Hands-On HPC Application Development Using C++ and SYCL

James Reinders, Phuong Nguyen, Thomas Applencourt, Rod Burns, Alastair Murray

ENQUEUING A KERNELS AND SHARING DATA

LEARNING OBJECTIVES

- Key concepts: Queues, Kernels, Sharing Data
- Cool useful extras:
    - sycl::stream (debug out)
    - Profiling (kernel timing)

LEARNING OBJECTIVES

- Key concepts: Queues, Kernels, Sharing Data
- Cool useful extras:
    - sycl::stream (debug out)
    - Profiling (kernel timing)

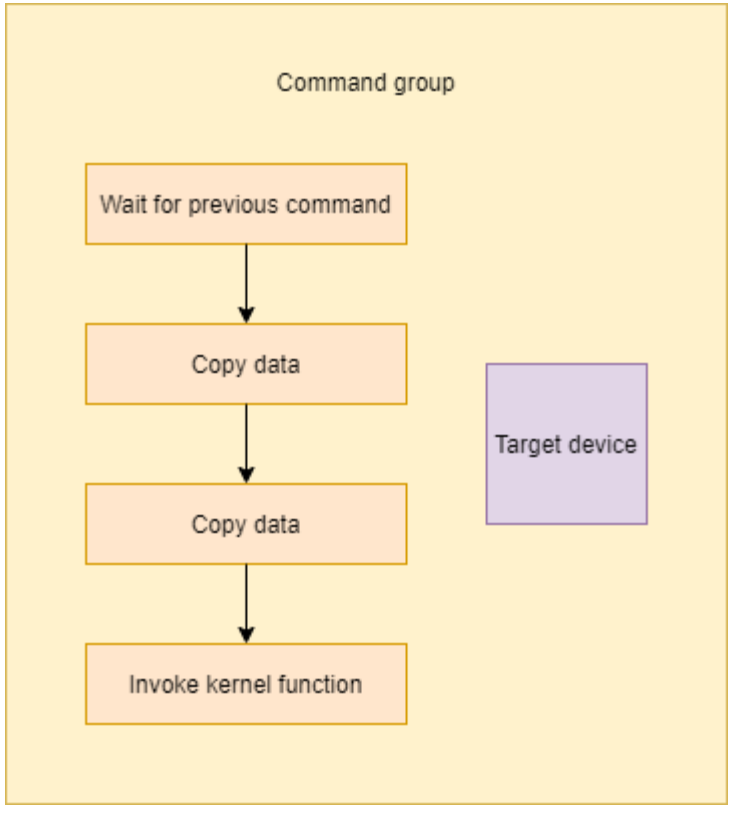
THE QUEUE

- 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

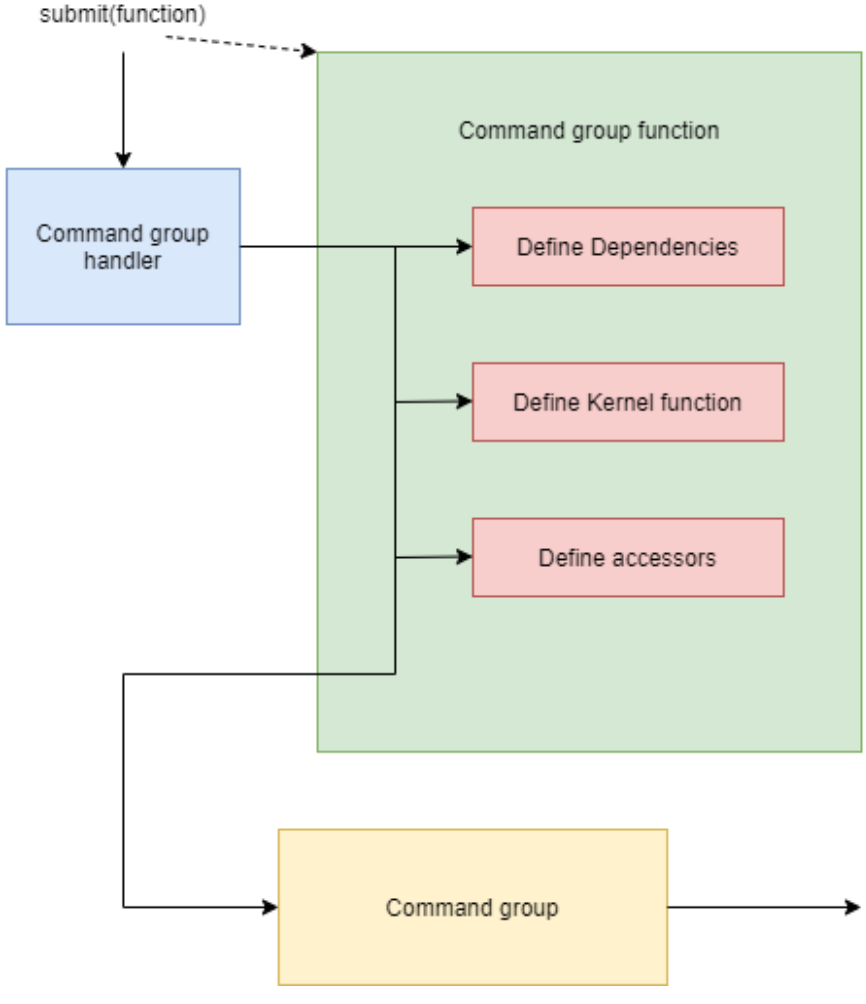
- In SYCL there are two models for managing data:
  - The buffer/accessor model.
  - The USM (unified shared memory) model.
- Which model you choose can have an effect on how you enqueue kernel functions.
- For now we are going to focus on the buffer/accessor model.

