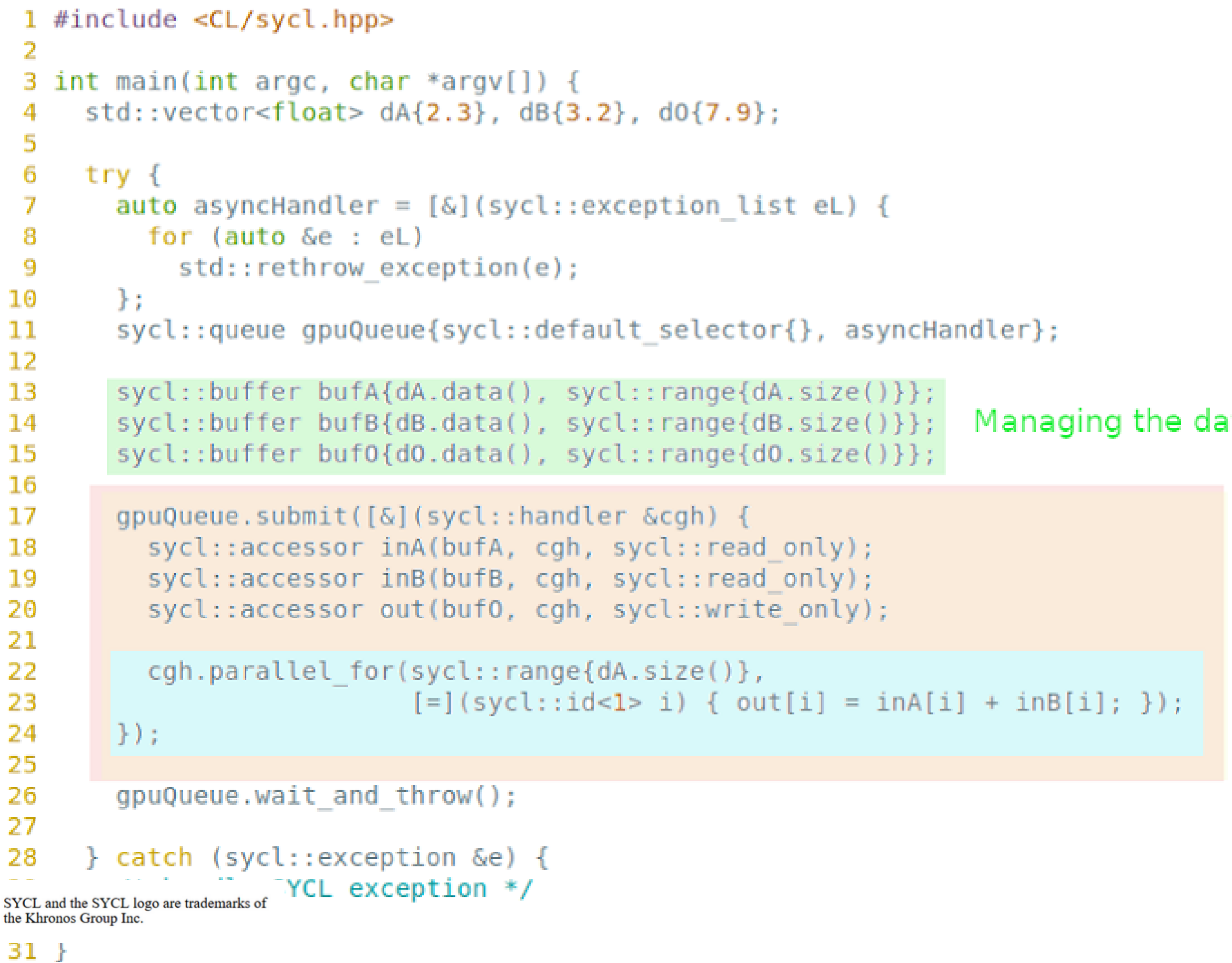
- Scientific applications need to be performant across a range of processors
- Need to write applications in (heterogeneous) parallel programming model
    - Open Standards: SYCL, OpenMP, …
    - DSLs and abstractions: Kokkos, Raja, …
    - Language parallelism: ISO C++, Fortran, …

WHAT IS SYCL?

SYCL is a single source, high-level, standard C++ programming model, that can target a range of heterogeneous platforms

WHAT IS SYCL?

A first example of SYCL code. Elements will be explained in coming sections!



```
#include <CL/sycl.hpp>

int main(int argc, char *argv[]) {
  std::vector<float> dA{2.3}, dB{3.2}, dO{7.9};

  try {
    auto asyncHandler = [&](sycl::exception_list eL) {
      for (auto &e : eL)
        std::rethrow_exception(e);
    };
    sycl::queue gpuQueue{sycl::default_selector{}, asyncHandler};

    sycl::buffer bufA{dA.data(), sycl::range{dA.size()}};
    sycl::buffer bufB{dB.data(), sycl::range{dB.size()}};
    sycl::buffer bufO{dO.data(), sycl::range{dO.size()}};

    gpuQueue.submit([&](sycl::handler &cgh) {
      sycl::accessor inA(bufA, cgh, sycl::read_only);
      sycl::accessor inB(bufB, cgh, sycl::read_only);
      sycl::accessor out(bufO, cgh, sycl::write_only);

      cgh.parallel_for(sycl::range{dA.size()},
                       [=](sycl::id<1> i) { out[i] = inA[i] + inB[i]; });
    });

    gpuQueue.wait_and_throw();

  } catch (sycl::exception &e) {
                 YCL exception */

}
```

Managing the data

Work unit

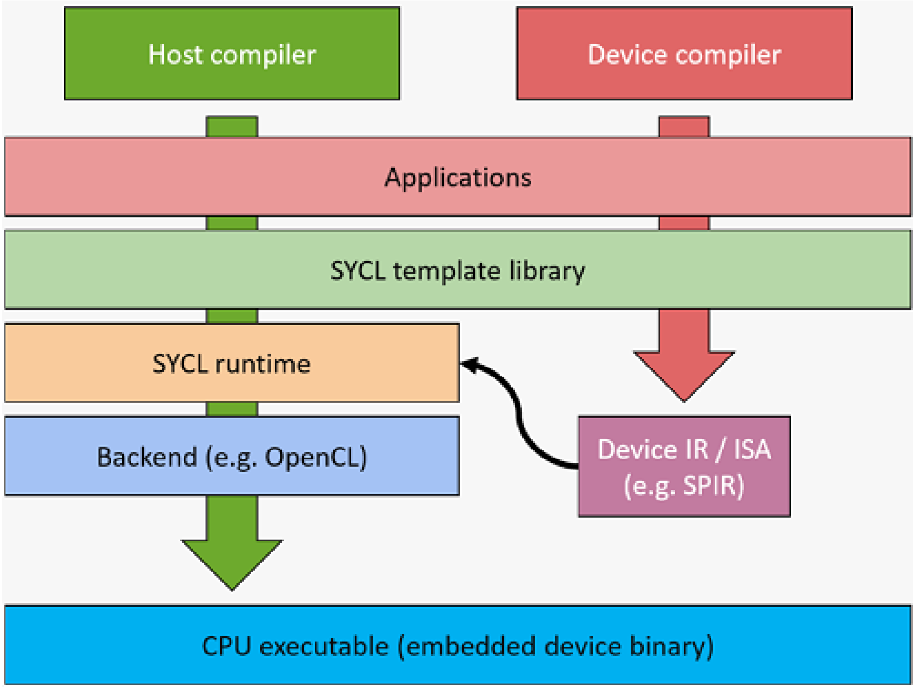
Device code

SYCL IS...

- SYCL extends C++ in two key ways:
    - heterogeneous memory
    - heterogeneous parallel compute
- SYCL is modern C++, with APIs for
    - device discovery (and information)
    - device control (kernels of work, memory)
- SYCL doesn't add extensions to the core language
- SYCL is an open standard
    - multivendor and multiarchitecture support

WHAT IS SYCL?

SYCL is a **single source**, high-level, standard C++ programming model, that can target a range of heterogeneous platforms



- SYCL allows you to write both host CPU and device code in the same C++ source file
- This requires two compilation passes; one for the host code and one for the device code

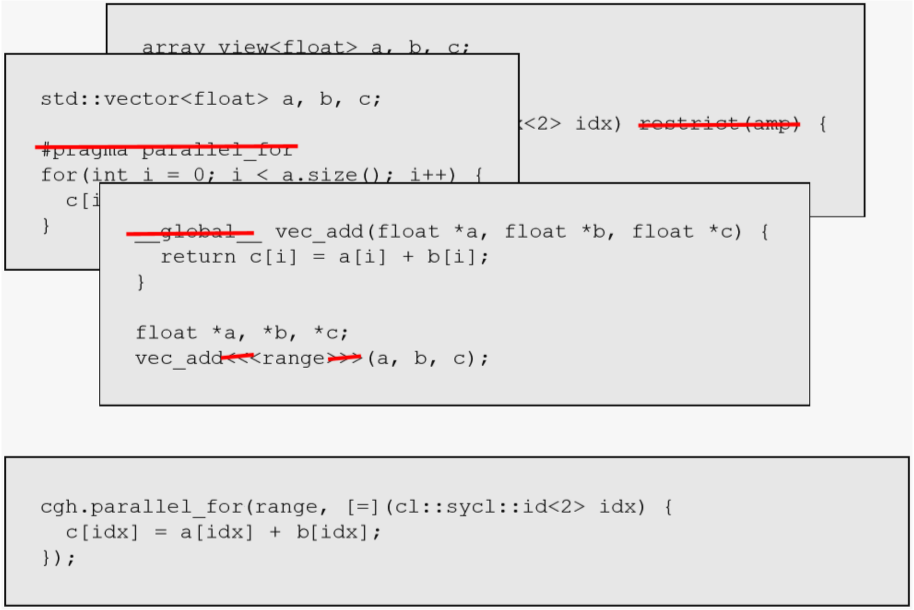
WHAT IS SYCL?

SYCL is a single source, **high-level**, standard C++ programming model, that can target a range of heterogeneous platforms

- SYCL provides high-level abstractions over common boilerplate code
    - Platform/device selection
    - Buffer creation and data movement
    - Kernel function compilation
    - Dependency management and scheduling
- High-level abstractions are good for productivity
    - SYCL has layers of abstractions when control is needed

WHAT IS SYCL?

SYCL is a single source, high-level **standard C++** programming model, that can target a range of heterogeneous platforms



```
array_view<float> a, b, c;

<2> idx) ~~restrict(amp)~~ {
```

```
std::vector<float> a, b, c;

~~#pragma parallel_for~~
for(int i = 0; i < a.size(); i++) {
  c[i
}
```

```
~~__global__~~ vec_add(float *a, float *b, float *c) {
  return c[i] = a[i] + b[i];
}

float *a, *b, *c;
vec_add~~<<<~~range~~>>>~~(a, b, c);
```

```
cgh.parallel_for(range, [=](cl::sycl::id<2> idx) {
  c[idx] = a[idx] + b[idx];
});
```

- SYCL allows you to write standard C++
    - SYCL 2020 is based on C++17
- Unlike the other implementations shown on the left there are:
    - No language extensions
    - No pragmas
    - No mandatory attributes

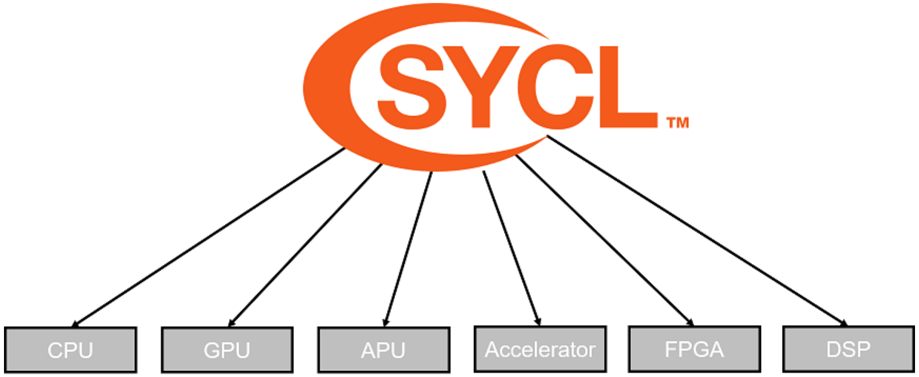
SYCL AND ISO C++

- ISO C++ has some notion of concurrency via threads and futures
- and data parallelism via algorithm and numeric libraries
- Assumes single execution space and single memory
- No control of where to run (yet)
- No asynchrony of algorithms (yet)

SYCL is aligning with and helping shape the future for heterogeneous compute in C++

WHAT IS SYCL?

SYCL is a single source, high-level standard C++ programming model, that can **target a range of heterogeneous platforms**



- SYCL can target any device supported by its backend
- SYCL can target a number of different backends

SYCL has been designed to be implemented on top of a variety of backends. Current implementations support backends such as OpenCL, CUDA, HIP, OpenMP and others.

