

# MANAGING DATA

# LEARNING OBJECTIVES

- Learn about the buffer/accessor model and USM for managing data
- Learn how to use these
- Learn how to use data in a kernel function
- Learn how to synchronize data

## MEMORY MODELS

- In SYCL there are two models for managing data:
  - The buffer/accessor model.
  - The USM (unified shared memory) model.
- Which model you choose can have an effect on how you enqueue kernel functions.

# 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

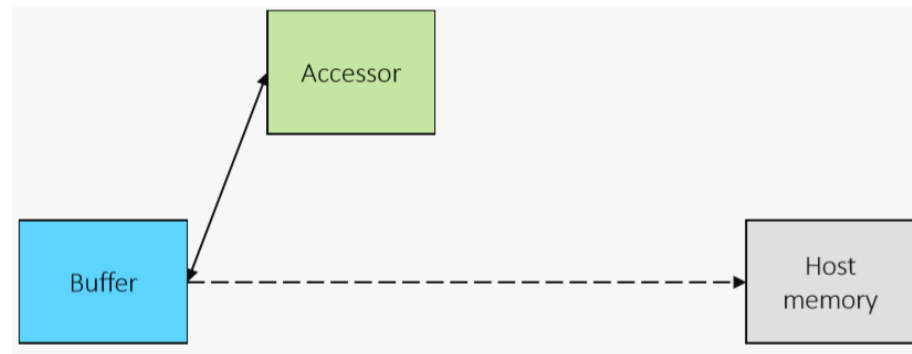
## SYCL BUFFERS & ACCESSORS

- 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



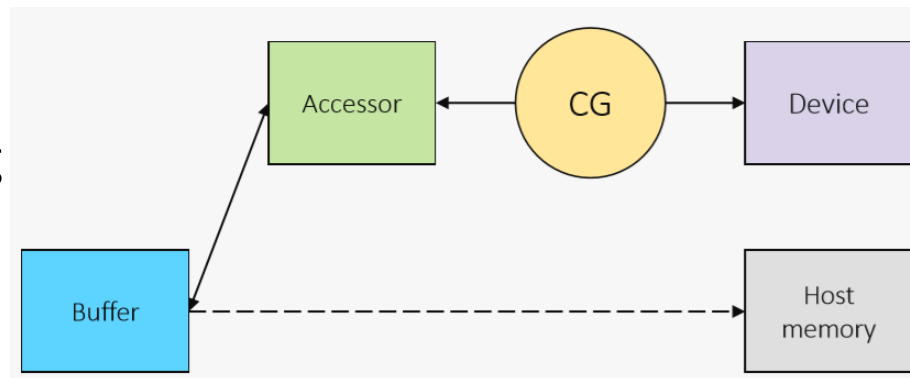
## SYCL BUFFERS & ACCESSORS

- 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



## SYCL BUFFERS & ACCESSORS

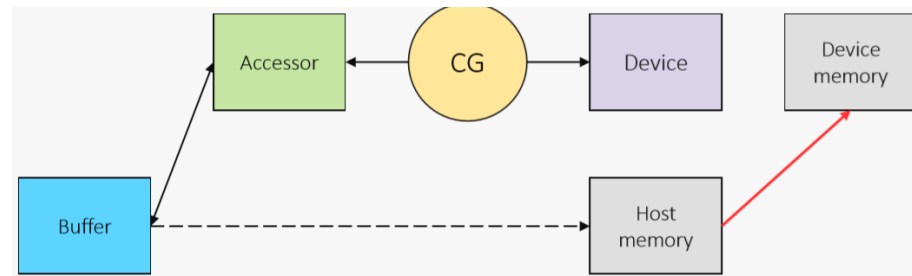
- When an accessor is constructed it is associated with a command group via the handler object
- This connects the buffer that is being accessed, the way in which it's being accessed and the device that the command group is being submitted to





## SYCL BUFFERS & ACCESSORS

- 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



## SYCL BUFFERS & ACCESSORS

- 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



# BUFFER CLASS

```
template <typename dataT, int dimensions>  
sycl::buffer;
```

- A buffer manages data across the host application and kernel functions executing on device(s).
- It has a typename which specifies the type of the elements of data it manages.
- It has a dimensionality which specifies the dimensionality that the elements of data are represented in.

# CONSTRUCTING A BUFFER

```
int var = 42;  
auto buf = sycl::buffer{&var, sycl::range{1}};
```

- A buffer can be constructed from a pointer to data for it to manage and a range which describes the number of elements of data.
- Using CTAD the type and the dimensionality can be inferred.

# ACCESSOR CLASS

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

## Element type

The element type of an accessor can be any non-pointer type that is standard layout and trivially copyable

## Dimensions

The dimensionality of an accessor can be 0, 1, 2 or 3

## Access mode

The access mode of an accessor can be read, write, read\_write, discard\_write, discard\_read\_write or atomic

## Access target

The access target of an accessor can be host\_buffer, global\_buffer, constant\_buffer or local

## Placeholder

An accessor can optionally be a placeholder accessor, which allows it to be constructed in advance outside of a command group

# ACCESSOR CLASS

- There are many different ways to use the accessor class.
  - Accessing data on a device.
  - Accessing data immediately in the host application.
  - Allocating local memory.
- For now we are going to focus on accessing data on a device.

