OpenCL

UNSHARP MASK REPORT

B00235610 2017

This reports sets out the development and techniques used to optimize a C++ program that sharpened an input image. GPGPU was used in tandem with various optimizations to provide a version that was significantly faster and more accurate as well.

TABLE OF CONTENTS

Table of Contents

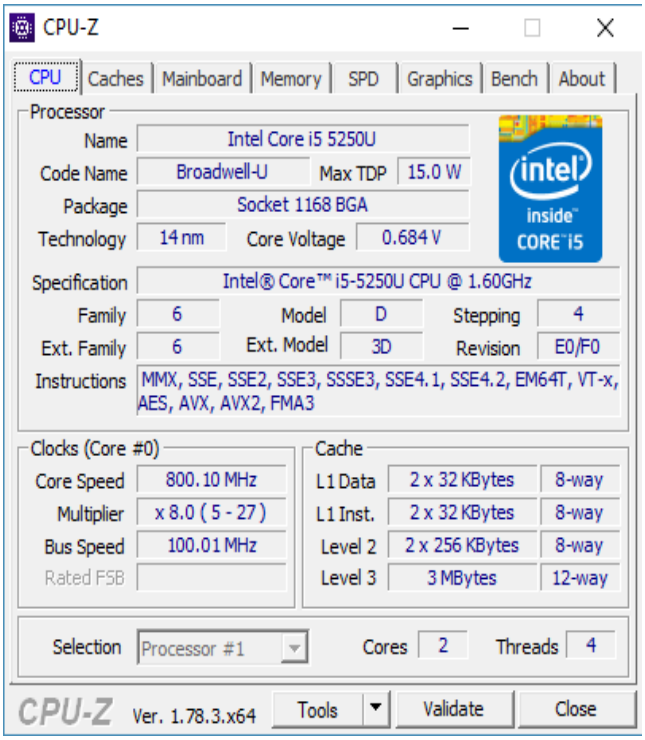
Introduction

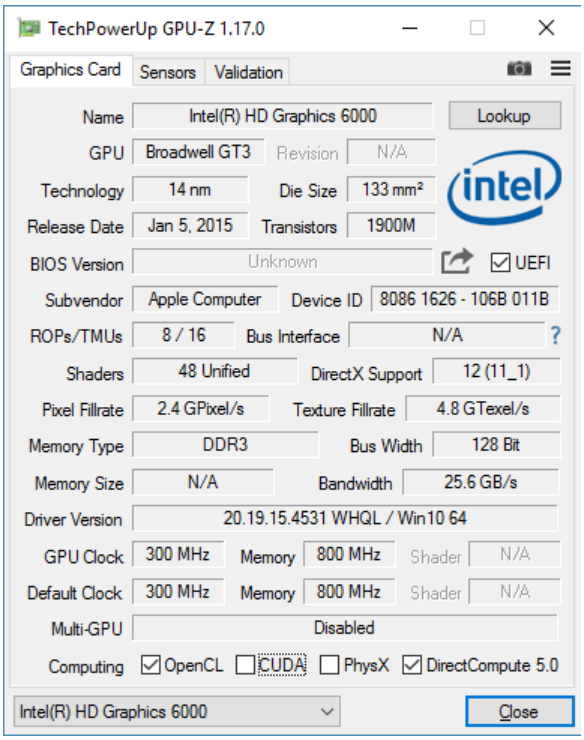
PROBLEM

C++ code was given which takes a PPM input image and performs an unsharp mask operation and writes the result to disk. The unsharp mask is implemented as a box blur whose result is subtracted from the input image, however the code is slow and could be made faster by using parallel compute and this is what is set out in this report where I describe how this application was made faster using OpenCL.

HARDWARE

As we will be comparing performance I decided to give an overview of the hardware which the timings were recorded on. On the CPU side the code was run on an i5 dual core processor with hyper threading. For the GPU, I am using an integrated chip the Intel HD 6000 rather than a dedicated graphics card. It was necessary to dispense this information as the performance increase shown may have been even larger had I run the kernels on a dedicated graphics card which has more horsepower at its disposal.





INTRODUCTION

OBJECTIVES

There are several objectives for this coursework. First is that the OpenCL version should be significantly faster than the original by factors not percent. Additionally, the optimized version should produce a more accurate result by not taking shortcuts as the given method does. Finally, utilities should be developed to help and test the program. This includes a Visualiser which renders the generated output in real time with key inputs to increase and decrease the blur radius. Additionally, there should be a Viewer program that loads and displays a PPM file quickly and accurately.

Development

OPENCL DEVELOPMENT

When porting the code over to OpenCL serval implementation decisions were taken.

