# GPGPU Assessment 2

## Introduction

The problem this project aims to tackle was speeding up an image sharping algorithm using the an GPGU technology of our choice. For this project, OpenCL (Kronos Group, 2009) was chosen.

## Why OpenCL

OpenCL was chosen to take on this approach. Many other candidates existed, including; CUDA (Nvidia, 2007), Thrust (Nvidia, 2009), AMP (Microsoft, 2008), and SYCL (Kronos Group, 2015). However, OpenCL was chosen for a number of reasons. It is a standard, and is widely used, and a lot of resources exist online for using it.

It is also much more portable than CUDA, which will only work on NVIDIA GPU. By contrast, OpenCL will work on NVIDIA, AMD, and Intel GPUs, which are the three main ones on the desktop platform.

## Tools used

Various tools were used throughout the development of this project. These include; Sublime Text 3 (Skinner, 2016), Microsoft Visual Studio 2013 (Microsoft, 2013), OpenCL, Microsoft Word 2013 (Microsoft, 2013), Git (Torvalds, 2005), and Github (Preston-Werner et all).

The code was run on Microsoft Windows 10 Home operating system. The processor was Intel Core i7 Ivy Bridge x64-based processor (Intel, 2012). The computer had 8GB RAM installed.

For the GPU code, it was run on the AMD Radeon HD 7700 Series (AMD, 2012).

## Blur

This first part of the project which I moved onto the GPU was the blurring of an image. Because the boxed blur algorithm used works on individual pixels, it was a very good candidate to become parallel.

First, the blur code will allocate memory on the GPU to store the original image. Then it will copy the memory from the CPU to the GPU. The blur then allocates two extra buffers, which are used to store the blurred data. For however many blurs the user wants (the default is three), the program will calculate the blur and write it into the first buffer. Then it will calculate a blur on the first buffer and write it into the second. Then it calculates a blur on the second buffer and write it back into the first. Having these buffers swap allows there to be an infinite number of blurs performed while keeping the number of memory allocates on the GPU at two.

Once the blur is done, the passed the final blurred image into the sharpening code. The code which does the box blur is shown in Figure 1.

|  |
| --- |
| \_\_kernel void pixel\_average(\_\_global char unsigned \*out,  \_\_global char unsigned const \*in,  int const x, int const y, int const blur\_radius,  int const w, int const h, int const nchannels)  {  float total[4] = {0, 0, 0, 0};  int const nsamples = (blur\_radius\*2-1) \* (blur\_radius\*2-1);  int byte\_offset;  for(int j = y-blur\_radius+1; j < y+blur\_radius; ++j)  {  for(int i = x-blur\_radius+1; i < x+blur\_radius; ++i)  {  int const r\_i = i < 0 ? 0 : i >= w ? w-1 : i;  int const r\_j = j < 0 ? 0 : j >= h ? h-1 : j;  byte\_offset = (r\_j\*w+r\_i)\*nchannels;  for(int channel\_index = 0;  (channel\_index < nchannels);  ++channel\_index)  {  total[channel\_index] += in[byte\_offset + channel\_index];  }  }  }  for(int channel\_index = 0; (channel\_index < nchannels); ++channel\_index)  {  out[byte\_offset + channel\_index] =  (char unsigned)(total[channel\_index] / nsamples);  }  }  \_\_kernel void blur(\_\_global unsigned char \*out, \_\_global const unsigned char \*in,  const int blur\_radius,  const unsigned w, const unsigned h, const unsigned nchannels) {  int x = get\_global\_id(0);  int y = get\_global\_id(1);  if((x <= w) && (y <= h))  {  pixel\_average(out, in, x, y, blur\_radius, w, h, nchannels);  }  } |

Figure 1 - Box Blur in OpenCL.

## Sharpening

The sharping code works by subtracting the original image from the sharpened image pixel-by-pixel. This means that, for each pixel, the Red, Green, and Blue components will be subtracted.

The sharpening code was moved to the GPU much more directly than the blurring code. The sharpening function was changed into an OpenCL kernel function. To call the function, two simple utility functions were created; *set\_argument\_helper* and *run\_kernel*. The first of these is just used to abstract away passing parameters to OpenCL. It increments a global counter variable, so that the parameters can be passed sequentially without the user having to worry about it. The second function is used to launch a kernel. It also sets the global counter back to zero.

## Increasing Blur Radius

Increasing the blur radius had the effect of brightening the colours of the image. Figure 2 shows an image which had a blur radius of 5 pixels, and Figure 3 shows one that had a blur radius of 50 pixels.

|  |
| --- |
|  |

Figure 2 - Image using a blur radius of 5 pixels.

|  |
| --- |
|  |

Figure 3 - Image using a blur radius of 50 pixels.

