Lincoln School of Computer Science
University of Lincoln
CMP3110M/CMP9057M Parallel Computing

Introduction to OpenCL

1 Introduction to Workshop Sessions

The aim of these workshops is to introduce you to practical aspects of parallel programming using the OpenCL framework. There are 4 tutorial sessions for this module in total, each designed to take 2 weeks or 4 hours. Each session is meant to serve as a tutorial for a particular topic in parallel programming with explanations and practical tasks which should help you to understand the theory covered during lectures. These practical tasks are indicated by the *Task4U* label and are highlighted in italics. These tasks are not assessed but are designed in a way which should lead you to the solutions required in the practical assessment for this module. To make the most out of these sessions, discuss your answers with your colleagues and/or ask demonstrators for feedback.

2 Tutorial

2.1 What is OpenCL?

OpenCL is an open standard for parallel programming of different devices and hardware platforms (https://www.khronos.org/opencl/). It is a C-like programming language which allows the same programs to be run on different parallel hardware. OpenCL also specifies API which allows for the management and control of parallel devices, and scheduling of the execution of parallel programs. The API is available in different programming languages including C++/Python/Java, etc. We are going to use C++ in our tutorials. The following document summarises all the functionality available through the C++ API v1.2 and should be used as a practical reference during our sessions: https://www.khronos.org/registry/cl/specs/opencl-cplusplus-1.2.pdf.

2.2 Programming Environment

OpenCL is a specification only and each major manufacturer of parallel hardware including Intel, NVidia and AMD has their own implementation of the standard, which runs on their hardware products. To *execute* an OpenCL program, one needs dedicated run-time drivers, which are typically installed with device drivers (e.g. graphics driver for NVidia cards). To *develop* the code, one needs specific libraries which are also supplied by each manufacturer. The computers in our lab have already run-time drivers installed for CPU and GPUs. For development, we are going to use Intel OpenCL SDK which is integrated into Visual Studio programming environment. If you would like to have a similar setup on your PC at home, refer to the instructions provided in Section 4 of this document.

2.3 Basic concepts

Programming in OpenCL consists of writing two different types of code: the device code, in so called “kernels”, which is executed on the device, and the host code which is responsible for managing

devices, dispatching of kernels, data flow between the host and devices, etc. The host is always the main CPU and it can only be a single device. The host code is more or less standard C++ code and in majority consists of the boilerplate code. Whilst not very friendly for beginners, it can be reused with little modification for different applications.

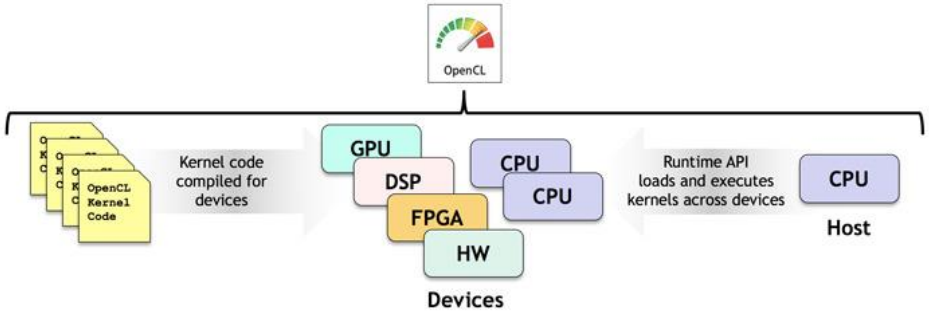

Figure 1 OpenCL hardware architecture: host and devices (after https://www.khronos.org/opencl/).

The kernel code is the actual program executed on parallel devices. The devices range from CPUs to GPUs, FPGAs, DSP and so on (see Fig. 1). By default, OpenCL kernels are loaded and compiled at run-time in a process similar to the execution of Java programs, called "just-in-time" compilation. Thanks to this, the same code can be executed on different platforms without the necessity of rebuilding it for each device. Therefore, the code built with one OpenCL toolkit (e.g. from NVidia) can be run on other devices (AMD, Intel), as long as the run-time drivers for these devices are present.

