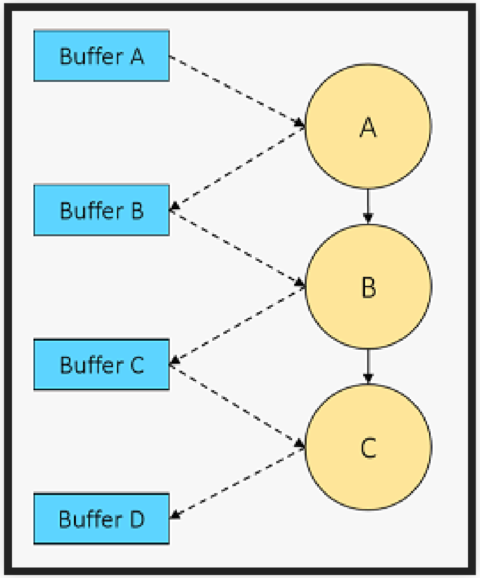
SYCL DATA DEPENDENCY ANALYSIS

LEARNING OBJECTIVES

- Learn about how the SYCL runtime orders execution using data dependencies
- Learn about how SYCL synchronizes data

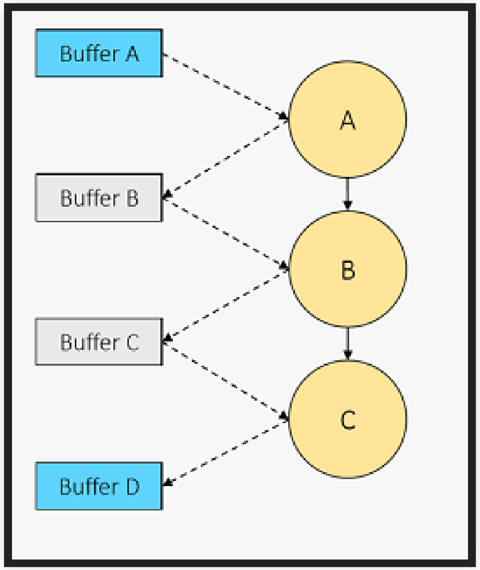
- When a command group is submitted to a SYCL queue the runtime performs dependency analysis
  - If a command group requests access to a memory object, such as a buffer
    - A pre-requisite is the data must be available before kernel execution
  - The scheduler uses these pre-requisites to order the execution of commands
- Data is copied when required or when explicitly requested
  - Data will stay on the device to avoid unnecessary copies back to the host

```
queue cpuQueue(cpu_selector{}, async_handler{});
cpuQueue.submit([&](handler &cgh){ // CG A
  auto in = bufA.get_access<access::mode::read>(cgh);
  auto out = bufB.get_access<access::mode::write>(cgh);
  cgh.parallel_for(range<1>(dA.size()), func(in, out)); });

cpuQueue.submit([&](handler &cgh){ // CG B
  auto in = bufB.get_access<access::mode::read>(cgh);
  auto out = bufC.get_access<access::mode::write>(cgh);
  cgh.parallel_for(range<1>(dA.size()), func(in, out)); });

cpuQueue.submit([&](handler &cgh){ // CG C
  auto in = bufC.get_access<access::mode::read>(cgh);
  auto out = bufD.get_access<access::mode::write>(cgh);
  cgh.parallel_for(range<1>(dA.size()), func(in, out)); });

cpuQueue.wait_and_throw();
```

