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

Parallel Patterns in OpenCL

Download the source code for Tutorial 2 from Blackboard ("Study Materials/Week B4"), unpack the archive into a directory and open the solution file "OpenCL Tutorials.sln". The solution consists of two projects, the original Tutorial 1 and new Tutorial 2, and we are going to use both projects in this session. Refer to the previous tutorial descriptions for more information about the workshops and their structure, and for more technical details on OpenCL.

1 Kernel Execution Model

Let us now look in more detail at the inner workings of OpenCL kernel execution. As mentioned in the previous tutorial, a "device" will run its code on "computing units" which in turn consists of "processing elements" (see Fig. 1). Each separate kernel execution is performed on a processing element and in OpenCL terminology it is called a "work item". You can think about work items as separate threads. In our basic example we have used vectors of length 10, so the kernel launch in this case will consists of 10 work items.

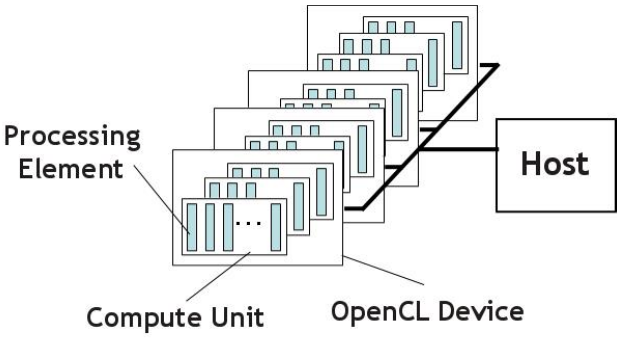

Figure 1 Each OpenCL device consists of at least 1 Compute Unit, which in turn contains one or more Processing Elements (after https://github.com/HandsOnOpenCL).

Each computing unit consists of a fixed number of processing elements which naturally limits the number of work items which might be executed at the same time. In addition, the work items often need to communicate between each other (e.g. share the same variable, report the intermediate results, etc.) and therefore OpenCL introduces a concept of "work groups" which are collections of work items executed by a single compute unit. Finally, to enable work on larger problems, the device will launch as many work groups as possible at the same time, depending on the number of the existing compute units. This process will be repeated until all the necessary work items associated with the problem are completed (see Fig. 2).

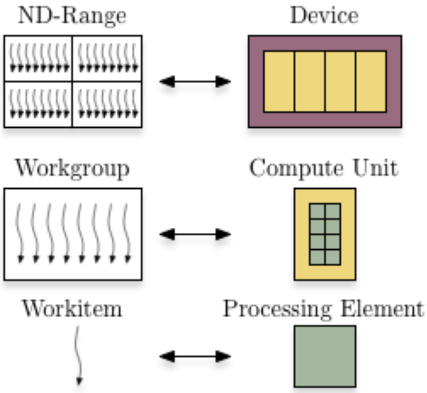

Figure 2 The hierarchy of execution: a problem is executed on a device, a workgroup on a compute unit and a work item on a processing element (after http://mygsoc.blogspot.co.uk/).

Let us see some examples to get a better intuition about these terms. For that purpose, we will need to be able to inspect the inner workings of individual kernels. A simple debugging facility is provided by the `printf` function which can be used to output kernel variables used by each work item. The formatting used by this function is exactly the same is in `printf` used in other languages (see https://en.wikipedia.org/wiki/Printf_format_string for more details). The following example shows how to use this function for printing out the `id` variable from our simple vector addition kernel:

```
int id = get_global_id(0);
printf("work item id = %d\n", id);
```

Open the solution file, navigate to Tutorial 1 and add the code above. If you run the program, you will see the id values for each consecutive work item being displayed. We can also now better understand the purpose of the `get_global_id` function: it returns back a unique id for each work item from 0 to N-1 where N is the total number of work items required to solve a particular problem. Be careful with the `printf` function though – if you use it on larger problems the function might interfere with the execution of kernels and even freeze your computer!

Task4U: Run the modified kernel on different devices and note the order of the execution of work items. Is it the same for each device and each run?

In general, work items are executed in an arbitrary order so one cannot make any assumptions which work item is executed first.

