DATA AND DEPENDENCIES

LEARNING OBJECTIVES

- Learn about how to create dependencies between kernel functions
- Learn about how to move data between the host and device(s)
- Learn about the differences between the buffer/accessor and USM data management models
- Learn how to represent basic data flow graphs

ACCESS/BUFFER AND USM

There are two ways to move data and create dependencies between kernel functions in SYCL

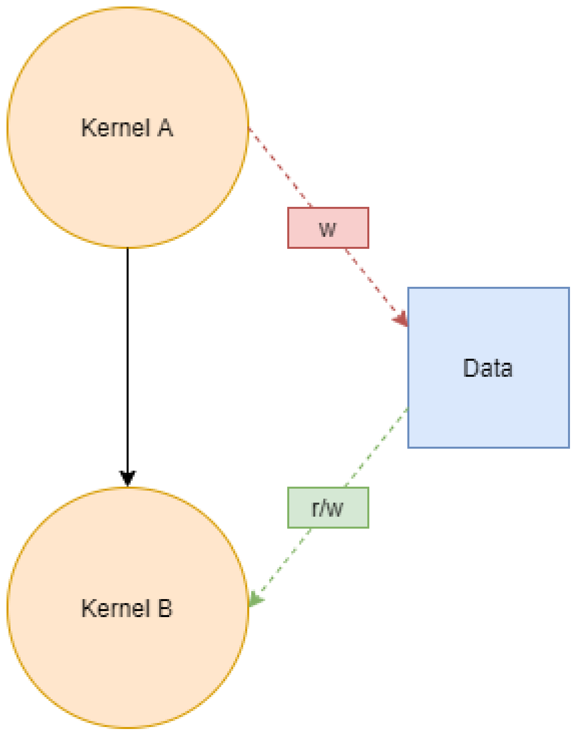
Buffer/accessor data movement model

- Data dependencies analysis
- Implicit data movement

USM data movement model

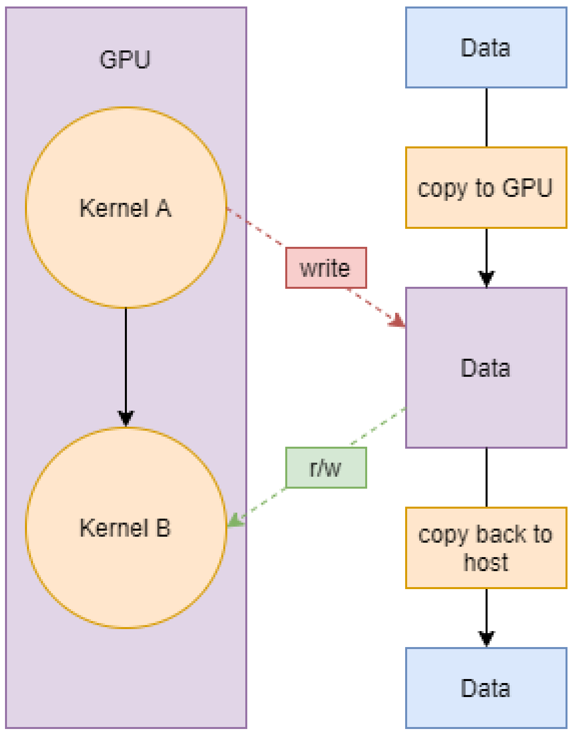
- Manual chaining of dependencies
- Explicit data movement