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

});
```

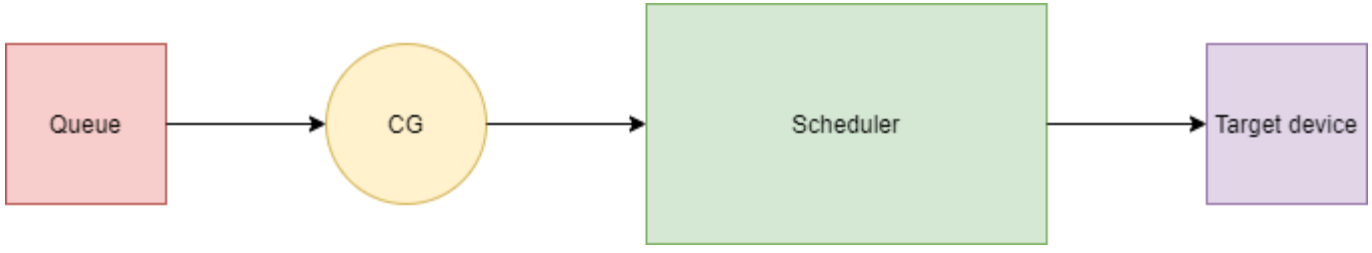
- The command group function is processed exactly once when `submit` is called.
- At this point all the commands and requirements declared inside the command group function are processed to produce a command group.
- The command group is then submitted asynchronously to the scheduler.

COMPOSING COMMAND GROUPS

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

   /* Command group function



}).wait();
```

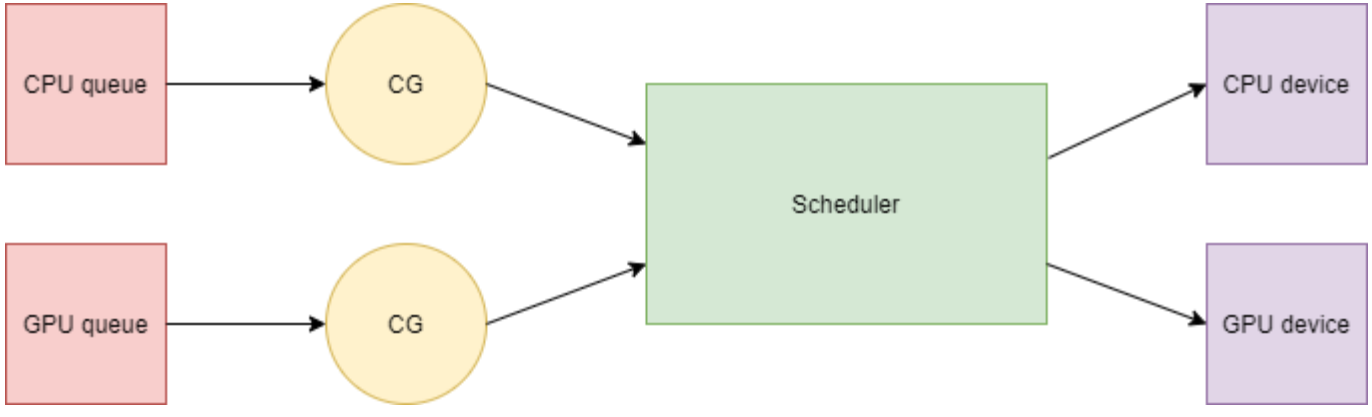
- The `queue` will not wait for commands to complete on destruction.
- However `submit` returns an `event` to allow you to synchronize with the completion of the commands.
- Here we call `wait` on the `event` to immediately wait for it to complete.
- There are other ways to do this, that will be covered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

