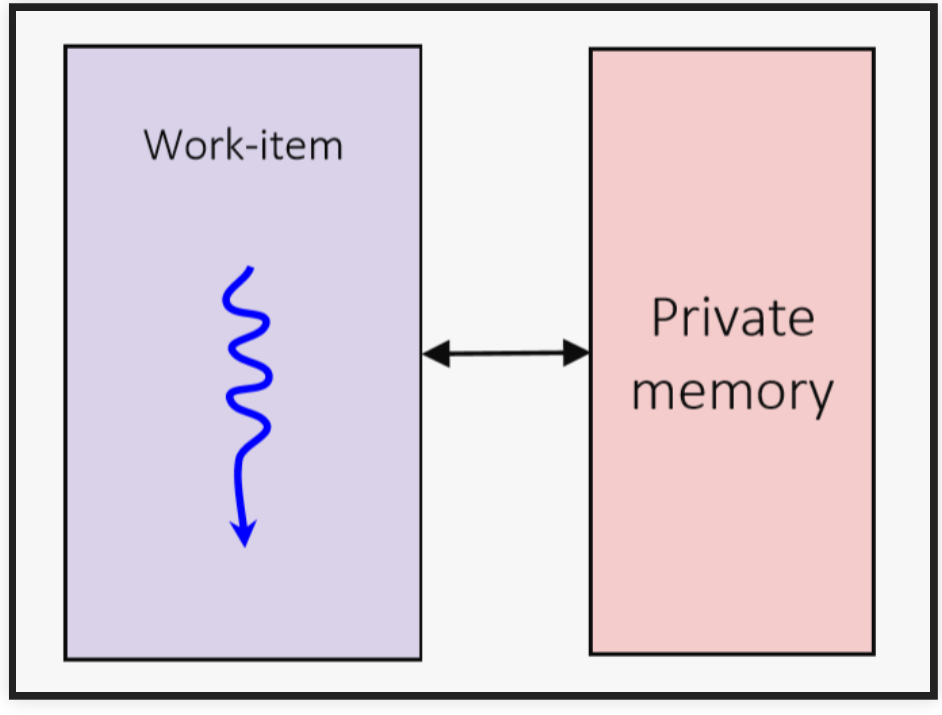
MANAGING DATA IN SYCL APPLICATIONS

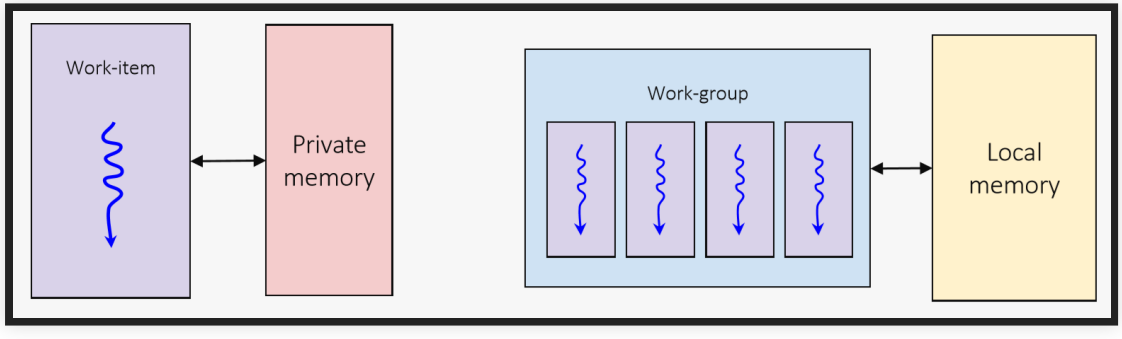
LEARNING OBJECTIVES

- Understand the SYCL memory model
- Learn how to use SYCL buffers and accessors
- Understand memory access in different address spaces
- Learn about different ways to access the data
- Learn about execution ordering using data dependencies
- Learn about how SYCL synchronizes data

