UOW AUSTRALIA

SCIT

School of Computing & Information Technology

CSCI376 – Multicore and GPU Programming

Data Management

The aim of this tutorial is to introduce you to the OpenCL runtime memory objects and to data management between host memory and memory objects. By now, you should already be familiar with the OpenCL Platform Layer.

To start off, have a look at the code in common.h and common.cpp files. These contain functions that were described in tutorials 1 & 2. They have been placed in these two files for convenience, and will be used in all other tutorials.

Memory objects

In OpenCL, a memory object is an instantiation of the `cl::Memory` class. There are two types of memory objects:

- `cl::Buffer` – stores one-dimensional data
- `cl::Image` – stores multi-dimensional data

For memory objects that do not involve images, you should use a buffer object. This tutorial examines buffer objects. Image objects will be in a different tutorial. To create a buffer object, you can use the following constructor:

```
cl::Buffer(const Context& context, cl_mem_flags flags, ::size_t size,
      void* host_ptr=NULL, cl_int* err=NULL)
```

Memory objects are associated with a context, which is passed as the first parameter. The second parameter identifies the read/write capability of the buffer relative to the device and the nature of the buffer memory's allocation on the host. The third parameter is the size in byte. The fourth is a pointer to where the data is stored in host memory. If the buffer object is to be write-only, e.g., intended to hold output exclusively, the argument can be left as `NULL`. Finally, the fifth parameter (optional) is to store a returned error code.

Have a look at the code in Tutorial3a.cpp. Several vectors are declared in the `main()` function:

```
std::vector<cl_float> floatDataA(LENGTH);
std::vector<cl_float> floatDataB(LENGTH);
std::vector<cl_float> floatOutput(LENGTH);
std::vector<cl_int> intData(LENGTH);
std::vector<cl_int> intOutput(LENGTH);
```

These will be used to store data use in the program. Note that they have all been initialised to size LENGTH, which was defined as:

```
#define LENGTH 40
```

Next, three buffer object variables were declared:

```
cl::Buffer floatBufferA;
cl::Buffer floatBufferB;
cl::Buffer intBuffer;
```

These will be created later in the code. The floatDataA, floatDataB and intData vectors are then initialised with a range of values:

```
for (int i = 0; i < LENGTH; i++) {
	floatDataA[i] = i + 1.0f;
	floatDataB[i] = -1.0f * (i + 1);
	intData[i] = i;
}
```

Have a look further down in the code, at the section where the buffers are created. The following shows an example of a buffer object where the data is intended to be read-only (`CL_MEM_READ_ONLY`) on the device and `CL_MEM_USE_HOST_PTR` indicates that the application wants the OpenCL implementation to use memory referenced by `floatDataA` for the memory object. The size of the data is all the data contained in `floatDataA`. It can be set using `sizeof(cl_float) * floatDataA.size()`, or `sizeof(cl_float) * LENGTH`, or even explicitly using `4 x 40`. The next argument provides a reference to where the data is stored:

```
floatBufferA = cl::Buffer(context, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
	sizeof(cl_float) * floatDataA.size(), &floatDataA[0]);
```

The buffer object is created differently. This memory object is intended to be for read and write access on the device. Since no host pointer was passed, the following only reserves memory:

```
floatBufferB = cl::Buffer(context, CL_MEM_READ_WRITE, sizeof(cl_float) * LENGTH);
```

Next, another buffer is created using a

```
intBuffer = cl::Buffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
	sizeof(cl_int) * intData.size(), &intData[0]);
```

`CL_MEM_COPY_HOST_PTR` indicates that the application wants the OpenCL implementation to allocate memory for the memory object and copy the data from memory referenced by `intData`. The `CL_MEM_COPY_HOST_PTR` flag can be used with `CL_MEM_ALLOC_HOST_PTR`, which specifies that the memory should be allocated from host accessible memory. Note that `CL_MEM_USE_HOST_PTR` is mutually exclusive with `CL_MEM_COPY_HOST_PTR` and `CL_MEM_ALLOC_HOST_PTR`.