CREATING DEPENDENCIES



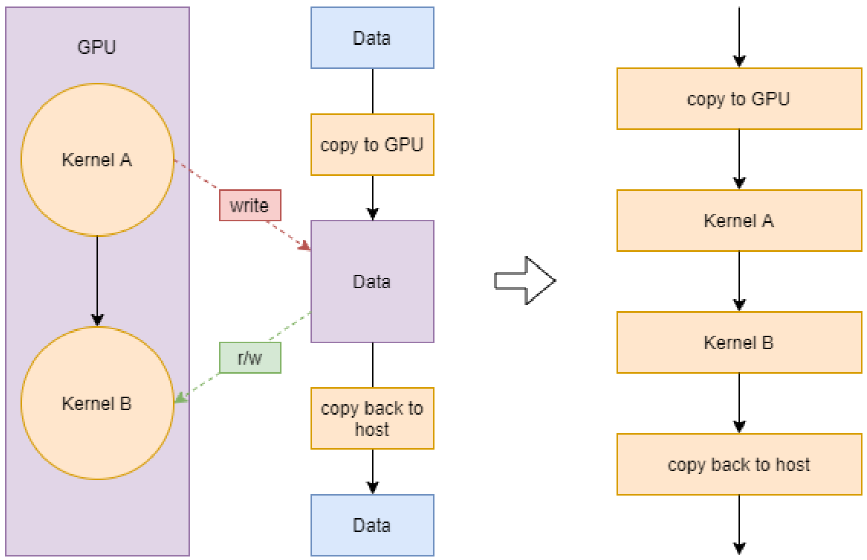
- Kernel A first writes to the data
- Kernel B then reads from and writes to the data
- This creates a read-after-write (RAW) relationship
- There must be a dependency created between Kernel A and Kernel B

MOVING DATA



- Here both kernel functions are enqueued to the same device, in this case a GPU
- The data must be copied to the GPU before the Kernel A is executed
- The data must remain on the GPU for Kernel B to be executed
- The data must be copied back to the host after Kernel B has executed

DATA FLOW



- Combining kernel function dependencies and the data movement dependencies we have a final data flow graph
- This graph defines the order in which all commands must execute in order to maintain consistency
- In more complex data flow graphs there may be multiple orderings which can achieve the same consistency

DATA FLOW WITH BUFFERS AND ACCESSORS

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

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

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

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

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

gpuQueue.wait();
```

- The buffer/accessor data management model data model is descriptive
- Dependencies and data movement is inferred from the access requirements of command groups
- The SYCL runtime is responsible for guaranteeing that data dependencies and consistency are maintained

DATA FLOW WITH BUFFERS AND ACCESSORS

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

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

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

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

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

gpuQueue.wait();
```

- A `buffer` object is responsible for managing data between the host and one or more devices
- It is also responsible for tracking dependencies on the data it manages
- It will also allocating memory and move data when necessary.
- Note that a `buffer` is lazy and will not allocate or move data until it is asked to

DATA FLOW WITH BUFFERS AND ACCESSORS

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

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

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

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

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

gpuQueue.wait();
```

- An `accessor` object is responsible for describing data access requirements
- It describes what data a kernel function is accessing and how it is accessing it
- The `buffer` object uses this information to create infer dependencies and data movement

DATA FLOW WITH BUFFERS AND ACCESSORS

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

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

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

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

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

gpuQueue.wait();
```

- Associating the `accessor` object with the `handler` connects the access dependency to the kernel function
- It also associates the access requirement with the device being targeted

DATA FLOW WITH USM

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

auto e1 = gpuQueue.memcpy(devicePtr, data, sizeof(int

auto e2 = gpuQueue.parallel_for<kernel_a>(
  sycl::range{1024}, e1, [=](sycl::id<1> idx) {
    devicePtr[idx] = /* some computation */
  });

auto e3 = gpuQueue.parallel_for<kernel_b>(
  sycl::range{1024}, e2, [=](sycl::id<1> idx) {
    devicePtr[idx] = /* some computation */
  });

auto e4 = gpuQueue.memcpy(data, devicePtr,
                          sizeof(int), e3);

e4.wait();
```

- The USM data management model data model is prescriptive
- Dependencies are defined explicitly by passing around event objects
- Data movement is performed explicitly by enqueuing `memcpy` operations
- The user is responsible for ensuring data dependencies and consistency are maintained

DATA FLOW WITH USM

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

auto e1 = gpuQueue.memcpy(devicePtr, data, sizeof(int

auto e2 = gpuQueue.parallel_for<kernel_a>(
  sycl::range{1024}, e1, [=](sycl::id<1> idx) {
    devicePtr[idx] = /* some computation */
  });

auto e3 = gpuQueue.parallel_for<kernel_b>(
  sycl::range{1024}, e2, [=](sycl::id<1> idx) {
    devicePtr[idx] = /* some computation */
  });

auto e4 = gpuQueue.memcpy(data, devicePtr,
                          sizeof(int), e3);

e4.wait();
```

