Unified Shared Memory (USM)

- What is USM?
- Types of USM
- *Code:* Implicit USM
- *Code:* Explicit USM
- Data Dependency in USM
- *Code:* Data Dependency in-order queues
- *Code:* Data Dependency out-of-order queues
- *Lab Exercise:* Unified Shared Memory

Learning Objectives

- Use new SYCL2020 features such as Unified Shared Memory to simplify programming.
- Understand implicit and explicit way of moving memory using USM.
- Solve data dependency between kernel tasks in optimal way.

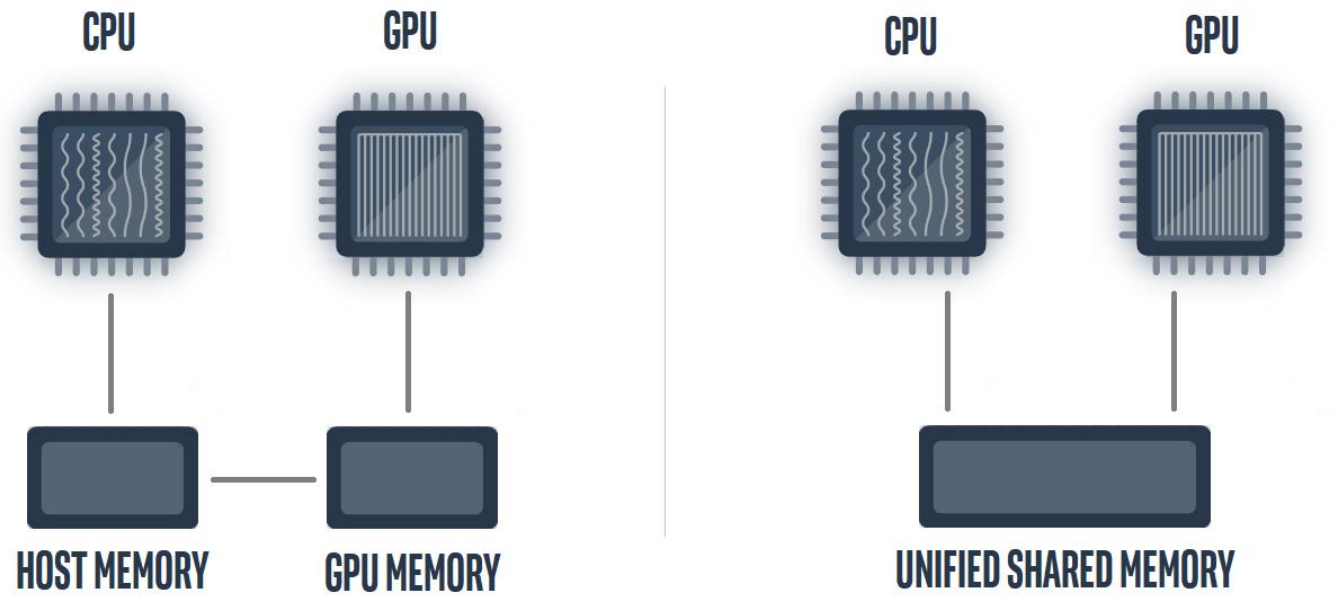
What is Unified Shared Memory?

Unified Shared Memory (USM) is a pointer-based memory management in SYCL. USM is a **pointer-based approach** that should be familiar to C and C++ programmers who use malloc or new to allocate data. USM **simplifies development** for the programmer when **porting existing C/C++ code** to SYCL.

Developer view of USM

The picture below shows **developer view of memory** without USM and with USM.

With USM, the developer can reference that same memory object in host and device code.



Types of USM

Unified shared memory provides both **explicit** and **implicit** models for managing memory.