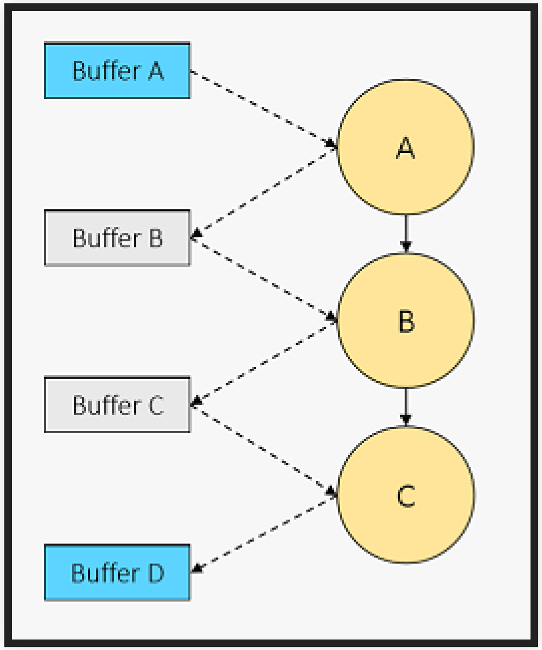


- To use a C++ function object you simply construct an instance of the type initializing the accessors and pass it to parallel_for
- Notice you no longer need to name the SYCL kernel

```
queue cpuQueue(cpu_selector{}, async_handler{});
cpuQueue.submit([&](handler &cgh){ // CG A
  auto in = bufA.get_access<access::mode::read>(cgh);
  auto out = bufB.get_access<access::mode::write>(cgh);
  cgh.parallel_for(range<1>(dA.size()), func(in, out)); });

cpuQueue.submit([&](handler &cgh){ // CG B
  auto in = bufB.get_access<access::mode::read>(cgh);
  auto out = bufC.get_access<access::mode::write>(cgh);
  cgh.parallel_for(range<1>(dA.size()), func(in, out)); });

cpuQueue.submit([&](handler &cgh){ // CG C
  auto in = bufC.get_access<access::mode::read>(cgh);
  auto out = bufD.get_access<access::mode::write>(cgh);
  cgh.parallel_for(range<1>(dA.size()), func(in, out)); });

cpuQueue.wait_and_throw();
```

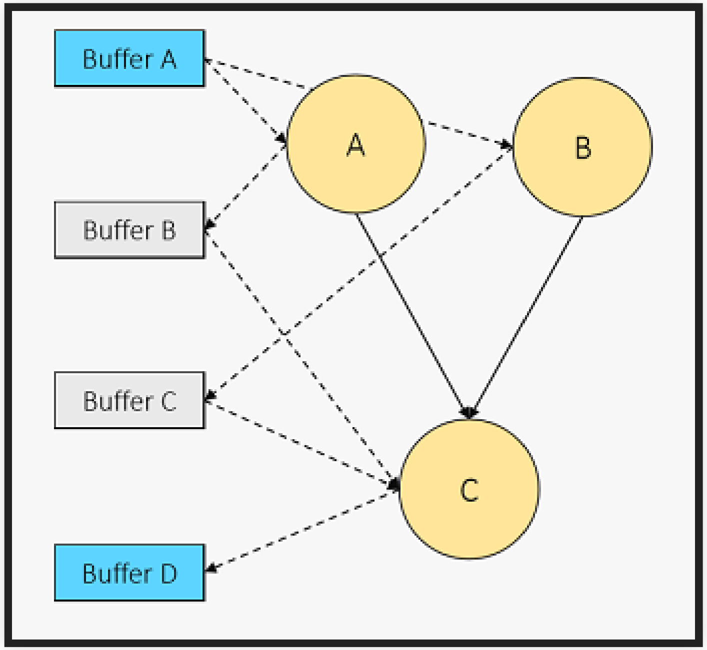


Buffers B and C are not accessed on the host so they can be optimized to remain on the device

