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.

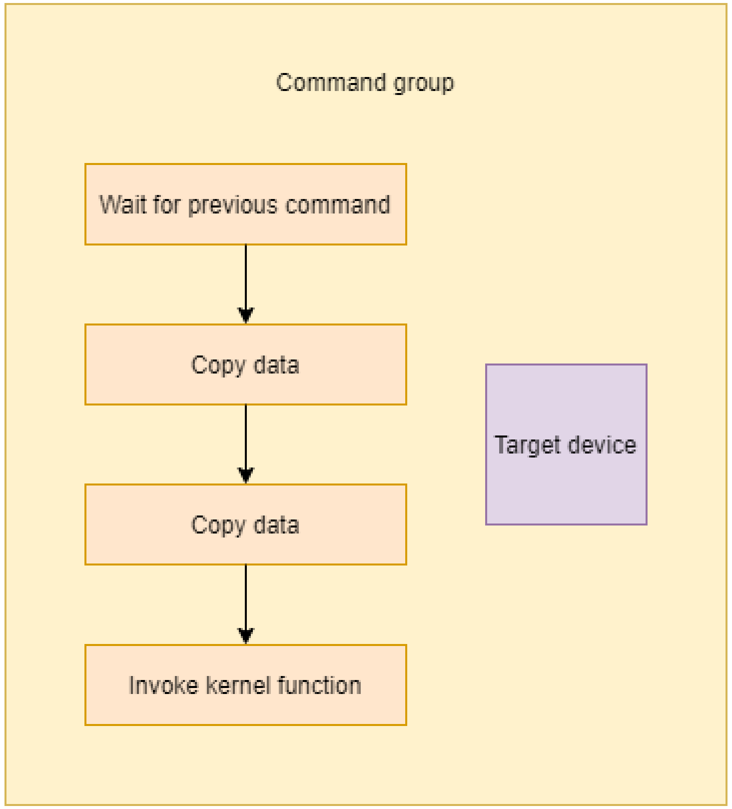
THE QUEUE

- 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

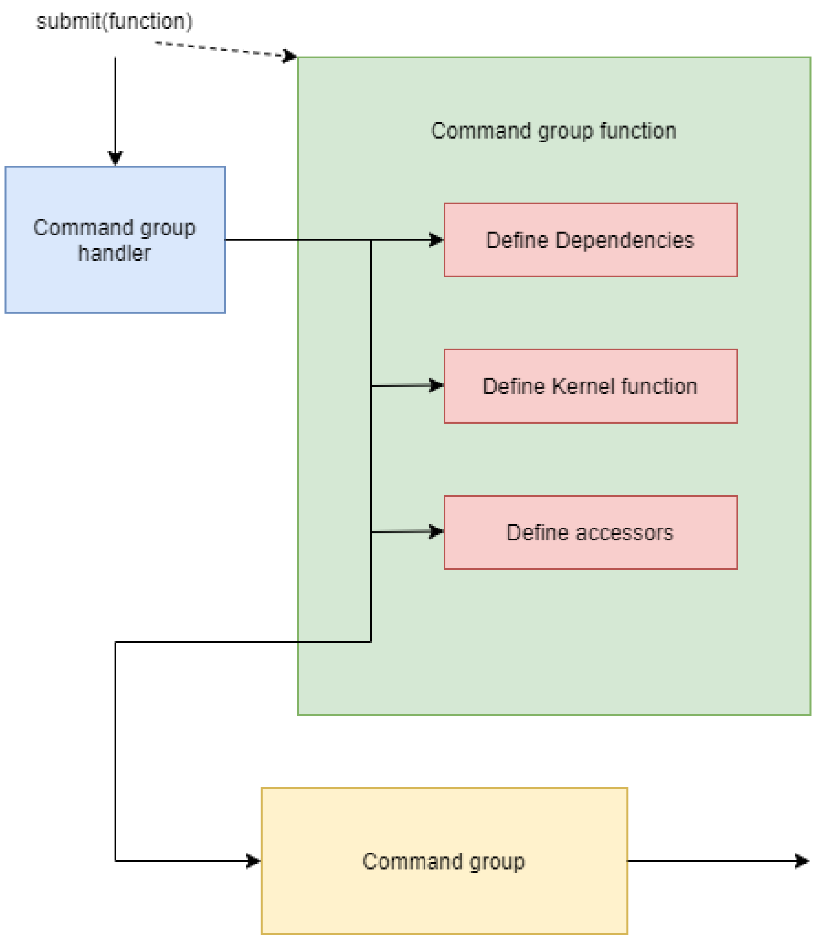
- In SYCL there are two models for managing data:
    - The buffer/accessor model.
    - The USM model.
- Which model you choose can have an effect on how you enqueue kernel functions.
- For now we are going to focus on the buffer/accessor model.