2.4 Running the code

Let us start with some simple code example written in OpenCL – a vector addition `C = A + B` should serve well our purpose. The tutorial code is hosted on GitHub https://github.com/gcielniak/OpenCL-Tutorials. You can either download and extract the repository from a zip file (the green "Clone or download" button) or clone to a local directory by issuing '`git clone https://github.com/gcielniak/OpenCL-Tutorials.git`' from the command line (search for "cmd" and open the Command Prompt application). Afterwards, open the solution file "OpenCL Tutorials.sln". The solution contains several projects but for this workshop we are interested only in "Tutorial 1". To run the code, first build the solution ("Build/Build Solution" or press "F7") which contains a command line application. The most flexible way of executing this application is by using a separate command line window. Then navigate to the directory containing your built project (e.g. `cd "C:\Desktop\OpenCL Tutorials\x64\Debug"`) and then run the compiled application ("`Tutorial1.exe`"). You should see a line with a short description of the hardware executing the code and three lines of numbers representing two input vectors A and B and their element-wise sum contained in vector C. Nothing particularly special but the addition operation was performed on a parallel device! If you make any modifications, you must repeat this procedure: build the solution first and then run the application from the command line. Let us now go through the code and try to understand how it works.

The host program is included in "Tutorial 1.cpp" file. The main function starts first with checking the provided command line arguments which define the behaviour of our program (Part 1). You can set for example a specific device on which you want to run the code, list all existing devices, etc. Part 2 is responsible for host operations: here you select the computing device, load & build device code, etc. Part 3 allocates memory both on the host and device. We will use host buffers to manually provide

input values (vectors A & B) and read the results of the operation (vector C), whilst the device buffers will be used by device code/kernels – in general, an OpenCL device cannot access the host's memory! Part 4 communicates directly with the device and consists of copying initial values of vectors A and B to device memory, executing the kernel code and copying the result vector C back to the host so we can check its content. The header file "Utils.h" contains additional helper functions which wrap up some more cumbersome OpenCL functionality. That is a general overview of the code. In the remainder of this tutorial, we will try to understand more specific details about this simple application.

2.5 OpenCL concepts

There are several key concepts which form the basis of the OpenCL framework. The first one is a "platform". The OpenCL platform is simply a specific implementation of the standard by individual hardware vendors. On a single computer there might be several OpenCL platforms/implementations (e.g. CPU supported by the Intel platform and GPU by the AMD platform). The next term is "device" which is an actual hardware device supporting OpenCL. A "device" will run its code on "computing units" (e.g. a single core of a modern CPU or an NVidia's GPU streaming multiprocessor) which in turn consists of "processing elements" (e.g. ALU in CPU, an NVidia's GPU streaming processor) (see Fig. 2). Each platform can support multiple devices (e.g. Intel platform can support a CPU device and an integrated GPU device which are both present in the modern Core family of Intel processors). Multiple devices from the same platform can be then assembled into different "contexts" which enable flexible configurations on machines with multiple devices (e.g. consisting of 4 GPUs). In our case, we will constraint ourselves to basic setups consisting of a single device and therefore will not have to worry too much about that aspect of OpenCL. The final term is a "queue". A queue is associated with a specific context (i.e. a specific device in our case) and allows the programmer to interact with the device. This is realised by sending different commands to the queue. For example, a command can request a copy operation between the host and device (see Part 4.1 and 4.3) or an execution of a specific kernel (see Part 4.2). All these terms will become more familiar after we start implementing our own programs.

