



# Hands-On HPC Application Development Using C++ and SYCL

Dounia Khaldi, Hugh Delaney, James Reinders, Michael Wong, Ronan Keryell

# MANAGING DATA

# LEARNING OBJECTIVES

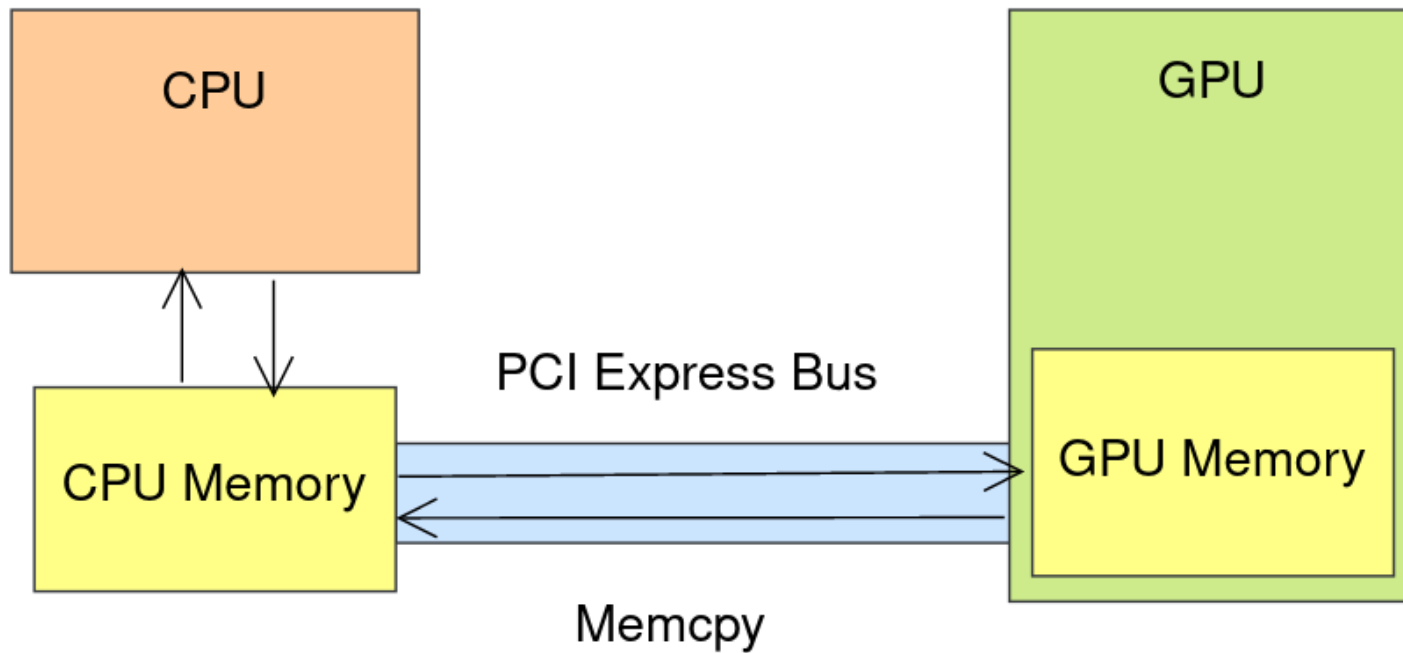
- Learn about the USM and buffer/accessor models for managing data
- Learn how to allocate, transfer and free memory using USM.
- Learn how a buffer synchronizes data
- Learn how to access data in a kernel function