OpenCL specifies a number of functions similar to `get_global_id` which help to identify different launch parameters (see https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf#page=244 for more details). For example, `get_local_id` returns back a work item id within a specific work group. The values vary from 0 to M-1 where M is the size of a work group. This size can be determined by calling `get_local_size`. Now, add the following lines to your code and run the program again and note the differences between local and global ids but also work group size for different devices.

```
if (id == 0) { //perform this part only once i.e. for work item 0
	printf("work group size %d\n", get_local_size(0));
}
```

```
int loc_id = get_local_id(0);
printf("global id = %d, local id = %d\n", id, loc_id); //do it for each work item
```

Task4U: Print out the total number of work items for a given problem (just once) and a work group id for each work item. Note the differences between devices.

The work group size in our example varies depending on which device it is being running on. This is due to the fact that our kernel execution command has left out the fourth parameter as empty (`cl::NullRange`) which is signalling to OpenCL that this value should be set automatically. It is possible however to define the work group size (or local size) manually when calling the kernel from the host:

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

There is a requirement imposed by OpenCL 1.2 that the input data size should be divisible by the work group size and therefore not all local sizes are permitted. There are several ways to overcome this limitation which we will see in future workshops, but at the moment we will only work with datasets which meet that condition.

Different kernels will have different preferred work group size (due to device architecture, memory usage, etc.). The recommended values for the work group size can be identified by reading a kernel property called `CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE`. This parameter can be accessed in the following way in the host code:

```
cl::Device device = context.getInfo<CL_CONTEXT_DEVICES>()[0]; //get device
cerr << kernel_add.getWorkGroupInfo<CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE>
(device) << endl; //get info
```

This value is the smallest work group size suggested, but its multiples are also possible, up to the maximum work group size specified by another kernel parameter which can be accessed in the following way:

```
kernel_add.getWorkGroupInfo<CL_KERNEL_WORK_GROUP_SIZE>(device)
```

Task4U: Run the vector addition kernel on different devices for the work group size equal to the value specified by the preferred work group size parameter and note the differences in local and group ids. You will need to manually adjust the length of the input vector so it is divisible by the work group size.

The work groups play an important role in applications which require work items to communicate with each other. We will see such examples in the following tutorial sessions.

2 Parallel Pattern – Map

Parallel programming patterns define standard abstract components from which larger algorithms and programs can be built. One of the most straightforward parallel patterns is the Map (see Fig. 3).

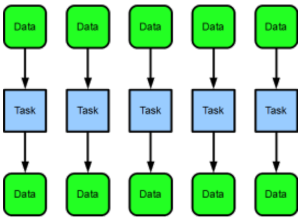

Figure 3 The Map parallel pattern (after http://parallelbook.com/sites/parallelbook.com/files/ SC13_20131117_Intel_McCool_Robison_Reinders_Hebenstreit.pdf).

The map pattern describes the same computation performed on different data without the need for communication between work items. So far in these workshops, we have been working solely with the map pattern, even if we have not mentioned that explicitly. Let us now see how to apply the map pattern to solve practical problems.

For this purpose, we will use some simple applications in digital image processing due to the fact that the results of operations on digital images can be easily visible and provide instant feedback. This is however not, by any means, a tutorial on digital image processing and we will only use these examples as illustrations to generic problems in parallel processing.

The second project in the solution folder called Tutorial 2 is setup to enable I/O operations involving digital images. The framework is based on one of the OpenGL implementations called Glut (https://www.opengl.org/resources/libraries/glut/) to enable simple visualisation and interaction with the user, and the "lodepng" library (http://lodev.org/lodepng/) to handle reading and writing of digital images from/to files in a png format. A digital image in our project is represented as a 1D byte array (`vector<unsigned char>`) with each pixel being represented by 4 consecutive bytes storing the intensity level for each colour channel (RGB) and transparency level (A). Each RGBA component can assume values from 0 to 255. The image has also its width and height defining rows and columns.