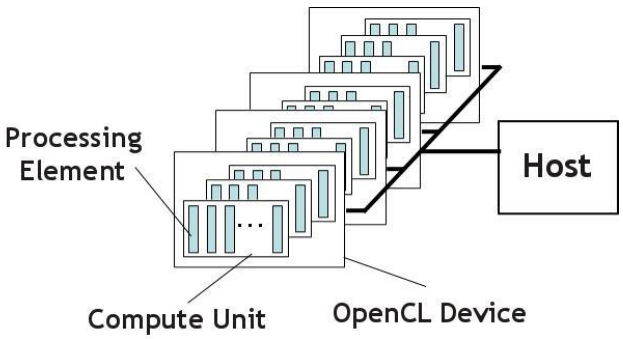

Figure 2 Each OpenCL device consists of at least one Compute Unit, which in turn contains one or more Processing Elements (after https://github.com/HandsOnOpenCL).

Let us now explore the available OpenCL platforms and devices present on a PC in front of you. Our program accepts different command line arguments - you can see all supported arguments by specifying "-h" option when calling the program from the command line. For example, option "-l" lists all the platforms and devices present with additional technical information. Run the program now with this option enabled and inspect all platforms and devices present on your PC. You should

also see different capabilities listed such as available device memory, clock speed, number of computing units, etc. By default, our vector summation program is performed on the first available platform and device. You can select a different platform/device by using options "-p" and "-d" respectively.

Check the values in vector C after running the code on different devices - they should all be the same! The vectors used in our program are very short and therefore there should not be any noticeable difference between executing the code on different platforms. In general, we will not rely on our senses only and will measure the execution time by additional functionality provided by OpenCL.

2.6 Basic Profiling

OpenCL enables the measurement of execution time (i.e. profiling) by the use of so called events. The events are attached to different queue commands and collect information about the timing of the respective command. Let us now profile the kernel execution time with the use of events for which we will need to make the following modifications to our host code:

1. Enable profiling for the queue:

```
cl::CommandQueue queue(context, CL_QUEUE_PROFILING_ENABLE);
```

2. Create an event and attach it to a queue command responsible for the kernel launch:

```
cl::Event prof_event;
queue.enqueueNDRangeKernel(kernel_add, cl::NullRange,
cl::NDRange(vector_elements), cl::NullRange, NULL, &prof_event);
```

3. Display the kernel execution time at the end of the program:

```
std::cout << "Kernel execution time [ns]:" <<
prof_event.getProfilingInfo<CL_PROFILING_COMMAND_END>() -
prof_event.getProfilingInfo<CL_PROFILING_COMMAND_START>() << std::endl;
```

Task4U: You can now check the kernel execution time for different devices. Does the execution time on a single device vary between consecutive launches of the program? How big are the differences between different devices?

You can also get full information about each profiling event including enqueueing and preparation time by using the provided helper function `GetFullProfilingInfo()`. The function accepts two parameters: an event and time unit format. For example, to print out the detailed breakdown of our event in microseconds add the following line of code:

```
std::cout << GetFullProfilingInfo(prof_event, ProfilingResolution::PROF_US) <<
endl;
```

Task4U: How do total times compare to the execution time for a specific event?

2.7 Large datasets

It should be rather difficult to see any significant difference between the execution time on different devices because our input vectors are very short. Let us change the host code to accommodate larger input arrays:

```
std::vector<int> A(1000);
std::vector<int> B(1000);
```

This time we will not care too much about the actual vector values and leave them initialised to 0. You can also comment out the lines of code printing out the vectors – these are long and not particularly informative in this case.

Task4U: Profile your code for different vector lengths (e.g. small, medium, large values (10K, 100K, 1M) on a CPU and GPU device. When does the performance of the GPU device exceed 2x and 10x that of CPU device? What is the maximum allowed length of vectors A, B and C for each of the device? How would you calculate sums of vectors longer than those allowed by the max limit?