LEARNING OBJECTIVES

- Key concepts: Queues, Kernels, Sharing Data
- Cool useful extras:
    - sycl::stream (debug out)
    - Profiling (kernel timing)

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`.

```
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.

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 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`.

LEARNING OBJECTIVES

- Key concepts: Queues, Kernels, Sharing Data
- Cool useful extras:
    - sycl::stream (debug out)
    - Profiling (kernel timing)

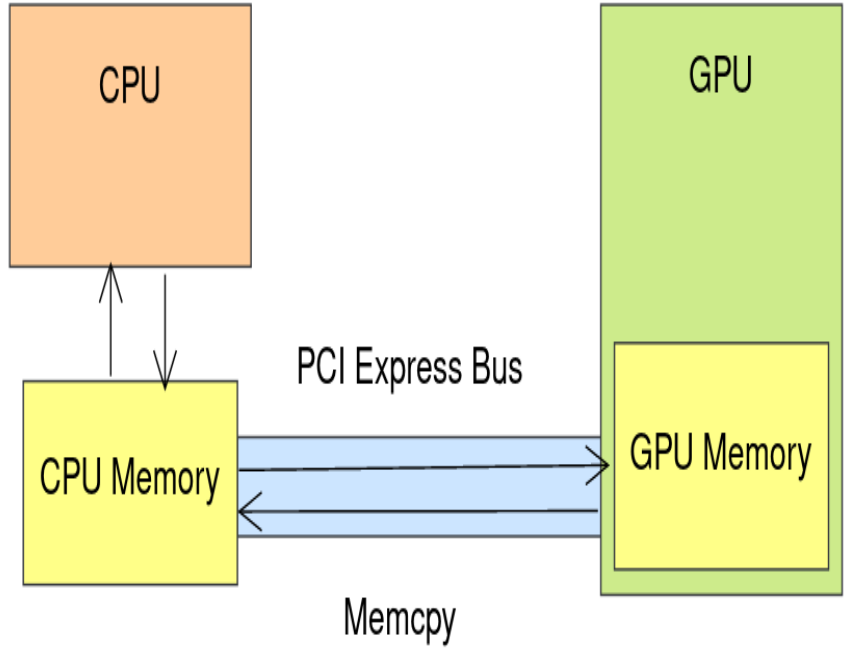
SHARING (MANAGING) DATA

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.

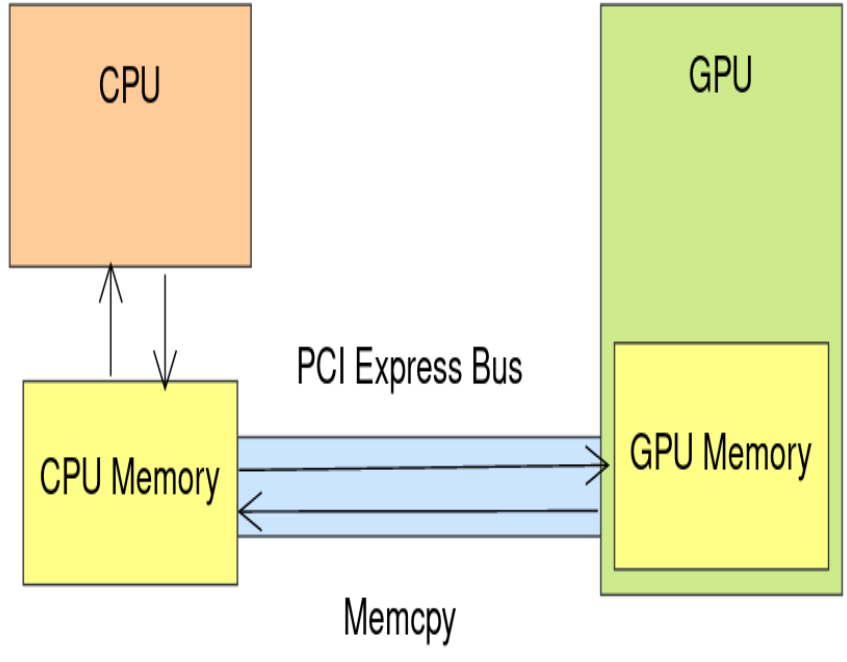
CPU AND GPU MEMORY

- A GPU has its own memory, separate to CPU memory.
- In order for the GPU to use memory from the CPU, the following actions must take place (either explicitly or implicitly):
  - Memory allocation on the GPU.
  - Data migration from the CPU to the allocation on the GPU.
  - Some computation on the GPU.
  - Migration of the result back to the CPU.



CPU AND GPU MEMORY

- Memory transfers between CPU and a device (GPU) are a bottleneck.
- We want to minimize these transfers, when possible.



USM ALLOCATION TYPES

There are different ways USM memory can be allocated: host, device and shared.

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)

USING USM - MALLOC DEVICE

