2016

Accelerating the Unsharp Mask

GPGPU & ACCELERATOR PROGRAMMING – ASSIGNMENT TWO

DAVID MIMNAGH – B00254398

Contents

Introduction

The purpose of this project is to analyse the degree of speedup from converting serialised code working on the central processing unit (CPU), to the parallelised version on the graphics processing unit (GPU). The program itself is an unsharp mask which is the most common form of sharpening for an image using the computer. It is common to have it as a feature within different pieces of image editing software like Adobe Photoshop, if it is not available as standard there will no doubt be a plug-in for it such as seen in Paint.NET.

This program can use either OpenCL or Nvidia's CUDA. Due to the program for the first coursework being created on CUDA, it felt wise to choose it again for this coursework. When creating this project it is important that the load times of each program be analysed for comparison between the two versions.

The Unsharp mask doesn't work by adding more detail to an image, instead it works by enhancing the image itself "by increasing small-scale acutance" (Cambridge in Colour, n.d.). The first step in the process is to create a mask of the original image, afterwards it is combined with a negative image which in turn creates an image containing less blur than the original image. There is a downside to this process, however, as this will most likely lose its correctness with regards to the image itself.

Project Outline

The original build of the program works with on the CPU by making use of five files and seven functions within. Each of the functions have their specific part to play within the code and work as follows:

- add_weighted – This single function file takes in several parameters to calculate the weighted summation of the two arrays that are passed into it.
- blur – This file contains two functions within it pixel_average and blur. The purpose of this file is to average the current samples pixels within the designated blur radius. If a pixel is out-with the image bounds then the value will be replicated at the border of the image.
- ppm – The purpose of this file is to handle the reading and writing of the ppm file types. It has three functions within it: Write (write file contents), Read (read file contents) and get_file_contents (called within the read function). All the functions are self-explanatory for their purpose within the program.
- unsharp_mask – There are two files with this name, the first is the header file. The header part to these two files contains a single function called unsharp_mask. This function takes in a few parameters:
    - Out data, this is fed back out for the write to file.
    - In data, the initial image.
    - Blur radius, the blur size of the current program (default is five).
    - W, the width of the image.
    - H, the height of the image.
    - Nchannels, the number of channels within the image.

  This function then does the following: It creates three vectors to hold the data for the blur process, resizes the new vectors to the image size, calls the blur three times for the masking and sharpening effect, and then finally calls add weighted passing in all the newly created/updated variables to receive the final output.

  The .cpp file for unsharpen mask contains the main for the program. It works by having 2 constant character variables for the file input and output name, an integer for the blur radius, a ppm object for the image and two vectors for the data before and after. The first official step in the program is to read in the image file, done by calling the .read function of the ppm file, afterwards the output vector data_sharp is resized to be allow for space for the final image. To ensure proper testing a ‘chrono’ clock is started before the call of the unsharp_mask function (this is the main function for handling all of the image altering process), a second clock is started after the program has finished and subtracted from the first to get the total time taken for the function. After unsharp_mask was complete, the new image is written to file by calling the .write function of the ppm file.

Implementation

The implementation for this project did not involve as much work in comparison to the first project, however, it involved learning some new things like the creation of 2D dim3 grids. To begin the first change made to all of the files was converting them the appropriate 'CUDA' format, the .hpp files to .cuh and the .cpp to .cu. After the files were successfully converted, the process of optimising the code was started.

Add Weighted

The first file to be looked at was add_weighted. The only change made to this file was the addition of two integers pixel_ROW and pixel_COLUMN, these were used to retrieve a pixel co-ordinate on the x and y of the image respectively (ROW is the x and COLUMN is the y). The previously implemented for loops were then changed and replaced with simple if statements to ensure the row and column were within the boundaries of the image. The reason this was done is due to the inability to effectively distribute the tasks and data amongst blocks/threads, in an attempt to keep the program parallel.

```
int pixel_ROW = blockIdx.x * blockDim.x + threadIdx.x;
int pixel_COLUMN = blockIdx.y * blockDim.y + threadIdx.y;

if (pixel_ROW < w)
{
    if (pixel_COLUMN < h)
    {
```

Figure 1: Changes made to add_weighted function.

PPM

This file remained untouched for this project. This file could possibly be changed to allow for the image reading and writing process to be done on the GPU. By moving this process to the GPU the time taken for the program to complete as a whole would be most likely improved a considerable amount. The time taken for the reading and writing image files was upwards of two minutes in some cases, this may not seem substantial but with the larger images being tested multiple times, the wait time added up.