Our profiling so far has considered only the kernel execution time. Most likely the performance on a GPU is much better than that on CPU for longer vectors. However, copying large arrays between host and device does not come for free and must be taken into consideration when assessing the overall performance of the system. We can add additional events to measure the upload time (from host to device) for input vectors A and B and download time (from device to host) for the output vector C. For example, a line of code for profiling a copy operation on vector A would look like this:

```
queue.enqueueWriteBuffer(buffer_A, CL_TRUE, 0, vector_size, &A[0], NULL,
&A_event);
```

The total time required for all memory transfers is equal to the sum of all time periods reported by the three vector events. The overall operation time is then a sum of the memory transfers and kernel execution.

Task4U: How do memory transfer times compare to kernel execution times on CPU/GPU and for different vector lengths? What is the difference between the overall operation time for GPU and CPU devices? Are the memory operations between the host and a CPU device (which is exactly the same hardware) performed without any additional cost?

2.8 Kernels

Let us look at the kernel code now. Kernel (or device) code is provided in a separate file "my_kernels_1.cl". At the moment, the file contains a single kernel function only called add.

```
kernel void add(global const int* A, global const int* B, global int* C){
	int id = get_global_id(0);
	C[id] = A[id] + B[id];
}
```

You can see that this is an ordinary C function with some extra qualifiers. `kernel` label indicates that this is an OpenCL kernel function and `global` label indicates that the input parameters are located in the main memory of the device (we will discover other types of device memory in the following sessions). The add function takes pointers of two input integer vectors A and B (since we only read them there is a `const` qualifier in front) and a pointer to output vector C where we store the result

of the addition operation. The parallel add consist of a single line of code which adds individual elements of vectors A and B and stores the sum in a corresponding element of vector C indicated by the `id` variable. As you can see, there is no for loop here which would normally go through all elements of a vector. The kernel function is launched in parallel on as many computing units as available and as many times as necessary. Each launch gets a separate and unique id which is obtained by `get_global_id(0)` function. The total number of kernel launches is equal to the vector length which is specified as a parameter for the `enqueueNDRangeKernel` function: `cl::NDRange(vector_elements)`.

Task4U: Let us try to modify the kernel code. Revert to the original input vectors consisting of 10 values only. Change the kernel code so it does multiplication instead of addition (you do not need to change the function's name at this point). Run the code again and check if the output results are as expected.

Each ".cl" file can contain multiple kernel functions. Let us now revert back to our original kernel code which does parallel addition and create a separate function called `mult` which will perform a parallel multiplication operation. You can now choose which kernel function to call from the host code by specifying the kernel function name in the `cl::Kernel` declaration (i.e. `cl::Kernel(program, "mult")`).

More complex parallel programs usually involve running multiple kernels to achieve the intended outcomes. Let us now create a program which will launch two separate kernels in a sequence: first we will perform vector multiplication and then vector addition corresponding to the following operation: `C = A * B + B`. First, create another kernel variable called `kernel_mult` in a similar way to `kernel_add`. The next step will require setting up the kernel function arguments which can be done in the following way:

```
kernel_mult.setArg(0, buffer_A);
kernel_mult.setArg(1, buffer_B);
kernel_mult.setArg(2, buffer_C);
```

Then you need to change the input arguments for the addition kernel so that the first argument is the intermediate vector C: `kernel_mult.setArg(0, buffer_C)`. The result of parallel addition will simply override the values originally stored in vector C.

Then you need to add the kernel launches into the queue in the right order:

```
queue.enqueueNDRangeKernel(kernel_mult, cl::NullRange,
cl::NDRange(vector_elements), cl::NullRange);
queue.enqueueNDRangeKernel(kernel_add, cl::NullRange,
cl::NDRange(vector_elements), cl::NullRange);
```

In our current setup, both kernels will be executed in a sequence so first multiplication and then, when the results are ready, the addition operation. Run the code and check the results of the operation.

Task4U: A kernel can perform more complex functions rather than just a single arithmetic operation. To illustrate that, implement our previous operation `C = A * B + B` in a single kernel function called `multadd`. Check if the results are exactly the same as in the previous task where two kernels were

called separately. Then compare the overall performance and kernel execution times of both methods on CPU and GPU devices for different vector lengths (e.g. small, medium, large). Which version is more efficient, in what circumstances and why?

