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 Lena - 5 Blur Radius

Figure 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 Ghost town - Original

Figure 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, out-of-order execution mode is chosen, allowing queued commands to run in any order. 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