Similar to other OpenCL data structures, we can also query information from memory objects. The following code obtains and displays the memory size and host memory location of the memory objects. In addition, it also displays the memory location of the vectors on the host:

```
std::cout << "floatBufferA size: " << floatBufferA.getInfo<CL_MEM_SIZE>() << std::endl;
std::cout << "floatBufferB size: " << floatBufferB.getInfo<CL_MEM_SIZE>() << std::endl;
```

```
std::cout << "intBuffer size: " << intBuffer.getInfo<CL_MEM_SIZE>() << std::endl;
std::cout << "floatDataA memory location: " << &floatDataA[0] << std::endl;
std::cout << "floatBufferA memory location: " << floatBufferA.getInfo<CL_MEM_HOST_PTR>() <<
	std::endl;
std::cout << "floatDataB memory location: " << &floatDataB[0] << std::endl;
std::cout << "floatBufferB memory location: " << floatBufferB.getInfo<CL_MEM_HOST_PTR>() <<
	std::endl;
std::cout << "intData memory location: " << &intData[0] << std::endl;
std::cout << "intBuffer memory location: " << intBuffer.getInfo<CL_MEM_HOST_PTR>() <<
	std::endl;
```

Data transfer

Remember that for `floatBufferB`, memory was reserved but the data was not initialised:

```
floatBufferB = cl::Buffer(context, CL_MEM_READ_WRITE, sizeof(cl_float) * LENGTH);
```

We can transfer data from host memory to device memory and vice versa. From host to device is known as writing the data, whereas, from device to host is known as reading the data. As with all device commands we must enqueue the command using the command queue:

```
queue.enqueueWriteBuffer(floatBufferB, CL_TRUE, 0, sizeof(cl_float) *
	floatDataB.size(), &floatDataB[0]);
```

What this does is enqueues a write buffer command to transfer data referenced by `floatDataB` of size `sizeof(cl_float) * floatDataB.size()` into `floatBufferB`. The second argument specifies whether the function is a blocking or non-blocking function, and the third argument specifies an offset location where the data should be stored.

Note that instead of using the `enqueueWriteBuffer()` function, we could have initialised the data the same way as for `floatBufferA`, e.g.:

```
floatBufferB = cl::Buffer(context, CL_MEM_READ_WRITE | CL_MEM_USE_HOST_PTR,
	sizeof(cl_float) * floatDataB.size(), &floatDataB[0]);
```

Further down in the code, you will see an example of transferring data from the device to the host (i.e. reading the data). The following reads the data `floatBufferA` of size `sizeof(cl_float) * LENGTH` and transfer it into the host memory location reference by `&floatOutput[0]`:

```
queue.enqueueReadBuffer(floatBufferA, CL_TRUE, 0, sizeof(cl_float) * LENGTH,
	&floatOutput[0]);
```

Kernel arguments

Kernels, i.e. functions that run on devices, need to process data. Therefore, kernel functions must be able to accept data. Have a look at the code in the blank_kernel.cl file:

```
__kernel void blank( int a,              // index 0
                     __global float *b,  // index 1
                     __global float *c,  // index 2
                     __global int *d)    // index 3
```

Notice that it accepts an `int`, two `float` arrays and an `int` array. The indices for the parameters are in the order that they are defined, starting from index 0.

Go back to the code in Tutorial3a.cpp. We have to set the kernel arguments in the host code using the `cl::Kernel` class's `setArg()` member function. The first argument is the index in the kernel function, and the second argument identifies the data to be sent to the kernel. The following shows examples of how the kernel arguments can be set:

```
int a = 0;
kernel.setArg(0, a);
kernel.setArg(1, floatBufferA);
kernel.setArg(2, floatBufferB);
kernel.setArg(3, intBuffer);
```

Next, the kernel is enqueued for execution:

```
queue.enqueueTask(kernel);
```

The code after that reads the data from device to host, once in host memory the contents are displayed, e.g.:

```
queue.enqueueReadBuffer(floatBufferA, CL_TRUE, 0, sizeof(cl_float) * LENGTH,
	&floatOutput[0]);

std::cout << "\nContents of floatBufferA: " << std::endl;

for (int i = 0; i < LENGTH / 10; i++)
{
	for (int j = 0; j < 10; j++)
	{
		std::cout << floatOutput[j + i * 10] << " ";
	}
	std::cout << std::endl;
}
```

Compile and run the program to see what it does.