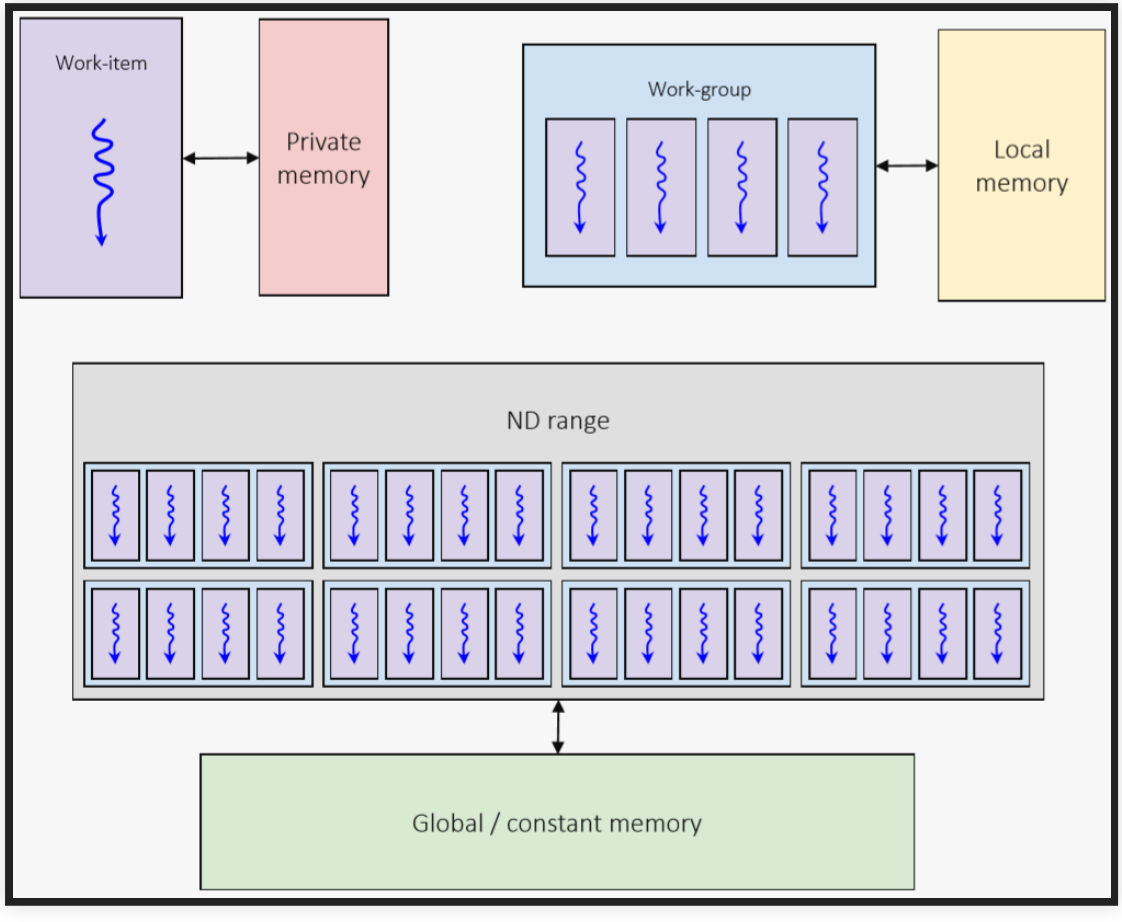
THE SYCL MEMORY MODEL

- Each work-item can access a dedicated region of **private memory**
- A work-item cannot access the private memory of another work-item



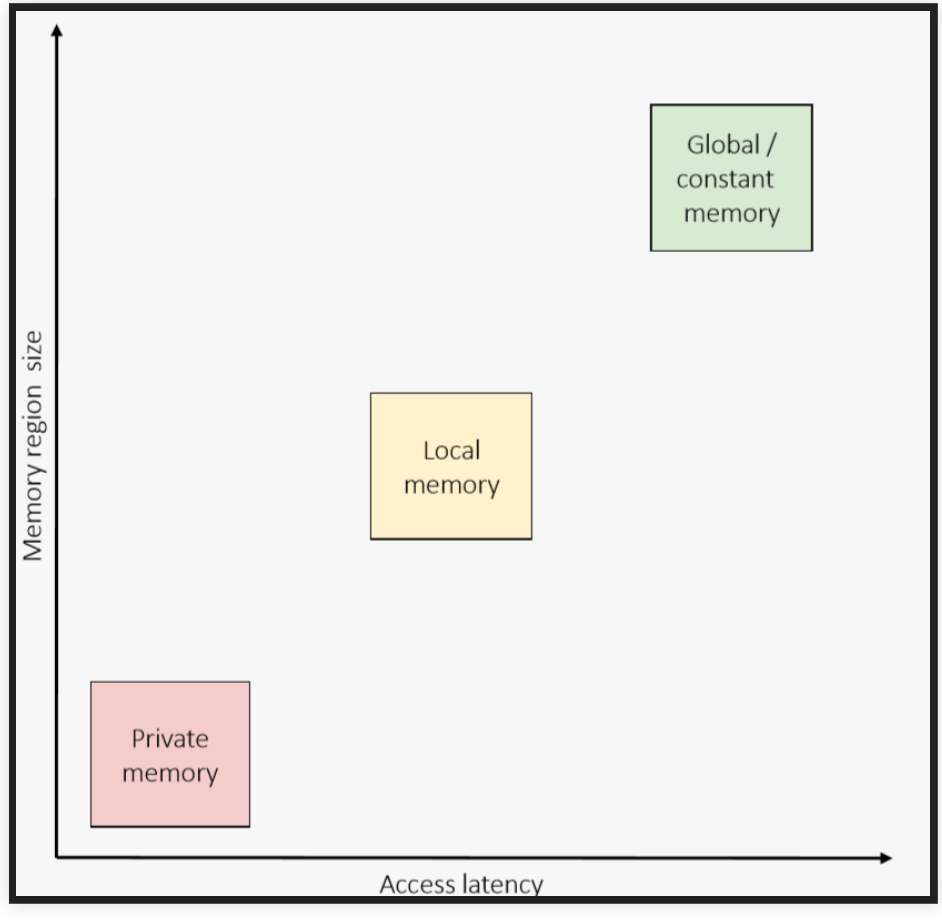


- Each work-item can access a dedicated region of **local memory** accessible to all work-items in a work-group
- A work-item cannot access the local memory of another workgroup