## Gaussian Blur

A Gaussian blur is the result of blurring an image using a Gaussian function.

|  |
| --- |
| \_\_kernel void blur(\_\_global char unsigned \*out, \_\_global char unsigned \*in,  int const blur\_radius, int const width, int const height,  int const nchannels)  {  int const x\_pixel = get\_global\_id(0);  int const h\_pixel = get\_global\_id(1);  if((h\_pixel <= height) && (x\_pixel <= width))  {  int byte\_offset;  int const significant\_radius = ceil(blur\_radius \* 2.57f);  float val[4] = {0, 0, 0, 0};  float weighted\_sum[4] = {0, 0, 0, 0};  for(int y\_index = (h\_pixel - significant\_radius);  (y\_index < h\_pixel + significant\_radius + 1);  ++y\_index)  {    for(int x\_index = (x\_pixel - significant\_radius);  (x\_index < x\_pixel + significant\_radius + 1);  ++x\_index)  {    int x = min(width - 1, max(0, x\_index));  int y = min(height - 1, max(0, y\_index));  byte\_offset = (y \* width + x) \* nchannels;  float dsq = (x\_index - x\_pixel) \* (x\_index - x\_pixel) +  (y\_index - h\_pixel) \* (y\_index - h\_pixel);  float weight = exp(-dsq / (2 \* blur\_radius\*blur\_radius) ) /  (M\_PI \* 2 \* blur\_radius\*blur\_radius);  for(int channel\_index = 0;  (channel\_index < nchannels);  ++channel\_index)  {  val[channel\_index] +=  in[byte\_offset + channel\_index] \* weight;  weighted\_sum[channel\_index] += weight;  }  }  }    for(int channel\_index = 0;  (channel\_index < nchannels);  ++channel\_index)  {  out[byte\_offset + channel\_index] =  round(val[channel\_index] / weighted\_sum[channel\_index]);  }  }  } |

Figure 4 - Gaussian Blur in OpenCL.

The Gaussian Blur code used was an adapted version which originally posted by Kuckir, 2014. The original code was in JavaScript and only worked on one output channel. The code was ported over to C++, and then to OpenCL, and was made to work on an arbitrary number of channels, up to 4.

The Gaussian Blur version of the code actually produces a sharper image than the box blur version. Figure 5 shows the output using a Box Blur and Figure 6 shows the output using a Gaussian Blur.

|  |
| --- |
|  |

Figure 5 - Box Blur

|  |
| --- |
|  |

Figure 6 - Gaussian Blur

## Timings

The times presented in Figure 7 represent the time it takes to do the blur on the provided image, *ghost-town-8k.ppm*. The timings do not account for time taken to load the image from disk, or write it back. These timings assume the blur radius is 5 pixels, and the image is blurred 3 times. They are presented in milliseconds.

|  |  |  |
| --- | --- | --- |
| **CPU Box Blur** | **GPU Box Blur** | **GPU Gaussian Blur** |
| 5347 | 291 | 7983 |
| 5343 | 292 | 7974 |
| 5339 | 239 | 7973 |

Figure 7 - GPU/CPU comparisons of Box Blur and Gaussian Blur

As you can see from the results in Figure 7, the GPU version of the code is significantly faster than the GPU version. The GPU box blur is almost 18 times faster than the default CPU code that was provided to us.

The Gaussian Blur implemented in this project is very similar to the original Gaussian Blur algorithm, compared to the modern variant of the Gaussian Blur, often called a *Fast Blur*. Because of this, however, the Gaussian Blur is almost 25 times slower than the box blur, on average. It is even slower than the CPU blur code, by about 2 seconds. Extrapolating, this means that the Gaussian Blur could, in theory, take about 143,694 milliseconds, which is just over 143 seconds, or over two minutes.

## Conclusion

As you can see from the timings in Figure 7, using general purpose GPU technologies can be hugely beneficial for getting a large performance gain in code. This is especially true for code which works on a large set of data, performing very similar actions of the data.

## References

Advanced Micro Devices. (2012). *Radeon HD 7000 GPU series.*

Intel. (2012). *Ivy Bridge x64 microarchitecture*.

Kronos Group. (2015). *C++ Single-source Heterogeneous Programming for OpenCL.*

Kronos Group. (2009). *Open Computing Language.*

Kuckir, I. (2014). *Fastest Gaussian Blur (in linear time).* Available at: <http://blog.ivank.net/fastest-gaussian-blur.html> [Accessed 30th March 2017].

Microsoft. (2008). *Accelerated Massive Parallelism.*

Microsoft (2013). *Microsoft Visual Studio 2013.*

Microsoft (2013). *Microsoft Office 2013.*

Nvidia. (2007). *CUDA.*

Nvidia. (2009). *Thurst.*

Preston-Werner, T. Wanstrath, C. Hyett, P. (2008). *Github hosing site.* Available at: <https://github.com/> [Accessed 30th March 2017].

Skinner J. (2016). *Sublime Text 3.* [Online] Available at: <http://www.sublimetext.com/> [Accessed 30th March 2017].

Torvalds, L. (2005). *Git version control system.*