GPGPU & Accelerator Programming

Assignment 2

B00241075

# Introduction

The aim of this project is to take some existing code and speed it up by making use of GPGPU programming tools such as OpenCL, CUDA, C++ Amp, etc. These tools and libraries allow calculations to be run on hardware other than the CPU, or by making use of powerful features of the hardware being used (including a CPU). The existing code, in this case, is an implementation of an unsharp-mask. This is a method of sharpening an image by adding a blurred mask of the image onto the original.

# Original Code

The original code is split up into five files:

* add\_weighted.hpp
  + adds two arrays together according to a formula:, where and are the two inputs to be added.
* blur.hpp
  + Holds the logic for blurring an image by averaging the pixels in a given radius.
* ppm.hpp
  + Used for reading and writing the .ppm image file type.
* unsharp\_mask.hpp
  + Contains the Unsharp-Mask logic.
* unsharp\_mask-cpp
  + Contains program entry-point.

Looking at the code it can be seen that the two files with the most processor intensive code are blur.hpp and add\_weighted.hpp. Both require iterating over all pixels in an image and either averaging it with the pixels surrounding it, or adding its value to another image’s according to the formula mentioned above.

Two tests were initially done on the overall time of the program for different sized images. First one is 512x512, nearing 3MB in size, and the second is 3841x2160 and 80MB. Running the mask for the first, smaller image, with a blur radius of 5 pixels, takes a total of about 1.3 seconds. Running it for the second image, also with a blur radius of 5, took 42 seconds. The originals and results from these two tests can be seen below.



Figure 1 Lena - 5 Blur Radius

Figure 2 Lena - Original

# C:\Users\Floofy\Projects\unsharp-mask\unsharp_mask\x64\Debug\ghost-town-8k-processed.jpgC:\Users\Floofy\Projects\unsharp-mask\unsharp_mask\x64\Debug\ghost-town-8k.jpg

Figure 3 Ghost town - Original

Figure 4 Ghost town - 5 Blur Radius

The blur radius is important both to the effect of the mask and the time taken to apply it. Increasing the blur radius from 1 to 15 in subsequent runs increased run time from less than a second to almost 10 seconds, as shown below, where the time taken seems to increase exponentially to the radius.

The effect the blur can have on the final, sharpened image, can also be seen in the following images, showing a blur radius of 1 on the left, and 15 on the right.

# Parallelisation

In order to speed up the program, the intensive parts of the program should be identified and either made more efficient or moved onto more powerful or appropriate hardware. The intensive parts have already been identified in this case, and as this project is focussed on GPU programming, that is what the focus for implementation will be, to parallelise code and use the GPU’s mass heterogeneous computing capabilities.

OpenCL was chosen as the library to implement this parallelisation with. It was chosen over CUDA because of its cross-platform capabilities, and over the likes of C++ Amp due to its relative flexibility.

## Setting Up OpenCL SDK

The OpenCL specification is created and maintained by the Khronos group, but is implemented by interested parties such as NVIDIA, AMD, Apple, Intel and many others. Installation of appropriate drivers and SDK for the distributors of the hardware being used must be done before development can be started. In this case, two OpenCL implementations have been set up for the project and tested on, namely, OpenCL 1.1 through NVIDIA CUDA installation, and OpenCL 3.0 through the AMD APP installation. Binaries developed using OpenCL should run as long as correct drivers have been stalled, but to use the Visual Studio Solution for this project, include, library and binary directories will require changing to allow for distributions other than the two mentioned above.

## Initialising OpenCL

Initialisation of OpenCL requires some boiler-plate code which may often be repeated for different projects, as such, we will not be going into a lot of detail with this code. There are, however, a few interesting points to bring up.