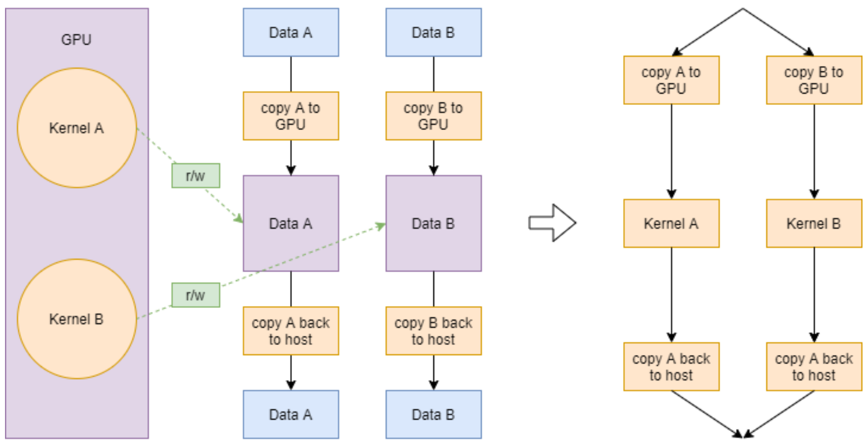
- Each command enqueued to the queue produces an event object which can be used to synchronize with the completion of that command
- Passing those event objects when enqueueing other commands creates dependencies

DATA FLOW WITH USM

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

auto e1 = gpuQueue.memcpy(devicePtr, data, sizeof(int

auto e2 = gpuQueue.parallel_for<kernel_a>(
  sycl::range{1024}, e1, [=](sycl::id<1> idx) {
    devicePtr[idx] = /* some computation */
  });

auto e3 = gpuQueue.parallel_for<kernel_b>(
  sycl::range{1024}, e2, [=](sycl::id<1> idx) {
    devicePtr[idx] = /* some computation */
  });

auto e4 = gpuQueue.memcpy(data, devicePtr,
                          sizeof(int), e3);

e4.wait();
```

- The `memcpy` member functions are used to enqueue data movement commands, moving the data to the GPU and then back again

CONCURRENT DATA FLOW



- If two kernels are accessing different buffers then there is no dependency between them
- In this case the two kernels and their respective data movement are independent
- By default queues are out-of-order which means that these commands can execute in any order
- They could also execute concurrently if the target device is able to do so

CONCURRENT DATA FLOW WITH BUFFERS AND ACCESSORS

```
sycl::buffer bufA {dataA, sycl::range{1024}};
sycl::buffer bufB {dataB, sycl::range{1024}};

gpuQueue.submit([&](sycl::handler &cgh) {
  auto accA = bufA.get_access(cgh);

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

gpuQueue.submit([&](sycl::handler &cgh) {
  auto accB = bufB.get_access(cgh);

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

gpuQueue.wait();
```

- The buffer/accessor data management model automatically infers dependencies
- As each of the two kernel functions are accessing different `buffer` objects the SYCL runtime can infer there is no dependency between them
- Data movement is still performed for the two kernels as normal
- The two kernels and their respective copies collectively can be executed in any order

CONCURRENT DATA FLOW WITH USM

```
auto devicePtrA = sycl::malloc_device<int>(1024, gpuQueue);
auto devicePtrB = sycl::malloc_device<int>(1024, gpuQueue);

auto e1 = gpuQueue.memcpy(devicePtrA, dataA, sizeof(int));
auto e2 = gpuQueue.memcpy(devicePtrB, dataB, sizeof(int));

auto e3 = gpuQueue.parallel_for<kernel_a>(sycl::range{1024}, e1, [=](sycl::id<1> idx) {
  devicePtrA[idx] = /* some computation */ });

auto e4 = gpuQueue.parallel_for<kernel_b>(sycl::range{1024}, e2, [=](sycl::id<1> idx) {
  devicePtrB[idx] = /* some computation */ });

auto e5 = gpuQueue.memcpy(dataA), devicePtrA, sizeof(int), e3);

auto e6 = gpuQueue.memcpy(dataB, devicePtrB, sizeof(int), e4);

e5.wait(); e6.wait();
```

- Dependencies are defined explicitly
- We don't create dependencies between kernel functions but we do create dependencies on the data movement

CONCURRENT DATA FLOW WITH USM