- Each work-item can access a single region of **global memory** that's accessible to all work-items in a ND-range
- Each work-item can also access a region of global memory reserved as **constant memory**, which is read-only

- Each memory region has a different size and access latency
- Global / constant memory is larger than local memory and local memory is larger than private memory
- Private memory is faster than local memory and local memory is faster than global / constant memory



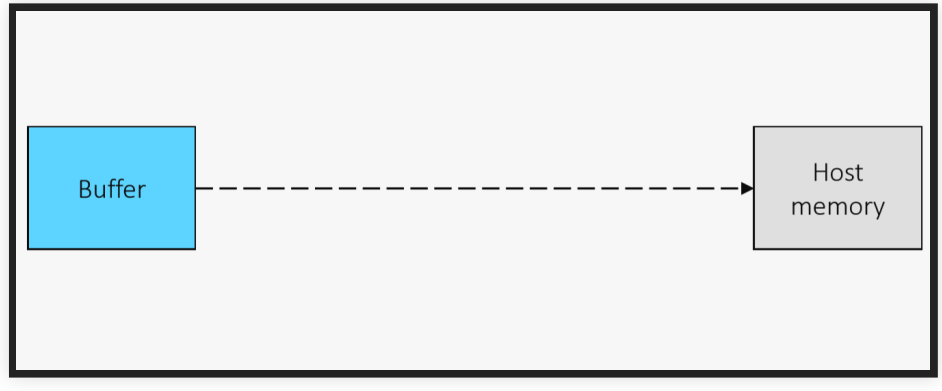
SYCL BUFFERS & ACCESSORS

- SYCL separates the storage and access of data
  - A SYCL buffer manages data across the host and any number of devices
  - A SYCL accessor requests access to data on the host or on a device for a specific SYCL kernel function
- Accessors are also used to access data within a SYCL kernel function
  - This means they are declared in the host code but captured by and then accessed within a SYCL kernel function

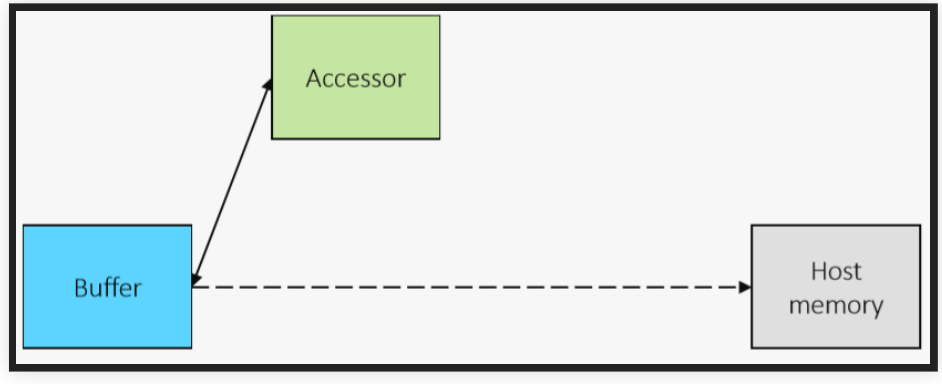
```
buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

gpuQueue.submit([&](handler &cgh){
  auto inA = bufA.get_access<access::mode::read>(cgh);
  auto inB = bufB.get_access<access::mode::read>(cgh);
  auto out = bufO.get_access<access::mode::write>(cgh);
  cgh.parallel_for<add>(range<1>(dA.size()),
    [=](id<1> i){ out[i] = inA[i] + inB[i]; });
});
```

- In order to achieve this the host compiler and the SYCL device compiler interpret accessors differently
- The host compiler interprets accessors as host objects which instruct the SYCL runtime of data dependencies for a SYCL kernel function
- The SYCL device compiler interprets accessors as a wrapper on the argument to the SYCL kernel function (a pointer to device memory) that is used to access data on the device