Blur – Pixel Average

Following the similar process from the add_weighted function the loop was removed initially for this function. Although the program still produced an output image, the image was incorrect in the sense that it appeared that it was not being sharpened and the image was in fact being brightened by a large amount (see Figure 2.2).

```
if (ROW < y+blur_radius)
{
    COLUMN = x - blur_radius + 1;
  if (COLUMN < x+blur_radius)
  {
    const unsigned r_i = COLUMN < 0 ? 0 : COLUMN >= w ? w-1 : COLUMN;
    const unsigned r_j = ROW < 0 ? 0 : ROW >= h ? h-1 : ROW;
    unsigned byte_offset = (r_j*w+r_i)*nchannels;
    red_total   += in[byte_offset+0];
    green_total += in[byte_offset+1];
    blue_total  += in[byte_offset+2];
    ++COLUMN;
  }
  ++ROW;
```

Figure 2.1: Initial change made to Pixel Average.

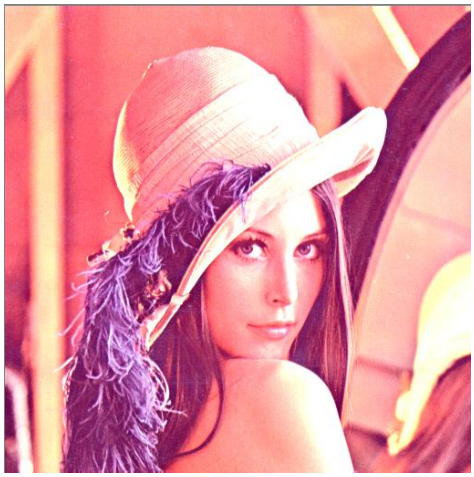

Figure 2.2: Output of pixel average change.

After the program was fully complete and running on the GPU it was realised that the resulting image was too bright. The choice was made to try reverting the code back to its original state, resulting in the program working as expected.

Blur – Blur

The next function changed was the blur function within blur.hpp. The way in which this function was implemented was the same as add_weighted. There are two integer values created (pixel_ROW and pixel_COLUMN) that are used to index the current pixel on the x and y axis of the image. The for loops where then replaced with two simple if statements to ensure that the program is still within the bounds of the image before calling the pixel average function, note the change of the x and y parameter variables in the function call.

```
int pixel_ROW = blockIdx.x * blockDim.x + threadIdx.x;
int pixel_COLUMN = blockIdx.y * blockDim.y + threadIdx.y;
if (pixel_ROW < w)
{
    if (pixel_COLUMN < h)
    {
        pixel_average(out, in, pixel_ROW, pixel_COLUMN, blur_radius, w, h, nchannels);
    }
}
```

Figure 3: Blur function after being changed.

Unsharp_Mask – Header

The biggest change to the original file was within the header file of unsharp mask. This is where the kernels were launched and was essentially the root of the graphics processing unit source code. The choice was made to call the kernel within this function as it was were all of the other functions were being called from and could therefore be viewed as the 'main' function.

To begin, some new variables were going to be needed that would hold the data and information being used by the GPU. This lead to the creation of five unsigned char variables

- Gout and Gin. These two variables hold the data being input to and output from the GPU.
- Gblur1, Gblur2 and Gblur3. These values hold the different set of blur data that is being applied on the GPU.

Created next were two integer variables that were used to hold the maximum number of threads available on the current device. This was done by using the cudaDeviceGetAttribute function, that returns a requested device attribute value (this was done for both X and Y).

```
unsigned char *Gout, *Gin;
unsigned char *Gblur1, *Gblur2, *Gblur3; // GPU blur variables

int maxThreadsX, maxThreadsY;
cudaDeviceGetAttribute(&maxThreadsX, cudaDevAttrMaxGridDimX, 0);
cudaDeviceGetAttribute(&maxThreadsY, cudaDevAttrMaxGridDimY, 0);
```

Figure 4: Variables needed for kernels and retrieving max thread count.

In order to guarantee that the grid size and block size are sufficient enough for all images, the values had to be calculated correctly. First an integer was created to hold the maximum block size to be used in the program, this was set to 32 due to the 32x32 giving 1024 (i.e. the maximum number of threads per any given block). Two float values where that are used to calculate the exact amount of blocks needing to be launched to cover the X and Y (width and height) of the image. When it came to calculating this, incorrect sizes were being obtained for the amount of blocks needed to cover an image, the sizes were in fact rounding down and was later fixed by discussing with Conor about the use of the ceil function.

