ADVANCED DATA FLOW

LEARNING OBJECTIVES

- Learn about ways to use initial data and pinned memory
- Learn about uninitialized buffers and when to use them

PRECURSOR

Parts of this lecture will focus primarily on the buffer/accessor model.

INITIAL DATA

- Often when writing SYCL kernel functions there is initial data which has been allocated somewhere else in the application.

INITIAL DATA (USM)

```
auto devicePtr = sycl::malloc_device(sizeInBytes, gpuQueue);

gpuQueue.memcpy(devicePtr, initialData, sizeInBytes).wait();
```

- When using the USM model (unless using system USM) pointers passed to kernel functions must be allocated by the SYCL runtime.
- This means you have to copy the initial data to the USM memory allocation using `memcpy`.

INITIAL DATA (USM)

```
auto sharedData = sycl::malloc_shared(sizeInBytes, gpuQueue);
```

- Alternatively if your device supports shared USM allocations you can allocate memory which is shared across the host and device.
- Then there is no need to copy the data to the device.

INITIAL DATA (BUFFER/ACCESSOR)

```
auto buf = sycl::buffer{initialData, sycl::range{size}};
```

- When using the buffer/accessor model a `buffer` can manage already allocated memory or have the SYCL runtime allocate it.
- To do this simply provide an initial pointer when constructing a `buffer`.
- Note that the SYCL runtime is free to allocate memory and copy this into it, which can introduce an overhead.

USE_HOST_POINTER PROPERTY

```
auto buf = sycl::buffer{initialData, sycl::range{size},
  {sycl::property::buffer::use_host_ptr{}}};
```

- To prevent the runtime allocation memory you can provide the `property::buffer::use_host_ptr` property when constructing the `buffer`.
- This instructs the SYCL runtime that it may not allocate any additional memory.
- Though note that the backend (such as OpenCL) may still allocate memory.

COPY BACK

- A `buffer` will synchronize the latest modified copy of the data it manages back to the initial pointer on destruction.

SET_FINAL_DATA

```
auto buf = sycl::buffer{initialData, sycl::range{size}};

buf.set_final_data(finalData);
```

- To change the destination that a `buffer` will synchronize to on destruction you can call `set_final_data` with another.
- The address provided must be capable of holding the size of the data the `buffer` manages.

SET_FINAL_DATA

```
auto buf = sycl::buffer{initialData, sycl::range{size}};

buf.set_final_data(nullptr);
```

- Alternatively to prevent the `buffer` from synchronizing back to the initial data entirely you can call `set_final_data` with `nullptr`.
- A `buffer` with no final data address is useful because the data can left on a device and not copied back to the host from the device.

UNINITIALIZED BUFFERS (BUFFER/ACCESSOR)

```
auto buf = sycl::buffer{sycl::range{size}};
```

- As we've seen in the USM model all memory is allocated initialized, but `buffers` can be constructed within initial data.
- Alternatively a `buffer` can be constructed as uninitialized.
- To do this simply construct a `buffer` without initial data.
- Uninitialized `buffers` are useful for a couple of reasons because they can be allocated directly on a device and not copied to the device from the host.

UNINITIALIZED BUFFERS (BUFFER/ACCESSOR)

```
auto buf = sycl::buffer{sycl::range{size}};
```

- As we've seen in the USM model all memory is allocated initialized, but `buffers` can be constructed within initial data.
- Alternatively a `buffer` can be constructed as uninitialized.
- To do this simply construct a `buffer` without initial data.
- Uninitialized `buffers` are useful for temporary data or data which is initialized within a kernel function.

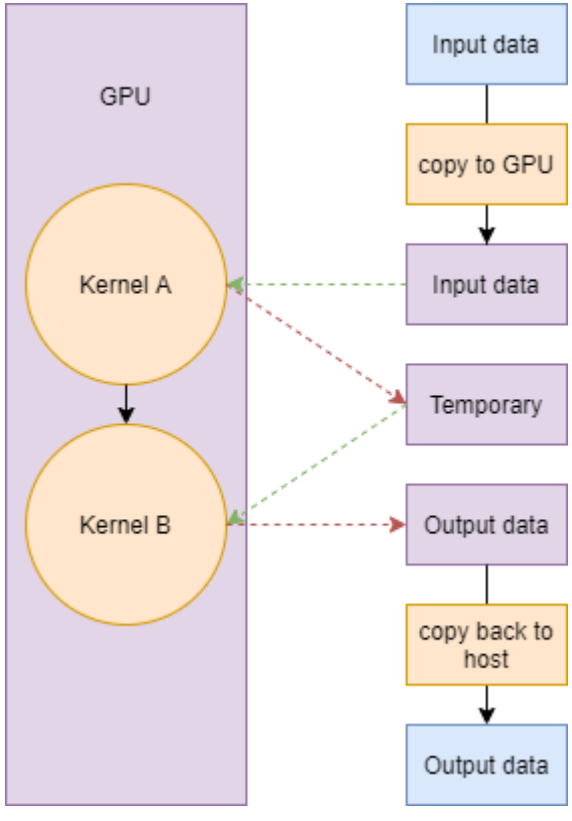
UNINITIALIZED BUFFERS (BUFFER/ACCESSOR)

```
auto buf = sycl::buffer{sycl::range{size}};

buf.set_final_data()
```

- As we've seen in the USM model all memory is allocated initialized, but `buffers` can be constructed within initial data.
- Alternatively a `buffer` can be constructed as uninitialized.
- To do this simply construct a `buffer` without initial data.
- Uninitialized `buffers` are useful for a couple of reasons because they can be allocated directly on a device and don't require moving data from the host.

USING INITIAL DATA AND UNINITIALIZED BUFFERS



- Here we have an example of using these techniques:
  - **Input data** is initialized with initial data but doesn't need to be copied back so it can use `set_final_data(nullptr)`.
  - **Temporary** is only used on the device so can be an uninitialized `buffer`.
  - **Output data** is initialized on the data and needs to be copied back so it can be an uninitialized `buffer` and use `set_final_data` to provide the final data address.

PINNED MEMORY (BUFFER/ACCESSOR)

- Pinned memory is a feature supported by most SYCL backends and devices.
- It allows you to allocate memory which can be mapped between the host and device more efficiently, providing similar benefits to USM.
- Though the requirements can vary from one device to another.
- It's always best to check the vendor's programming guide.

PINNED MEMORY (BUFFER/ACCESSOR)

- The SYCL runtime will always aim to manage the memory for you in the most efficient way for the target device.
- Generally there are two approaches to facilitate pinned memory:
  - Allocate memory according to the vendor's programming guide, usually involved allocating a size fo a particular multiple and aligned to a particular size, and then use the `property::buffer::use_host_ptr` property.
  - Create an uninitialized `buffer` and allow the runtime to allocate the memory the appropriate way.

QUESTIONS

EXERCISE

Code_Exercises/Exercise_12_Temporary_Data/source

Write a SYCL application which uses uninitialized `buffers` and disabling write back.