UOW AUSTRALIA

SCIT

School of Computing & Information Technology

CSCI376 – Multicore and GPU Programming

Platforms, Devices and Contexts

The aim of this tutorial is to introduce you to the OpenCL Platform layer, which allows a host program to discover OpenCL platforms, devices and their capabilities as well as to create contexts. This covers the following fundamental OpenCL data structures: platforms, devices and contexts.

Before you start this tutorial, you should go through the 'Setting up the Environment' document which shows you how to set up the Visual Studio/Xcode project environment. While the projects for the tutorials have already been set up for you, it is a good idea to know how to set it up yourself.

Accessing platforms and devices

This section is based on the contents in Tutorial1a.

Different computers have different hardware. When writing an OpenCL program that is meant to run on different computers, we need to assume that we do not know anything about the underlying hardware before we run the application. OpenCL addresses this by providing the `cl::Platform` and `cl::Device` classes, and allowing us to query platform and device information.

Getting started

Have a look at the code in Tutorial1a.cpp, it starts with:

```
#define CL_USE_DEPRECATED_OPENCL_2_0_APIS
```

The purpose of this line is because we will be using some OpenCL 1.2 functions. Without this, the compiler will use the default OpenCL 2.x, where some OpenCL 1.2 functions are deprecated.

```
#define __CL_ENABLE_EXCEPTIONS
```

By default, OpenCL disables exceptions. This line enables it.

Next the headers from the C++ standard library and STL that are used in the code are included:

```
#include <iostream>
#include <vector>
```

This is followed by including the OpenCL header. Depending on which Operating System (OS) we are using, this checks whether we are running the program on Mac OS or other OS:

```
#ifdef __APPLE__
#include <OpenCL/cl.hpp>
#else
#include <CL/cl.hpp>
```

```
#endif
```

After this we include a user defined header file:

```
#include "error.h"
```

Have a look at the contents in the error.h file. It contains three simple functions:

```
void handle_error(cl::Error e);
void quit_program(const std::string str);
const std::string lookup_error_code(cl_int error_code);
```

The first function is what we will use to catch exceptions (you will see this later in the `main()` function).

```
void handle_error(cl::Error e)
{
	std::cout << "Error in: " << e.what() << std::endl;
	std::cout << "Error code: " << e.err() << " (" <<
		lookup_error_code(e.err()) << ")" << std::endl;
}
```

It accepts a `cl::Error` object, which allows us to obtain error information using two of its class functions:

- `what()`– identifies what cause the error
- `err()` – returns the error code

Notice that the function displays what caused the error and displays the error code. In addition, it calls the `lookup_error_code()` function to convert the error code into a string. This will make it easier to determine what caused the error. Have a look at that function:

```
const std::string lookup_error_code(cl_int error_code)
{
	switch (error_code) {
	case CL_SUCCESS:
		return "CL_SUCCESS";
	case CL_DEVICE_NOT_FOUND:
		return "CL_DEVICE_NOT_FOUND";
…

…

	default:
		return "Unknown error code";
	}
}
```

The `quit_program()` function simply displays an error message and quits the program.

Now return to the Tutorial1a.cpp file. The following lines have been deliberately commented out:

```
//using namespace std;
//using namespace cl;
```

You can use these if you want to avoid having to type `std::` and `cl::` prefixes in front of the C++ standard library or OpenCL functions. However, the prefixes are used in the tutorial code to clearly show which library the functions/data structures belong to.

Platforms

Now examine at the `main()` function. It starts by declaring a number of data structures and variables, the most important are:

```
std::vector<cl::Platform> platforms;
std::vector<cl::Device> devices;
```

These use the `std::vector` class, and will be used to store the platforms and devices available to the system.