```
// Allocate memory on device
T *device_ptr = sycl::malloc_device<T>(n, myQueue);

// Copy data to device
myQueue.memcpy(device_ptr, cpu_ptr, n * sizeof(T));

// ...
// Do some computation on device
// ...

// Copy data back to CPU
myQueue.memcpy(result_ptr, device_ptr, n * sizeof(T)).wait();

// Free allocated data
sycl::free(device_ptr, myQueue);
```

- It is important to free memory after it has been used to avoid memory leaks.

USING USM - MALLOC SHARED

```
// Allocate shared memory
T *shared_ptr = sycl::malloc_shared<T>(n, myQueue);

// Shared memory can be accessed on host as well as device
for (auto i = 0; i < n; ++i)
  shared_ptr[i] = i;

// ...
// Do some computation on device
// ...

// Free allocated data
sycl::free(shared_ptr, myQueue);
```

- Shared memory is accessible on host and device.
- Performance of shared memory accesses may be poor depending on platform.

SYCL BUFFERS & ACCESSORS

- SYCL provides an API which takes care of allocations and memcpys, as well as some other things.

SYCL BUFFERS & ACCESSORS

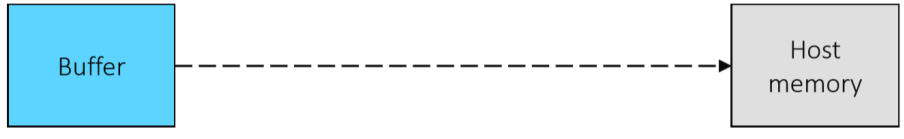
- The buffer/accessor model separates the storage and access of data
  - A SYCL buffer manages data across the host and any number of devices
  - A SYCL accessor requests access to data on the host or on a device for a specific SYCL kernel function

SYCL BUFFERS & ACCESSORS

- 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

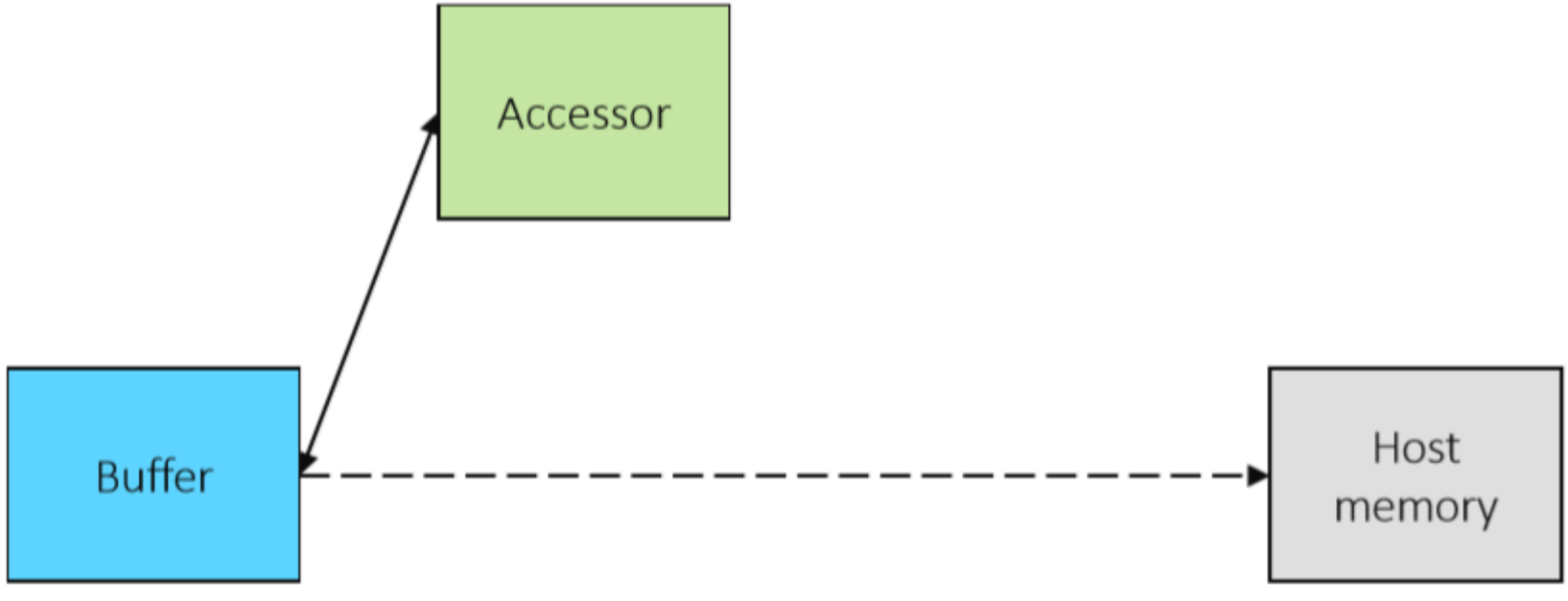
SYCL BUFFERS & ACCESSORS