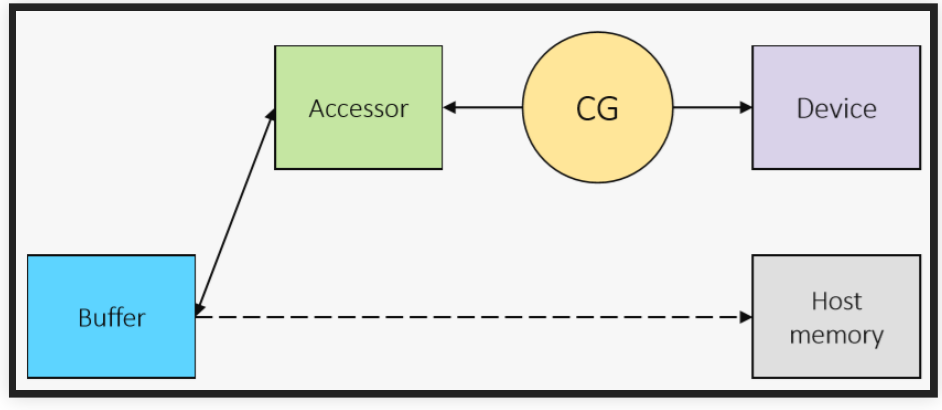
- A SYCL buffer can be constructed with a pointer to host memory
- For the lifetime of the buffer this memory is owned by the SYCL runtime
- When a buffer object is constructed it will not allocate or copy to device memory at first
- This will only happen once the SYCL runtime knows the data needs to be accessed and where it needs to be accessed



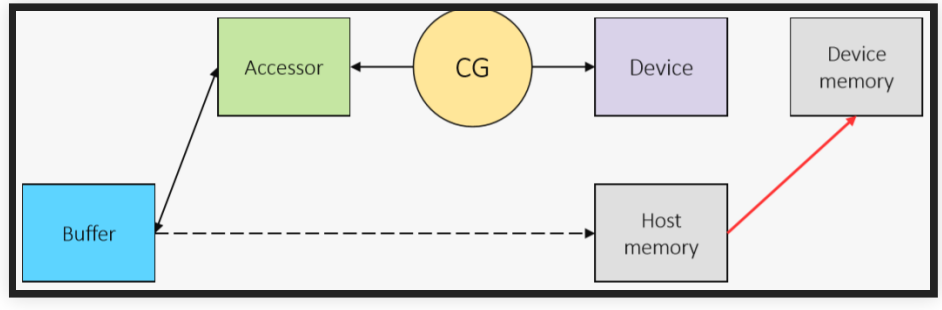
- Constructing an accessor specifies a request to access the data managed by the buffer
- There are a range of different types of accessor which provide different ways to access data



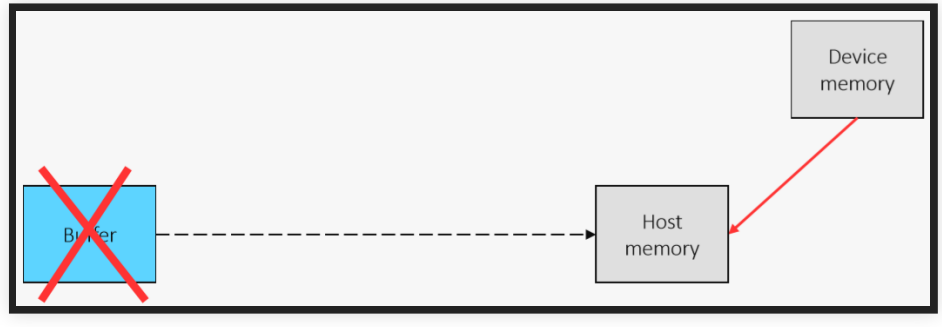
- When an accessor is constructed it is associated with a command group via the handler object
- This connects the buffer that is being accessed, the way in which it's being accessed and the device that the command group is being submitted to



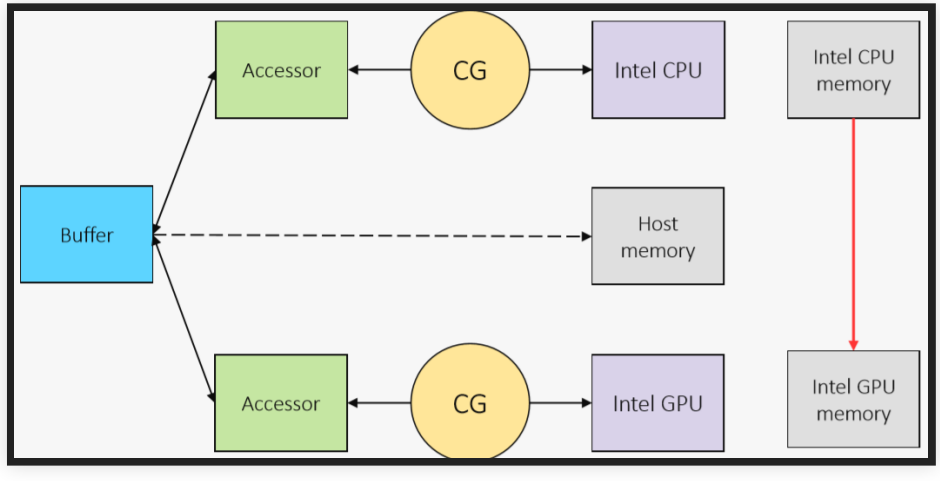
- Once the SYCL scheduler selects the command group to be executed it must first satisfy its data dependencies
- This means allocating and copying data to the device the data is being accessed on if necessary
- If the most recent copy of the data is already on the device then the runtime will not copy again