A basic OpenCL program (such as this one), involves creating a single OpenCL context, command queue and program. Creation of the context is done by finding an appropriate platform available, and a list of devices for that platform. In our case we also do some checking on the platform’s and devices’ version, name and vendor, as well as the devices’ maximum compute units, memory allocation size and global memory size. Our platform is:

* Version: OpenCL 1.1 AMD-APP-SDK-v2.5 (709.2)
* Name: AMD Accelerated Parallel Processing
* Vendor: Advanced Micro Devices, Inc.

While two devices which are found are

* Version: OpenCL 1.1 AMD-APP-SDK-v2.5 (709.2)
* Name: Turks
* Vendor: Advanced Micro Devices, Inc.
* Max compute units: 6
* Max alloc size: 536870912
* Global memory size: 2147483648

And

* Version: OpenCL 1.1 AMD-APP-SDK-v2.5 (709.2)
* Name: Intel(R) Core(TM) i7-2630QM CPU @ 2.00GHz
* Vendor: GenuineIntel
* Max compute units: 8
* Max alloc size: 2147483648
* Global memory size: 8535261184

Creation of the command queue is simply done by calling the *clCreateCommandQueue* function, passing it a context and device ID, and set what mode it should be in. In this case, no mode is selected so commands are executed in order and no profiling is enabled. It should be noted however, that the *clCreateCommandQueue* function is deprecated as of OpenCL version 2.0. As this project is partly developed with SDK version 3.0, but targets 1.1, this deprecation must be worked around. One work-around is to define the CL\_USE\_DEPRECATED\_OPENCL\_1\_2\_APIS flag before including cl.h. Unfortunately, this did not work for the AMD APP SDK, so cl.h was instead modified to remove the deprecation flags around the *clCreateCommandQueue* declaration.

Creation of the program is slightly more involved. This is where any kernels that have been written are loaded and built. To do this, each kernel is placed in its own file which are all read into an array of strings. The program is then created using *clCreateProgramWithSource*, which takes the array of strings as a parameter, along with the OpenCL context. Once the program has been successfully created, it needs to be built. This is done with *clBuildProgram*. A very useful thing to do after this method is to check the result of the build. This is done by checking the result of the function, and if an error occurred we check the build logs, as below.



This retrieves the build log and prints it to stdio. This turns out to be very helpful when debugging kernels, as it provides compilation errors.

## Blur

It was speculated that a significant speedup could be attained by parallelising the code responsible for blurring the image. To do this, a kernel was created which would do what the *blur* function does in blur.hpp on the GPU, using OpenCL. The kernel is shown below.



The ­*\_kernel* keyword specified that the following function is an OpenCL kernel. The kernel signature has four parameters, including pointers to the input and output buffers, specified as residing in global memory, and the blur radius and number of channels in the image. The pixel index is retrieved using *get\_global\_id*, and the width and height of the image using *get\_global\_size*. The code is then very similar to the *pixel\_average* function inside blur.hpp.

Inside unsharp\_mask.hpp, an overloaded *unsharp\_mask* function is created which takes an additional three parameters, namely, the OpenCL context, command queue and program. This overloaded function is then called when running on the gpu. A reference to the blur kernel is created, along with two buffers to store the image data on the device, and writes the image data to the first, using the command queue.



The kernel parameters for the blur radius and number of channels are set once.



In the original code, the blur code is run consecutively three times, compounding the blur. To allow this with the kernel, we also call it three times, swapping which buffer is uses for the in an out parameters.



Where *globalWorkSizes* is defined as an array containing , the width and height of the image. After this we read the buffer back into main memory, and use the *add\_weighted* function to perform the final part of the unsharp-mask algorithm, and release the buffers and kernel used.



For the smaller image, the change to OpenCL resulted in a time of 0.11 seconds (from 1.3 in the original code), and the larger image took 3.31 seconds (from 42 seconds in the original code). Over ten times quicker for both. The result from both can be seen below on the right, next to the originals on the left.



The results from running the code on different blur radii can be seen below.

While the times are significantly smaller than they were for the original code, the times still rise exponentially and could possibly get high for large radii and images.