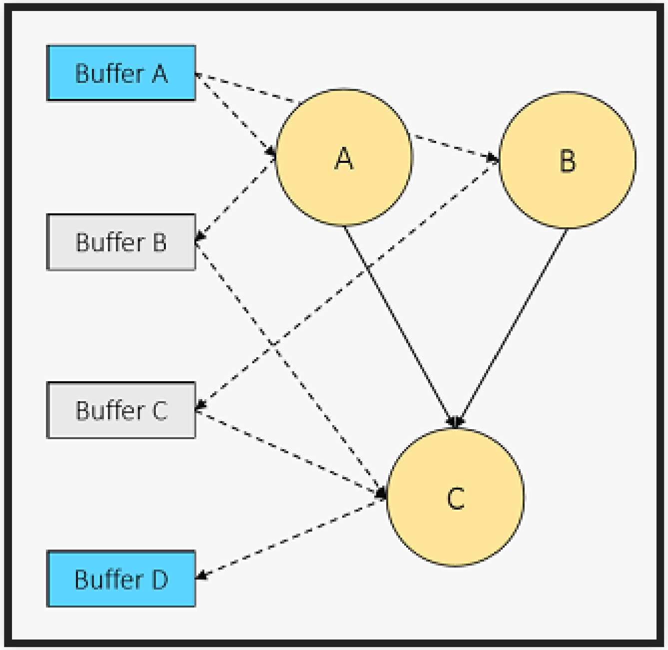


- As these commands are required to execute in sequence, they are enqueued to OpenCL with events between each one
- There are no copies required for buffers B and C as they remain on the device

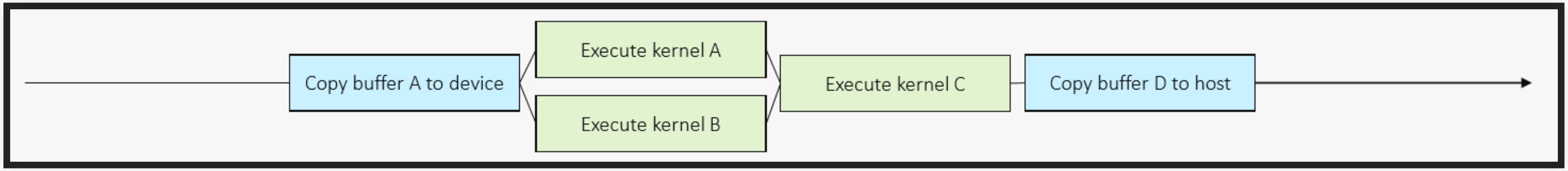
```
queue cpuQueue(cpu_selector{}, async_handler{});
cpuQueue.submit([&](handler &cgh){ // CG A
  auto in = bufA.get_access<access::mode::read>(cgh);
  auto out = bufB.get_access<access::mode::write>(cgh);
  cgh.parallel_for(range<1>(dA.size()), func(in, out)); });

cpuQueue.submit([&](handler &cgh){ // CG B
  auto in = bufB.get_access<access::mode::read>(cgh);
  auto out = bufC.get_access<access::mode::write>(cgh);
  cgh.parallel_for(range<1>(dA.size()), func(in, out)); });

cpuQueue.submit([&](handler &cgh){ // CG C
  auto in = bufC.get_access<access::mode::read>(cgh);
  auto out = bufD.get_access<access::mode::write>(cgh);
  cgh.parallel_for(range<1>(dA.size()), func(in, out)); });

cpuQueue.wait_and_throw();
```



The third command group has data dependencies on the previous, the first two command groups run concurrently