Next, the code uses a `try` and `catch` block to catch any exceptions that may occur. The catch block simply catches any OpenCL exceptions and calls the function to display the error information.

```
catch (cl::Error e) {
	handle_error(e);
}
```

In the try block, the first line obtains the OpenCL platforms that are available on the system, and the second line simply displays the number of available platforms.

```
cl::Platform::get(&platforms);
std::cout << "Number of OpenCL platforms: " << platforms.size() << std::endl;
```

A `cl::Platform` object represents an installed platform. The static function `cl::Platform::get()` function will return a vector containing a `cl::Platform` object for each platform that is installed on the system. Its signature is as follows:

```
static cl_int cl::Platform::get(VECTOR_CLASS<cl::Platform>* platforms);
```

Note that it is useful to refer to the OpenCL C++ Wrapper 1.2 Reference Card for information on what's available. You can find information on the Platform class in section 2.1.

Once we obtain the platform objects, we can query the platform information for each platform. The next section contains a loop which goes through the platforms and displays information about the platform:

```
for (i = 0; i < platforms.size(); i++)
{
```

The `getInfo()` member function allows us to obtain platform information. Note that all the key OpenCL classes have a `getInfo()` member function, you will see this when other OpenCL classes are introduced.

There are two ways this can be used to obtain information. For example:

```
platforms[i].getInfo(CL_PLATFORM_NAME, &outputString);
```

This gets the name of the platform (using: `CL_PLATFORM_NAME`) and stores it in the output string.

The other way is to use the template parameter:

```
outputString = platforms[i].getInfo<CL_PLATFORM_VENDOR>();
```

This gets the platform vendor's name (using: `CL_PLATFORM_VENDOR`) and stores it in the output string. There is a number of other platform information that you can get. Please refer to section 2.1 of the OpenCL C++ Wrapper 1.2 Reference Card:

cl_int **getInfo**(cl_platform_info *name*,
STRING_CLASS **param*) const;
name:

CL_PLATFORM_EXTENSIONS	*STRING_CLASS*
CL_PLATFORM_NAME	*STRING_CLASS*
CL_PLATFORM_PROFILE	*STRING_CLASS*
CL_PLATFORM_VENDOR	*STRING_CLASS*
CL_PLATFORM_VERSION	*STRING_CLASS*

Devices

An important capability of a Platform object is that it allows us to access Device objects associated with that platform, using `getDevices()`. In the code, we get the devices for each platform. The code gets all devices (i.e. `CL_DEVICE_TYPE_ALL`), regardless of device type.

```
platforms[i].getDevices(CL_DEVICE_TYPE_ALL, &devices);
```

If you only want specific device types, you can do so using for example: `CL_DEVICE_TYPE_CPU` or `CL_DEVICE_TYPE_GPU`

cl_int **getDevices**(cl_device_type *type*,
VECTOR_CLASS<Device> **devices*) const;
type: CL_DEVICE_TYPE_{ACCELERATOR, ALL, CPU},
CL_DEVICE_TYPE_{CUSTOM, DEFAULT, GPU}

After displaying the number of devices on each platform:

```
std::cout << "\nNumber of devices available to platform " << i << ": " <<
        devices.size() << std::endl;
```

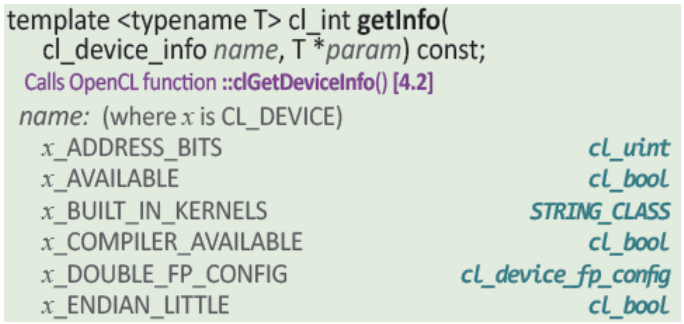
The code goes through each device:

```
for (j = 0; j < devices.size(); j++)
{
```

and obtains information about each device using the `cl::Device::getInfo()` member function, e.g.:

```
outputString = devices[j].getInfo<CL_DEVICE_NAME>();
```

Note that there is a lot of information that you can obtain. Please refer to the `cl::Device` class Section 2.2 in the OpenCL C++ Wrapper 1.2 Reference Card. An example of a small snippet:



Note that the data type of each piece of information is different. The example code below shows how to get and identify the device type:

```
cl_device_type type;
devices[j].getInfo(CL_DEVICE_TYPE, &type);
if (type == CL_DEVICE_TYPE_CPU)
	std::cout << "\tType: " << "CPU" << std::endl;
else if (type == CL_DEVICE_TYPE_GPU)
	std::cout << "\tType: " << "GPU" << std::endl;
else
	std::cout << "\tType: " << "Other" << std::endl;
```

Contexts

The next example in Tutorial1b shows an example of creating a context from one or more devices. Look at the code in Tutorial1b.cpp, near the top it has:

```
#define NUMBER_OF_DEVICES 2
```

This defines the minimum number of devices you want to create a context for. Note that while you can create as many contexts as you want, all devices in a context must be from the same platform.

As an overview, the program searches for the first platform that has the requested minimum number of devices, creates a context and displays the names of devices within the context.

Examine the code within the try block of the `main()` function. It gets all available platforms, then for each platform it gets all available devices:

```
cl::Platform::get(&platforms);

for (i = 0; i < platforms.size(); i++)
{
	platforms[i].getDevices(CL_DEVICE_TYPE_ALL, &devices);
```

Next, it checks whether the number of available devices for that platform meets the minimum number that is requested:

```
	if (devices.size() >= NUMBER_OF_DEVICES)
	{
```

If it does, the program creates a context for all the devices in that platform:

```
context = cl::Context(devices);
```

To check the devices within the context, we can query the context for information using the `cl::Context` class's `getInfo()` member function. The following code gets all the devices within the context and for each device, displays the device's name:

```
std::vector<cl::Device> contextDevices =
	context.getInfo<CL_CONTEXT_DEVICES>();

for (j = 0; j < contextDevices.size(); j++)
{
	outputString = contextDevices[j].getInfo<CL_DEVICE_NAME>();
	std::cout << "  Device " << j << ": " <<
		outputString << std::endl;
}
```

Once a platform with the requested number of devices is found, a context created and device names displayed. The program no longer searches through the platforms:

```
break;
```

If the system does not have a platform with the requested number of devices, it will display this before exiting the program:

```
if (i == platforms.size()) {
	std::cout << "Unable to find a platform with " << NUMBER_OF_DEVICES <<
		" or more devices." << std::endl;
}
```

Try changing the number of requested devices (i.e. `NUMBER_OF_DEVICES`) to 1 and 3 and see what the program will output.

Selecting a platform and device

The next example in Tutorial1c is a program that displays all the available platforms and devices on a system. The program then allows the user to select a platform and a device, and creates a context using the selected device.

Most of the code is similar to Tutorial1a and Tutorial1b. So only some of the differences will be highlighted here. You should examine the code yourself to try to understand it.

The code defines a function that displays all available platform and device options, allows the user to select one option, then returns the selected platform and device. Note that the code in later tutorials will use this function.

```
bool select_one_device(cl::Platform* platfm, cl::Device* dev)
```

The devices per platform are stored using:

```
std::vector< std::vector<cl::Device> > platformDevices;
```

The code goes through each platform and fills in the available devices per platform. Then it displays each platform and device option, and stores each option:

```
std::vector< std::pair<int, int> > options;
unsigned int optionCounter = 0;

for (i = 0; i < platforms.size(); i++)
{
	for (j = 0; j < platformDevices[i].size(); j++)
	{
		std::cout << "Option " << optionCounter << ": Platform - ";

		outputString = platforms[i].getInfo<CL_PLATFORM_VENDOR>();
		std::cout << outputString << ", Device - ";

		outputString = platformDevices[i][j].getInfo<CL_DEVICE_NAME>();
		std::cout << outputString << std::endl;

		options.push_back(std::make_pair(i, j));
		optionCounter++;
	}
}
```

The code then allows the user to enter a selection, checks whether the selection is valid. If valid, the code will return the selected platform and device. If not, it will display a message about the invalid option and exit.

Things to try

By now you should have a basic grasp of platforms, devices and contexts, and how to create and obtain information from these objects.

- Try to get other platform and device information.

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”