To build the code, you need to switch the active project to Tutorial 2 (right-click on the project in the Solution Explorer/Set as Start-up Project). The code is very similar to Tutorial 1 with additional functions for image loading and display, and changes to variables. This time instead of input and output vectors we have two images: `image_before` which is our input and `image_after` – our output. Before running the code, study it first and note the differences to the host program from Tutorial 1. The provided kernel code demonstrates a simple copy operation so the output image is exactly the same as the input after the execution of the program. There are two images provided with the code: smaller version called "test.png" which is good for development and testing and "test_large.png" for evaluation and analysis.

The identity kernel uses a special OpenCL data type for our input and output arguments called `uchar4` which is simply a collection of 4 bytes. This data type will make it easier to work on individual pixels. The `uchar4` type has additional convenience methods and functions. The `xyzw` fields give access to individual bytes and in our case they are equivalent to RGBA pixel components. For example, to set the intensity value of a green channel to a maximum, one has to perform the following operation:

```
uchar4 pixel;
pixel.y = 255; //set the maximum value for the second field of uchar4 -
               //green channel
```

More details on other vector data types similar to uchar4 can be found here: https://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/vectorDataTypes.html.

Task4U: Write a kernel code called `filter_r` *which will perform colour channel filtering. The output image should only contain values for red colour component and all other components (GB) should be set to 0. Leave the transparency component unaffected. Try out the filter for other colour channels too.*

Task4U: Write a kernel code called `invert` *which will invert the intensity value of each pixel (e.g. 255 becomes 0, 0 becomes 255, 100 becomes 155, etc.). Leave the transparency component unaffected.*

Task4U: Profile the image inversion code on a large image (i.e. test_large.png) for at least three different work group sizes. Compare the results to those obtained with the default settings used by OpenCL. Which configuration produced the best results?

Task4U: Write a kernel function called rgb2grey which would convert an input colour image into greyscale. Keep the RGBA format of the output image (it is necessary for displaying purposes) but set each RGB channel to the same value of Y = 0.2126R + 0.7152G + 0.0722B.

3 Parallel Pattern – Stencil

A stencil pattern takes multiple data inputs from a pre-defined neighbourhood and combines them into a single value. Similarly to the map pattern, each task/work item is independent. This pattern has several important applications in signal processing such as noise filtering, image blurring/sharpening, etc.

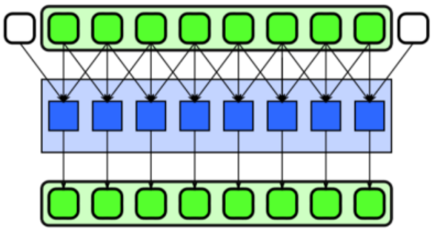

Figure 4 The Stencil parallel pattern (after http://parallelbook.com/sites/parallelbook.com/files/SC13_20131117_Intel_McCool_Robison_Reinders_Hebenstreit.pdf).

3.1 1D Stencil

Let us go back to our project Tutorial 1 so we can have a look at a simple 1D example first. In the kernel file, you can find a kernel function called `avg_filter` which performs an averaging operation in a local window of size 3 (or range 1). The result is an average of values at the central position and the immediate neighbours to the left and right. This operation is equivalent to a simple smoothing filter which can be applied to filter out noise from data (e.g. audio signal). First, adapt the host code so it can run the kernel and then test its operation on different input values.

Task4U: Modify the code such it uses a window of size 5 (or range 2). How does this larger window affect the smoothness of the signal?

Task4U: Currently, the kernel is not handling the boundary conditions. In our case, the leftmost and rightmost vector elements (i.e. id = 0, id = length-1) have one neighbour less and require different treatment. Add the boundary handling to the kernel and compare the results with the original implementation.

3.2 2D Stencil

So far we have only worked with one dimensional arrays. In certain applications, however, it is more convenient to specify the problem in more dimensions. For example, when dealing with digital images it is easier to refer to pixel x and y coordinates rather than their location in a 1D array. Kernel dimensionality is specified within a kernel execution command. In 2D case, the first parameter of the `NDRange` function specifies the number of columns (width) and the second one the number of rows (height). For example, to call a 2D kernel with data arranged into a 5x2 array, the following modification to the host code is required:

```
queue.enqueueNDRangeKernel(kernel_add, cl::NullRange, cl::NDRange(5, 2),
cl::NullRange);
```

The kernel code also needs to be modified. The global indices are now arranged into two dimensions and the `get_global_id(d)` function will return the number of elements along the dth dimension. In our example, for d=0 this value is 5 and for d=1 it is 2. These indices can be combined into a global linear index as it is demonstrated in the kernel code called `add2D`.

