Graphical user interface, text, application, email

Description automatically generatedA picture containing text, clock

Description automatically generatedUnified Shared Memory (USM) = 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**.



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.

**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**.

**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 simultaneousl**y.

Programmers **may explicitly wait on event objects** or use the depends\_on method inside a command group to s**pecify 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*

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 **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; });

**Use in\_order queue property**

Use **in\_order** queue *property for the queue*, this will **serialize all the kernel 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;});

**Use depends\_on queue property**

Use h.depends\_on(e) method in the 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;});

**^**