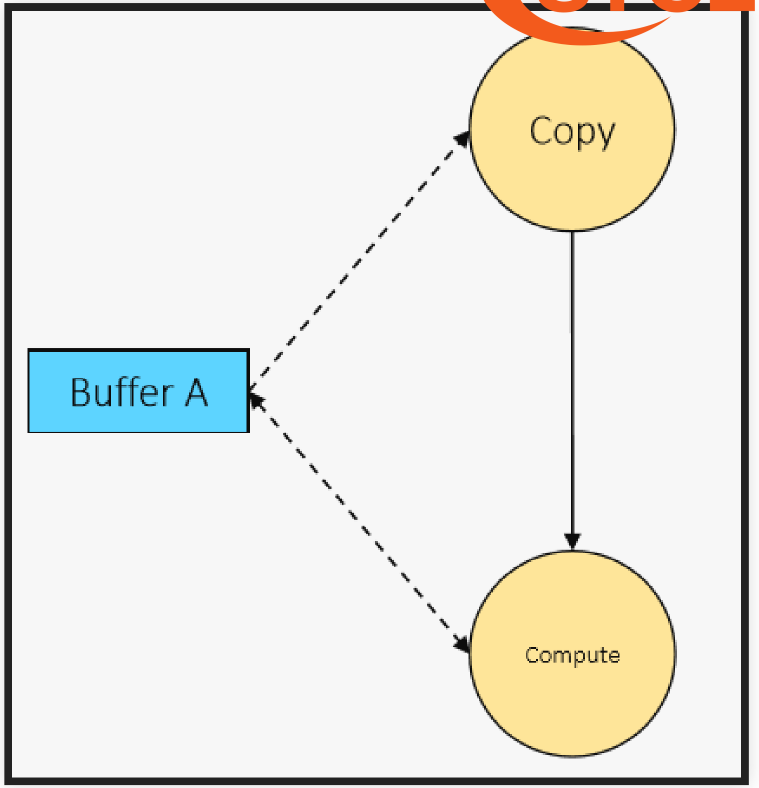
- As command groups A and B are only reading from buffer A, they can both access it concurrently
- As there are no dependencies between command groups A and B they can be run in parallel
- Again, there are no copies required for buffers B and C as they remain on the device



EXPLICIT COPY COMMANDS

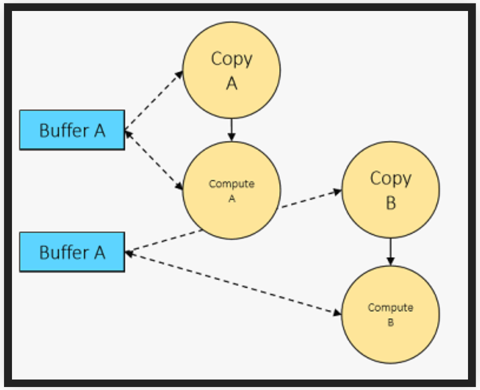
- As well as SYCL kernel functions a command group can also contain explicit copy commands
  - These commands enqueue a copy operation to the SYCL scheduler with the same data dependency analysis
  - This can be used to perform double buffering of copy and compute

```
queue cpuQueue(cpu_selector{}, async_handler{});
cpuQueue.submit([&](handler &cgh){ // Copy
  auto ptr = bufA.get_access<access::mode::read>(cgh);
  cgh.copy(data, ptr); });

cpuQueue.submit([&](handler &cgh){ // Compute
  auto ptr = bufA.get_access<access::mode::read_write>(cgh);
  cgh.parallel_for(range<1>(dA.size()), func(ptr)); });

cpuQueue.wait_and_throw();
```