So far, our kernels worked on integer values only. It is easy to change the desired data type (e.g. to `float`) but this change must be done both in the host and kernel code. On the host side, vectors A,B,C need to be specified as the new type (e.g. `float`) and the `vector_size` variable needs to reflect the size of the new data type (so `sizeof(float)` in our case). The kernel code needs changing the type of the input parameters (vectors A,B,C) too which will require changing from `int` to `float`. That is it!

Task4U: Create a new kernel called `addf` which will perform parallel addition of two vectors containing floating point values (`float`). Make necessary adjustments in the host code including changing the name of the executed kernel. Is there any difference in performance of different devices for large vector sizes between `int` and `float` data types? Next, repeat the same task but this time for double precision vectors (`double`), perform the comparisons and note the differences. Note: not all OpenCL devices support double variables: if this happens, your program will not execute correctly and report an error.

3 ** Additional Tasks

The following tasks are designed for those eager students who finished all the tasks and would like an extra challenge - the instructions provided are deliberately sparse! These tasks are optional and not necessary to progress further in the following tutorials.

Task4U: Implement and profile a pure serial version of the vector equation: `C = A * B + B`. You can use any programming language you like for this task. You will need to use a loop and some system timing functions. Then compare the overall performance of the serial implementation to the parallel implementation run on CPU and GPU for different vector lengths (e.g. small, medium, large). Does the parallel solution bring any performance boost?

Task4U: Add an additional command line parameter to the OpenCL program performing `C = A * B + B` operation which will specify the length of the vectors. This way, you will not need to recompile the program every time you want to change this value. Then collect the profiling info (the overall performance) for CPU and GPU devices and different size of vector length (at least 10 different values) and plot the values in a graph (e.g. in Excel or Matlab). Repeat the same thing with the serial code and add the results to the graph.

Task4U: Adopt the tutorial project so it can be built and run on Linux or OS X. What changes were necessary to make the project fully multi-platform?

4 Additional Information

4.1 Installation Instructions

These additional instructions might be helpful if you decide to develop OpenCL code on your PC at home. The above code was tested on Windows 10, Visual Studio 2017 and Intel SDK for OpenCL

Applications. With some small modifications, however, it should be possible to run it on different operating systems, programming environments and OpenCL SDKs. The following instructions provide only a short summary of necessary steps but if you are interested in more details please check the following link: http://streamcomputing.eu/blog/2015-03-16/how-to-install-opencl-on-windows/.

4.1.1 Run-time drivers

The runtime drivers are necessary to run the OpenCL code on your hardware. Both NVidia and AMD GPUs have OpenCL runtime included with their card drivers. For CPUs, you will need to install a dedicated driver by Intel (https://software.intel.com/en-us/articles/opencl-drivers) or APP SDK for older AMD processors. It seems that AMD's OpenCL support for newer CPU models was dropped unfortunately. You can check the existing OpenCL support on your PC using GPU Caps Viewer (http://www.ozone3d.net/gpu_caps_viewer/).

4.1.2 OpenCL SDK

The OpenCL SDK enables you to develop and compile the code. In our case, we use Intel SDK for OpenCL Applications (https://software.intel.com/en-us/intel-opencl). After installation, you should be able to build and run the tutorial code in Visual Studio without any problems. You are not tied to that choice, however, and can use SDKs by NVidia or AMD. Each SDK comes with a range of additional tools which make development of OpenCL programs much easier.

4.2 Contributing Your Improvements

If you have great ideas for code improvements or functionality enhancements, please follow the classic git workflow: make a clone of the repository, implement your changes and then make a pull request. Your code will be then reviewed and merged into the main repository. Some additional information about this process can be found here: http://archaeogeek.github.io/foss4gukdontbeafraid/fixes/pullrequest.html. If you spot some bug in the code, please file an issue on GitHub.