SYCL SPECIFICATION

SYCL 1.2

- *Released Apr 2014*

SYCL 1.2.1

- *Released Dec 2017*

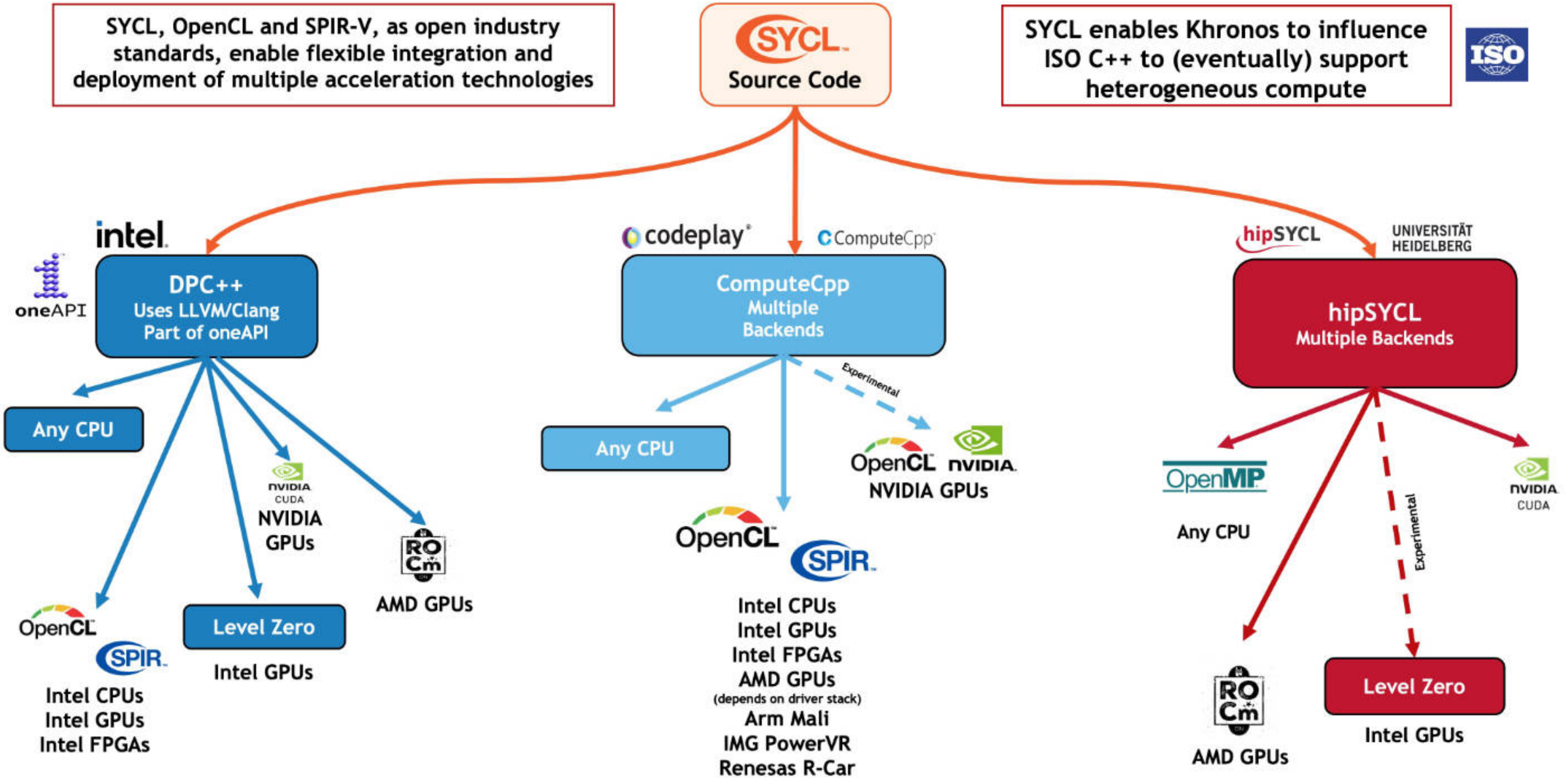
SYCL 2020 (provisional)

- *Released Jun 2020*

SYCL 2020 (final)

- *Released Feb 2021*

SYCL IMPLEMENTATIONS



SYCL, OpenCL and SPIR-V, as open industry standards, enable flexible integration and deployment of multiple acceleration technologies

SYCL Source Code

SYCL enables Khronos to influence ISO C++ to (eventually) support heterogeneous compute

DPC++
Uses LLVM/Clang
Part of oneAPI

- Any CPU
- OpenCL, SPIR: Intel CPUs, Intel GPUs, Intel FPGAs
- Level Zero: Intel GPUs
- NVIDIA CUDA: NVIDIA GPUs
- ROCm: AMD GPUs

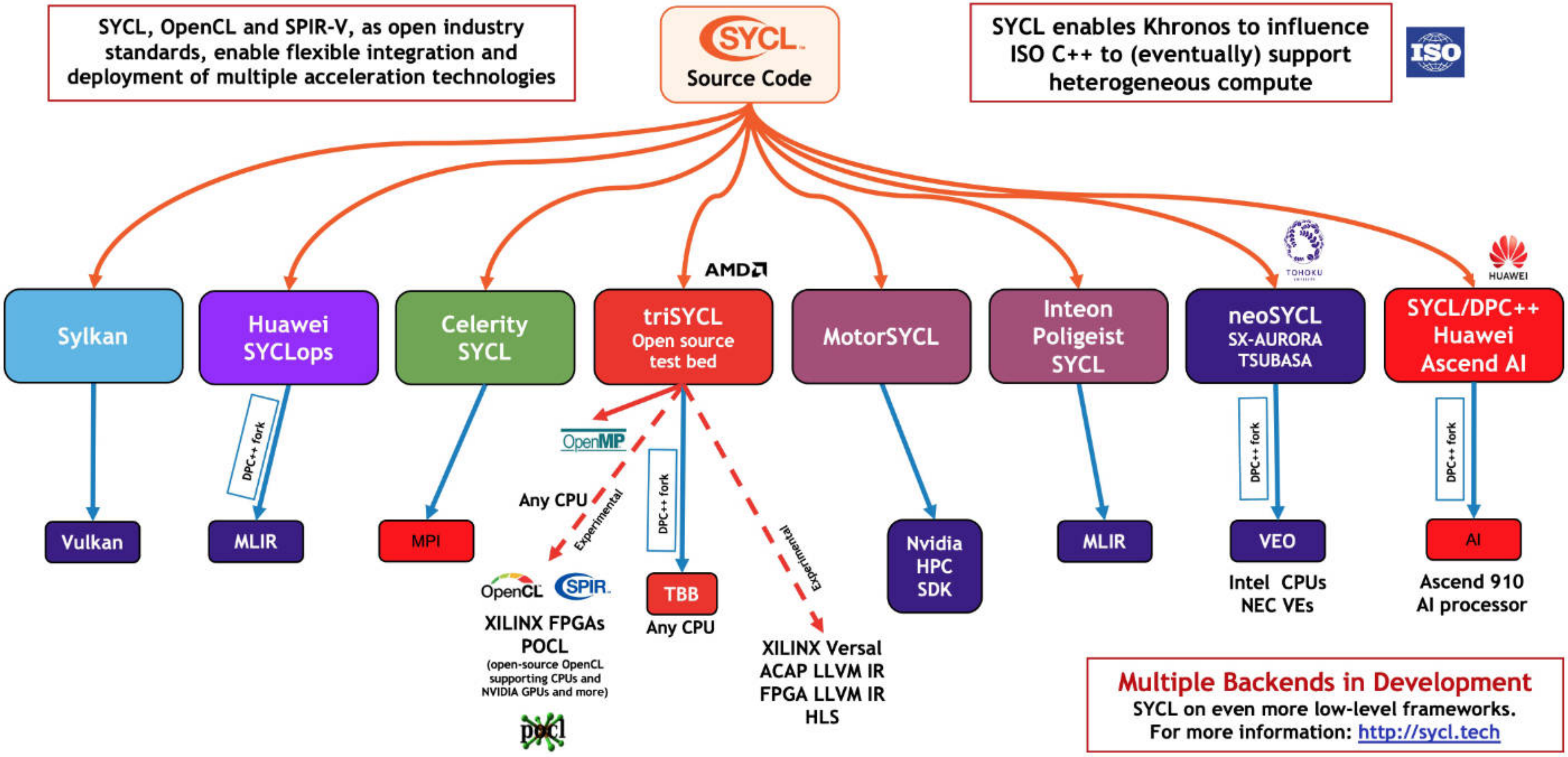
ComputeCpp
Multiple Backends

- Any CPU
- OpenCL, SPIR: Intel CPUs, Intel GPUs, Intel FPGAs, AMD GPUs (depends on driver stack), Arm Mali, IMG PowerVR, Renesas R-Car
- Experimental: OpenCL, NVIDIA: NVIDIA GPUs

hipSYCL
Multiple Backends

- OpenMP: Any CPU
- ROCm: AMD GPUs
- Experimental: Level Zero: Intel GPUs
- NVIDIA CUDA

SYCL IMPLEMENTATIONS



SYCL, OpenCL and SPIR-V, as open industry standards, enable flexible integration and deployment of multiple acceleration technologies

SYCL Source Code

SYCL enables Khronos to influence ISO C++ to (eventually) support heterogeneous compute

Sylkan → Vulkan

Huawei SYCLops → MLIR (DPC++ fork)

Celerity SYCL → MPI

triSYCL Open source test bed (AMD)
- OpenMP → Any CPU
- Experimental → OpenCL, SPIR: XILINX FPGAs, POCL (open-source OpenCL supporting CPUs and NVIDIA GPUs and more)
- TBB (DPC++ fork) → Any CPU
- Experimental → XILINX Versal ACAP LLVM IR, FPGA LLVM IR, HLS

MotorSYCL → Nvidia HPC SDK

Inteon Poligeist SYCL → MLIR

neoSYCL SX-AURORA TSUBASA (TOHOKU) → VEO (DPC++ fork): Intel CPUs, NEC VEs

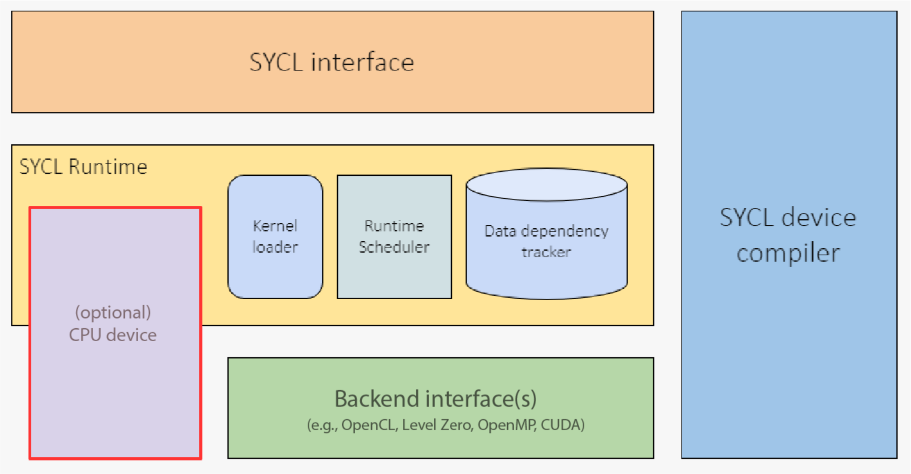
SYCL/DPC++ Huawei Ascend AI (HUAWEI) → AI (DPC++ fork): Ascend 910 AI processor

Multiple Backends in Development
SYCL on even more low-level frameworks.
For more information: http://sycl.tech

IMPLEMENTATIONS OF A STANDARD

- SYCL is a *standard*
- Document defines behaviour of API:
    - Platform, device model
    - Memory and execution model
    - What the APIs are and what they do
- Implementations (like DPC++, hipSYCL, etc) *implement* the standard
    - Once conformant, guarenteed all APIs are supported by the implementation

WHAT A SYCL IMPLEMENTATION LOOKS LIKE



- The SYCL interface is a C++ template library that developers can use to access the features of SYCL
- The same interface is used for both the host and device code

- The host is generally the CPU and is used to dispatch the parallel execution of kernels
- The device is the parallel unit used to execute the kernels, such as a GPU

WHERE TO GET STARTED WITH SYCL

- Visit **https://sycl.tech** to find out about all the SYCL book, implementations, tutorials, news, and videos
- Visit **https://www.khronos.org/sycl/** to find the latest SYCL specifications
- Checkout the documentation provided with one of the SYCL implementations.

QUESTIONS

EXERCISE

Code_Exercises/Exercise_1_Compiling_with_SYCL/source.cpp

Configure your environment for using SYCL and compile a source file with the SYCL compiler.

Task: Include the SYCL header and successfully build and run a binary.