The next step was the creation of the 2D grid and block to be used in the kernel launch. The dim3 type was used to create a variable "nBlocks" were the X and Y were the previously created XSize and YSize. Another dim3 object was then created called nThreads, the X and Y for this object are both set to the maxnThreads (32 because the maximum amount of threads available per block is 1024, or 32x32).

```
//Set block size and grid size to size of the image.
int maxnThreads = 32;

float XSize = ceil(w / 32.0f);
float YSize = ceil(h / 32.0f);
```

Figure 5: Maximum threads per block and floats used for amount of blocks in grid thread count.

```
dim3 nBlocks(XSize, YSize);
std::cout << " nBlocks: " << "X: " << nBlocks.x << " Y: " << nBlocks.y << std::endl;
dim3 nThreads(maxnThreads, maxnThreads);
```

Figure G: 2D grid and block

Following on from the previous creations, a size variable was created for the allocating the correct amount of space in memory for all of the variables. The size is equal to the size of the image. The variable is of type size_t due to the cudaMalloc looking for a variable of type size_t in its second parameter and so for continuity sake this type was used. However, the code would still work if the type was an integer. The next step was the allocation of memory by using the CUDA function cudaMalloc. This works by looking for a devicePtr as the first parameter and the size variable for the needed memory space, this is done for all of the unsigned char variables that have just been created for use on the GPU.

```
size_t size = ((w * h) * nchannels); // size of the image

//Allocating memory
cudaMalloc((void**)&Gblur1, size);
cudaMalloc((void**)&Gblur2, size);
cudaMalloc((void**)&Gblur3, size);
cudaMalloc((void**)&Gin, size);
cudaMalloc((void**)&Gout, size);
```

Figure 6: Calculating size and allocating memory.

In order to pass through the right data to the kernel launched function the data being passed in has to be copied to the newly created Gin variable. This was done by using the cudaMemcpy function. The function takes in the following parameters: destination, source, size, and kind (device to host or vice versa).

```
//copy to device
cudaMemcpy(Gin, in, size, cudaMemcpyHostToDevice);
```

Figure 7: Copying in data to Gin (CPU to GPU).

After the needed data was copied across the blur and add_weighted kernels were launched with the previously created nBlocks as the number of blocks and nThreads as the number of threads. The only thing that changed from the original function calls was the use of the newly created GPU variables instead of the CPU ones.

```
//run/launch kernals
std::cout << "Starting Blur process..." << std::endl;
blur << <gridSize, blockSize >> >(Gblur1, Gin, blur_radius, w, h, nchannels);
blur << <gridSize, blockSize >> >(Gblur2, Gblur1, blur_radius, w, h, nchannels);
blur << <gridSize, blockSize >> >(Gblur3, Gblur2, blur_radius, w, h, nchannels);
std::cout << "Starting Addition process..." << std::endl;
add_weighted << <gridSize, blockSize >> >(Gout, Gin, 1.5f, Gblur3, -0.5f, 0.0f, w, h, nchannels);
```

Figure 8: Launching kernels for blur and add_weighted functions.

Once the functions had finished the finished data on the GPU had to be copied back over to the CPU. This was done again by using cudaMemcpy with the destination being out and source being Gout.

```
cudaMemcpy(out, Gout, size, cudaMemcpyDeviceToHost); // copy back to host
```

Figure 9: Copying data back from GPU to CPU

To ensure that all memory had been freed after the computer was done with the variables the cudaFree function had to be called the freed up the space on the computer's memory.

```
//Free Memory
cudaFree(&Gblur1);
cudaFree(&Gblur2);
cudaFree(&Gblur3);
cudaFree(&Gin);
cudaFree(&Gout);
```

Figure 10: Freeing memory of computer by using cudaFree function

Unsharp_Mask - CPP

There were no changes made that affected the GPU implementation section of this project. The only real change made within this file was for testing and ease of use, like the creation of a write to file function. This function was a simple C++ write to file function that took an ofstream object to open and write to file.

```
void writeToFile(std::string imagename, int blurRadius, double timeToComplete)
{
    file << imagename << "," << blurRadius << "," << timeToComplete << "\n";
}
```

Figure 11: Write to file function

```
writeToFile("Image name: Lena", blurRadius, timeTaken);
```

Figure 12: Write to file function call within main

Problems Faced

Throughout the duration of this project there were several problems faced. These problems did not have any major effects on the outcome of the problem and were easily resolved in a few hours work. Some of the problems faced through the project duration were:

Pixel average inaccuracy

As previously mentioned, the implementation of the pixel average function on the GPU caused the output to be a lot brighter than expected. This was realised after the whole GPU version was working, resulting in the choice being made to revert the pixel_average function back to its original state. This was done to check if there would be any change to the output image, which there was. It is assumed that the reason for this was due to the image in fact not being subtracted from the original image to create the sharpening effect.