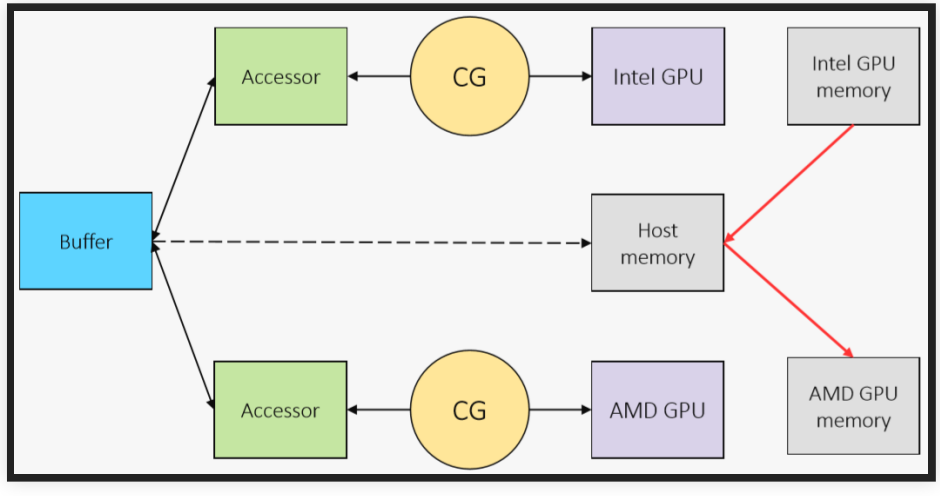
- Data will remain in device memory after kernels finish executing until another command group requests access in a different device or on the host
- When the buffer object is destroyed it will wait for any outstanding work that is accessing the data to complete and then copy back to the original host memory



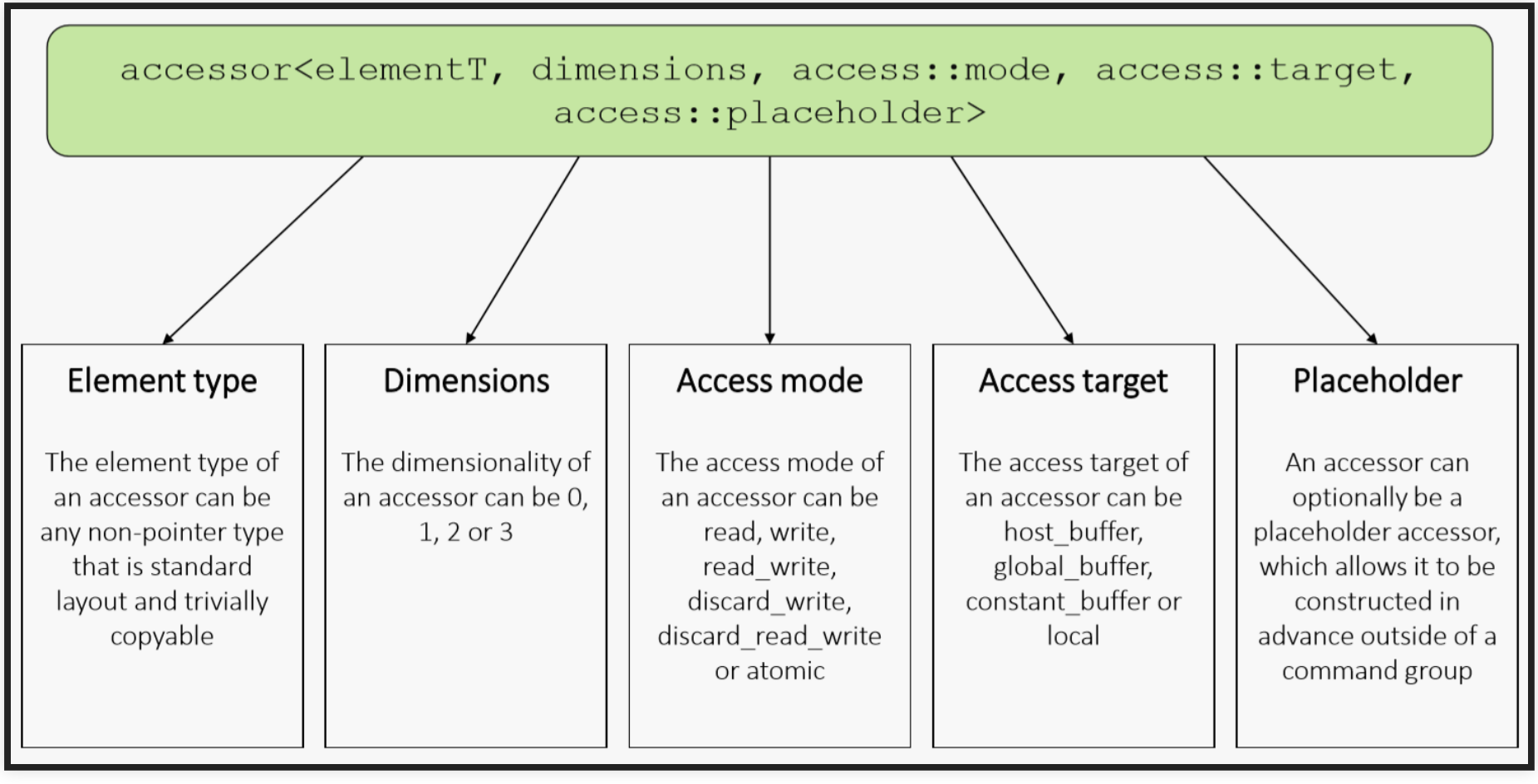
- If a buffer is accessed on one device when the latest copy of the data is on another device, the data will be copied
- If the two devices are of the same context the data can be copied directly