Type	function call	Description	Accessible on Host	Accessible on Device
Device	malloc_device	Allocation on device (explicit)	NO	YES
Host	malloc_host	Allocation on host (implicit)	YES	YES
Shared	malloc_shared	Allocation can migrate between host and device (implicit)	YES	YES

USM Syntax

USM Initialization: The initialization below shows example of shared allocation using malloc_shared, the "q" queue parameter provides information about the device that memory is accessible.

```
int *data = malloc_shared<int>(N, q);
```

OR you can use familiar C++/C style malloc:

```
int *data = static_cast<int *>(malloc_shared(N * sizeof(int), q));
```

Freeing USM:

```
free(data, q);
```

USM Implicit Data Movement

The SYCL code below shows an implementation of USM using malloc_shared, in which data movement happens implicitly between host and device. Useful to **get functional quickly with minimum amount of code** and developers will not having worry about moving memory between host and device.

The SYCL code below demonstrates USM Implicit Data Movement:

```
#include <sycl/sycl.hpp>
using namespace sycl;

static const int N = 16;

int main() {
  queue q;
  std::cout << "Device : " << q.get_device().get_info<info::device::name>() << "\n";

  //# USM allocation using malloc_shared
  int *data = malloc_shared<int>(N, q);

  //# Initialize data array
  for (int i = 0; i < N; i++) data[i] = i;

  //# Modify data array on device
  q.parallel_for(range<1>(N), [=](id<1> i) { data[i] *= 2; }).wait();

  //# print output
  for (int i = 0; i < N; i++) std::cout << data[i] << "\n";
  free(data, q);
  return 0;
}
```

Exercise #1

1. Inspect the code in lab2/usm.cpp file
2. Compile and run this code
   icpx -fsycl usm.cpp -o usm
   ./usm

USM Explicit Data Movement

The SYCL code below shows an implementation of USM using malloc_device, in which data movement between host and device should be done explicitly by developer using memcpy. This allows developers to have more **controlled movement of data** between host and device.

The SYCL code below demonstrates USM Explicit Data Movement:

```
#include <sycl/sycl.hpp>
using namespace sycl;

static const int N = 16;

int main() {
  queue q;
  std::cout << "Device : " << q.get_device().get_info<info::device::name>() << "\n";

  //# initialize data on host
  int *data = static_cast<int *>(malloc(N * sizeof(int)));
  for (int i = 0; i < N; i++) data[i] = i;

  //# Explicit USM allocation using malloc_device
  int *data_device = malloc_device<int>(N, q);

  //# copy mem from host to device
  q.memcpy(data_device, data, sizeof(int) * N).wait();

  //# update device memory
  q.parallel_for(range<1>(N), [=](id<1> i) { data_device[i] *= 2; }).wait();

  //# copy mem from device to host
  q.memcpy(data, data_device, sizeof(int) * N).wait();

  //# print output
  for (int i = 0; i < N; i++) std::cout << data[i] << "\n";
  free(data_device, q);
  free(data);
  return 0;

}
```

Exercise #2

1. Inspect the code in lab2/usm_explicit.cpp file
2. Compile and run this code
   icpx -fsycl usm_explicit.cpp
   ./icpx -fsycl usm_explicit.cpp -o usm_explicit

3. What happens if you don't wait on the event returned from parallel_for on line 27?

When to use USM?

SYCL* Buffers are **powerful and elegant**. Use them if the abstraction applies cleanly in your application, and/or if buffers aren't disruptive to your development. However, replacing all pointers and arrays with buffers in a C++ program can be a burden to programmers so in this case consider using USM.

USM provides a familiar pointer-based C++ interface:

- Useful when **porting C++ code** to SYCL by minimizing changes
- Use shared allocations when porting code to **get functional quickly**. Note that shared allocation is not intended to provide peak performance out of box.
- Use explicit USM allocations when **controlled data movement** is needed.

Data dependency in USM

When using unified shared memory, dependences between tasks must be specified using events since tasks execute asynchronously and multiple tasks can execute simultaneously.

Programmers may either explicitly wait on event objects or use the depends_on method inside a command group to specify a list of events that must complete before a task may begin.