Black image

The second problem faced was that the smaller image "Lena" would work fine, however, the ghost town image would not and only half of the image would sharpened with the rest being completely black. This problem was resolved by taking time to go over the blur and add_weighted functions and was realised fairly quickly that a simple mistake had been made. The mistake was simply checking the pixel_ROW against the height instead of the width resulting in the incorrect sizes being calculated (half of the image).

Loading other Images

The other major problem faced throughout this projects duration was within the extended testing phase for other images. The only problem with this was that it wasn't as straight forward as getting an image, exporting to the .ppm format and then opening it within the program. The image had to be first opened in a text file format, notepad++ in this case, the comment usually made with exporting had to be then removed in order for the ppm .read function to accept the image format without any problems.

Watchdog

The final major problem encountered in this project was the watchdog function of Nvidia. This simply puts a timer on how long the kernel can run before CUDA will cut off the process, at standard this is two seconds. When running a test on the bigger image, with a blur radius above 8, the process takes around 3 seconds to complete. During the running of the program the kernel/process is completely stopped by the Nvidia program to ensure there is no device crash. Unfortunately due to the way in which the computers are set up in the labs, this could not be turned off and the setting saved (when the setting is changed, a restart is required. When the computer is restarted, all changes are reverted back to default).

Testing & Results

To start the testing section of this project in order to get a fair result a blur radius from 1 through to 15 was tested on the main two images on both the CPU and GPU code versions.

The test results are as follows.

Lena blur 1-15 CPU

The first test that was run on the CPU was for the smaller image “Lena”. The blur radius started at 1 and then progressed up to 15 to notice what changes were apparent in the unsharp_mask process time and the overall effect that it had on the image itself. The reason that this is a good test run is due to the amount of computations being increased as the blur radius is increased by just one, a standard 5 blur radius is 10x10 (100 pixels per computation) but increase it to a blur radius to 10 it becomes 100x100 it becomes 10000 pixels per computation.

The time taken to load the image had a direct correlation to the increase in the blur radius. As can be seen from figure 13 below.

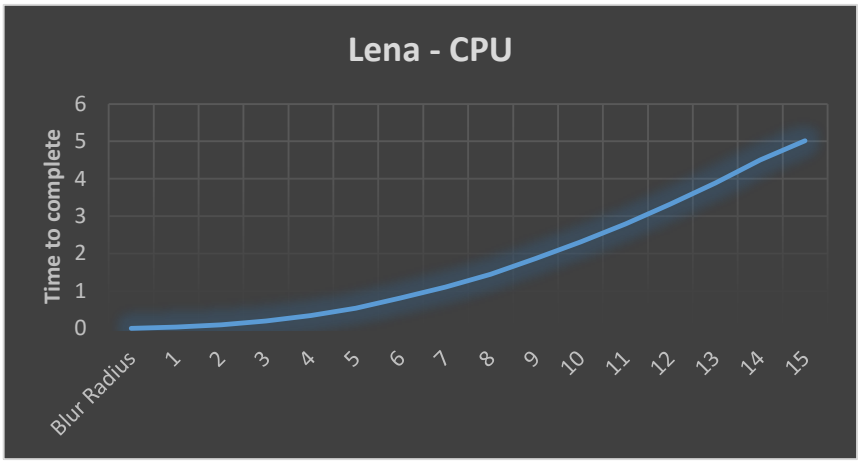

Figure 13: Results from Lena tests on CPU.

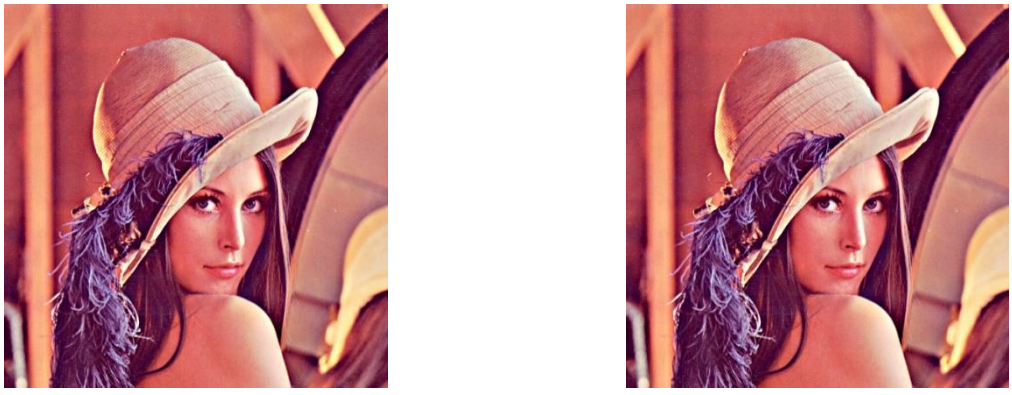

Figure 14: Blur radius 1 on left, 15 on right.