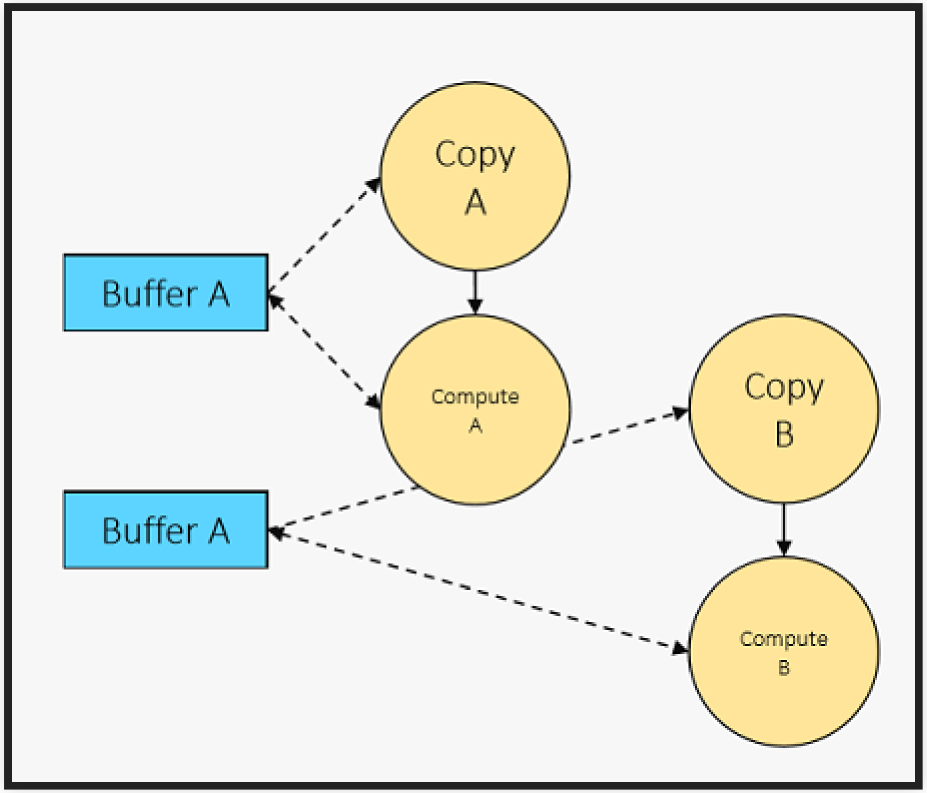


The command group performing the copy must complete before the command group performing the computation

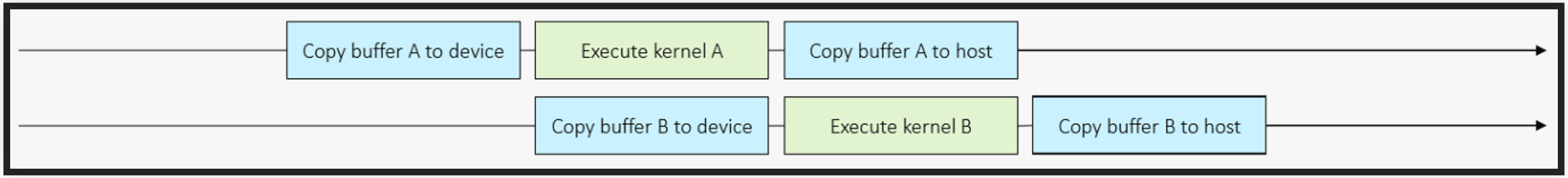
```
queue cpuQueue(cpu_selector{}, async_handler{});
cpuQueue.submit([&](handler &cgh){ // Copy A
  auto ptr = bufA.get_access<access::mode::read>(cgh);
  cgh.copy(data, ptr); });

cpuQueue.submit([&](handler &cgh){ // Compute A
  auto ptr = bufA.get_access<access::mode::read_write>(cgh);

  cgh.parallel_for(range<1>(dA.size()), func(ptr)); });

cpuQueue.submit([&](handler &cgh){ // Copy B
  auto ptr = bufB.get_access<access::mode::read>(cgh);
  cgh.copy(data, ptr); });

cpuQueue.submit([&](handler &cgh){ // Compute B
  auto ptr = bufB.get_access<access::mode::read_write>(cgh);
  cgh.parallel_for(range<1>(dA.size()), func(ptr)); });

cpuQueue.wait_and_throw();
```



The command group copying buffer B can execute concurrently with the compute of buffer A



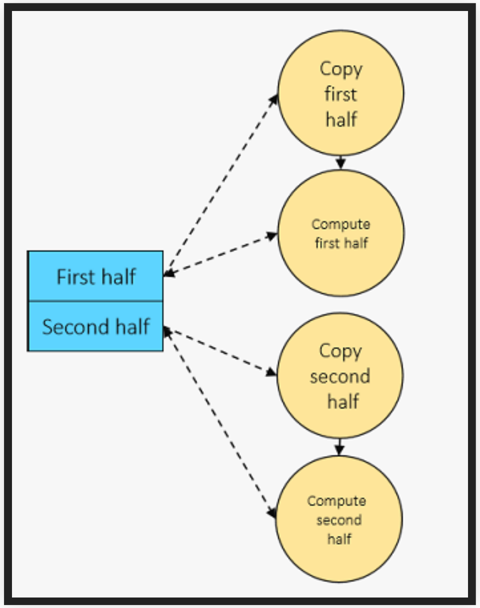
- The copy and compute on buffer A and buffer B are independent so they have separate chains of events
- This means that they can be run in parallel, double buffering copy and compute