An indication that there may be a problem with the kernel or how the kernel is used, is that there is a deformity in the processed image along the left and right edges, which grows in size as the blur radius does, as is shown below with the results from the mask applied using a radius of 15 with the parallelised code (left) and the original code (right).



This may have something to do with how the kernel attempts to access memory when the radius causes access outside of the image.

In an attempt to address the growth of time as blur radius does, the pixel averaging code was refactored to make use of OpenCL work items too, rather than the nested for loops it currently had. In order to do this, private memory, work-groups and barrier were required to keep the functionality. The kernel was modified to what is shown below.



The important changes are the calculation of the pixel to be added to the total, namely *pixel\_x* and *pixel\_y*. Originally, two extra dimensions were going to be added to the work-items with the size of the blur diameter, allowing an extra work item for each pixel in the blur radius. Unfortunately, three was the maximum dimensions allowed in this case so the method above was used, where work-items are in a single extra dimension. The totals for the three colour channels are then declared as private, to persist their values among a work-group. A work-group in this case has work-items in it, enough to calculate the total for a given radius. A barrier is then placed after the totals have been added to, to synchronise the items in the work-group before setting the output to the totals. Inside *unsharp\_mask.cpp*, in our overloaded *unsharp\_mask* function, we modify the work dimensions so it looks like the following.



This adds an extra dimension to the work-items and specifies the size of the work-groups. Each *clEnqueueNDRangeKernel* call is modified to also take the *workGroupSizes* array as the sixth parameter.

Unfortunately, these changes increased times to around two minutes for a blur radius of 15. The final result also did not match those given by the original or previous version of the code, a can be see below.



Figure 5 Original

Figure 6 Modified

## Weighted Add

The second intensive part of the algorithm is the weighted add, which adds the blurred image generated with the code explored above, to the original one, giving a sharper looking image. The kernel and the code to call it are simpler to the blur ones. A kernel was created in a separate file to define what should be run on the device.



The kernel takes 7 parameters, a pointer to three buffers, one for the result, and two for the two to be added, three floats for the alpha, beta and gamma components of the equation and the number of channels in the image. The kernel then retrieves the current pixel index and image size using the same method as in the blur kernel. Finally, it applies the weighted addition formula to the image data and saves the result in the *out* parameter.

In the *unsharp\_mask* function, where *add\_weighted* would normally be called, we now intiailise the kernel and add it to the command queue.



The kernel requires an extra buffer to write the result to, and the original image data, which is written again into *buffer1* before setting the parameter data. The result from the kernel is then read from the device into the *out* parameter supplied to the *unsharp­\_mask* function.

The timing results from this change look very similar to the previous ones with blur radii ranging from 1 – 15.

The differences are slight, but are more noticeable in table format.

|  |  |  |
| --- | --- | --- |
|  | Blur | Blur & Add Weighted |
| Blur Radius | Time | Time |
| 1 | 0.014599 | 0.0087631 |
| 2 | 0.046868 | 0.03254 |
| 3 | 0.044389 | 0.0386788 |
| 4 | 0.068969 | 0.0844453 |
| 5 | 0.119943 | 0.104942 |
| 6 | 0.158416 | 0.187235 |
| 7 | 0.233708 | 0.230941 |
| 8 | 0.295781 | 0.27776 |
| 9 | 0.33591 | 0.336491 |
| 10 | 0.39697 | 0.407518 |
| 11 | 0.495832 | 0.503194 |
| 12 | 0.57771 | 0.55665 |
| 13 | 0.672067 | 0.682386 |
| 14 | 0.753117 | 0.800174 |
| 15 | 0.897818 | 0.889284 |

The newer version definitely seems to start off better than the previous version, but still does worth in some of the subsequent runs. A count of the number of runs which has a better time for the newer code finds that 8 did better with the new code, almost exactly half.

# Problems Encountered

# Conclusion