From the tests above it shows that as the blur radius increased the time taken for the function to complete also increased. Starting out at a total of 0.043 seconds. From the above graphs it can be seen that as time progressed, the time taken went up. The total time to run the 15 blur radius test was 5.2 seconds. It can therefore be safely presumed that as the value of the blur radius increased then the time taken to complete the function would increase respectively.

Lena blur 1-15 GPU

The next step was to run the same test set but on the GPU version of the code. The obtained results showed another direct correlation in the time taken for the process to complete with respect to the value of the blur radius. The important thing to note however is the difference in the time taken to complete the functions on the GPU compared to the CPU. The process took a similar amount of time to complete in the beginning, however, as time progressed it became apparent as to just how much faster the parallelised version was compared to the serialised version.

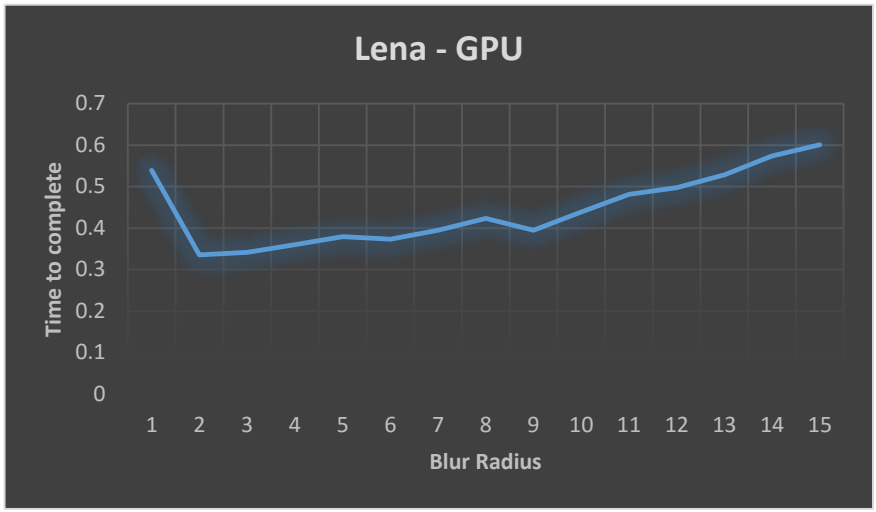

Figure 15: Results from Lena tests on GPU.

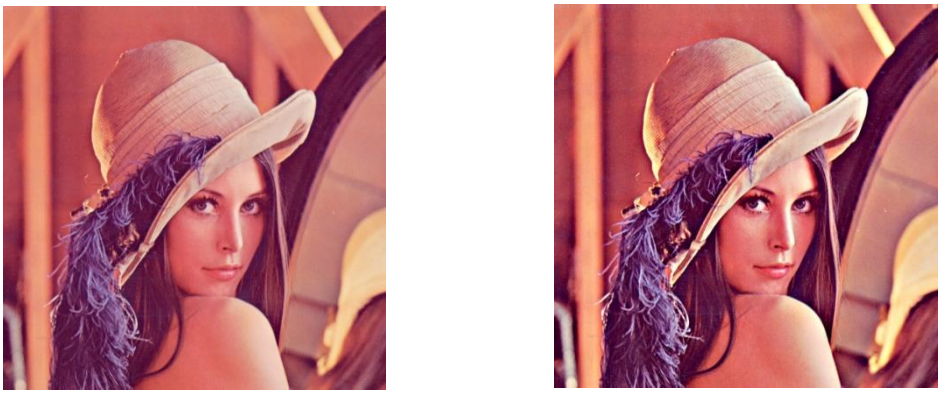

Figure 16: Blur radius 1 on left, 15 on right on GPU.

Analysing the parallelised version of the Lena tests show that a total time taken to run the first tests with a blur radius of one was 0.53 seconds. It can be seen that as time progressed the time taken went up to a total of 0.6 seconds. Comparing this to the first test shows that, as the blur radius increased on the GPU version, the time taken gradually reached a peak at around five seconds for the high blur radius. Comparing this to the first test, the amount of time taken is a massive difference now with the blur radius test of 15 on the CPU taking 5.2 seconds, where the GPU version only took 0.6 Seconds

Ghost town blur 1-15 CPU

Moving on to the bigger image, the file size increased from a simple 512x512 of Lena to the 3840x2160 size of Ghost town. This meant that the ppm's .read and .write function would take considerably longer to work. The testing of the larger images took a lot longer purely due to this fact. For the first test on the CPU, the difference in image size can be immediately noticed by examining figure 17.

