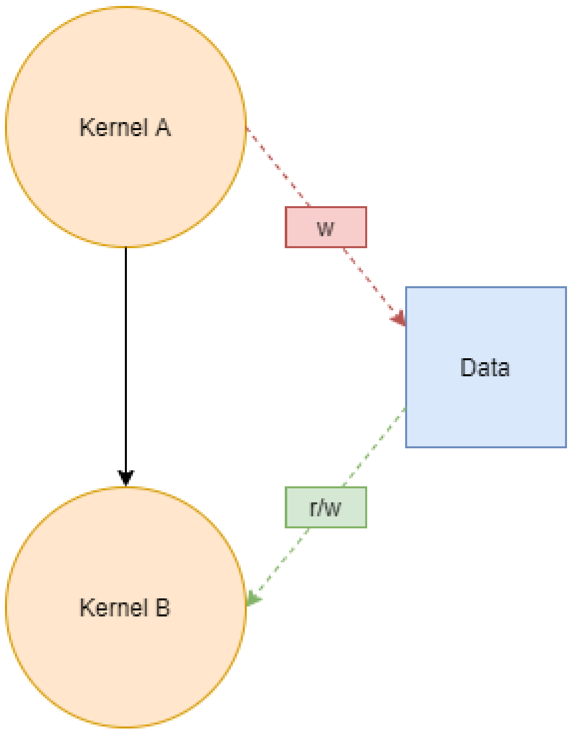
DATA AND DEPENDENCIES

LEARNING OBJECTIVES

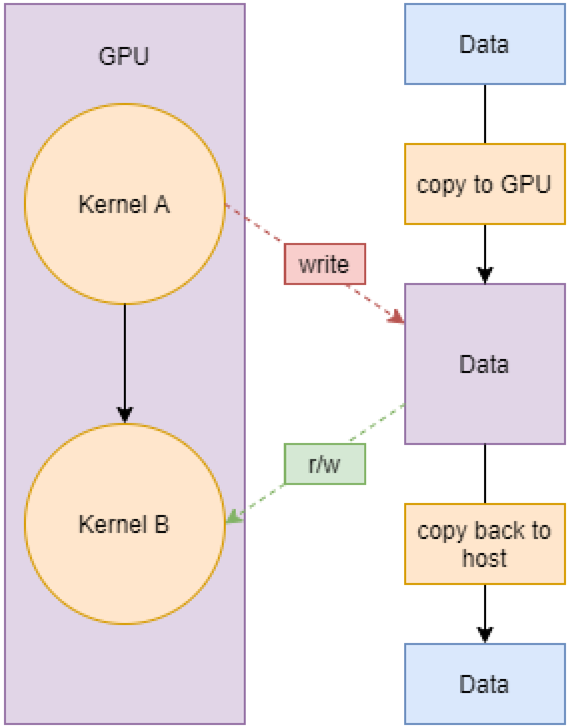
- Learn about how to create dependencies between kernel functions
- Learn about how to move data between the host and device(s)
- Learn how to represent basic data flow graphs

CREATING DEPENDENCIES



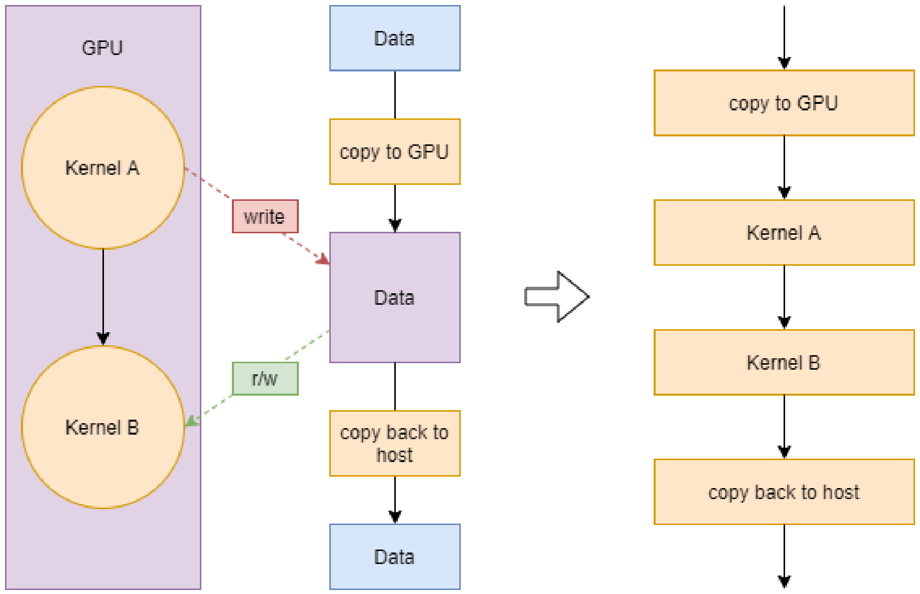
- Kernel A first writes to the data
- Kernel B then reads from and writes to the data
- This creates a read-after-write (RAW) relationship
- There must be a dependency created between Kernel A and Kernel B

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 USM

```
auto devPtr = sycl::malloc_device<int>(1024, q);

auto e1 = q.memcpy(devPtr, data, sizeof(int));
auto e2 = q.parallel_for(sycl::range{1024},
  e1,
  [=](sycl::id<1> idx) {
    devPtr[idx] = /* some computation */
  });

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

auto e4 = q.memcpy(data, devPtr, sizeof(int), e3);
```

- 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 devPtr = sycl::malloc_device<int>(1024, q);

auto e1 = q.memcpy(devPtr, data, sizeof(int));
auto e2 = q.parallel_for(sycl::range{1024},
  e1,
  [=](sycl::id<1> idx) {
    devPtr[idx] = /* some computation */
  });

