ENQUEUEING A KERNEL

LEARNING OBJECTIVES

- Learn about queues and how to submit work to them
- Learn how to define kernel functions
- Learn about the rules and restrictions on kernel functions

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 USM model.

ENQUEUEING SYCL KERNEL FUNCTIONS

```
gpuQueue.single_task(
    [=]() {
      /* kernel code */
    }
  ).wait();
```

- SYCL kernel functions are defined using one of the kernel function invoke APIs provided by the `queue`.
- These enqueue a SYCL kernel function to the SYCL implementation's scheduler.
- Here we use `single_task`.

```
gpuQueue.single_task(
    [=]() {
      /* 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.

```
gpuQueue.single_task(
    [=]() {
      /* kernel code */
    }
).wait();
```

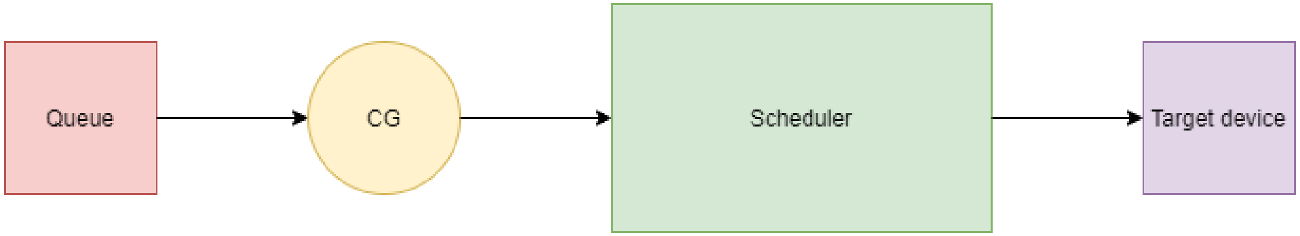
- Different kernel invoke APIs take different parameters describing the iteration space to be invoked in.
- Different kernel invoke APIs can also expect different arguments to be passed to the function object.
- The `single_task` function describes a kernel function that is invoked exactly once, so there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

SCHEDULING

```
gpuQueue.single_task(
    [=]() {
      /* kernel code */
    }
  ).wait();
```

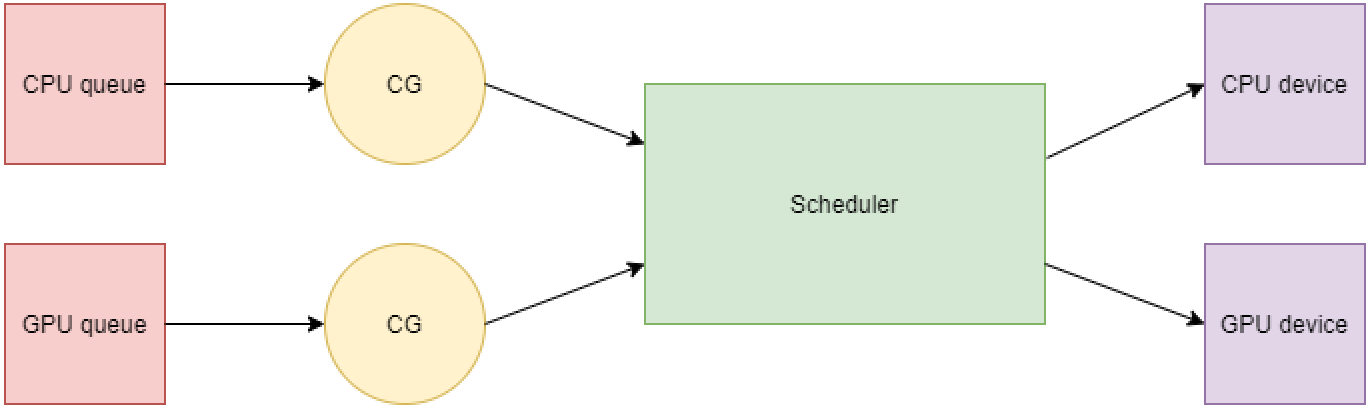
- The `queue` will not wait for commands to complete on destruction.
- However the invoke APIs return an `event` to allow you to synchronize with the completion of the commands.
- Here we call `wait` on the `event` to immediately wait for it to complete.
- There are other ways to do this, that will be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

SCHEDULING



- The invoke APIs (e.g. `single_task`) submit their work 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.

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 RESTRICTIONS

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

KERNELS AS FUNCTION OBJECTS

```
gpuQueue.single_task(
    [=]() {
      /* 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 */
  }
};
```

- However, 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.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`.

QUESTIONS

EXERCISE

Code_Exercises/Enqueueing_a_Kernel_USM/source

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