UOW AUSTRALIA

# SCIT

School of Computing & Information Technology

## CSCI376 – Multicore and GPU Programming

# Programs, Kernels and Command Queues

The aim of this tutorial is to introduce you to some of the OpenCL runtime objects. In particular, it will cover the following OpenCL objects: program objects, kernel objects and command queues. Before you start this tutorial, you need to be familiar with the OpenCL Platform layer covered in tutorial 1.

### Program objects

Have a look at the files in Tutorial2a.

Note: if you try to run the program, will produce a runtime error (or may not even run on Xcode). This is deliberate.

An OpenCL program consists of a set of kernels that are identified as functions declared with the `__kernel` qualifier in a program source. Note that programs and kernels are different, even though they both store executable code. A kernel represents a single function to be executed on a device, whereas a program is a container of one or more kernels. OpenCL programs can also contain auxiliary functions and constant data that can be used by kernel functions.

A program object is represented by a `cl::Program` class and encapsulates the following information:

- An associated context.
- A program source or binary.
- The latest successfully built program executable, the list of devices for which the program executable is built, the build options used and a build log.
- The number of kernel objects currently attached.

### Creating a program object

Examine the code in Tutorial2a.cpp. The line:

```
std::ifstream programFile("blank_bad.cl");
```

opens a program source file as an input stream, and checks whether it opened successfully. If not, it quits the program:

```
if (!programFile.is_open())
{
```

```
        quit_program("File not found.");
}
```

If the file was opened, it loads the contents from the file into the string `programString` and displays its contents:

```
std::string programString(std::istreambuf_iterator<char>(programFile),
        (std::istreambuf_iterator<char>()));


std::cout << programString << std::endl;
```

The reason for this is because to build the program, you before you can build the program, the source code must be stored in a buffer. While it is possible to define the source code as a collection of strings in the host application code itself, most proper programs will read in the source code from a separate file. This keeps the host and device code separate. Thus, it is a good idea to have a function to read the contents of a file as shown above.

Then the code below creates a program `cl::Program::Sources` object from 1 string. Note that the syntax requires the creation of a `pair` containing the string containing the program source and the size of the string:

```
cl::Program::Sources source(1, std::make_pair(programString.c_str(),
        programString.length() + 1));
```

Once created, a Program object can be created from a context and a source object:

```
cl::Program program(context, source);
```

**Building a program**

Once a program object is created the program can be built using the `cl::Program` class's `build()` member function. An OpenCL framework must provide a runtime compiler since it builds device specific executables. As such, the build function accepts a vector containing the device objects to be targeted by the compiler:

```
program.build(contextDevices);
```

If there were build errors, an exception will be thrown. We can check the build log to see what caused the error. The following code checks whether the exception was caused by a failure to build the program, the displays the device name and build log, using the `getBuildInfo()` function, for all devices in the context:

```
if (e.err() == CL_BUILD_PROGRAM_FAILURE)
{
        std::cout << e.what() << ": Failed to build program." << std::endl;

        for (unsigned int i = 0; i < contextDevices.size(); i++)
        {
                outputString = contextDevices[i].getInfo<CL_DEVICE_NAME>();
                std::string build_log =
```

```
                program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(contextDevices[i]);

            std::cout << "Device - " << outputString << ", build log:"<< std::endl;
            std::cout << build_log << "--------------------" << std::endl;
        }
    }
```

Note that since each OpenCL framework has its own compiler, the contents of the build log will be different for different compilers.

Run the Tutorial2a program and observe the output. It will produce an error and display the program log. (Note: that on Mac OS, Xcode does a pre-check of the source code, so it may already show the error and will not run). The program log can be used to debug kernel source code.

Open the blank_bad.cl file. It contains a black kernel function. The problem with the source code is that it contains a syntax error: kernels are defined with a double underscore, i.e. `__kernel`, the kernel in the source code only has one underscore. Add another underscore and run the program again. This time it should run.

**Kernel objects**

Tutorial2b shows how to create kernel objects.

After you have compiled and linked a program, you can package its functions into data structures called kernels. Kernels can be dispatched to a command queue and sent to a device. A kernel object is represented by a `cl::Kernel` class.

There are two ways of creating kernel objects. The first method is to create individual kernel objects by using the `cl::Kernel` class's constructor. Note that the program object and kernel name is passed to the constructor:

```
cl::Kernel addKernel(program, "add");
cl::Kernel subKernel(program, "subtract");
cl::Kernel multKernel(program, "divide");
```

Have a look at the code in the kernels.cl file. Notice that the file contains four kernels; namely, add, subtract, multiply and divide. The code above creates three kernel objects using the kernel names.

The second method creates kernel objects from all the kernels in a given program using the `cl::Program` class's `createKernels()` member function:

```
std::vector<cl::Kernel> allKernels;

program.createKernels(&allKernels);
```

The following code goes through all the kernels in the vector, queries and displays their respective kernel names:

```
for (i = 0; i < allKernels.size(); i++) {
    outputString = allKernels[i].getInfo<CL_KERNEL_FUNCTION_NAME>();
```

```
        std::cout << "Kernel " << i << ": " << outputString << std::endl;
    }
```

### Command queues

This section is based on the code in Tutorial2c.

Examine the code in Tutorial2c.cpp. For starters, every OpenCL program will require the program source code to be built. Therefore, to make things easier, the boilerplate code for doing this has been placed in the following function:

```
bool build_program(cl::Program* prog, const cl::Context* ctx, const std::string filename);
```

Note that this function will be used in all the other tutorials. It accepts a context object and the name of the file containing the source code. It will open the file and read its contents, build the program, check for built errors and return the built program.

### Creating command queues

To execute a kernel, it must be deployed to a device through a command queue. Each device has its own command queue. Kernel execution is only one type of command that can be dispatched to a command queue. A command is a message sent from the host that tells a device to perform an operation. Besides kernel execution, many OpenCL command operations involve transferring data: e.g., reading data from the device to the host, writing data from the host to the device, and copying data between devices. Command queues in OpenCL are represented by the `cl::CommandQueue` class.

Notice that at the start of the `main()` function, the six fundamental OpenCL data structures are declared:

```
cl::Platform platform;
cl::Device device;
cl::Context context;
cl::Program program;
cl::Kernel kernel;
cl::CommandQueue queue;
```

We have seen how to create and use each of them. Further down the code, you will see how to create a command queue from a context and device:

```
queue = cl::CommandQueue(context, device);
```

### Enqueuing a kernel execution command

As previously mentioned, all commands are enqueued to a device via a command queue. An example of this can be observed in the following lines of code. The following creates a kernel from a kernel function named “blank”:

```
        kernel = cl::Kernel(program, "blank");
```

Next a command queue is created:

```
        queue = cl::CommandQueue(context, device);
```

Like normal functions, kernel functions can accept arguments. To do this, we need to set the arguments in the host application code. We will look at data management in the next tutorial. However, because some compilers will not allow you to enqueue a kernel that does not have any arguments, we will set one here:

```
int a = 0;
kernel.setArg(0, a);
```

To dispatch the kernel to the command queue, we can use the `enqueueTask()` member function:

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

With this, the kernel will be executed on the device when ready.

## Things to try

Try writing a program to do the following:

- Read a program source from a file and build the program
- Display the program build log
- Create kernels from a program and display the kernel function names

### 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”