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

auto e4 = q.memcpy(data, devPtr,
nt), e3);
e4.wait();
```

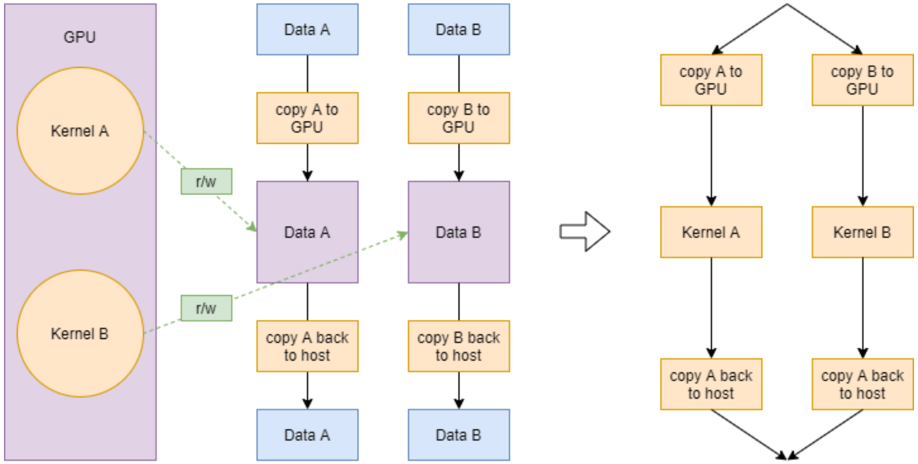
- Each command enqueued to the queue produces an event object which can be used to synchronize with the completion of that command
- Passing those event objects when enqueueing other commands creates dependencies

DATA FLOW WITH USM

```
auto devPtr = sycl::malloc_device<int>(1024, q);

auto e1 = q.memcpy(devPtr, data, sizeof(int));
auto e2 = q.parallel_for(sycl::range{1024}, 
  e1,
  [=](sycl::id<1> idx) {
    devPtr[idx] = /* some computation */
  });

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

auto e4 = q.memcpy(data, devPtr, 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 USM

```
auto devPtrA = sycl::malloc_device<int>(1024, q);
auto devPtrB = sycl::malloc_device<int>(1024, q);

auto e1 = q.memcpy(devPtrA, dataA, sizeof(int));
auto e2 = q.memcpy(devPtrB, dataB, sizeof(int));

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

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

auto e5 = q.memcpy(dataA, devPtrA, sizeof(int), e3);
auto e6 = q.memcpy(dataB, devPtrB, sizeof(int), e4);

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

sycl::free(devPtrA, q);
sycl::free(devPtrB, q);
```


- Dependencies are defined explicitly

CONCURRENT DATA FLOW WITH USM

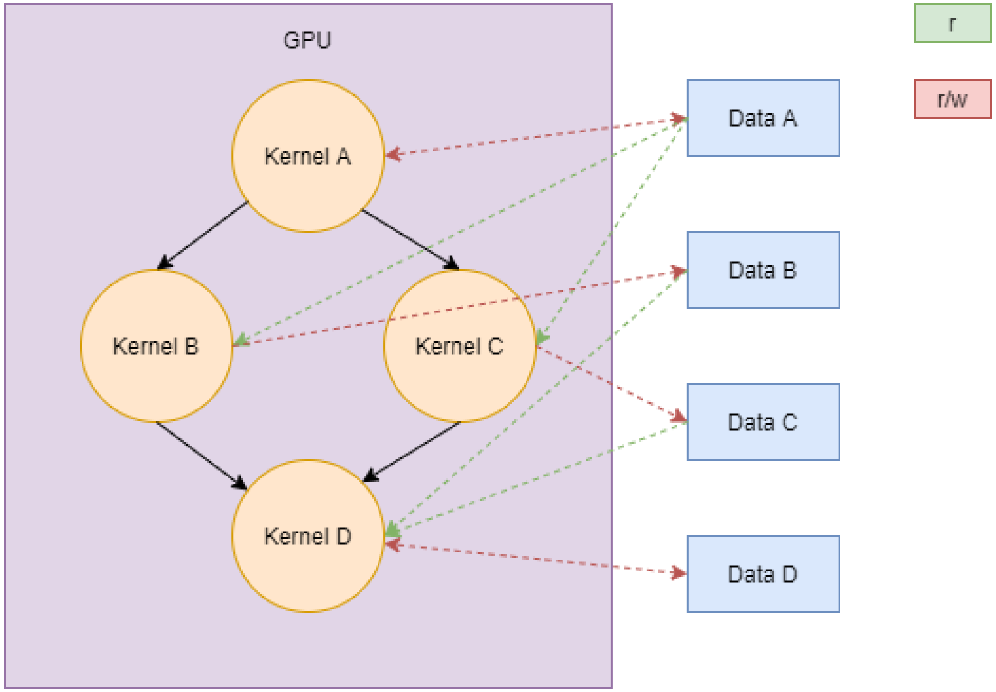
```
auto devPtrA = sycl::malloc_device<int>(1024, q);
auto devPtrB = sycl::malloc_device<int>(1024, q);

auto e1 = q.memcpy(devPtrA, dataA, sizeof(int));
auto e2 = q.memcpy(devPtrB, dataB, sizeof(int));

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

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

auto e5 = q.memcpy(dataA, devPtrA, sizeof(int), e3);
auto e6 = q.memcpy(dataB, devPtrB, sizeof(int), e4);

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

sycl::free(devPtrA, q);
sycl::free(devPtrB, q);
```


- The dependencies of each chain of commands is independant of the other

QUESTIONS

EXERCISE

Code_Exercises/Data_and_Dependencies/source



Put together what you've seen here to create the above diamond data flow graph.