INTEL DEVCLOUD

1. Register for the Intel DevCloud
2. Follow instructions to set up SSH

https://devcloud.intel.com/oneapi/documentation/

DEVCLOUD DEMO

```
$ ssh devcloud
$ git clone https://github.com/illuhad/syclacademy -b cluster22 --recursive
$ cd syclacademy
$ mkdir build
$ dpcpp -fsycl -o sycl-ex-1 ../Code_Exercises/Exercise_01_Compiling_with_SYCL/source.cpp
$ qsub -I -l nodes=1:gpu:ppn=2 -d .
$ ./sycl-ex-1
```

ENQUEUING A KERNEL

LEARNING OBJECTIVES

- Learn about queues and how to submit work to them
- Learn how to compose command groups
- Learn how to define kernel functions
- Learn about the rules and restrictions on kernel functions
- Learn how to stream text from a kernel function to the console.

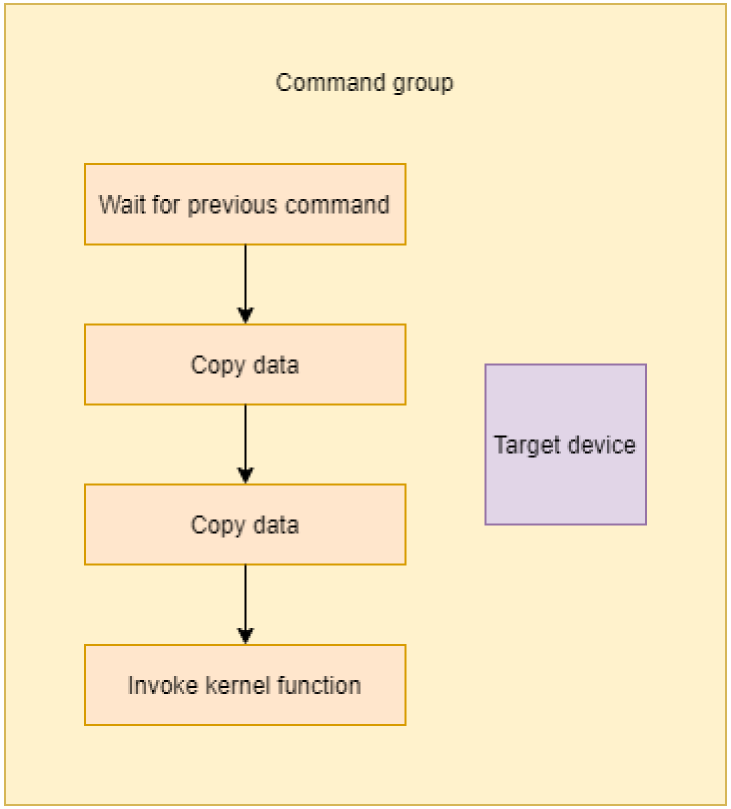
THE QUEUE

- In SYCL all work is submitted via commands to a `queue`.
- The `queue` has an associated device that any commands enqueued to it will target.
- There are several different ways to construct a `queue`.
- The most straight forward is to default construct one.
- This will have the SYCL runtime choose a device for you.

PRECURSOR

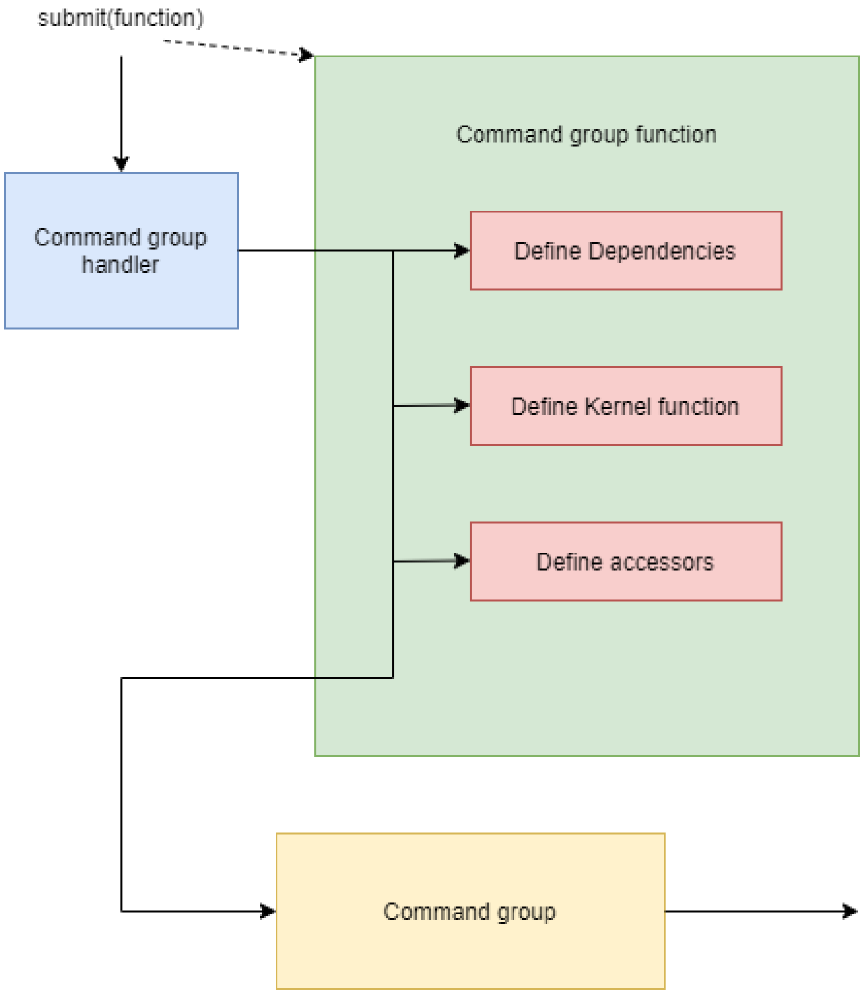
- In SYCL there are two models for managing data:
    - The buffer/accessor model.
    - The USM (unified shared memory) model.
- Which model you choose can have an effect on how you enqueue kernel functions.
- For now we are going to focus on the buffer/accessor model.

COMMAND GROUPS



- In the buffer/accessor model commands must be enqueued via command groups.
- A command group represents a series of commands to be executed by a device.
- These commands include:
  - Invoking kernel functions on a device.
  - Copying data to and from a device.
  - Waiting on other commands to complete.

COMPOSING COMMAND GROUPS



- Command groups are composed by calling the `submit` member function on a `queue`.
- The `submit` function takes a command group function which acts as a factory for composing the command group.
- The `submit` function creates a `handler` and passes it into the command group function.
- The `handler` then composes the command group.

COMPOSING COMMAND GROUPS

```
gpuQueue.submit([&](handler &cgh){

  /* Command group function */

});
```

- The `submit` member function takes a C++ function object, which takes a reference to a `handler`.
- The function object can be a lambda expression or a class with a function call operator.
- The body of the function object represents the command group function.

COMPOSING COMMAND GROUPS

```
gpuQueue.submit([&](handler &cgh){

  /* Command group function */

});
```

- The command group function is processed exactly once when `submit` is called.
- At this point all the commands and requirements declared inside the command group function are processed to produce a command group.
- The command group is then submitted asynchronously to the scheduler.

COMPOSING COMMAND GROUPS

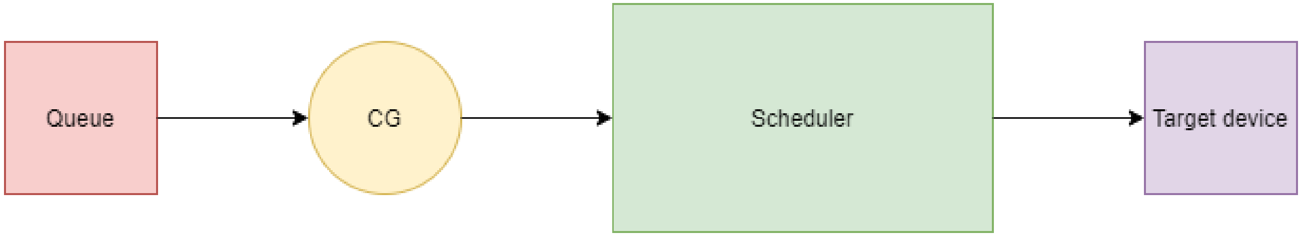
```
gpuQueue.submit([&](handler &cgh){

  /* Command group function */

}).wait();
```

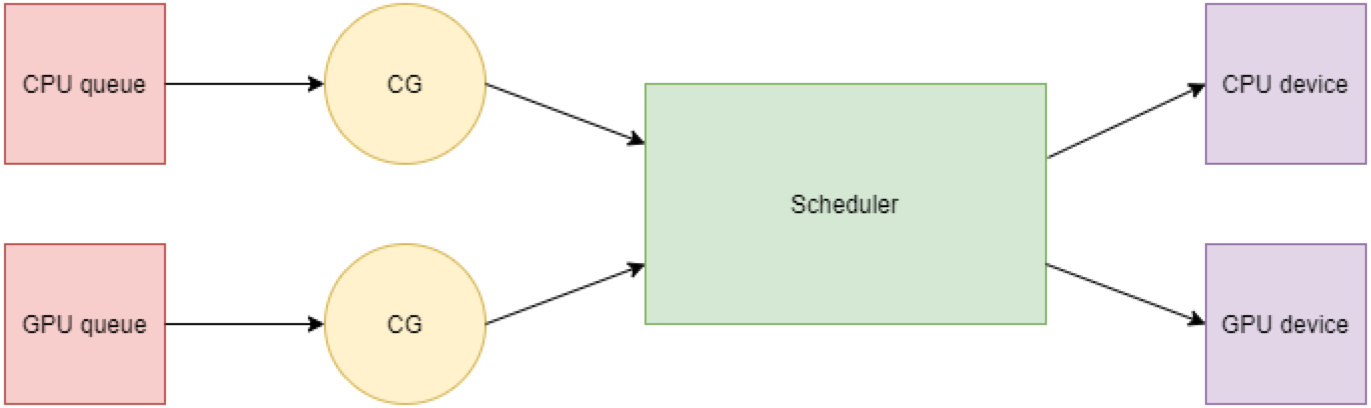
- The `queue` will not wait for commands to complete on destruction.
- However `submit` returns an `event` to allow you to synchronize with the completion of the commands.
- Here we call `wait` on the `event` to immediately wait for it complete.
- There are other ways to do this, that will be covered in later lectures.

SCHEDULING



- Once `submit` has created a command group it will submit it to the scheduler.
- The scheduler will then execute the commands on the target device once all dependencies and requirements are satisfied.

SCHEDULING



- The same scheduler is used for all queues.
- This allows sharing dependency information.

```
class my_kernel;

gpuQueue.submit([&](handler &cgh){

  cgh.single_task<my_kernel>([=]() {
    /* kernel code */
  });
}).wait();
```

- The kernel function invoke APIs take a function object representing the kernel function.
- This can be a lambda expression or a class with a function call operator.
- This is the entry point to the code that is compiled to execute on the device.

```
class my_kernel;

gpuQueue.submit([&](handler &cgh){

  cgh.single_task<my_kernel>([=]() {
    /* kernel code */
  });
}).wait();
```

- Different kernel invoke APIs take different parameters describing the iteration space to be invoked in.
- Different kernel invoke APIs can also expect different arguments to be passed to the function object.
- The `single_task` function describes a kernel function that is invoked exactly once, so there are no additional parameters or arguments.

```
class my_kernel;

gpuQueue.submit([&](handler &cgh){

  cgh.single_task<my_kernel>([=]() {
    /* kernel code */
  });
}).wait();
```

- The template parameter passed to `single_task` is used to name the kernel function.
- This is necessary when defining kernel functions with lambdas to allow the host and device compilers to communicate.
- SYCL 2020 allows kernel lambdas to be unnamed, but not all implementations support that yet.

SYCL KERNEL FUNCTION RULES

- Must be defined using a C++ lambda or function object, they cannot be a function pointer or std::function.
- Must always capture or store members by-value.
- SYCL kernel functions declared with a lambda must be named using a forward declarable C++ type, declared in global scope.
- SYCL kernel function names follow C++ ODR rules, which means you cannot have two kernels with the same name.

SYCL KERNEL FUNCTION RESTRICTIONS

- No dynamic allocation
- No dynamic polymorphism
- No function pointers
- No recursion

KERNELS AS FUNCTION OBJECTS

```
class my_kernel;

queue gpuQueue;
gpuQueue.submit([&](handler &cgh){

  cgh.single_task<my_kernel>([=]() {
    /* kernel code */
  });
}).wait();
```

- All the examples of SYCL kernel functions up until now have been defined using lambda expressions.

KERNELS AS FUNCTION OBJECTS

```
struct my_kernel {
  void operator()(){
    /* kernel function */
  }
};
```

- As well as defining SYCL kernels using lambda expressions, You can also define a SYCL kernel using a regular C++ function object.

KERNELS AS FUNCTION OBJECTS

```
struct my_kernel {
  void operator()(){
    /* kernel function */
  }
};
```

```
queue gpuQueue;
gpuQueue.submit([&](handler &cgh){

  cgh.single_task(my_kernel{});
}).wait();
```

- To use a C++ function object you simply construct an instance of the type and pass it to `single_task`.
- Notice you no longer need to name the SYCL kernel.