- If the devices are of different contexts the data must be copied via host memory
- It's important to consider this as it could incur further overhead when copying between devices

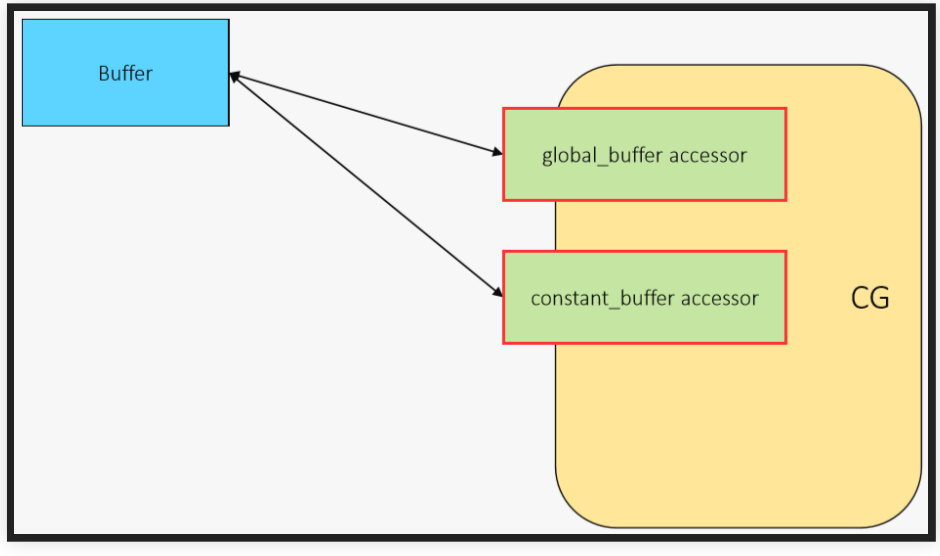


DIFFERENT KINDS OF ACCESSORS



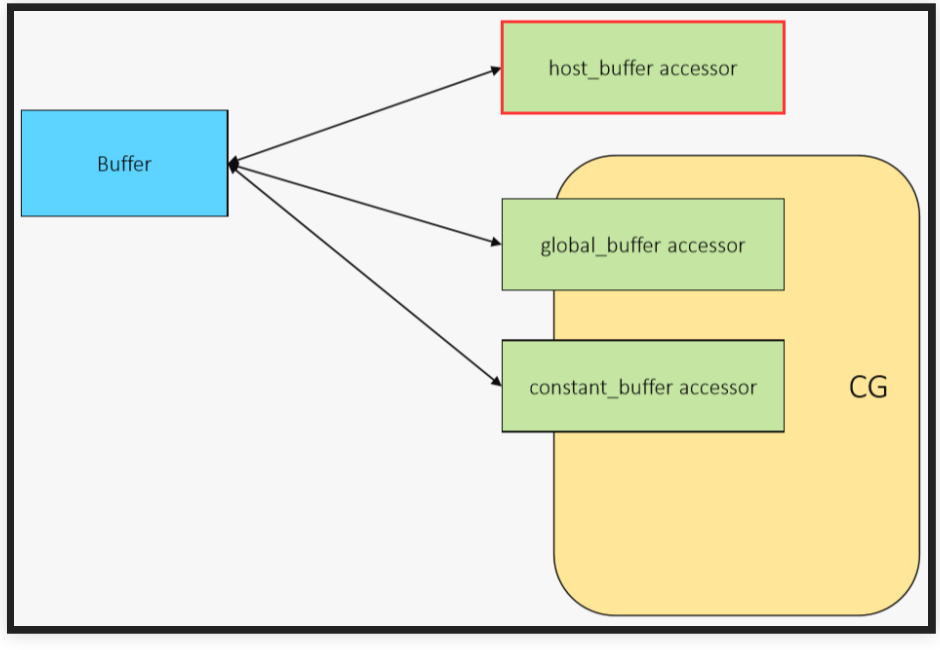
```
accessor<elementT, dimensions, access::mode, access::target, access::placeholder>
```