## 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.

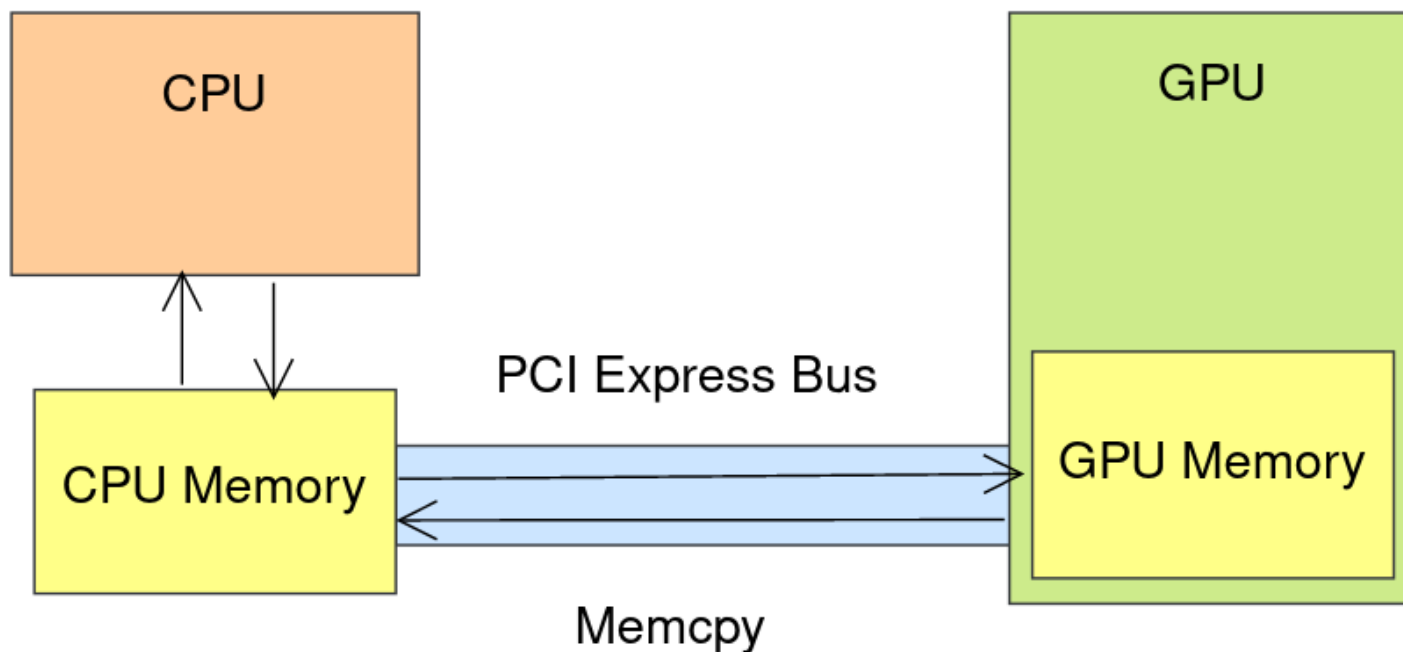
## CPU AND GPU MEMORY

- A GPU has its own memory, separate to CPU memory.
- In order for the GPU to use memory from the CPU, the following actions must take place (either explicitly or implicitly):
  - Memory allocation on the GPU.
  - Data migration from the CPU to the allocation on the GPU.
  - Some computation on the GPU.
  - Migration of the result back to the CPU.



## CPU AND GPU MEMORY

- Memory transfers between CPU and GPU are a bottleneck.
- We want to minimize these transfers, when possible.



## USM ALLOCATION TYPES

- There are different ways USM memory can be allocated: host, device and shared.

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)



## USING USM - MALLOC DEVICE

```
// Allocate memory on device
T *device_ptr = sycl::malloc_device<T>(n, myQueue);

// Copy data to device
myQueue.memcpy(device_ptr, cpu_ptr, n * sizeof(T));

// ...
// Do some computation on device
// ...

// Copy data back to CPU
myQueue.memcpy(result_ptr, device_ptr, n * sizeof(T)).wait();

// Free allocated data
sycl::free(device_ptr, myQueue);
```

- It is important to free memory after it has been used to avoid memory leaks.

## USING USM - MALLOC SHARED

```
// Allocate shared memory
T *shared_ptr = sycl::malloc_shared<T>(n, myQueue);

// Shared memory can be accessed on host as well as device
for (auto i = 0; i < n; ++i)
    shared_ptr[i] = i;

// ...
// Do some computation on device
// ...

// Free allocated data
sycl::free(shared_ptr, myQueue);
```

- Shared memory is accessible on host and device.
- Performance of shared memory accesses may be poor depending on platform.

## SYCL BUFFERS & ACCESSORS

- SYCL provides an API which takes care of allocations and memcpy, as well as some other things.

## SYCL BUFFERS & ACCESSORS

- The buffer/accessor model 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