STREAMS

- A `stream` can be used in a kernel function to print text to the console from the device, similarly to how you would with `std::cout`.
- The `stream` is a buffered output stream so the output may not appear until the kernel function is complete.
- The `stream` is useful for debugging, but should not be relied on in performance critical code.

STREAMS

```
stream::stream(size_t bufferSize, size_t workItemBufferSize, handler &cgh);
```

- A `stream` must be constructed in the command group function, as a `handler` is required.
- The constructor also takes a `size_t` parameter specifying the total size of the buffer that will store the text.
- It also takes a second `size_t` parameter specifying the work-item buffer size.
- The work-item buffer size represents the cache that each invocation of the kernel function (in the case of `single_task` 1) has for composing a stream of text.

STREAMS

```
class my_kernel;

queue gpuQueue;
gpuQueue.submit([&](handler &cgh){

  auto os = sycl::stream(1024, 128, cgh);

  cgh.single_task<my_kernel>([=]() {
    /* kernel code */
  });
}).wait();
```

- Here we construct a `stream` in our command group function with a buffer size of `1024` and a work-item size of `128`.
- This means that the total text that the stream can receive is 1024 bytes.

STREAMS

```
class my_kernel;

queue gpuQueue;
gpuQueue.submit([&](handler &cgh){

  auto os = sycl::stream(1024, 128, cgh);

  cgh.single_task<my_kernel>([=]() {
    os << "Hello world!\n";
  });
}).wait();
```

- Next we capture the `stream` in the kernel function's lambda expression.
- Then we can print `"Hello World!"` to the console using the << operator.
- This is where the work-item size comes in, this is the cache available to store text on the right-hand-size of the << operator.

ENQUEUING SYCL KERNEL FUNCTIONS

```
class my_kernel;

gpuQueue.submit([&](handler &cgh){

  cgh.single_task<my_kernel>([=]() {
    /* kernel code */
  });
}).wait();
```

- SYCL kernel functions are defined using one of the kernel function invoke APIs provided by the `handler`.
- These add a SYCL kernel function command to the command group.
- There can only be one SYCL kernel function command in a command group.
- Here we use `single_task`.

QUESTIONS

EXERCISE

Code_Exercises/Exercise_2_Hello_World/source

Implement a SYCL application which enqueues a kernel function to a device and streams "Hello world!" to the console.

MANAGING DATA

LEARNING OBJECTIVES

- Learn about the buffer/accessor model and USM for managing data
- Learn how to use these
- Learn how to use data in a kernel function
- Learn how to synchronize data

MEMORY MODELS

- In SYCL there are two models for managing data:
  - The buffer/accessor model.
  - The USM (unified shared memory) model.
- Which model you choose can have an effect on how you enqueue kernel functions.

SYCL BUFFERS & ACCESSORS

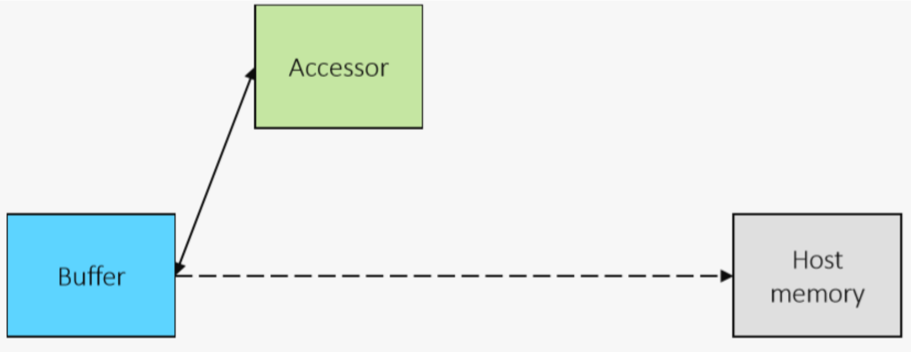
- SYCL separates the storage and access of data
    - A SYCL buffer manages data across the host and any number of devices
    - A SYCL accessor requests access to data on the host or on a device for a specific SYCL kernel function
- Accessors are also used to access data within a SYCL kernel function
    - This means they are declared in the host code but captured by and then accessed within a SYCL kernel function

SYCL BUFFERS & ACCESSORS

- A SYCL buffer can be constructed with a pointer to host memory
- For the lifetime of the buffer this memory is owned by the SYCL runtime
- When a buffer object is constructed it will not allocate or copy to device memory at first
- This will only happen once the SYCL runtime knows the data needs to be accessed and where it needs to be accessed

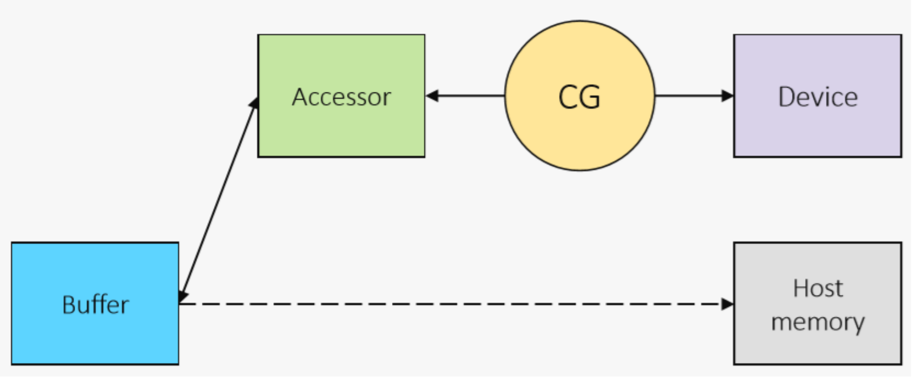
SYCL BUFFERS & ACCESSORS

- Constructing an accessor specifies a request to access the data managed by the buffer
- There are a range of different types of accessor which provide different ways to access data



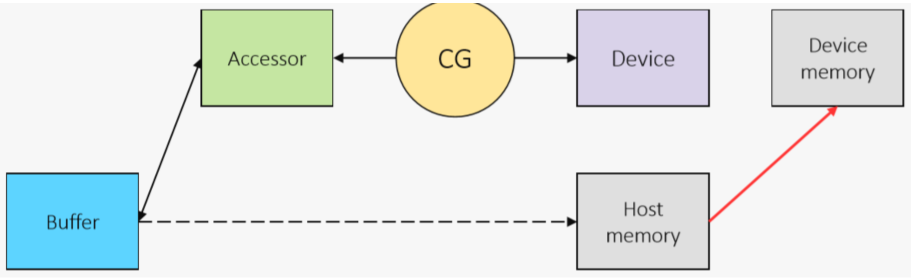
SYCL BUFFERS & ACCESSORS

- When an accessor is constructed it is associated with a command group via the handler object
- This connects the buffer that is being accessed, the way in which it's being accessed and the device that the command group is being submitted to



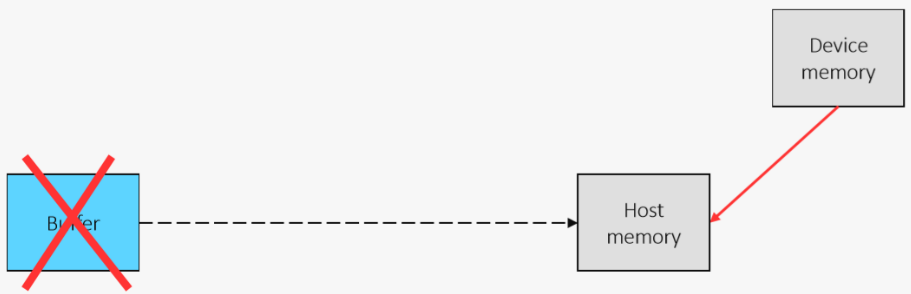
SYCL BUFFERS & ACCESSORS

- Once the SYCL scheduler selects the command group to be executed it must first satisfy its data dependencies
- This means allocating and copying data to the device the data is being accessed on if necessary
- If the most recent copy of the data is already on the device then the runtime will not copy again



SYCL BUFFERS & ACCESSORS

- Data will remain in device memory after kernels finish executing until another command group requests access in a different device or on the host
- When the buffer object is destroyed it will wait for any outstanding work that is accessing the data to complete and then copy back to the original host memory



BUFFER CLASS

```
template <typename dataT, int dimensions>
sycl::buffer;
```

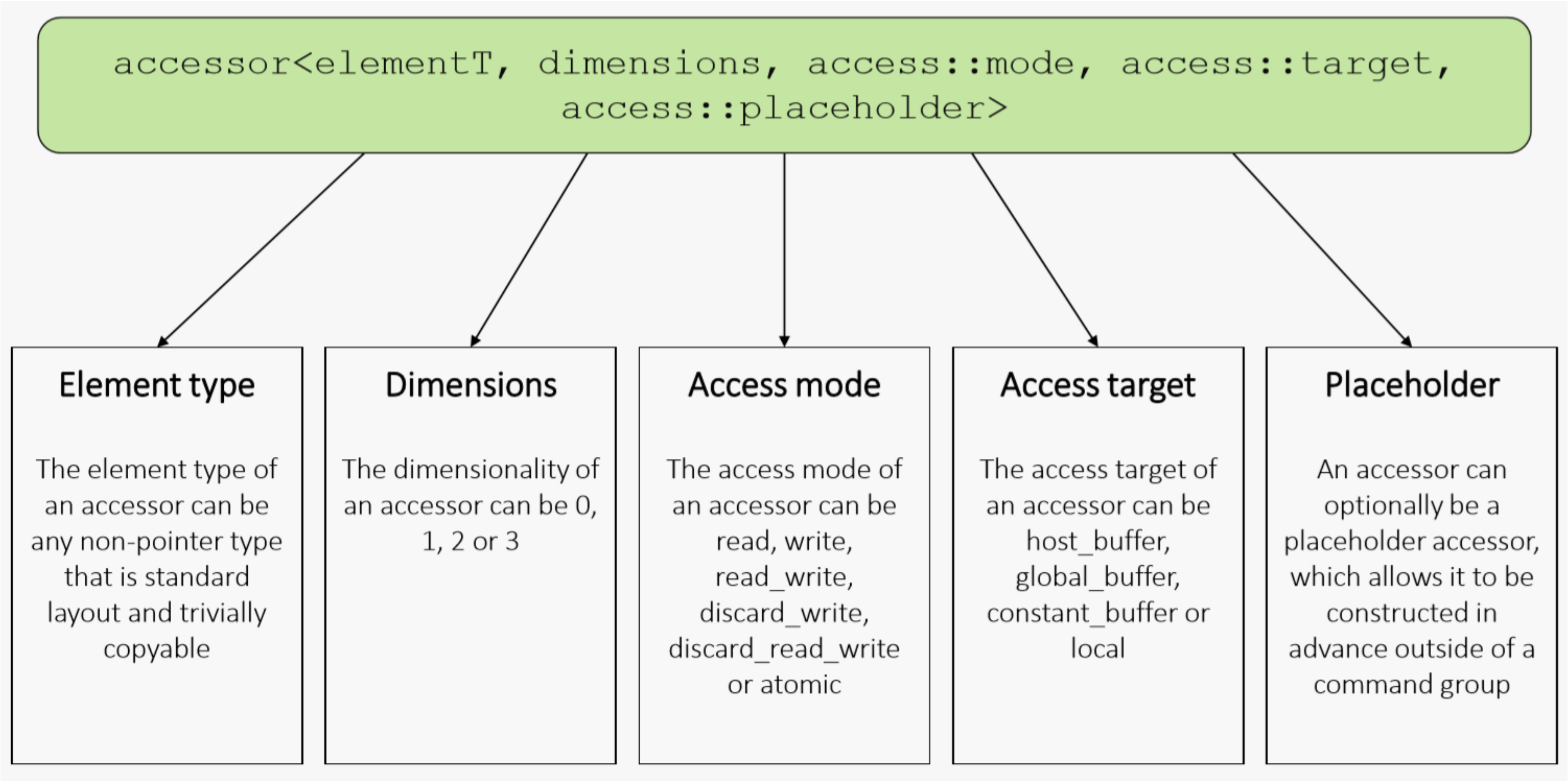
- A `buffer` manages data across the host application and kernel functions executing on device(s).
- It has a typename which specifies the type of the elements of data it manages.
- It has a dimensionality which specifies the dimensionality that the elements of data are represented in.

CONSTRUCTING A BUFFER

```
int var = 42;
auto buf = sycl::buffer{&var, sycl::range{1}};
```

- A `buffer` can be constructed from a pointer to data for it to manage and a `range` which describes the number of elements of data.
- Using CTAD the type and the dimensionality can be inferred.

ACCESSOR CLASS



`accessor<elementT, dimensions, access::mode, access::target, access::placeholder>`

Element type

The element type of an accessor can be any non-pointer type that is standard layout and trivially copyable

Dimensions

The dimensionality of an accessor can be 0, 1, 2 or 3

Access mode

The access mode of an accessor can be read, write, read_write, discard_write, discard_read_write or atomic

Access target

The access target of an accessor can be host_buffer, global_buffer, constant_buffer or local

Placeholder

An accessor can optionally be a placeholder accessor, which allows it to be constructed in advance outside of a command group

ACCESSOR CLASS

- There are many different ways to use the `accessor` class.
    - Accessing data on a device.
    - Accessing data immediately in the host application.
    - Allocating local memory.
- For now we are going to focus on accessing data on a device.