In the example below, the two kernel tasks are updating the same data array, these two kernels can execute simultaneously and may cause undesired result. The first task must be complete before the second can begin, the next section will show different ways the data dependency can be resolved.

```
q.parallel_for(range<1>(N), [=](id<1> i) { data[i] += 2; });

q.parallel_for(range<1>(N), [=](id<1> i) { data[i] += 3; });
```

Different options to manage data dependency when using USM:

- **wait()** on kernel task
- use **in_order** queue property
- use **depends_on** method

wait()

- Use **q.wait()** on kernel task to wait before the next dependent task can begin, however it will block execution on host.

```
q.parallel_for(range<1>(N), [=](id<1> i) { data[i] += 2; });
    q.wait();  // <--- wait() will make sure that task is complete before continuing
    q.parallel_for(range<1>(N), [=](id<1> i) { data[i] += 3; });
```

in_order queue property

- Use **in_order** queue property for the queue, this will serialize all the kerenel tasks. Note that execution will not overlap even if the queues have no data dependency.

```
queue q{property::queue::in_order()}; // <--- this will serialize all kernel tasks
    q.parallel_for(range<1>(N), [=](id<1> i) { data[i] += 2; });
    q.parallel_for(range<1>(N), [=](id<1> i) { data[i] += 3; });
```

depends_on

- Use **h.depends_on(e)** method in command group to specify events that must complete before a task may begin.

```
auto e = q.submit([&](handler &h) {  // <--- e is event for kernel task
      h.parallel_for(range<1>(N), [=](id<1> i) { data[i] += 2; });
    });

    q.submit([&](handler &h) {
```

```
    h.depends_on(e);  // <--- waits until event e is complete
    h.parallel_for(range<1>(N), [=](id<1> i) { data[i] += 3; });
  });
```

- You can also use a simplified way of specifying dependencies by passing an extra parameter in parallel_for

```
auto e = q.parallel_for(range<1>(N), [=](id<1> i) { data[i] += 2; });
    q.parallel_for(range<1>(N), e, [=](id<1> i) { data[i] += 3; });
                                ^
```

Code Example: USM and Data dependency 1

The code in lab2/usm_data.cpp uses USM and has three kernels that are submitted to the device. Each kernel modifies the same data array. There is data dependency between the three queue submissions, so the code needs to be fixed to get desired output of 20.

Exercise #3

1. Inspect the code in lab2/usm_data.cpp file and fix the bug.

There are three solutions: use **in_order** queue property or use **wait()** event or use **depends_on()** method.

HINT:

- Add **wait()** for each queue submit
- Implement **depends_on()** method in second and third kernel task
- Use **in_order** queue property instead of regular queue: queue q{property::queue::in_order()};

2. Compile and run this code
   icpx -fsycl usm_data.cpp -o usm_data
   ./usm_data

Code Example: USM and Data dependency 2

The code in lab2/usm_data2.cpp uses USM and has three kernels that are submitted to device. The first two kernels modify two different memory objects and the third one has a dependency on the first two. There is data dependency between the three queue submissions, so the code needs to be fixed to get the desired output of 25.

Exercise #4

1. Inspect the code in lab2/usm_data2.cpp file and implement the solution.

- Implementing **depends_on()** method gets the best performance
- Using **in_order** queue property or **wait()** will get results but not the most efficient

HINT:

```
auto e1 = ...
auto e2 = ...

q.parallel_for(range<1>(N), {e1, e2}, [=](id<1> i) {
```

2. Compile and run this code
   icpx -fsycl usm_data2.cpp -o usm_data2
   ./usm_data2

Lab Exercise: Unified Shared Memory

Complete the coding exercise using Unified Shared Memory concepts.

Exercise #5

1. Complete the code in lab2/usm_lab.cpp file by writing the missing code (look for comments)

- The code has two arrays data1 and data2 initialized on host
- Create USM device allocation for data1 and data2 and copy data to device
- Create two kernel tasks, one to update data1 with sqrt of values and another to update data2 with sqrt of values
- Create a third kernel task to add data2 into data1
- Copy data1 back to host and verify results

2. Compile and run this code

   icpx -fsycl usm_lab.cpp -o usm_lab

   ./usm_lab