```
auto devicePtrA = sycl::malloc_device<int>(1024, gpuQueue);
auto devicePtrB = sycl::malloc_device<int>(1024, gpuQueue);

auto e1 = gpuQueue.memcpy(devicePtrA, dataA, sizeof(int));
auto e2 = gpuQueue.memcpy(devicePtrB, dataB, sizeof(int));

auto e3 = gpuQueue.parallel_for<kernel_a>(sycl::range{1024}, e1, [=](sycl::id<1> idx) {
  devicePtrA[idx] = /* some computation */ });

auto e4 = gpuQueue.parallel_for<kernel_b>(sycl::range{1024}, e2, [=](sycl::id<1> idx) {
  devicePtrB[idx] = /* some computation */ });

auto e5 = gpuQueue.memcpy(dataA), devicePtrA, sizeof(int), e3);

auto e6 = gpuQueue.memcpy(dataB, devicePtrB, sizeof(int), e4);

e5.wait(); e6.wait();
```

- The dependencies of each chain of commands is independant of the other
- The two kernels and their respective copies collectively can be executed in any order

WHICH SHOULD YOU CHOOSE?

When should you use the buffer/accessor or USM data management models?

Buffer/accessor data movement model

- If you want to guarantee consistency and avoid errors
- If you want to iterate over your data flow quicker

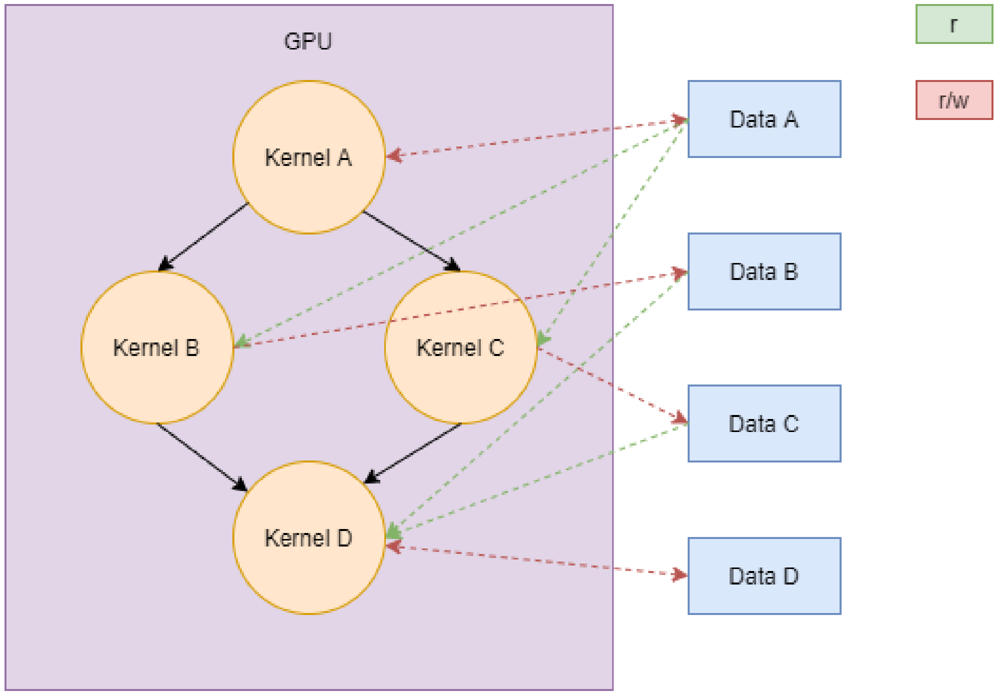
USM data movement model

- If you need to use USM
- If you want more fine grained control over data movement

QUESTIONS

EXERCISE

Code_Exercises/Exercise_10_Managing_Dependencies/source



Put together what you've seen here to create the above diamond data flow graph in either buffer/accessor or USM data management models