CONSTRUCTING AN ACCESSOR

```
auto acc = sycl::accessor{bufA, cgh};
```

- There are many ways to construct an `accessor`.
- The `accessor` class supports CTAD so it's not nessesary to specify all of the template arguments.
- The most common way to construct an `accessor` is from a `buffer` and a `handler` associated with the command group function you are within.
  - The element type and dimensionality are infered from the `buffer`.
  - The `access::target` is defaulted to `access::target::global_buffer`.
  - The `access::mode` is defaulted to `access::mode::read_write`.

SPECIFYING THE ACCESS MODE

```
auto readAcc = sycl::accessor{bufA, cgh, sycl::read_only};
auto writeAcc = sycl::accessor{bufB, cgh, sycl::write_only};
```

- When constructing an `accessor` you will likely also want to specify the `access::mode`
- You can do this by passing one of the CTAD tags:
  - `read_only` will result in `access::mode::read`.
  - `write_only` will result in `access::mode::write`.

SPECIFYING NO INITIALIZATION

```
auto acc = sycl::accessor{buf, cgh, sycl::no_init};
```

- When constructing an `accessor` you may also want to discard the original data of a `buffer`.
- You can do this by passing the `no_init` property.

ACCESS MODES

- A **read** `accessor` instructs the SYCL runtime that the SYCL kernel function will read the data – cannot be written to within a SYCL kernel function.
- A **write** `accessor` instructs the SYCL runtime that the SYCL kernel function will modify the data – creating a dependency for future command groups.
- A **no_init** `accessor` instructs the SYCL runtime that the SYCL kernel function does not need the initial values of the data – removing the dependency on previous command groups.

ACCESSOR RESOLUTION

- If a command group has more than one `accessor` to the same `buffer` with conflicting `access::mode` they are resolved into one:
    - read & write => read_write.
- If a command group has more than one `accessor` to the same `buffer` all must have the `no_init` property for it to apply.
- Within the SYCL kernel function there are still multiple `accessors`, but they alias to the same memory address.

ACCESSOR RESOLUTION

```
gpuQueue.submit([&](handler &cgh){
  auto in = sycl::accessor{buf, cgh, sycl::read_only
  auto out = sycl::accessor{buf, cgh, sycl::write_on
});
```

- Here `in` and `out` both point to `buf` but one is `access::mode::read` and one is `access::mode::write`.
- So the SYCL runtime will treat them both as `access::mode::read_write`.
- Both will point to a single allocation of global memory on the device(s).
- The runtime will resolve the data dependency into `access::mode::read_write`.

OPERATOR[]

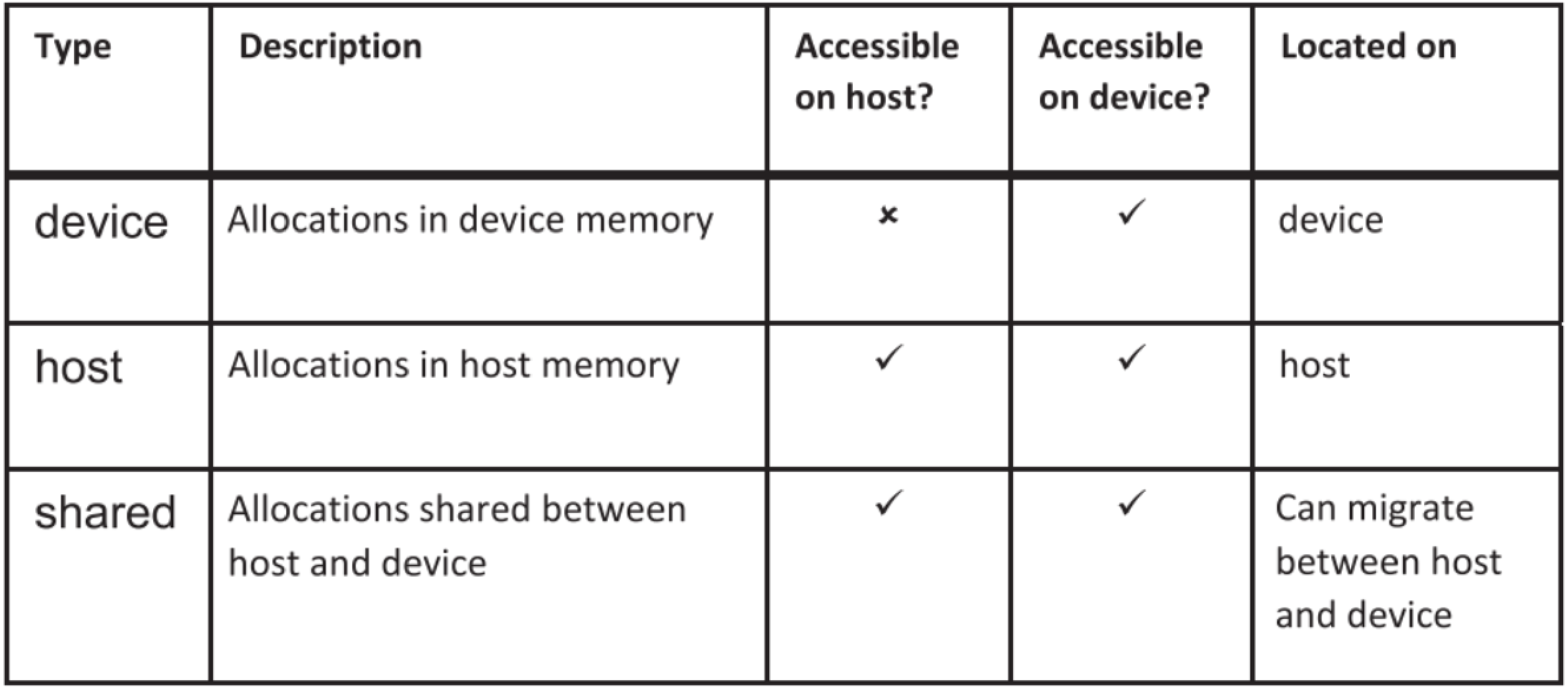
```
gpuQueue.submit([&](handler &cgh){
  auto inA = sycl::accessor{bufA, cgh, sycl::read_only};
  auto inB = sycl::accessor{bufB, cgh, sycl::read_only};
  auto out = sycl::accessor{bufO, cgh, sycl::write_only};
  cgh.single_task<add>([=]{
    out[0] = inA[0] + inB[0];
  });
});
```

- As well as specifying data dependencies an `accessor` can also be used to access the data from within a kernel function.
- You can do this by calling `operator[]` on the `accessor`.
    - This operator can take an `id` or a `size_t`.

USM MODEL

- There are different ways USM memory can be allocated; host, device and shared.
- We're going to focus on shared and device allocations.

USM ALLOCATION TYPES



Type	Description	Accessible on host?	Accessible on device?	Located on
device	Allocations in device memory	×	✓	device
host	Allocations in host memory	✓	✓	host
shared	Allocations shared between host and device	✓	✓	Can migrate between host and device

***Figure 6-1.** USM allocation types*

(from book)

MALLOC_DEVICE

```
void* malloc_device(size_t numBytes, const queue& syclQueue, const property_list &propList = {});

template <typename T>
T* malloc_device(size_t count, const queue& syclQueue,  const property_list &propList = {});
```

- A USM device allocation is performed by calling one of the `malloc_device` functions.
- Both of these functions allocate the specified region of memory on the `device` associated with the specified `queue`.
- The pointer returned is only accessible in a kernel function running on that `device`.
- Synchronous exception if the device does not have aspect::usm_device_allocations.
- This is a blocking operation.
- Calls the underlying `cudaMalloc` if using CUDA backend.

MALLOC_SHARED

```
void* malloc_shared(size_t numBytes, const queue& syclQueue, const property_list &propList = {});

template <typename T>
T* malloc_shared(size_t count, const queue& syclQueue,  const property_list &propList = {});
```

- Both of these functions allocate the specified region of memory on the `device` associated with the specified `queue`, as well as host.
- The pointer returned is accessible in CPU code as well as `device` kernel code, for the `device` attached to the `queue`.
- Synchronous exception if the device does not have aspect::usm_device_allocations
- This is a blocking operation.
- Calls the underlying `cudaMallocManaged` if using CUDA backend.
- Convenient API but potentially slower than `malloc_device` with explicit `memcpys`.

FREE

```
void free(void* ptr, queue& syclQueue);
```

- In order to prevent memory leaks USM device allocations must be free by calling the `free` function.
- The `queue` must be the same as was used to allocate the memory.
- This is a blocking operation.

MEMCPY

```
event queue::memcpy(void* dest, const void* src, size_t numBytes, const std::vector &depEvents);
```

- Data can be copied to and from a USM device allocation by calling the `queue`'s `memcpy` member function.
- The source and destination can be either a host application pointer or a USM device allocation.
- This is an asynchronous operation enqueued to the `queue`.
- An `event` is returned which can be used to synchronize with the completion of copy operation.
- May depend on other events via `depEvents`

MEMSET & FILL

```
event queue::memset(void* ptr, int value, size_t numBytes, const std::vector &depEvents);

event queue::fill(void* ptr, const T& pattern, size_t count, const std::vector &depEvents);
```

- The additional `queue` member functions `memset` and `fill` provide operations for initializing the data of a USM device allocation.
- The member function `memset` initializes each byte of the data with the value interpreted as an unsigned char.
- The member function `fill` initializes the data with a recurring pattern.
- These are also asynchronous operations.

EXERCISE

Code_Exercises/Exercise_03_Scalar_Add

Implement a SYCL application that adds two variables and returns the result using USM and Buffers.

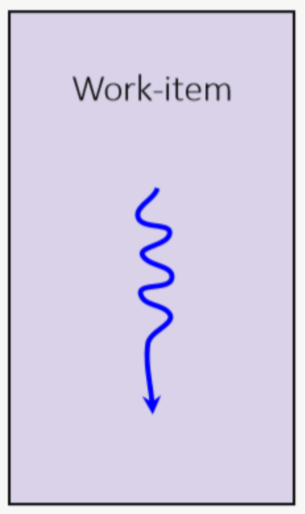
ND RANGE KERNELS

LEARNING OBJECTIVES

- Learn about the SYCL execution and memory model
- Learn how to enqueue an nd-range kernel functions

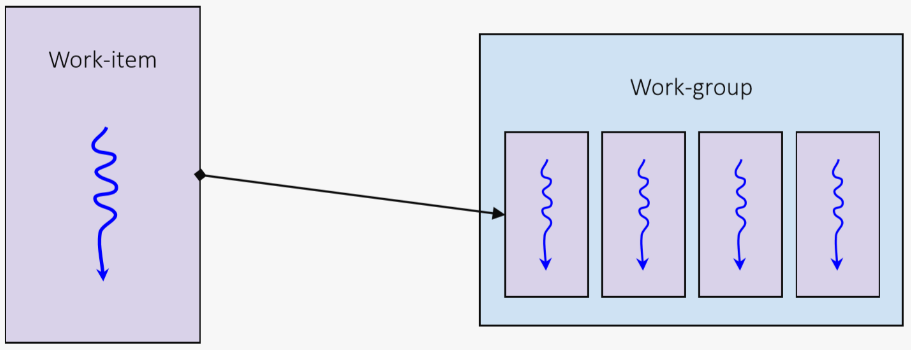
SYCL EXECUTION MODEL

- SYCL kernel functions are executed by **work-items**
- You can think of a work-item as a thread of execution
- Each work-item will execute a SYCL kernel function from start to end
- A work-item can run on CPU threads, SIMD lanes, GPU threads, or any other kind of processing element



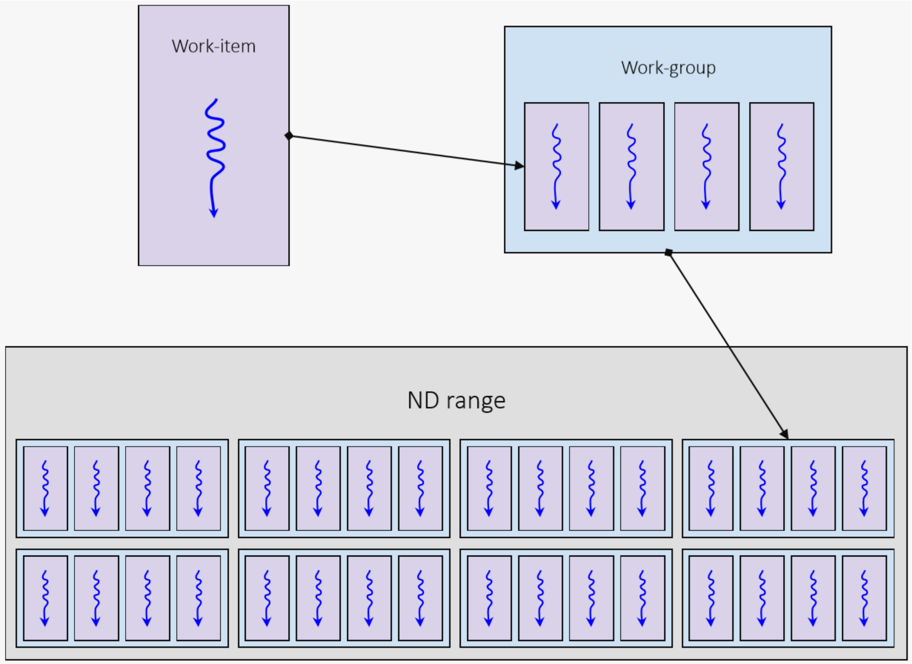
SYCL EXECUTION MODEL

- Work-items are collected together into **work-groups**
- The size of work-groups is generally relative to what is optimal on the device being targeted
- It can also be affected by the resources used by each work-item