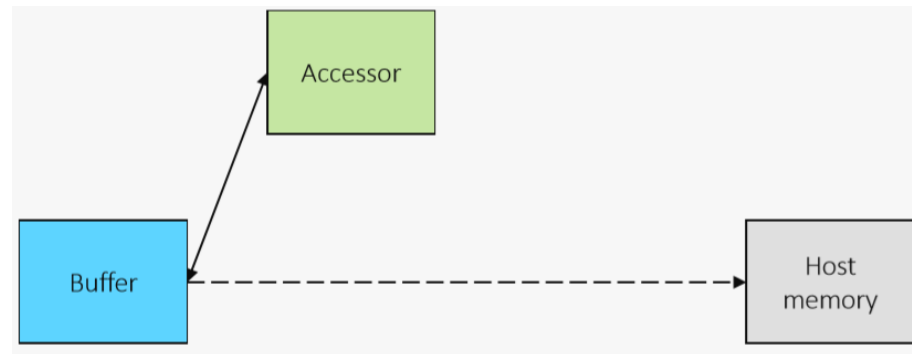
## SYCL BUFFERS & ACCESSORS

- 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



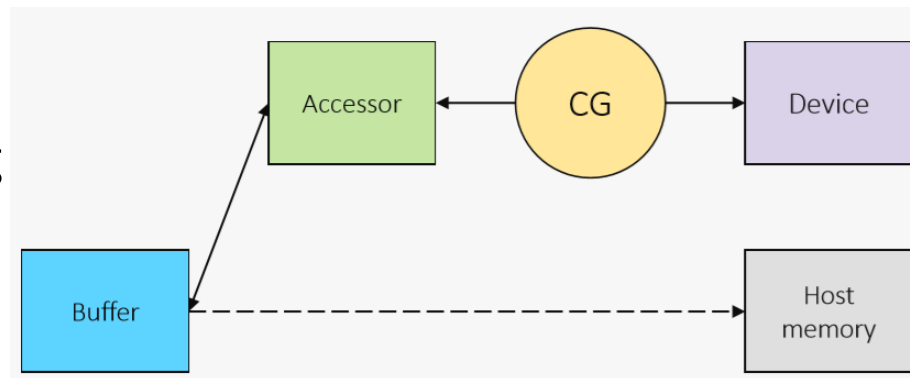
## SYCL BUFFERS & ACCESSORS

- 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



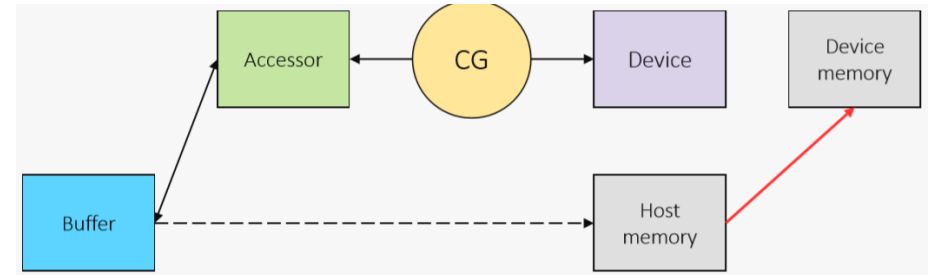
## SYCL BUFFERS & ACCESSORS

- 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

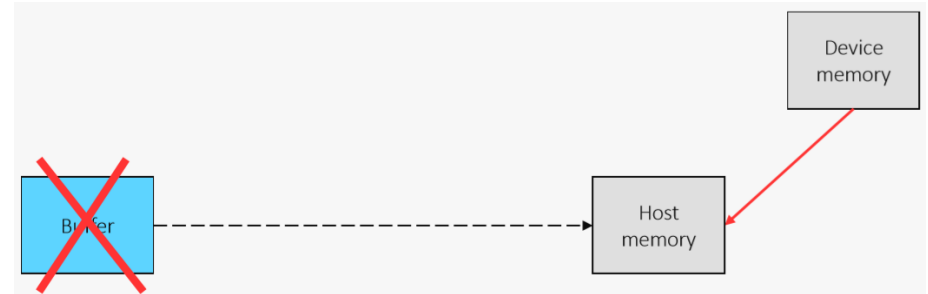
- Once the SYCL scheduler selects the command group to be executed it must first satisfy its data dependencies
- This means allocating and copying data to the device the data is being accessed on if necessary
- If the most recent copy of the data is already on the device then the runtime will not copy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



# SYCL BUFFERS & ACCESSORS

```
T var = 42;

{
    // Create buffer pointing to var.
    auto buf = sycl::buffer{&var, sycl::range<1>{1}};

    // ...
    // Do some computation on device. Use accessors to access buffer
    // ...

} // var updated here

assert(var != 42);
```

- A buffer is associated with a type, range and dimensionality. Dimensionality must be either 1, 2 or 3.
- Usually type and dimensionality can be inferred using CTAD.
- If a buffer is associated with some allocation in host memory, the host memory will be updated only once the buffer goes out of scope.

# 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::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.

# USING ACCESSORS

```
T var = 42;

{
    // Create buffer pointing to var.
    auto bufA = sycl::buffer{&var, sycl::range<1>{1}};
    auto bufB = sycl::buffer{&var, sycl::range<1>{1}};

    q.submit([&](sycl::handler &cgh) {
        auto accA = sycl::accessor{bufA, cgh, sycl::read_only};
        auto accB = sycl::accessor{bufA, cgh, sycl::no_init};

        cgh.single_task<mykernel>(...); // Do some work
    });

    // var updated here

    assert(var != 42);
}
```

- Buffers and accessors take care of memory migration, as well as dependency analysis.
  - More to come later on dependencies.



## 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<mykernel>([=] {  
        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.
  - `operator[]` for USM pointers must take a `size_t`, whereas `operator[]` for accessors can take a multi-dimensional `sycl::id` or a `size_t`.

# QUESTIONS

# EXERCISE

Code\_Exercises/Exercise\_3\_Scalar\_Add/source

Implement a SYCL application that adds two variables and returns the result using:

1. The USM memory model
2. The buffer/accessor memory model.