- 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



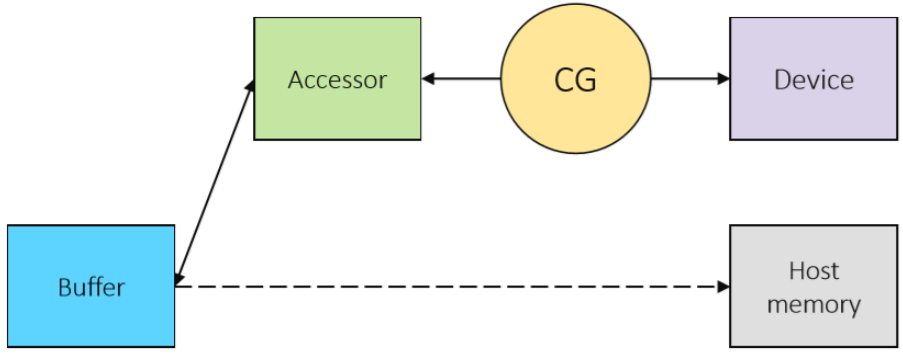
SYCL BUFFERS & ACCESSORS

- 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



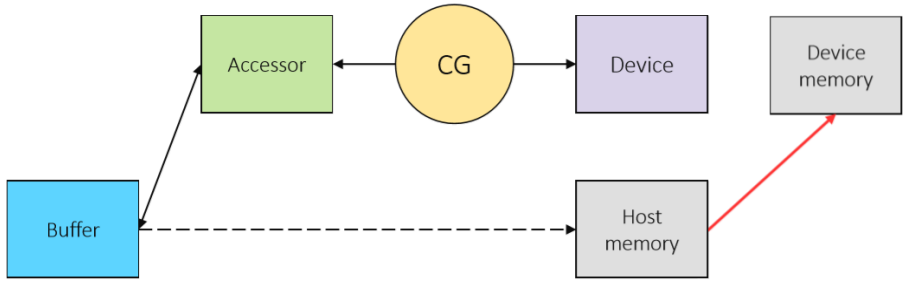
SYCL BUFFERS & ACCESSORS

- 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



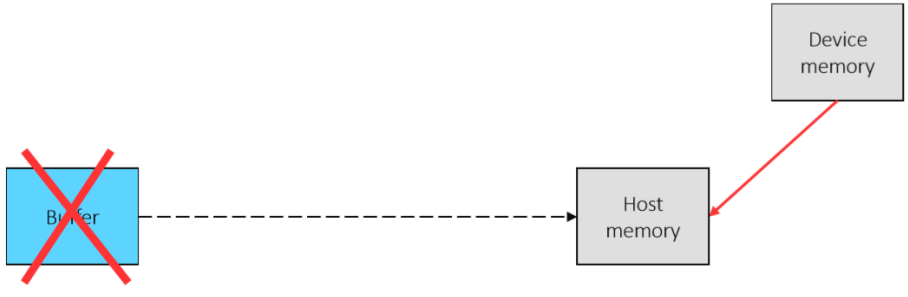
SYCL BUFFERS & ACCESSORS

- Once the SYCL scheduler selects the command group to be executed it must first satisfy its data dependencies
- If necessary, this includes allocating and copying the data to the device accessing that data
- 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 accessor 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

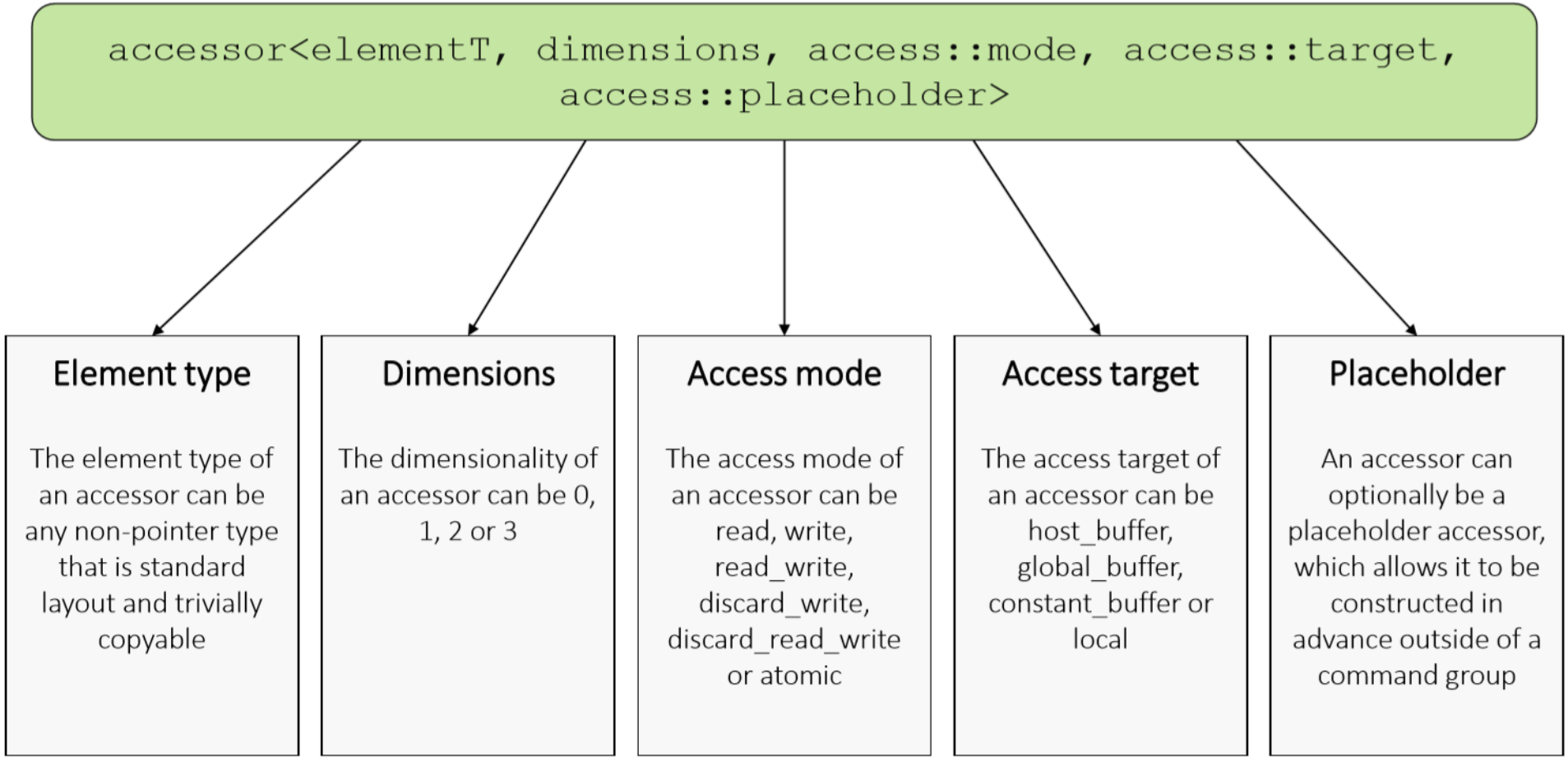


SYCL BUFFERS & ACCESSORS

