SYCL Academy

USING USM

LEARNING OBJECTIVES

- Learn how to allocate memory using USM
- Learn how to copy data to and from USM allocated memory
- Learn how to access data from USM allocated memory in a kernel function
- Learn how to free USM memory allocations

FOCUS ON EXPLICIT USM

- Remember that there are different variants of USM; explicit, restricted, concurrent and system.
- Remember also that there are different ways USM memory can be allocated; host, device and shared.
- We're going to focus explicit USM and device allocations - this is the minimum required variant.

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.

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.

PUTTING IT ALL TOGETHER

```
int square_number(int x){

  auto myQueue = queue{};

  myQueue.submit([&](handler &cgh){
    cgh.single_task<square_number>([=](){
      /* square some number */
    });
  }).wait();

  return x;
}
```

We start with a basic SYCL application which invokes a kernel function with `single_task`.

PUTTING IT ALL TOGETHER

```
int square_number(int x){

  auto myQueue = queue{usm_selector{}};

  myQueue.submit([&](handler &cgh){
    cgh.single_task<square_number>([=](){
      /* square some number */
    });
  }).wait();

  return x;
}
```

We initialize the `queue` with the `usm_selector` we wrote in the last exercise, which will choose a device which supports USM device allocations.

PUTTING IT ALL TOGETHER

```
int square_number(int x){

  auto myQueue = queue{usm_selector{}};

  auto devicePtr = malloc_device<int>(1, myQueue);

  myQueue.submit([&](handler &cgh){
    cgh.single_task<square_number>([=](){
      /* square some number */
    });
  }).wait();

  return x;
}
```

We allocate USM device memory by calling `malloc_device`. Here we use the template variant and specify type `int`.

PUTTING IT ALL TOGETHER

```
int square_number(int x){

  auto myQueue = queue{usm_selector{}};

  auto devicePtr = malloc_device<int>(1, myQueue);

  myQueue.memcpy(devicePtr, &x, sizeof(int)).wait();

  myQueue.submit([&](handler &cgh){
    cgh.single_task<square>([=](){
      /* square some number */
    });
  }).wait();

  return x;
}
```

We copy the value of `x` in the host application to the USM device memory by calling `memcpy` on `myQueue`. We immediately call `wait` on the returned `event` to synchronize with the completion of the copy operation.

PUTTING IT ALL TOGETHER

```
int square_number(int x){

  auto myQueue = queue{usm_selector{}};

  auto devicePtr = malloc_device<int>(1, myQueue);

  myQueue.memcpy(devicePtr, &x, sizeof(int)).wait();

  myQueue.submit([&](handler &cgh){
    cgh.single_task<square>([=](){
      *devicePtr = (*devicePtr) * (*devicePtr);
    });
  }).wait();

  return x;
}
```

We then pass the `devicePtr` directly to the kernel function and access it can then be deferenced and the data written to.

PUTTING IT ALL TOGETHER

```
int square_number(int x){

  auto myQueue = queue{usm_selector{}};

  auto devicePtr = malloc_device<int>(1, myQueue);

  myQueue.memcpy(devicePtr, &x, sizeof(int)).wait();

  myQueue.submit([&](handler &cgh){
    cgh.single_task<square>([=](){
      *devicePtr = (*devicePtr) * (*devicePtr);
    });
  }).wait();

  myQueue.memcpy(&x, devicePtr, sizeof(int)).wait();

  return x;
}
```

We then copy the result from USM device memory back to the variable x in the host application by calling `memcpy` on `myQueue`.

PUTTING IT ALL TOGETHER

```
int square_number(int x){

  auto myQueue = queue{usm_selector{}};

  auto devicePtr = malloc_device<int>(1, myQueue);

  myQueue.memcpy(devicePtr, &x, sizeof(int)).wait();

  myQueue.submit([&](handler &cgh){
    cgh.single_task<square>([=](){
      *devicePtr = (*devicePtr) * (*devicePtr);
    });
  }).wait();

  myQueue.memcpy(&x, devicePtr, sizeof(int)).wait();

  free(devicePtr, myQueue);

  return x;
}
```

Finally, we free the USM device memory that we allocated.

QUEUE SHORTCUTS

```
template <typename KernelName, typename KernelType>
event queue::single_task(const KernelType &KernelFunc);

template <typename KernelName, typename KernelType, int Dims>
event queue::parallel_for(range GlobalRange, const KernelType &KernelFunc);
```

- The `queue` provides shortcut member functions which allow you to invoke a `single_task` or a `parallel_for` without defining a command group.
- These can only be used when using the USM data management model.

WITH THE QUEUE SHORTCUT

```
int square_number(int x){

  auto myQueue = queue{usm_selector{}};

  auto devicePtr = malloc_device<int>(1, myQueue);

  myQueue.memcpy(devicePtr, &x, sizeof(int)).wait();

  myQueue.single_task<square>([=](){
    *devicePtr = (*devicePtr) * (*devicePtr);
  }).wait();

  myQueue.memcpy(&x, devicePtr, sizeof(int)).wait();

  free(devicePtr, myQueue);

  return x;
}
```

If we use the queue shortcut here it reduces the complexity of the code.

QUESTIONS

EXERCISE

Code_Exercises/Exercise_8_USM_Vector_Add/source

Implement the vector add from lesson 6 using the USM data management model.