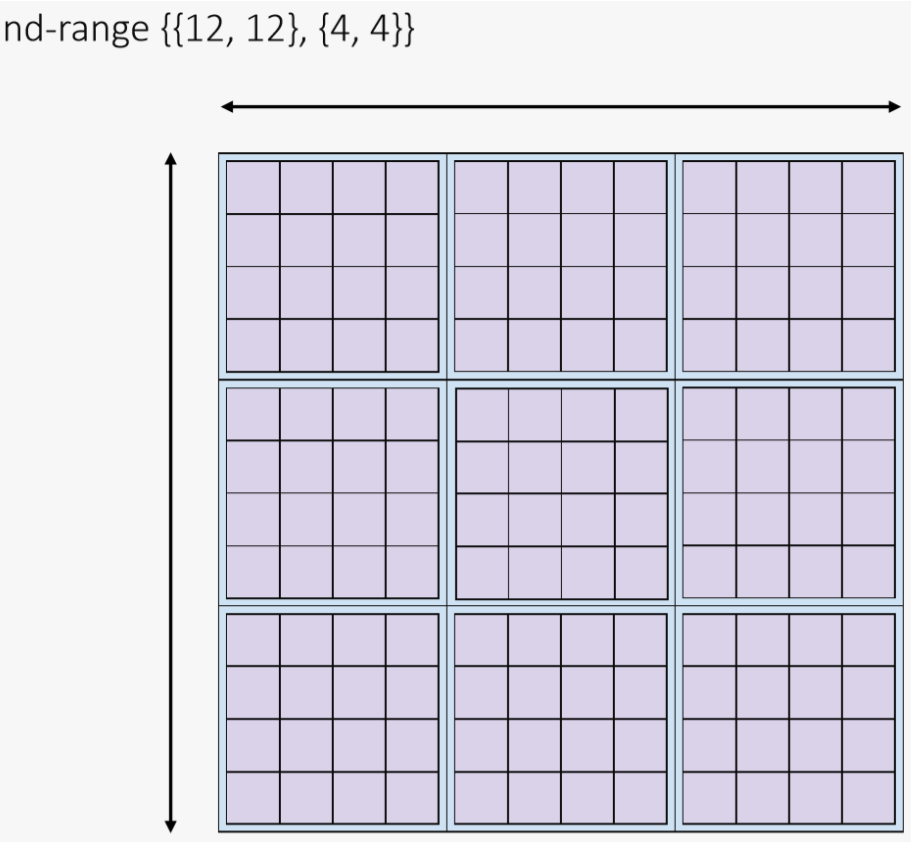
SYCL EXECUTION MODEL

- SYCL kernel functions are invoked within an **nd-range**
- An nd-range has a number of work-groups and subsequently a number of work-items
- Work-groups always have the same number of work-items



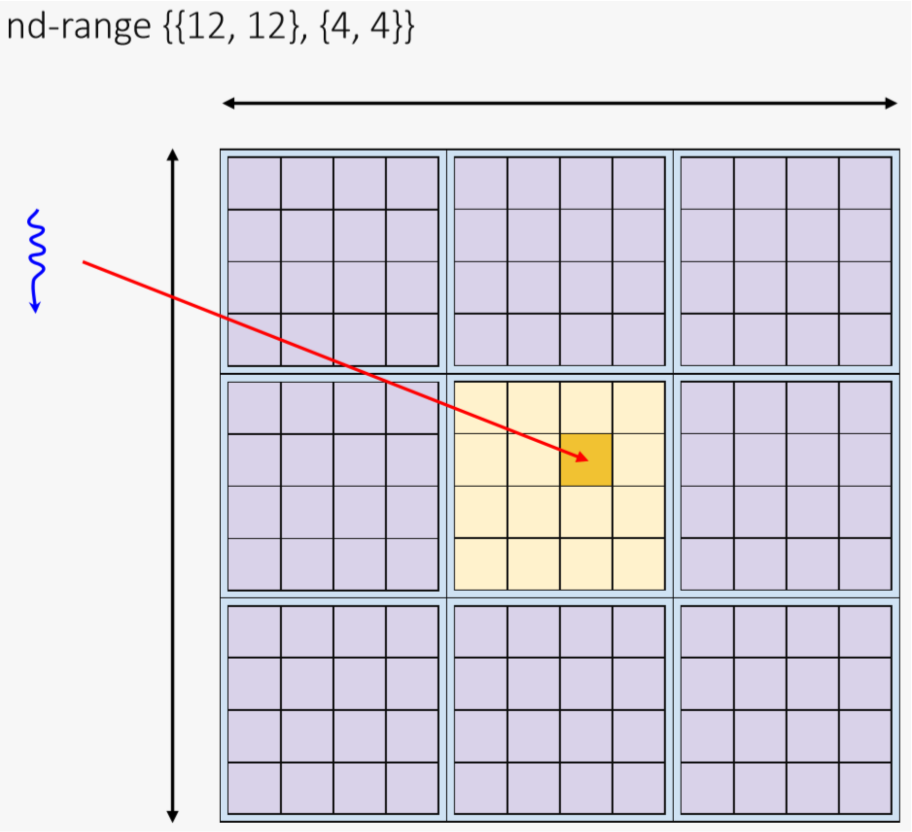
SYCL EXECUTION MODEL

- The nd-range describes an **iteration space**; how the work-items and work-groups are composed
- An nd-range can be 1, 2 or 3 dimensions
- An nd-range has two components
    - The **global-range** describes the total number of workitems in each dimension
    - The **local-range** describes the number of work-items in a work-group in each dimension



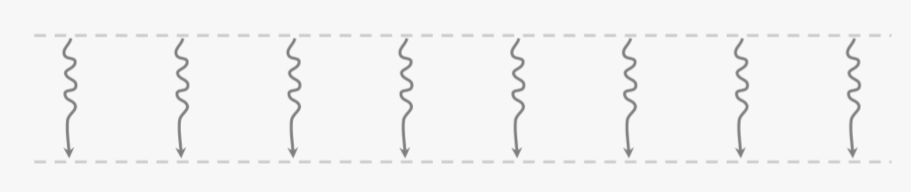
SYCL EXECUTION MODEL

- Each invocation in the iteration space of an nd-range is a work-item
- Each invocation knows which work-item it is on and can query certain information about its position in the nd-range
- Each work-item has the following:
  - **Global range**: {12, 12}
  - **Global id**: {5, 6}
  - **Group range**: {3, 3}
  - **Group id**: {1, 1}
  - **Local range**: {4, 4}
  - **Local id**: {1, 2}



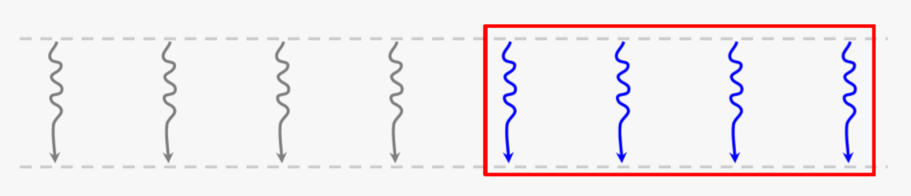
SYCL EXECUTION MODEL

Typically an nd-range invocation SYCL will execute the SYCL kernel function on a very large number of work-items, often in the thousands



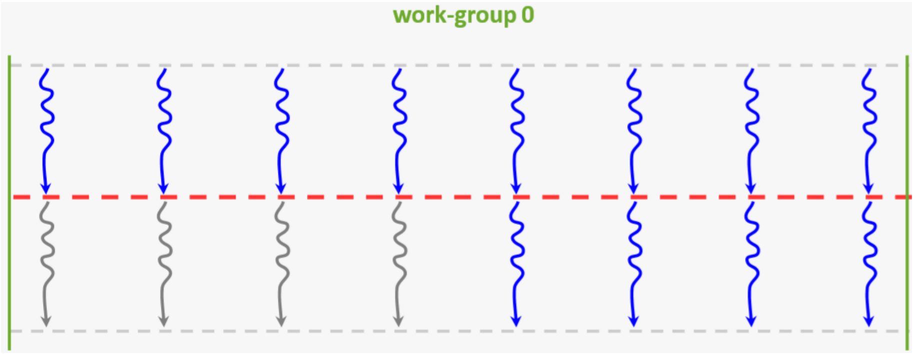
SYCL EXECUTION MODEL

- Multiple work-items will generally execute concurrently
- On vector hardware this is often done in lock-step, which means the same hardware instructions
- The number of work-items that will execute concurrently can vary from one device to another
- Work-items will be batched along with other work-items in the same work-group
- The order work-items and workgroups are executed in is implementation defined



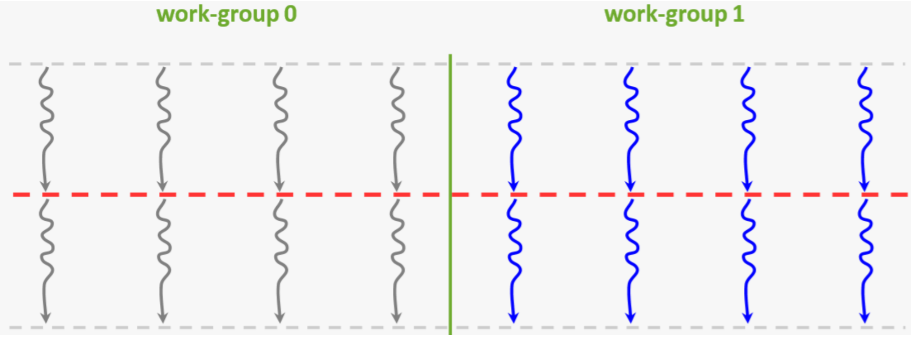
SYCL EXECUTION MODEL

- Work-items in a work-group can be synchronized using a work-group barrier
    - All work-items within a work-group must reach the barrier before any can continue on



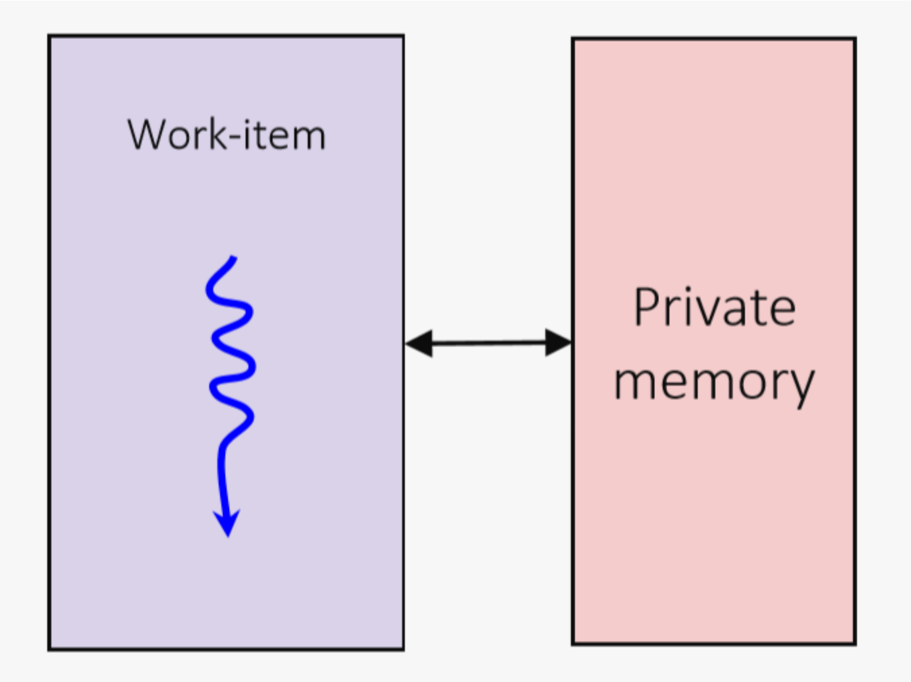
SYCL EXECUTION MODEL

- SYCL does not support synchronizing across all work-items in the nd-range
- The only way to do this is to split the computation into separate SYCL kernel functions