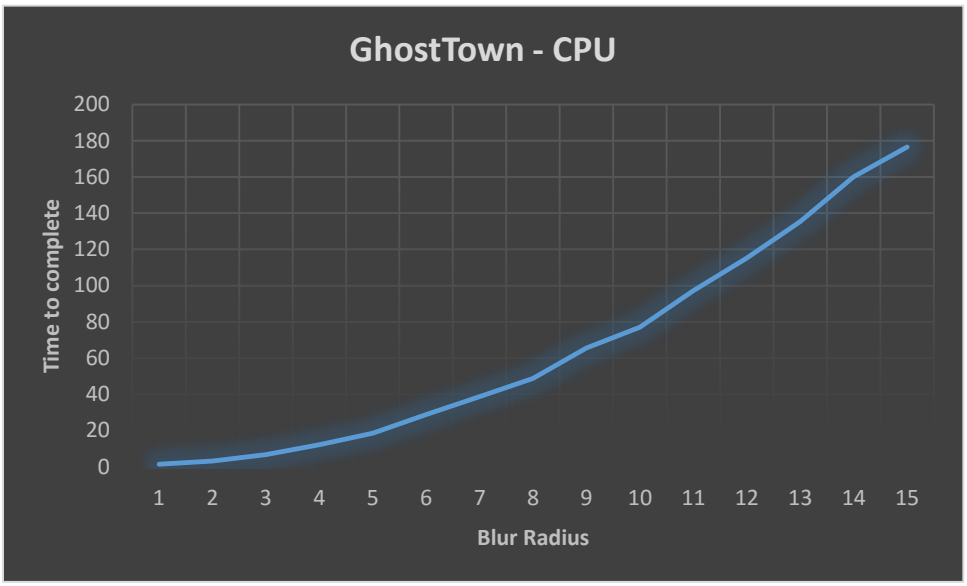

Figure 17: Testing Ghost Town with blur radius of 1-15 on CPU.

Figure 18: Blur radius 1 on left, 15 on right on CPU.

In the above test results it can show just how much the size of the image can affect the time taken for the process to complete. With the serialised version the code took only 1.3 seconds to complete with a blur radius of one. However, for the blur radius of 15 this shot up to 176 seconds, noting a massive increase of 174.7 seconds.

Ghost town blur 1-8 GPU

Unfortunately, the blur radius test could only go up to eight before the Nvidia watchdog stopped the process. The time taken to handle a blur radius of one through to eight can still be compared to the serialised version.

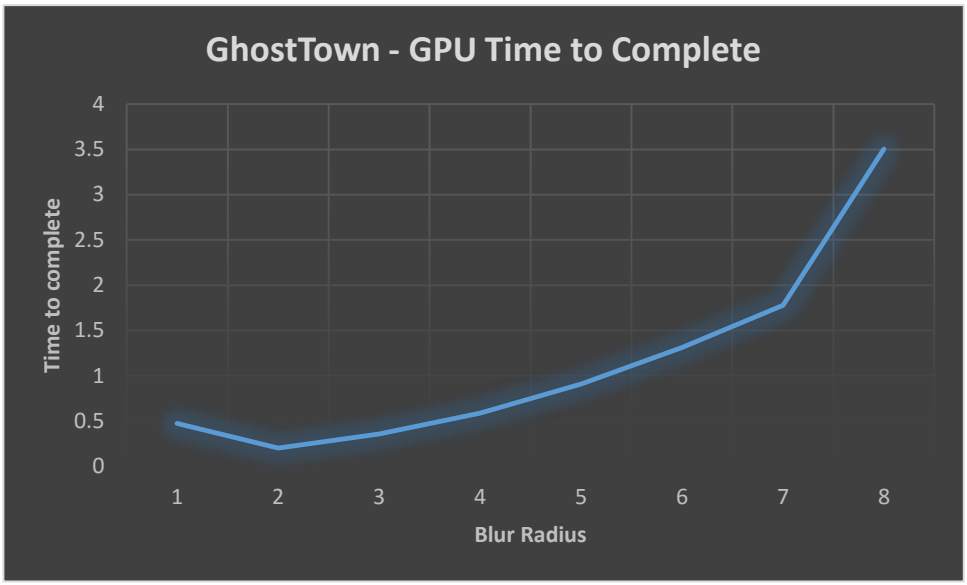

Figure 19: Test results from Ghost Town test on GPU.