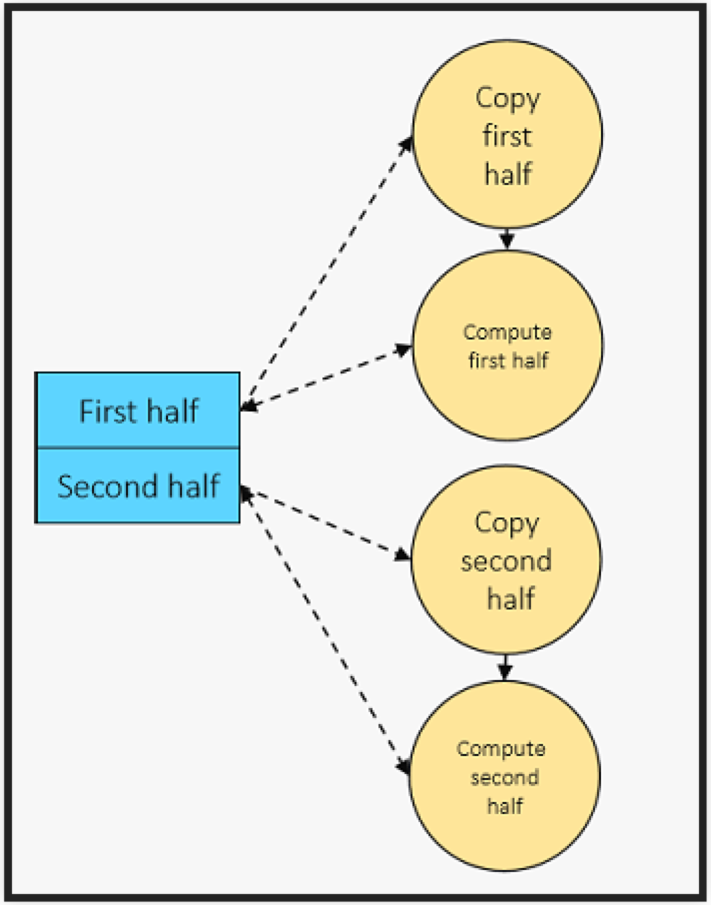
RANGED ACCESSORS

- By default accessors access the entire buffer, however it's possible to access only a region of a buffer
  - Only the region of the buffer that you are accessing is copied
  - This is particularly useful for tiling larger data

```
queue cpuQueue(cpu_selector{}, async_handler{});
cpuQueue.submit([&](handler &cgh){ // Copy first half
  auto ptr = bufA.get_access<access::mode::read>(cgh, halfSize, origin);
  cgh.copy(data, ptr);

cpuQueue.submit([&](handler &cgh){ // Compute first half
  auto ptr = bufA.get_access<access::mode::read_write>(cgh, halfSize, origin)
  cgh.parallel_for(range<1>(dA.size()), func(ptr)); });

cpuQueue.submit([&](handler &cgh){ // Copy second half
  auto ptr = bufB.get_access<access::mode::read>(cgh);
  cgh.copy(data, ptr);});

cpuQueue.submit([&](handler &cgh){ // Compute first half
  auto ptr = bufB.get_access<access::mode::read_write>(cgh, halfSize, origin)
  cgh.parallel_for(range<1>(dA.size()), func(ptr)); });

cpuQueue.wait_and_throw();
```



The first copy and compute operate on the first half of the buffer and the other copy and compute operate on the second half



- Each region of data is copied and then that region is computed
  - The entire buffer is copied back to the host at the end