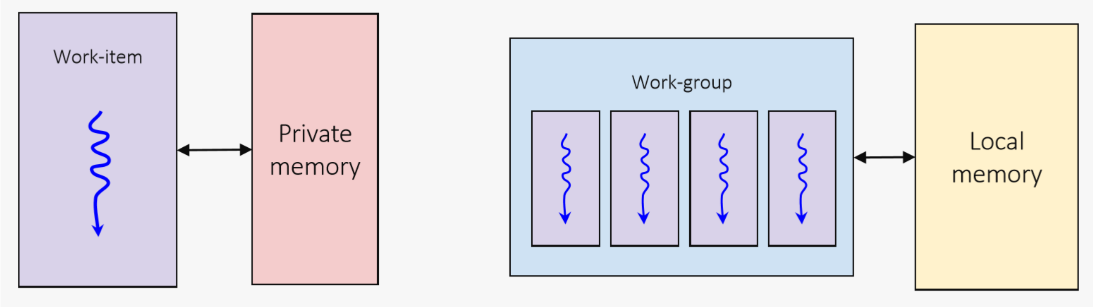
SYCL MEMORY MODEL

- Each work-item can access a dedicated region of **private memory**
- A work-item cannot access the private memory of another work-item



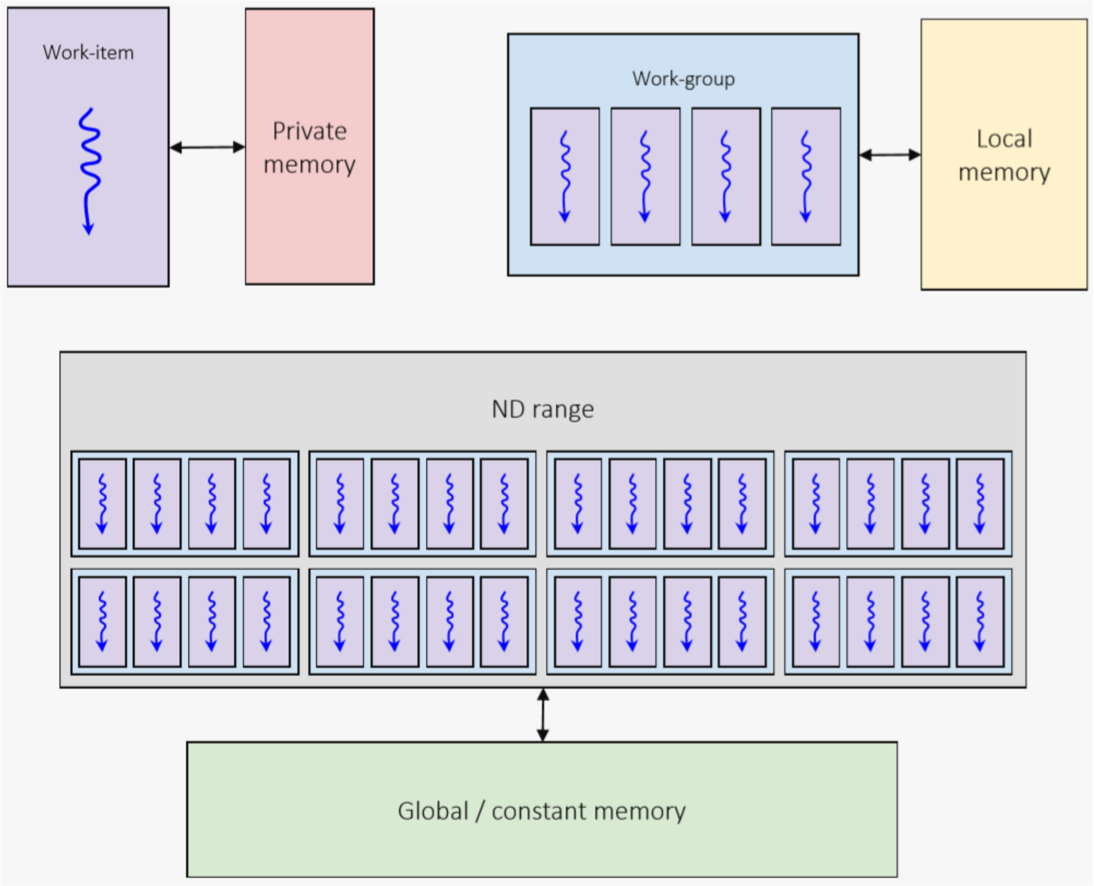
SYCL and the SYCL logo are trademarks of the Khronos Group Inc.

SYCL MEMORY MODEL



- Each work-item can access a dedicated region of **local memory** accessible to all work-items in a work-group
- A work-item cannot access the local memory of another workgroup

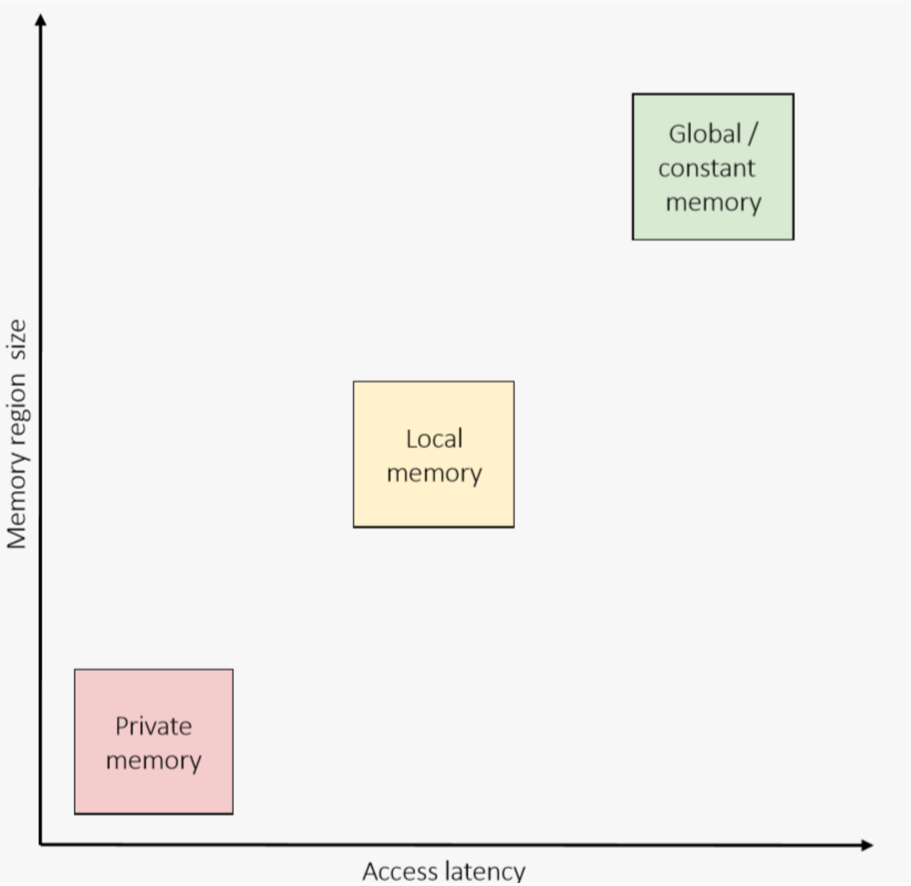
SYCL MEMORY MODEL



- Each work-item can access a single region of **global memory** that's accessible to all work-items in a ND-range
- Each work-item can also access a region of global memory reserved as **constant memory**, which is read-only

SYCL MEMORY MODEL

- Each memory region has a different size and access latency
- Global / constant memory is larger than local memory and local memory is larger than private memory
- Private memory is faster than local memory and local memory is faster than global / constant memory



EXPRESSING PARALLELISM

```
cgh.parallel_for<kernel>(range<1>(1024),
  [=](id<1> idx){
    /* kernel function code */
});
```

- Overload taking a **range** object specifies the global range, runtime decides local range
- An **id** parameter represents the index within the global range

---

```
cgh.parallel_for<kernel>(range<1>(1024),
  [=](item<1> item){
    /* kernel function code */
});
```

- Overload taking a **range** object specifies the global range, runtime decides local range
- An **item** parameter represents the global range and the index within the global range

```
cgh.parallel_for<kernel>(nd_range<1>(range<1>(1024),
  range<1>(32)),[=](nd_item<1> ndItem){
                unction code */
```

ACCESSING DATA WITH ACCESSORS

- There are a few different ways to access the data represented by an accessor
    - The subscript operator can take an **id**
        - Must be the same dimensionality of the accessor
        - For dimensions > 1, linear address is calculated in row major
- Nested subscript operators can be called for each dimension taking a **size_t**
    - E.g. a 3-dimensional accessor: acc[x][y][z] = …
- A pointer to memory can be retrieved by calling **get_pointer**
    - This returns a **multi_ptr**, which is a wrapper class for pointers to the memory in the relevant memory space

ACCESSING DATA WITH ACCESSORS

```
buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

gpuQueue.submit([&](handler &cgh){
  auto inA = bufA.get_access<access::mode::read>(cgh);
  auto inB = bufB.get_access<access::mode::read>(cgh);
  auto out = bufO.get_access<access::mode::write>(cgh);
  cgh.parallel_for<add>(range<1>(dA.size()),
    [=](id<1> i){
    out[i] = inA[i] + inB[i];
  });
});
```

- Here we access the data of the `accessor` by passing in the `id` passed to the SYCL kernel function.

ACCESSING DATA WITH ACCESSORS

```
buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

gpuQueue.submit([&](handler &cgh){
  auto inA = bufA.get_access<access::mode::read>(cgh);
  auto inB = bufB.get_access<access::mode::read>(cgh);
  auto out = bufO.get_access<access::mode::write>(cgh);
  cgh.parallel_for<add>(rng, [=](id<3> i){
    auto ptrA = inA.get_pointer();
    auto ptrB = inB.get_pointer();
    auto ptrO = out.get_pointer();
    auto linearId = i.get_linear_id();

    ptrA[linearId] = ptrB[linearId] + ptrO[linearId];
  });
});
```

- Here we retrieve the underlying pointer for each of the `accessors`.
- We then access the pointer using the linearized `id` by calling the `get_linear_id` member function on the `item`.
- Again this linearization is calculated in row-major order.

QUESTIONS

EXERCISE

Code_Exercises/Exercise_14_ND_Range_Kernel/source

Implement a SYCL application that will perform a vector add using `parallel_for`, adding multiple elements in parallel.

IMAGE CONVOLUTION

LEARNING OBJECTIVES

- Learn about image convolutions and what makes them a good problem for solving on a GPU
- Learn what a naive image convolution may look like

IMAGE CONVOLUTION

Over the next few lectures we will be looking at some common GPU optimizations with an image convolution as the motivational example.

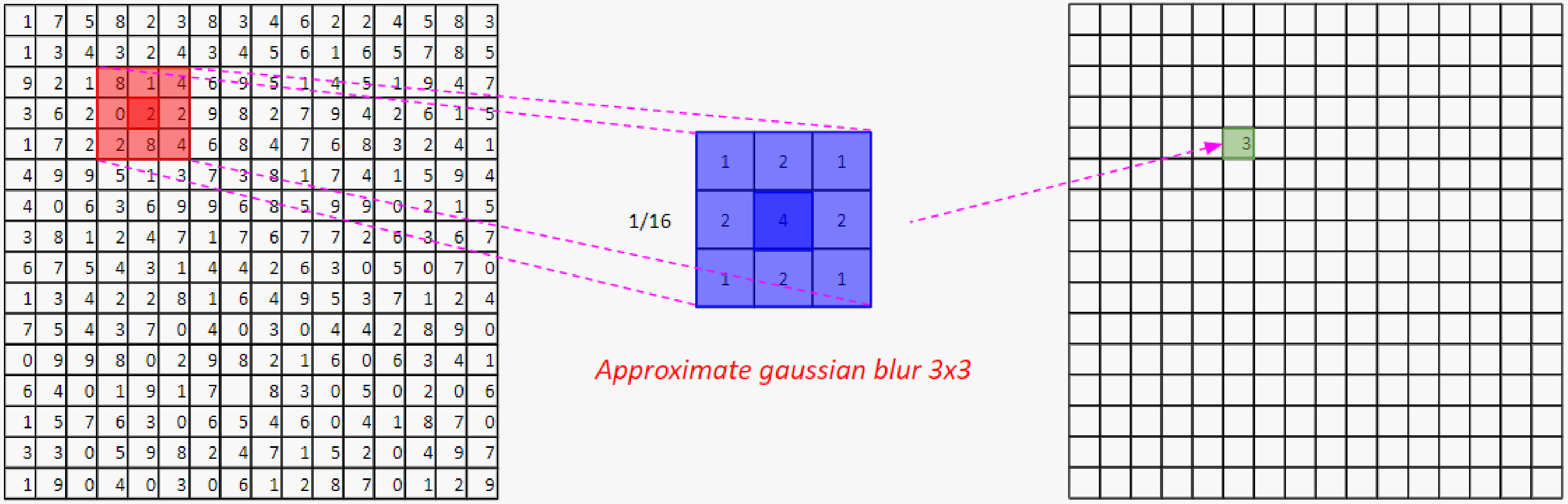
- A good problem to solve on a GPU.
- Can take advantage of a number of common optimizations.
- Convolution is a very powerful algorithm with many applications.
- Deep neural networks.
- Image processing.