Some other data transfer operations

For this section have a look at the code in Tutorial3b.cpp.

Copying

You have previously seen examples of reading and writing to memory objects. In this example, we will examine some other data transfer operations. First, we can transfer contents between memory objects on the device:

```
queue.enqueueCopyBuffer(floatBufferA, floatBufferB, 0, 0, sizeof(cl_float) * LENGTH);
```

The above code transfers contents of size `sizeof(cl_float) * LENGTH` from `floatBufferA` starting at offset 0, into `floatBufferB` starting at offset 0.

Memory mapping

We can also map a memory object on the device to host memory. Then transfer the data using mapped memory:

```
void* mappedMemory = queue.enqueueMapBuffer(floatBufferB, CL_TRUE, CL_MAP_READ, 0,
       sizeof(cl_float) * LENGTH);

memcpy(&floatOutput[0], mappedMemory, sizeof(cl_float) * LENGTH);

queue.enqueueUnmapMemObject(floatBufferB, mappedMemory);
```

The code above maps the data in `floatBufferB` memory object, starting at offset 0 of size `sizeof(cl_float) * LENGTH`. It then copies the contents from the memory object into `floatOutput` using the `memcpy()` function. Finally, it unmaps the memory object. The contents in `floatOutput` is then displayed.

Transferring a rectangular section of data

The next section of code shows an example of transferring a rectangular section of data. Note that the data itself is a contiguous sequence in memory. The dimensionality is specified through the function's arguments.

First, the output data was set to all 0s:

```
memset(&intOutput[0], 0, sizeof(cl_int)*LENGTH);
```

Then, the buffer origin, host origin and rectangular region of data that will be transferred were specified. These will be used as arguments in the rectangular data transfer function:

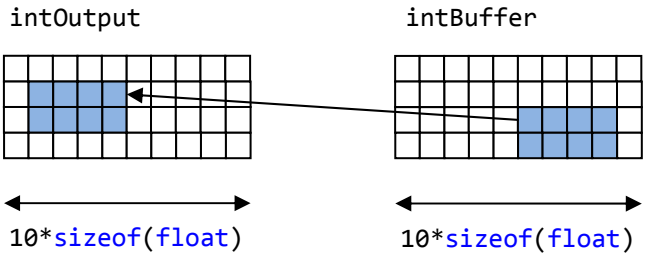
```
cl::size_t<3> bufferOrigin, hostOrigin, region;

bufferOrigin[0] = 5 * sizeof(cl_int);
bufferOrigin[1] = 2;
bufferOrigin[2] = 0;
hostOrigin[0] = 1 * sizeof(cl_int);
hostOrigin[1] = 1;
hostOrigin[2] = 0;
region[0] = 4 * sizeof(cl_int);
region[1] = 2;
region[2] = 1;
```

The rectangular read buffer function is enqueuer to transfer the rectangular section of data from the device to the host:

```
queue.enqueueReadBufferRect(intBuffer, CL_TRUE, bufferOrigin, hostOrigin, region,
       10 * sizeof(cl_int), 0, 10 * sizeof(cl_int), 0, &intOutput[0]);
```

The following diagram is a conceptual depiction of the data that is transferred:



After the transfer, the contents are displayed.

Compile and run the program, observe the data that the program displays.

References

Among others, much of the material in this tutorial was sourced from:

- M. Scarpino, “OpenCL in Action: How to Accelerate Graphics and Computation,” Manning Publications
- Khronos OpenCL Working Group, “The OpenCL Specifications”, version 1.2
- Khronos OpenCL Working Group, “OpenCL C++ Wrapper 1.2 Reference Card”