## **ASYNCHRONOUS EXECUTION**

#### **LEARNING OBJECTIVES**

- Learn about how commands are enqueued asynchronously
- Learn about the different reasons for synchronization
- Learn about the different ways to perform synchronization

#### **ASYNCHRONOUS EXECUTION**

- All command submitted to a queue are done so asynchronously.
- The functions return immediately and the command is run in a background thread.
- This includes individual commands like memcpy and collections of commands derived from a command group.
- This means you have to synchronize with those commands.

#### **SYNCHRONIZATION**

There are a number of reasons why you need to synchronize with commands

- Await completion of a kernel function.
- Await the results of a computation.
- Await error conditions which come from a failure to execute any of the commands.

#### SYNCHRONIZATION WITH KERNEL FUNCTIONS

There are two ways ways to synchronize with kernel functions.

- Calling wait on an event object returned from enqueuing a kernel function command, either via a command group or a shortcut function.
- Calling wait or wait\_and\_throw on the queue itself.

### SYNCHRONIZING WITH KERNEL FUNCTIONS (BUFFERS/ACCESSORS)

```
buf = sycl::buffer(data, sycl::range{1024});

gpuQueue.submit([&](sycl::handler &cgh){
   auto acc = sycl::accessor{buf, cgh};

   cgh.parallel_for<kernel_a>(sycl::range{1024},
        [=](sycl::id<1> idx){
        acc[idx] = /* some computation */
   });
}).wait();
```

- Calling wait on an event object returned from enqueuing a command group will wait for the commands from that command group to complete.
- This is how we have synchronized in our examples so far.
- This effectively creates a blocking operations that will complete in place by immediately synchronizing.

#### SYNCHRONIZING WITH KERNEL FUNCTIONS (BUFFERS/ACCESSORS)

```
buf = sycl::buffer(data, sycl::range{1024});

gpuQueue.submit([&](sycl::handler &cgh){
   auto acc = sycl::accessor{buf, cgh};

   cgh.parallel_for<kernel_a>(sycl::range{1024},
      [=](sycl::id<1> idx){
      acc[idx] = /* some computation */
   });
});

gpuQueue.wait();
```

- Calling wait or wait\_and\_throw on a queue will wait for all commands enqueued to it to complete.
- Note that command groups do not create commands to copy data back to the host application.

### SYNCHRONIZING WITH KERNEL FUNCTIONS (USM)

```
auto devicePtr = usm_wrapper<int>(
    malloc_device<int>(1024, gpuQueue));

gpuQueue.memcpy(devicePtr, data, sizeof(int)).wait();

gpuQueue.parallel_for<kernel_a>(sycl::range{1024},
    [=](sycl::id<1> idx){
    devicePtr[idx] = /* some computation */
}).wait();
```

- Calling wait on an event object returned from functions such as memcpy or the queue shortcuts will wait for that specific command to complete.
- Again this is how we have synchronized in our examples so far.

### SYNCHRONIZING WITH KERNEL FUNCTIONS (USM)

```
auto devicePtr = usm_wrapper<int>(
    malloc_device<int>(1024, gpuQueue));

gpuQueue.memcpy(devicePtr, data, sizeof(int));

gpuQueue.wait();

gpuQueue.parallel_for<kernel_a>(sycl::range{1024},
    [=](sycl::id<1> idx){
    devicePtr[idx] = /* some computation */
});

gpuQueue.wait();
```

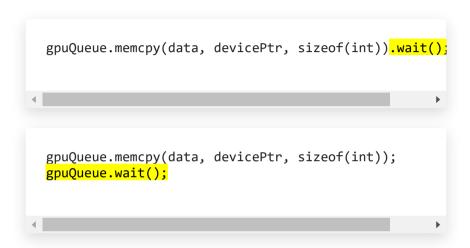
- Again calling wait or wait\_and\_throw on a queue will wait for all commands enqueued to it to complete.
- Note you generally don't want to call wait on the queue after every command, instead you want to create dependencies between commands, which we cover in the next lecture.

#### SYNCHRONIZING WITH DATA

There are multiple ways ways to synchronize with data, but it differs depending on the data management model you are using.

- When using the USM data management model you can synchronize the same way you would for kernel functions, calling wait on an event or the queue.
- When using the buffer/access data management model command groups don't automatically copy data back so there are other ways to synchronize with the data.
  - Creating a host\_accessor.
  - Destroying the buffer.

### SYNCHRONIZING WITH DATA (USM)



- Simply call wait on the event returned from memcpy.
- Alternatively call wait on the queue.

### SYNCHRONIZING WITH DATA (BUFFER/ACCESSOR)

```
buf = sycl::buffer(data, sycl::range{1024});

gpuQueue.submit([&](sycl::handler &cgh){
    auto acc = sycl::accessor{buf, cgh};

    cgh.parallel_for<kernel_a>(sycl::range{1024},
        [=](sycl::id<1> idx){
        acc[idx] = /* some computation */
    });
});

{
    auto hostAcc = buf.get_host_access(cgh);
    hostAcc[idx] = /* some computation */
}
```

- A host\_accessor gives immediate access to the data managed by a buffer in the host application.
- This will wait for any kernel functions accessing the buffer to complete and then copying the data back to the host.
- It will also block any other accessor accessing a buffer until it is destroyed.
- Note that the data

.... .... .... ....

managed by the buffer may not be copied back to the original address on the host, in this case data.

### SYNCHRONIZING WITH DATA (BUFFER/ACCESSOR)

```
buf = sycl::buffer(data, sycl::range{1024});

gpuQueue.submit([&](sycl::handler &cgh){
   auto acc = sycl::accessor{buf, cgh};

   cgh.parallel_for<kernel_a>(sycl::range{1024},
        [=](sycl::id<1> idx){
        acc[idx] = /* some computation */
    });
});
}
```

- A buffer will also synchronize the data it manages on destruction.
- It will wait for any kernel functions accessing it to complete and copy the data back to the origin address before completing destruction.

#### **SYNCHRONIZING WITH ERRORS**

- Errors are handled by a queue and any asynchronous errors can be produced during any of the synchronization methods we've looked at.
- The best way to ensure all errors are caught is to synchronize by calling wait or wait\_and\_throw on the the queue.

# **QUESTIONS**

#### **EXERCISE**

Code\_Exercises/Exercise\_9\_Synchronization/source

Try out the different methods of synchronizing with a kernel function and the resulting data from the computation.