WHY ARE IMAGE CONVOLUTIONS GOOD ON A GPU?

- The algorithm is **embarrassingly parallel**.
- Each work-item in the computation can be calculated entirely independently.
- The algorithm is computation heavy.
- A large number of operations are performed for each work-item in the computation, particularly when using large filters.
- The algorithm requires a large bandwidth.
- A lot of data must be passed through the GPU to process an image, particularly if the image is very high resolution.

IMAGE CONVOLUTION DEFINITION

$$G = h \otimes F \qquad G[i,j] = \sum_{u=-k}^{k} \sum_{v=-k}^{k} h[u,v]F[i+u, j+v]$$



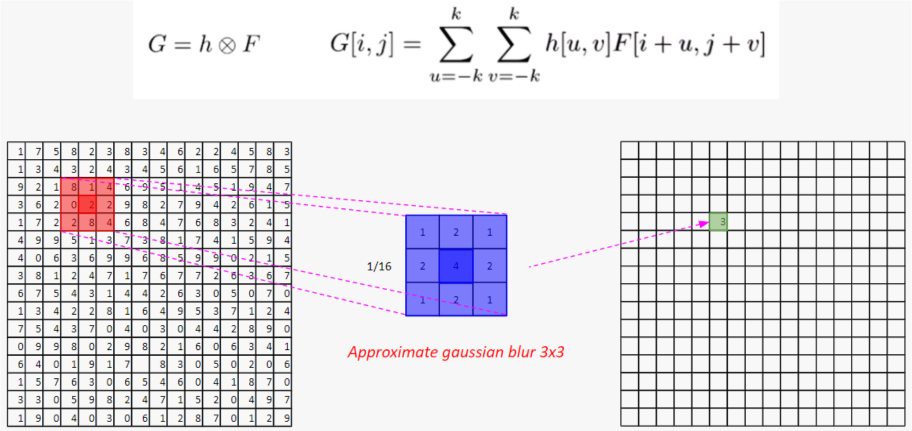
1	7	5	8	2	3	8	3	4	6	2	2	4	5	8	3
1	3	4	3	2	4	3	4	5	6	1	6	5	7	8	5
9	2	1	8	1	4	6	9	5	1	4	5	1	9	4	7
3	6	2	0	2	2	9	8	2	7	9	4	2	6	1	5
1	7	2	2	8	4	6	8	4	7	6	8	3	2	4	1
4	9	9	5	1	3	7	3	8	1	7	4	1	5	9	4
4	0	6	3	6	9	9	6	8	5	9	9	0	2	1	5
3	8	1	2	4	7	1	7	6	7	7	2	6	3	6	7
6	7	5	4	3	1	4	4	2	6	3	0	5	0	7	0
1	3	4	2	2	8	1	6	4	9	5	3	7	1	2	4
7	5	4	3	7	0	4	0	3	0	4	4	2	8	9	0
0	9	9	8	0	2	9	8	2	1	6	0	6	3	4	1
6	4	0	1	9	1	7		8	3	0	5	0	2	0	6
1	5	7	6	3	0	6	5	4	6	0	4	1	8	7	0
3	3	0	5	9	8	2	4	7	1	5	2	0	4	9	7
1	9	0	4	0	3	0	6	1	2	8	7	0	1	2	9

1/16

1	2	1
2	4	2
1	2	1

Approximate gaussian blur 3x3

IMAGE CONVOLUTION DEFINITION



$$G = h \otimes F \qquad G[i,j] = \sum_{u=-k}^{k} \sum_{v=-k}^{k} h[u,v]F[i+u,j+v]$$

1	7	5	8	2	3	8	3	4	6	2	2	4	5	8	3
1	3	4	3	2	4	3	4	5	6	1	6	5	7	8	5
9	2	1	8	1	4	6	9	5	1	4	5	1	9	4	7
3	6	2	0	2	2	9	8	2	7	9	4	2	6	1	5
1	7	2	2	8	4	6	8	4	7	6	8	3	2	4	1
4	9	9	5	1	3	7	3	8	1	7	4	1	5	9	4
4	0	6	3	6	9	9	6	8	5	9	9	0	2	1	5
3	8	1	2	4	7	1	7	6	7	7	2	6	3	6	7
6	7	5	4	3	1	4	4	2	6	3	0	5	0	7	0
1	3	4	2	2	8	1	6	4	9	5	3	7	1	2	4
7	5	4	3	7	0	4	0	3	0	4	4	2	8	9	0
0	9	9	8	0	2	9	8	2	1	6	0	6	3	4	1
6	4	0	1	9	1	7		8	3	0	5	0	2	0	6
1	5	7	6	3	0	6	5	4	6	0	4	1	8	7	0
3	3	0	5	9	8	2	4	7	1	5	2	0	4	9	7
1	9	0	4	0	3	0	6	1	2	8	7	0	1	2	9

1/16

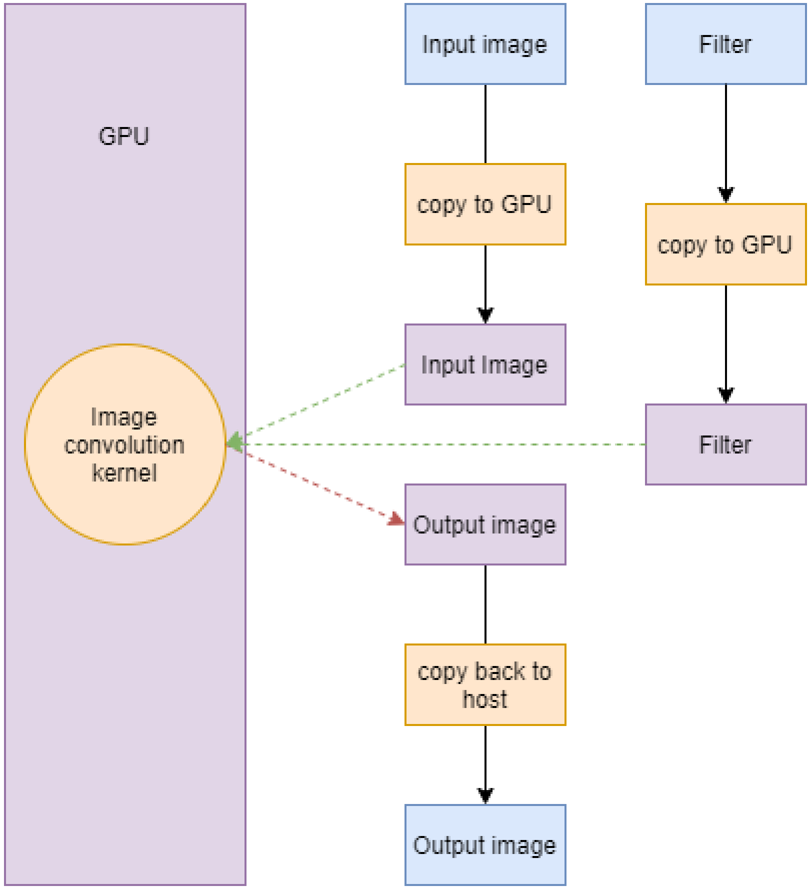
1	2	1
2	4	2
1	2	1

Approximate gaussian blur 3x3

- A filter of a given size is applied as a stencil to the position of each pixel in the input image.
- Each pixel covered by the filter is then multiples with the corresponding element in the filter.
- The result of these multiplications is then summed to give the resulting output pixel.
- Here we have a 3x3 gaussian blur approximation as an example.

IMAGE CONVOLUTION EXAMPLE

IMAGE CONVOLUTION DATA FLOW



- We have a single kernel function.
- It must read from the input image data and writes to the output image data.
- It must also read from the filter.
- The input image data and the filter don't need to be copied back to the host.
- The output image data can be uninitialized.

IMPLEMENTATION

- We provide a naive implementation of a SYCL application which implements the image convolution algorithm.
- This will be the basis for optimization in later lectures and exercises.
- The implementation uses the stb image library to allow us to visualize our results.
- The implementation also uses a benchmark function to allow us to measure the performance as we make optimizations.

REFERENCE IMAGE

- We provide a reference image to use in the exercise.
- This is in Code_Exercises/Images
- This image is a 512x512 RGBA png.
- Feel free to use your own image but we recommend keeping to this format.

INPUT/OUTPUT IMAGE LOCATIONS

```
auto inputImageFile = "../Code_Exercises/Images/dogs.png";
auto outputImageFile = ../Code_Exercises/Images/blurred_dogs.png";
```

- The reference code and the solutions to the remaining exercises use these strings to refernce the location of the input and output image.
- Before compiling these you will have to update this to point to the image in the development environment.

CONVOLUTION FILTERS

```
auto filter = util::generate_filter(util::filter_type filterType, int width);
```

- The utility for generating the filter data takes a `filter_type` enum which can be either `identity` or `blur` and a width.
- Feel free to experiment with different variations.
- Note that the filter width should always be an odd value.

NAIVE IMAGE CONVOLUTION PERFORMANCE

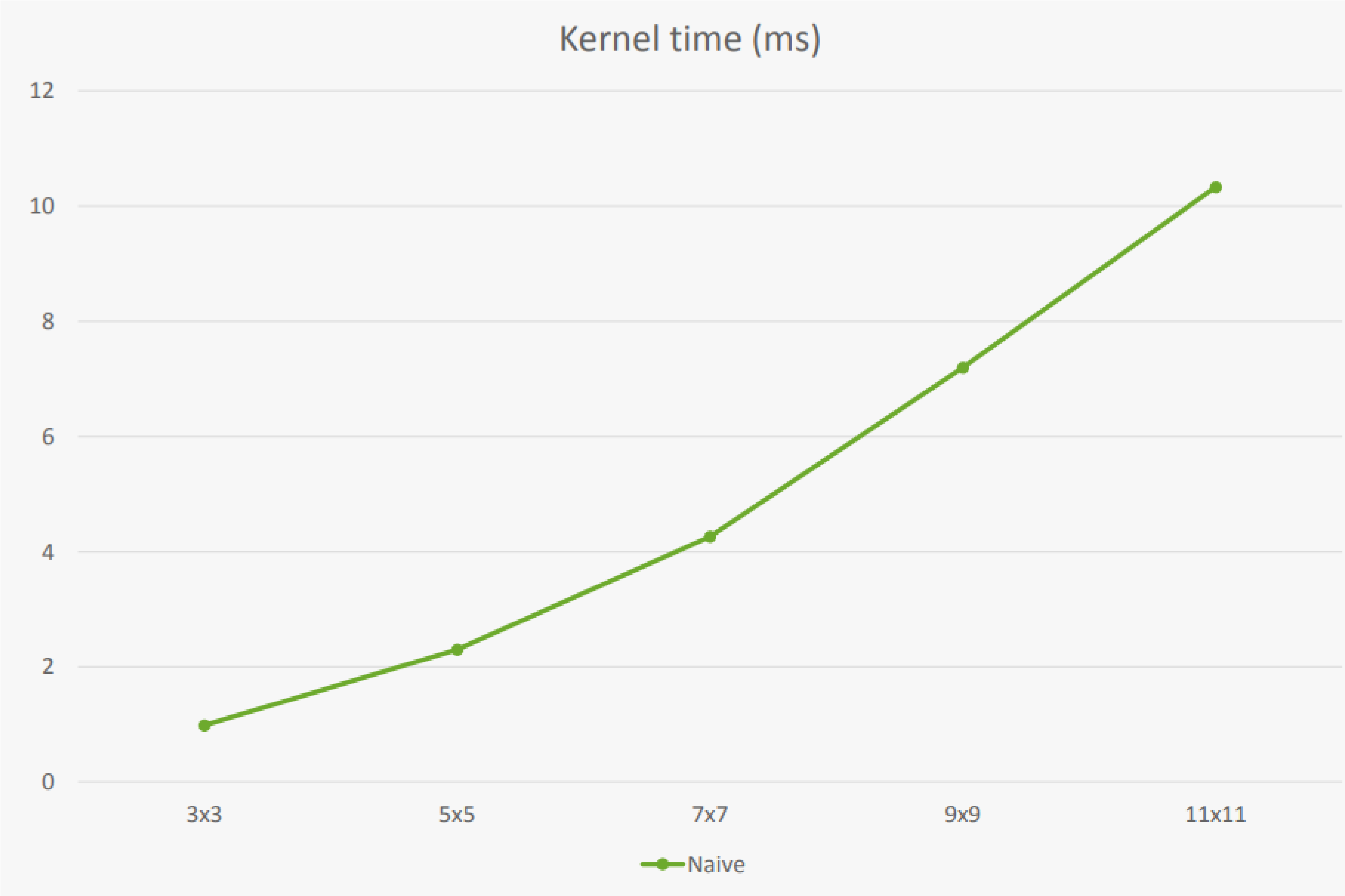


Image conv

512x512 source image

Intel HD Graphics 530

QUESTIONS

EXERCISE

Code Walkthrough - \Code_Exercises\Exercise_05_Image_Convolution

Let's walk through the code together and understand how it works and uses SYCL.