- Accessors using the access::target **global_buffer** and **constant_buffer** will allocate memory on the device in the **global** and **constant** address space respectively
- Accessors of these access targets must be constructed using a buffer
- A useful shortcut to construct an accessor with the access::target **global_buffer** is to use the **get_access** member function of the buffer
- Element type and dimensionality must be the same



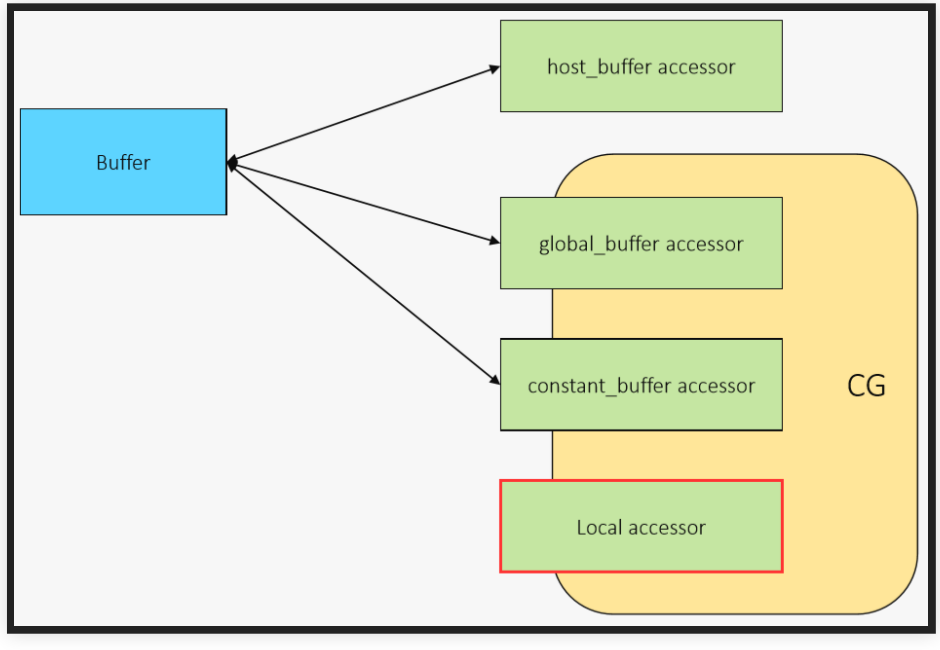
```
accessor<elementT, dimensions, access::mode, access::target, access::placeholder>
```

- Accessors using the access::target **host_buffer** will make the data available immediately on the host
- This access targets must be constructed using a buffer outside of a command group
- A useful shortcut to construct an accessor with the access::target **host_buffer** is to use the **get_access** member function of the buffer
- Note that a host accessor will block other accessors until it's destroyed

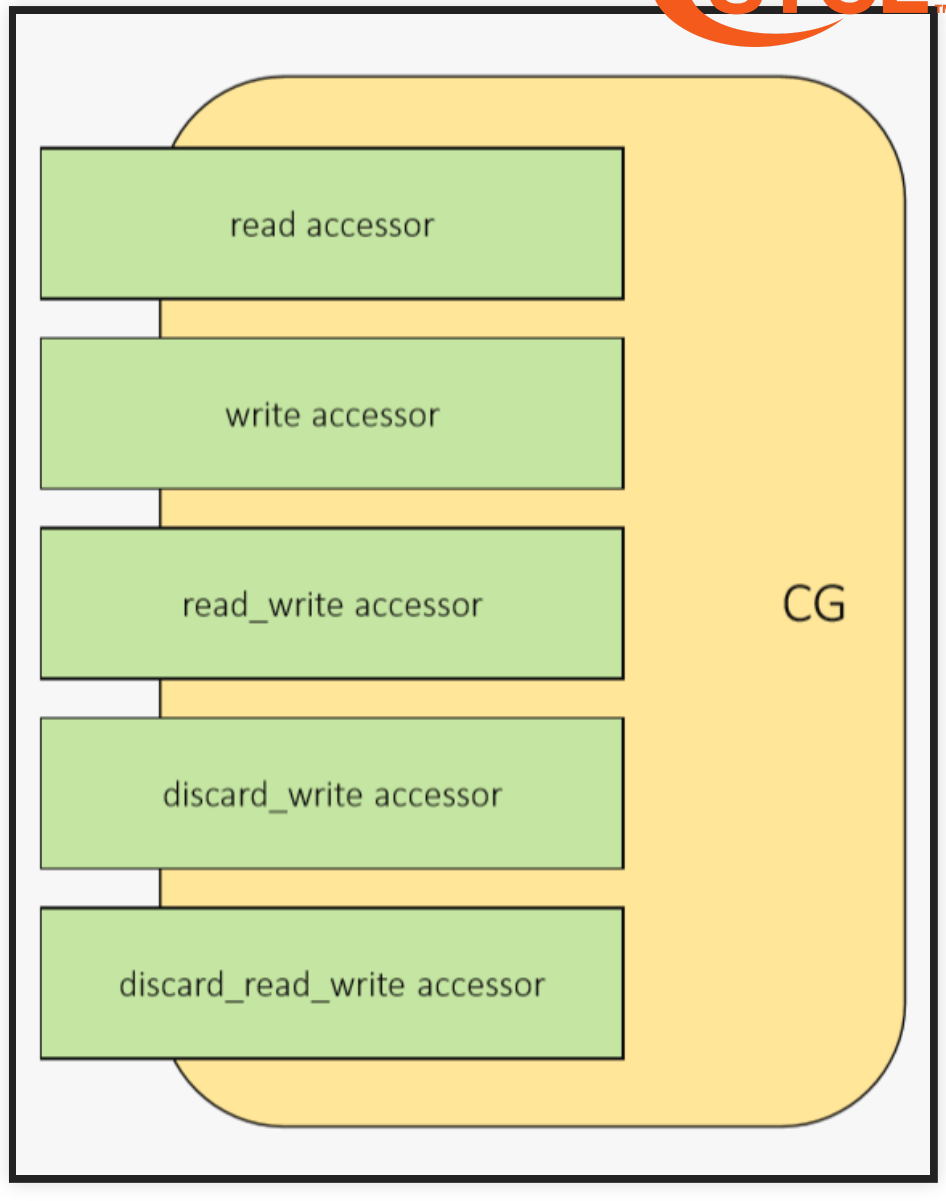


```
accessor<elementT, dimensions, access::mode, access::target, access::placeholder>
```