# CONSTRUCTING AN ACCESSOR

```
auto acc = sycl::accessor{bufA, cgh};
```

- There are many ways to construct an accessor.
- The accessor class supports CTAD so it's not necessary to specify all of the template arguments.
- The most common way to construct an accessor is from a buffer and a handler associated with the command group function you are within.
  - The element type and dimensionality are inferred from the buffer.
  - The `access::target` is defaulted to `access::target::global_buffer`.
  - The `access::mode` is defaulted to `access::mode::read_write`.

## SPECIFYING THE ACCESS MODE

```
auto readAcc = sycl::accessor{bufA, cgh, sycl::read_only};  
auto writeAcc = sycl::accessor{bufB, cgh, sycl::write_only};
```

- When constructing an accessor you will likely also want to specify the `access::mode`
- You can do this by passing one of the CTAD tags:
  - `read_only` will result in `access::mode::read`.
  - `write_only` will result in `access::mode::write`.



## SPECIFYING NO INITIALIZATION

```
auto acc = sycl::accessor{buf, cgh, sycl::no_init};
```

- When constructing an accessor you may also want to discard the original data of a buffer.
- You can do this by passing the `no_init` property.

## ACCESS MODES

- 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 **no\_init** 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 more than one accessor to the same buffer with conflicting `access::mode` they are resolved into one:
  - `read & write => read_write`.
- If a command group has more than one accessor to the same buffer all must have the `no_init` property for it to apply.
- Within the SYCL kernel function there are still multiple accessors, but they alias to the same memory address.

# ACCESSOR RESOLUTION

```
gpuQueue.submit([&](handler &cgh){  
    auto in = sycl::accessor{buf, cgh, sycl::read_only};  
    auto out = sycl::accessor{buf, cgh, sycl::write_only};  
});
```

- Here `in` and `out` both point to `buf` 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 device(s).
- The runtime will resolve the data dependency into `access::mode::read_write`.

# OPERATOR[]

```
gpuQueue.submit([&](handler &cgh) {  
    auto inA = sycl::accessor{bufA, cgh, sycl::read_only};  
    auto inB = sycl::accessor{bufB, cgh, sycl::read_only};  
    auto out = sycl::accessor{bufO, cgh, sycl::write_only};  
    cgh.single_task<add>([=] {  
        out[0] = inA[0] + inB[0];  
    });  
});
```

- As well as specifying data dependencies an accessor can also be used to access the data from within a kernel function.
- You can do this by calling operator[ ] on the accessor.
  - This operator can take an `id` or a `size_t`.

## USM MODEL

- There are different ways USM memory can be allocated; host, device and shared.
- We're going to focus on shared and device allocations.

## USM ALLOCATION TYPES

Type	Description	Accessible on host?	Accessible on device?	Located on
device	Allocations in device memory	✗	✓	device
host	Allocations in host memory	✓	✓	host
shared	Allocations shared between host and device	✓	✓	Can migrate between host and device

**Figure 6-1.** *USM allocation types*

(from book)

# MALLOC\_DEVICE

```
void* malloc_device(size_t numBytes, const queue& syclQueue, const property_list &propList = {});  
  
template <typename T>  
T* malloc_device(size_t count, const queue& syclQueue, const property_list &propList = {});
```

- A USM device allocation is performed by calling one of the `malloc_device` functions.
- Both of these functions allocate the specified region of memory on the device associated with the specified queue.
- The pointer returned is only accessible in a kernel function running on that device.
- Synchronous exception if the device does not have `aspect::usm_device_allocations`.
- This is a blocking operation.
- Calls the underlying `cudaMalloc` if using CUDA backend.



## MALLOC\_SHARED

```
void* malloc_shared(size_t numBytes, const queue& syclQueue, const property_list &propList = {});  
  
template <typename T>  
T* malloc_shared(size_t count, const queue& syclQueue, const property_list &propList = {});
```

- Both of these functions allocate the specified region of memory on the device associated with the specified queue, as well as host.
- The pointer returned is accessible in CPU code as well as device kernel code, for the device attached to the queue.
- Synchronous exception if the device does not have `aspect::usm_device_allocations`
- This is a blocking operation.
- Calls the underlying `cudaMallocManaged` if using CUDA backend.
- Convenient API but potentially slower than `malloc_device` with explicit `memcpy`s.

# FREE

```
void free(void* ptr, queue& syclQueue);
```

- In order to prevent memory leaks USM device allocations must be free by calling the free function.
- The queue must be the same as was used to allocate the memory.
- This is a blocking operation.

# MEMCPY

```
event queue::memcpy(void* dest, const void* src, size_t numBytes, const std::vector &depEvents);
```

- Data can be copied to and from a USM device allocation by calling the queue's `memcpy` member function.
- The source and destination can be either a host application pointer or a USM device allocation.
- This is an asynchronous operation enqueued to the queue.
- An event is returned which can be used to synchronize with the completion of copy operation.
- May depend on other events via `depEvents`

## MEMSET & FILL

```
event queue::memset(void* ptr, int value, size_t numBytes, const std::vector &depEvents);  
event queue::fill(void* ptr, const T& pattern, size_t count, const std::vector &depEvents);
```

- The additional queue member functions `memset` and `fill` provide operations for initializing the data of a USM device allocation.
- The member function `memset` initializes each byte of the data with the value interpreted as an unsigned char.
- The member function `fill` initializes the data with a recurring pattern.
- These are also asynchronous operations.

# EXERCISE

Code\_Exercises/Exercise\_03\_Scalar\_Add

Implement a SYCL application that adds two variables and returns the result using USM and Buffers.

