ENQUEUING A KERNEL

LEARNING OBJECTIVES

- Learn about queues and how to submit work to them.
- Learn how to allocate, transfer and free memory using USM.
- Learn how to define kernel functions.
- Learn about the rules and restrictions on kernel functions.
- Learn how to print from kernels to the console.

THE QUEUE

- In SYCL all work is submitted via commands to a queue.
- The queue has an associated device that any commands enqueued to it will target.
- There are several different ways to construct a queue.
- The most straight forward is to default construct one.
- This will have the SYCL runtime choose a device for you.

THE QUEUE

- The SYCL specification says that work submitted to a given queue, can be executed in any given order.
- It is necessary to define a given task's dependencies, or call `wait()` on `sycl::events` returned from enqueued tasks.

CONSTRUCTING QUEUES

```
int main() {

  // Constructs a queue with a default device.
  // Device can be specified at runtime using
  // SYCL_DEVICE_FILTER
  auto q1 = sycl::queue{};

  // Constructs a queue with a gpu, if one is available.
  // Throws a runtime error if no gpus available
  auto q2 = sycl::queue{gpu_selector{}};

  std::cout << "q1 using device "
    << q1.get_device().get_info<sycl::info::device::name>()
    << '\n';

  std::cout << "q2 using device "
    << q2.get_device().get_info<sycl::info::device::name>()
    << '\n';

}
```

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.
- For now we are going to focus on the USM model.

USM TYPES

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

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.

ENQUEUEING A KERNEL

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

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

- Kernels take the form of function objects or Lambdas.
- The queue provides member functions which allow you to invoke a `single_task` or a `parallel_for`.
- These can only be used when using the USM data management model.

PUTTING IT ALL TOGETHER - SHARED USM

We start with a basic SYCL application which used shared USM and invokes a kernel function with `single_task`.

```
T square_number(T x){

  auto q = sycl::queue{};

  T * sharedPtr = malloc_shared<T>(1, q);
  sharedPtr[0] = x;

  q.single_task([=](){
      (*sharedPtr) *= (*sharedPtr);
    });
  }).wait();

  return sharedPtr[0];
}
```

PUTTING IT ALL TOGETHER - MALLOC DEVICE

```
int square_number(T x){

  auto q = sycl::queue{};

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

  q.memcpy(devicePtr, &x, sizeof(T));

  q.single_task([=](){
      (*devicePtr) *= (*DevicePtr);
  });

  q.memcpy(&x, devicePtr, sizeof(T));

  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 - MANAGING DEPENDENCIES

```
int square_number(T x){

  auto q = sycl::queue{};

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

  q.memcpy(devicePtr, &x, sizeof(T)).wait();

  q.single_task([=](){
      (*devicePtr) *= (*DevicePtr);
    });
  }).wait();

  q.memcpy(&x, devicePtr, sizeof(T)).wait();

  return x;
}
```

- Since operations like `memcpy`, `single_task`, and `parallel_for` execute asynchronously, it is important to ensure that tasks have completed before their results are needed.
- Using `wait()` will only allow for sequential execution of tasks in a queue. DAG only one task wide.

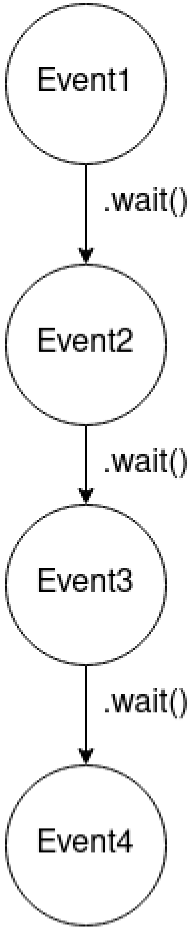
PUTTING IT ALL TOGETHER - MANAGING DEPENDENCIES

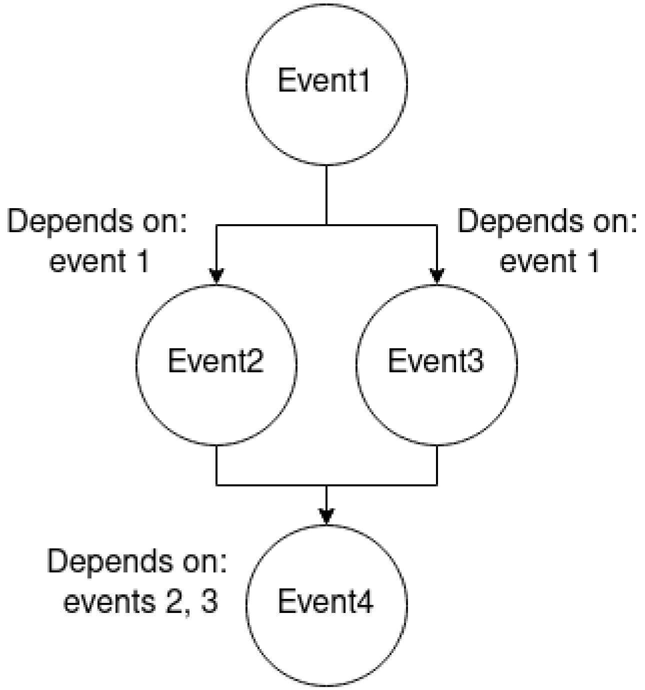
```
int square_number(T x){

  auto q = sycl::queue{};

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

  auto e1 = q.memcpy(devicePtr, &x, sizeof(T));

  auto e2 = q.single_task({e1}, [=](){
      (*devicePtr) *= (*DevicePtr);
    });

  auto e3 = q.memcpy(&x, devicePtr, sizeof(T), {e2});

  e3..wait();

  return x;
}
```

- You can also define an operation's dependent events, which allows construction of any execution DAG. This allows for concurrent execution of tasks.

CONTROL FLOW

- Here we can see the control flow difference between synchronizing using `wait()` and using explicit dependencies.





SYCL KERNEL FUNCTION RULES

- Must be defined using a C++ lambda or function object, they cannot be a function pointer or std::function.
- Must always capture or store members by-value.
- SYCL kernel functions declared with a lambda must be named using a forward declarable C++ type, declared in global scope.
- SYCL kernel function names follow C++ ODR rules, which means you cannot have two kernels with the same name.

SYCL KERNEL FUNCTION RESTRICTIONS

- No dynamic allocation
- No dynamic polymorphism
- No function pointers
- No recursion

KERNELS AS FUNCTION OBJECTS

```
sycl::queue gpuQueue;
gpuQueue.single_task([=]() {
    /* kernel code */
}).wait();
```

- All the examples of SYCL kernel functions up until now have been defined using lambda expressions.

KERNELS AS FUNCTION OBJECTS

```
struct my_kernel {
  void operator()(){
    /* kernel function */
  }
};
```

- As well as defining SYCL kernels using lambda expressions, You can also define a SYCL kernel using a regular C++ function object.

KERNELS AS FUNCTION OBJECTS

```
struct my_kernel {
  void operator()(){
    /* kernel function */
  }
};
```

```
sycl::queue gpuQueue;
gpuQueue.single_task(my_kernel{}).wait();
```

- To use a C++ function object you simply construct an instance of the type and pass it to `single_task`.

KERNEL PRINTF

- DPC++ provides an extension to allow in-kernel `printfs`. This is useful for debugging.

KERNEL PRINTF

```
sycl::queue gpuQueue;
gpuQueue.single_task([=]() {
        printf("Hello, World!\n");
}).wait();
```

EXERCISE

Code_Exercises/Exercise_2_Hello_World/source.cpp

Implement a SYCL application which enqueues a kernel function to a device and streams "Hello world!" to the console.