Task4U: Modify the host code in Tutorial 1 to run the 2D kernel add2D. Run the kernel in a 5x2 arrangement and see if the results are exactly the same as in 1D case. Inspect global indices and sizes along both dimensions.

Task4U: Modify the host code in Tutorial 2 to run the 2D kernel `identity2D`. Run the kernel in a 2D arrangement with image width and height specifying values for 2 dimensions. See if the results are exactly the same as in 1D case.

3.2.1 Averaging filter

Let us now look at an averaging filter in 2D. The kernel code `avg_filter2D` in Tutorial 2 performs a similar function to the 1D averaging filter developed in the previous task. The filter calculates an average of 9 pixels in a 3x3 local window. This operation results in a blurred version of the input image.

Task4U: Adjust the host code such that the 2d averaging filter kernel can be run on the input image. Specify `image_width` and `image_height` as values for kernel dimensions. Inspect the kernel code (note how 2D indices are used in the loop), run the code and visually test the results.

Task4U: Modify the kernel code such it uses a window of size 5x5 (or range 2). How does this larger window affect the results?

3.2.2 Convolution filter

The averaging filter can also be implemented as a generic image convolution operation. Image convolution uses a rectangular mask of weighting values (also called a convolution kernel) applied to every pixel and its neighbourhood. The result of this operation is a weighted sum of pixel values in the local neighbourhood. Each pixel is multiplied by the corresponding mask value and added together (see https://en.wikipedia.org/wiki/Kernel_(image_processing) for an illustrative example). Our averaging filter can be simply expressed as a 3x3 mask with all entries equal to 1/9. The result of such a convolution operation is exactly the same as for the avg_`filter2D`. There are many other combinations of mask values which result in different image processing operations including sharpening, soft blurring, edge detection, etc. There are several examples of such masks listed in the following Wikipedia entry: see https://en.wikipedia.org/wiki/Kernel_(image_processing). Tutorial 2 has a simple implementation of a 3x3 convolution filter provided in the kernel file. You will notice that the mask is provided as an additional input argument and initialised in the host code. To run the convolution example, you need to modify the host code by uncommenting the respective code lines responsible for allocating and transferring the mask values to the device.

Task4U: Study the averaging convolution kernel and run the code. Compare the blurring results with those obtained from the avg_`filter2D` kernel. You may want to use screenshots to perform direct visual comparison.

Task4U: Try out other convolution masks such as those specified in the mentioned Wikipedia article.

4 ** Additional Tasks

The following tasks are designed for those eager students who finished all 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: Write a 1D averaging filtering kernel avg_`filter` which will accept an additional parameter defining the radius. Make sure that boundary conditions are correctly considered for different radius size. Compare the results to the original fixed radius implementations from Task 3.1.

Task4U: Write a kernel function brightness which in addition to input and output images will also accept a separate integer parameter called beta. This value should be added to the intensity value of each colour component. This is a simple brightness adjustment function. Don't forget to convert types and clamp the output range to 0-255 for which you might want to use conversion functions such as `convert_uchar4_sat` (see for example: http://www.informit.com/articles/article.aspx?p=1732873&seqNum=8). Test your program on a larger test image for different values of the brightness parameter.

Task4U: Write a kernel function called `contrast` which will adjust the contrast of an input image. The contrast value should be specified as a `float` parameter which then should be multiplied by each colour component. Try different values of the parameters on your input image.

Task4U: Modify the kernel function avg_`filter2D` such it accepts an additional parameter called range defining the size of the averaging window. As an additional challenge, add proper handling of boundary conditions.

Task4U: Modify the kernel function convolution2D such it accepts an arbitrary mask size specified by an additional parameter called `mask_size`. Try the new functionality with larger convolution masks.

5 Additional Information

If you would like to use this project at home, you may need Glut's run-time libraries (glut32.dll). The files are provided in the Graphics/Bin directory.