Figure 20: Blur radius 1 on left, 8 on right on GPU (due to watchdog).

Looking at the above figure, the time taken to handle the first test was 0.4 seconds. However, as the blur radius value got bigger, the total time taking increased to 3.5 seconds. A comparison between the serialised and parallelised version of the code with the blur radius at eight was 48.8 seconds on the CPU whereas the GPU only took 3.5 seconds.

Other Images - CPU

For the purpose of expanded upon the original project three more images (of different sizes: small is 400x300, medium is 2513x1669 and large is 4608x3072) were chosen to be tested. For simplicity sake, the tested images were all at blur radius five. To begin, all of the images were tested on the serialised version of the code.

To start with the first image tested was the smallest in size. This image, called “programming”, was put through the CPU test and finished in only 0.26 seconds. The reason this image was chosen was to see how much a time difference there would be in an even smaller image than the original.

For the second test, an image sized between the original two was tested on the CPU. The purpose of this was see if there was any noticeable difference between two images, however, there was no noticeable difference with the image taking only 9.35 seconds for a fairly large file After this, it was found that this image took considerably longer to read and write, this was no doubt due to the amount of articulate details within this image.

Moving on to the third test, an image above the ghost town size was tested. This was used again to see if there was any noticeable difference if the image size was increased, however, there was no noticeable dissimilarities. The image took around 32.06 seconds, which in comparison the ghost town image on the CPU with blur radius of five was only around 18.47 seconds, meaning there was an increase of 14 seconds as the size of the image increased.

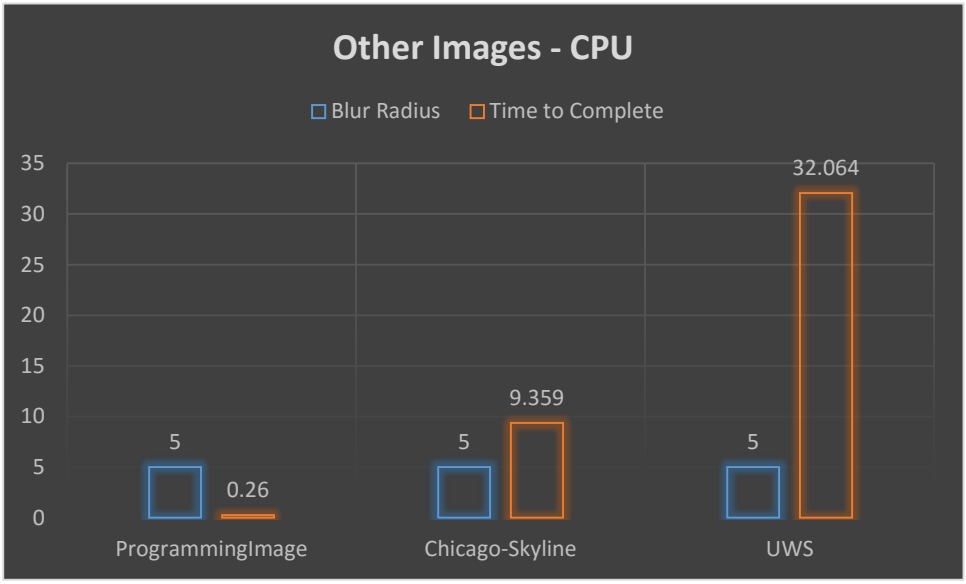

Figure 21: The above is the results from the three tests of different images on the CPU.

Other Images - GPU

The next logical step to take was to then run the new three images on the parallelised version of the code to note the difference.