```
T var = 42;

{
  // Create buffer pointing to var.

  auto buf = sycl::buffer{&var, sycl::range<1>{1}};

  // ...
  // Do some computation on device. Use accessors to access buffer
  // ...

} // var updated here

assert(var != 42);
```

- A buffer is associated with a type, range and dimensionality. Dimensionality must be either 1, 2 or 3.
- Usually type and dimensionality can be inferred using CTAD.
- If a buffer is associated with some allocation in host memory, the host memory will be updated only once the buffer goes out of scope.

ACCESSOR CLASS



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 necessary 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 inferred from the 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.

USING ACCESSORS

```
T var = 42;

{
  // Create buffer pointing to var.
  auto bufA = sycl::buffer{&var, sycl::range<1>{1}};
  auto bufB = sycl::buffer{&var, sycl::range<1>{1}};

  q.submit([&](sycl::handler &cgh) {
    auto accA = sycl::accessor{bufA, cgh, sycl::read_only};
    auto accB = sycl::accessor{bufA, cgh, sycl::no_init};

    cgh.single_task<mykernel>(...); // Do some work
  });

} // var updated here
```

- Buffers and accessors take care of memory migration, as well as dependency analysis.
- More to come later on dependencies.

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<mykernel>([=]{
    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.
- operator[] for USM pointers must take a size_t, whereas operator[] for accessors can take a multi-dimensional sycl::id or a size_t.

LEARNING OBJECTIVES

- Key concepts: Queues, Kernels, Sharing Data
- Cool useful extras:
  - sycl::stream (debug out)
  - Profiling (kernel timing)

SYCL::STREAM

- 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.

SYCL::STREAM

```
sycl::stream(size_t bufferSize, size_t workItemBufferSize, handler
```

- A `stream` must be constructed in the command group function, as a `handler` is required.
  - 1st Constructor `size_t` parameter specifies the total size of the buffer to store output text.
  - 2nd Constructor `size_t` parameter specifies the work-item buffer size.

SYCL::STREAM

```
sycl::stream(size_t bufferSize, size_t workItemBufferSize, handler
```

- 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.
- In other words, the maximum amount of text that invocation may produce.

SYCL::STREAM

```
class my_kernel;

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

   auto os = sycl::stream(1024, 1024, cg


  cgh.single_task(= {
    /* 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 also `1024`.
- This means that the total text that the stream can receive is 1024 bytes.

SYCL::STREAM

```
class my_kernel;

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

   auto os = sycl::stream(1024, 1024, cg

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

- Next we capture the `stream` in the kernel function.
- 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

LEARNING OBJECTIVES

- Key concepts: Queues, Kernels, Sharing Data
- Cool useful extras:
  - sycl::stream (debug out)
  - Profiling (kernel timing)

ENABLE KERNEL PROFILING

```
sycl::property::queue::enable_profiling{}
```

- Extremely valuable capability
- (Technically) SYCL does not require that every device supports this

ENABLE KERNEL PROFILING

```
double time1A =
    (e1.template get_profiling_info<
         sycl::info::event_profiling::
             command_end>() -
     e1.template get_profiling_info<
         sycl::info::event_profiling::
             command_start>());
```

- Data is available after a kernel completes
- Timing code is from SYCL book (2nd edition)
    - Covered in Chapter 13: look around figures 13-6 through 13-8.

QUESTIONS?

EXERCISE

Code_Exercises/Exercise_02_Data_Management

Modify the program to use USM instead of buffers for sharing the pi calculation results.