- Accessors using the access::target **local** will allocate memory in the **local** address space per work-group
- This access targets does not require a buffer to be constructed as it's temporary memory allocation for the duration of a SYCL kernel function invocation



- A **read** accessor instructs the SYCL runtime that the SYCL kernel function will read the data – cannot be written to within a SYCL kernel function
- A **write** accessor instructs the SYCL runtime that the SYCL kernel function will modify the data – creating a dependency for future command groups
- A **discard_*** accessor instructs the SYCL runtime that the SYCL kernel function does not need the initial values of the data – removing the dependency on previous command groups



ACCESSOR RESOLUTION

- If a command group has accessors to the same buffer with conflicting access modes they are resolved into one
  - read & write => `read_write`
  - `read_write` & `discard_write` => `read_write`
- Within the SYCL kernel function there are still multiple accessors, but they alias to the same memory address

```
buffer<float, 1> bufI(dA.data(), range<1>(dA.size()));
buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

gpuQueue.submit([&](handler &cgh){
  auto inA = bufI.get_access<access::mode::read>(cgh);
  auto inB = bufI.get_access<access::mode::write>(cgh);
  auto out = bufO.get_access<access::mode::write>(cgh);
  cgh.parallel_for<add>(range<1>(dA.size()),
    [=](id<1> i){ out[i] = inA[i] + inB[i]; });
});
```

- Here **inA** and **inB** both point to bufI but one is access::mode::read and one is access::mode::write
- So the SYCL runtime will treat them both as access::mode::read_write
- Both will point to a single allocation of global memory on the devices
- The runtime will resolve the data dependency into access::mode::read_write

USING DATA WITH ACCESSORS

- There are a few different ways to access the data represented by an accessor
  - The subscript operator can take an **id**
    - Must be the same dimensionality of the accessor
    - For dimensions > 1, linear address is calculated in row major
- Nested subscript operators can be called for each dimension taking a **size_t**
  - E.g. a 3-dimensional accessor: acc[x][y][z] = …
- A pointer to memory can be retrieved by calling **get_pointer**
  - This returns a **multi_ptr**, which is a wrapper class for pointers to the memory in the relevant memory space

```
buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

gpuQueue.submit([&](handler &cgh){
  auto inA = bufA.get_access<access::mode::read>(cgh);
  auto inB = bufB.get_access<access::mode::read>(cgh);
  auto out = bufO.get_access<access::mode::write>(cgh);
  cgh.parallel_for<add>(range<1>(dA.size()),
    [=](id<1> i){
    out[i] = inA[i] + inB[i];
  });
});
```

- Here we access the data of the accessor by passing in the id object passed to the SYCL kernel function

```
buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

gpuQueue.submit([&](handler &cgh){
  auto inA = bufA.get_access<access::mode::read>(cgh);
  auto inB = bufB.get_access<access::mode::read>(cgh);
  auto out = bufO.get_access<access::mode::write>(cgh);
  cgh.parallel_for<add>(rng, [=](id<3> i){
    auto ptrA = inA.get_pointer();
    auto ptrB = inB.get_pointer();
    auto ptrO = out.get_pointer();
    auto linearId = i.get_linear_id();

    ptrA[linearId] = ptrB[linearId] + ptrO[linearId];
  });
});
```

- Here we retrieve the underlying pointer for each of the accessors
- We then access the pointer using the linearized id by calling the **get_linear_id** member function on the **item** class
- Again this linearization is calculated in row major

QUESTIONS