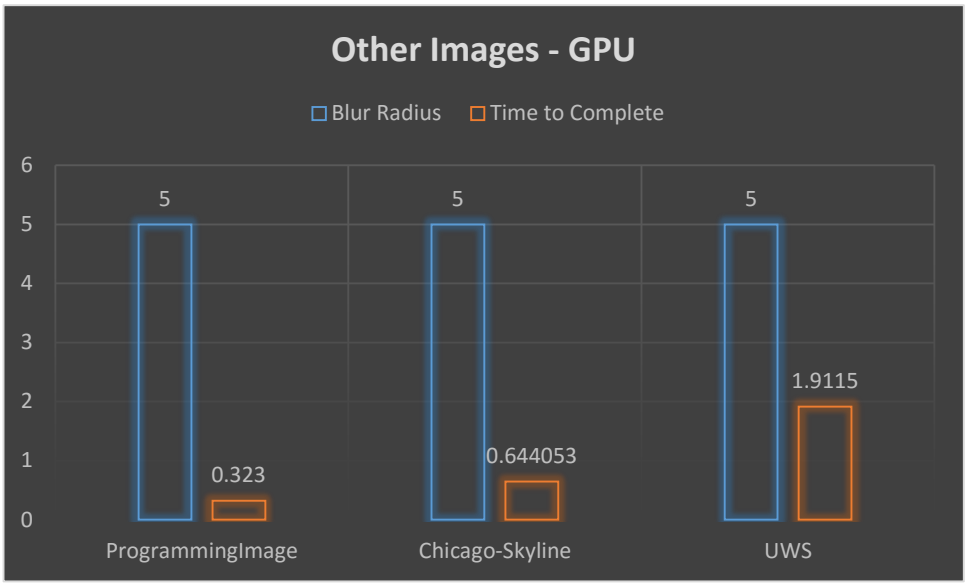

Figure 22: The above is the results from the three tests of different images on the GPU.

By examine figure 22 above, it can be seen that the first image took only 0.323 seconds to complete. Comparing this to the original code version the difference was too minimal to note. This is concurrent with the original set of tests, where there was no real notable difference until the larger images were tested at a higher blur radius.

Examining the second image and comparing it to the serialised version run saw a decrease in the time taken by around 8.7 seconds with the serialised version taking 9.3 seconds and the parallelised version only taking 0.644 seconds.

The final comparison can be made between the serialised and parallelised run of the third (largest) image. The speed up for this version was quite noticeable with the function only taking around 1.9 seconds to complete. This means that over the two versions, the parallelised version was a speed increase of around 30 seconds.

Changing MSVC options

The next thing that was analysed, but unfortunately not tested, was the difference the MSVC options could have on the code if they were changed. After consulting several different sources with regards to this subject, such as:

- Microsoft documentation,
- Stack overflow posts,
- Wikipedia section on optimization

It was found that each of these options have a different purpose, that may or may not have an effect on the output, or performance, of the program.
By going to the property page of the solution, configuration, C/C++ and then to Optimization there are several options which are available to the user which may, or may not optimise the code. For the purpose of this report, the "Optimization" section has been analysed.

Within this section there are seven different options for the user to change, all of which have a different effect on the program. Looking at the Microsoft documentation for the optimization section there are different "/O" options which control various options to help maximise speed or minimise the size of the program.

The different option controls are as follows:

- /O1 – This can optimise the code for minimum size
- /O2 – This can optimise the code for maximum speed
- /Ob – This controls the inline function expansion (replaces call site with body of function)
- /Od – Disables optimization which may speed up compilation and simplify the debugging process.
- /Og – This enables the global optimizations
- /Oi – The creates intrinsic function for the necessary function calls (compiler handles the function)
- /Os – This will tell the compiler to favour size optimisations over speed.
- /Ot – The opposite to the above. This tells the compiler favour speed over size optimisations.
- /Ox – This option allows for a full optimisation.
- /Oy – The purpose of this option is to remove the creation of frame pointers in the code, in turn allowing for snappier function calls.

Adding Gaussian Blur

This was unfortunately not implemented, however, the use of Gaussian blur would be ideal to implement as it would reduce the amount of blur calls needed to zero. The Gaussian would handle all of the blurring by taking a weighted average from around the pixel (Kuckir, 2012), instead of having to call it three times and calculate the average as seen in the normal. The Gaussian blur is more accurate, due to the blur method that is already implemented having a 3% error rate.

Conclusion

In the interest this report, it is important to understand the single purpose of the project. The aim of was to provide information and the creation of a program to prove understanding of the subject area, which in this case was enhancing the unsharp mask. As there are a number of ways in which this could be done, it was important to choose the most effective way for the allotted amount of time for the project. With CUDA being used for the first method of conversion, it made sense for it to be chosen as the implementation method for the second project. As previously mentioned, the program works by taking an image and creating a mask of it, afterwards it is combined with a negative version of the image which creates the final output containing less blur than the original image.

Once the necessary changes had been made, then came the process of testing the different variables within the program. The outcome of the performed tests saw a decrease by around 45 seconds on GPU in comparison to the CPU version. This was expected as there was bound to be a performance increase, however, an increase of 93% from a comparison of the two versions. The biggest improvement was seen throughout the larger tests of images and the higher blur radius values. There was a corresponding increase in time taken to complete the process and the current value of the blur radius. As the program was fully parallelised, the speed increase was around 93% for the larger images, or around 42 seconds for the blur radius of 8.