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 */
});
```

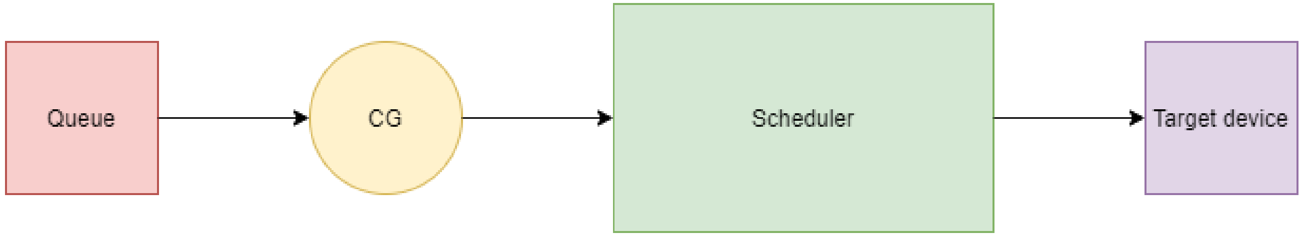
- The command group function is processed exactly once when `submit` is called.
- At this point all the commands and requirements declared inside the command group function are processed to produce a command group.
- The command group is then submitted asynchronously to the scheduler.

COMPOSING COMMAND GROUPS

```
gpuQueue.submit([&](handler &cgh){
  /* Command group function */
}).wait();
```

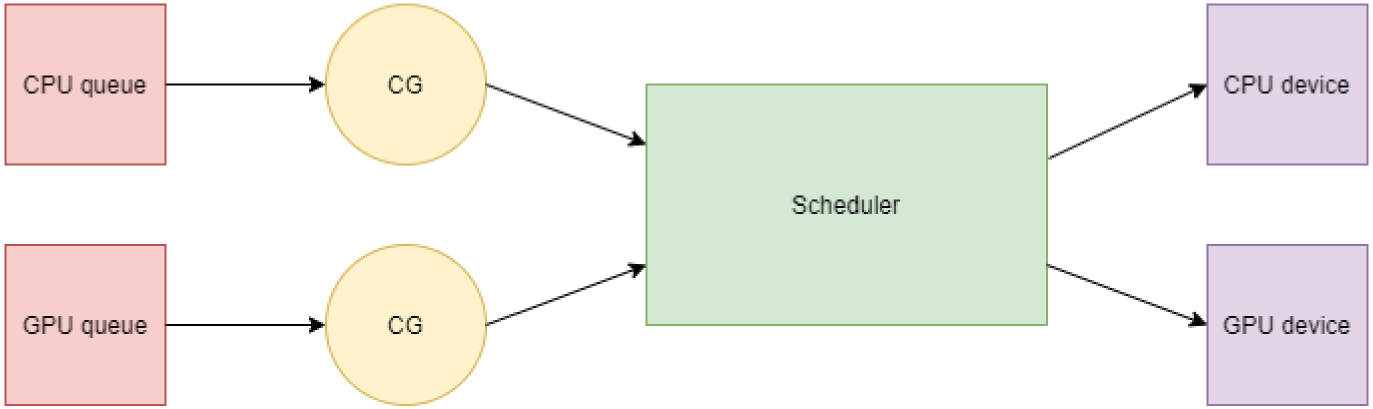
- The queue will not wait for commands to complete on destruction.
- However `submit` returns an `event` to allow you to synchronize with the completion of the commands.
- Here we call `wait` on the `event` to immediately wait for it complete.
- There are other ways to do this, that will be covered in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

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, 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.

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.