C++ Bindings

When writing the OpenCL code the C++ bindings were used to reduce the complexity of the code written. It also enabled easy clean up and resource management thanks to RAII (Resource Acquisition is Initialization). The benefits of this approach can be seen in the given source code and in the Figure 1.

```
// RAII, with constructors & deconstructors ensuring no leaks
cl::CommandQueue queue(context, device, CL_QUEUE_PROFILING_ENABLE);

// vs

auto commandQueue = clCreateCommandQueue(context(), device(), CL_QUEUE_CONTEXT);

// what if I forget this function?
clReleaseCommandQueue(commandQueue);
```

Figure 1: C++ vs C example with OpenCL

Gaussian Blur

The box blur was dropped in favour of a full Gaussian blur that would be optimized later. Continuous use of a box blur is an optimisation technique for a Gaussian blur, where repeated use results in an accurate approximation of a Gaussian blur. The central limit theorem explains why this is the case. By using a Gaussian blur, I was able to produce a result that is more accurate than the original as a box blur is merely an approximation. Additionally, by using a single pass blur I was able to combine the sharpen and blur operation into one OpenCL kernel which was nice.

Image Objects

From the start of the project OpenCL image objects were used for added performance. Not only does this allow the program to utilise automatic clamping if we access a pixel outside of the image but it allows us to choose how we sample pixels in the image. This is a great functional feature but there is an added performance benefit as well as GPUs have specific hardware to enable fast image manipulation and by using it we will likely see a performance increase over traditional use of OpenCL buffers.

DEVELOPMENT

OPENCL OPTIMISATIONS

The first implementation was a 2D Gaussian blur single pass program. This was simple however it had several key performance faults. First was that blur weights were calculated on the fly and were not pre-calculated which resulted in a lot of redundant calculations. Second was that it was a 2D single pass blur which resulted in a significant number of additional operations as well.

Blur Weights

The first thing to solve was to pre-calculate the Gaussian blur weight values. This was done on the CPU end then stored in constant memory which could then be read from inside the kernel. This reduced a lot of redundant operations found inside the kernel and made it a lot faster but there was much more to do.

```
inline std::vector<float> gaussianFilter2D(const int radii)
{
    auto radius = (int)ceil(radii * 2.57);
    auto dev = 2 * radii * radii;
    auto sum = 0.0f;

    std::vector<float> kernel;
    kernel.reserve((int)pow(radius * 2 + 1, 2));

    for (int row = -radius; row <= radius; row++)
    {
        for (int col = -radius; col <= radius; col++)
        {
            const auto dist = sqrt(pow(col, 2) + pow(row, 2));
            const auto value = (float)(exp(-(pow(dist, 2.0f)) / dev)) / (PI * dev);
            kernel.push_back(value);
            sum += value;
        }
    }

    for (auto& v : kernel)
    {
        v /= sum;
    }

    return kernel;
}
```

Figure 2: C++ code to pre-calculate Gaussian weight values

Constants & Flags

Several parameters such as blur radius are passed as constants to the program via command line arguments when the OpenCL program is built. We also pass additional optimisation parameters to ensure all optimisations are enabled when running the kernel. It also saves many parameters having to be passed in via the Kernel object as can be seen in Figure 3.

```
cl::Program program = cl::getKernel(context, device, "kernels.cl", [&](auto& options) {
    options << " -Dalpha=" << 1.5;
    options << " -Dgamma=" << 0.0;
    options << " -Dbeta=" << -0.5;
    options << " -Dradius=" << (int)ceil(radius * 2.57);
    options << " -cl-fast-relaxed-math";
});
```

Figure 3: Passing certain arguments as constants to the OpenCL program when it is compiled.

1D Blur

With pre-calculated weight values, I looked at ways to optimise the Gaussian blur. Thankfully the 2D Gaussian blur can be calculated by combing 2 one dimensional blurs in the horizontal and vertical directions. This means the program would need to launch two kernels instead of one but the amount of work or computations to be performed would be reduced tenfold.

```
kernel void unsharp_mask_pass_one(
    read_only image2d_t input,
    write_only image2d_t output,
    constant float* hori
)
{
    const int x = get_global_id(0);
    const int y = get_global_id(1);

    float4 blurred = (float4)0.0f;

    for(int i = -radius, index = 0; i <= radius; ++i)
    {
        blurred += read_imagef(input, sampler, (int2)(x + i, y)) * (float4)hori[index++];
    }

    const float4 colour = radius != 0 ? blurred : read_imagef(input, sampler, (int2)(x, y));
    write_imagef(output, (int2)(x, y), colour);
}
```

Figure 4: First pass of the Unsharp OpenCL kernel which performs a horizontal blur

DEVELOPMENT

VIEWER DEVELOPMENT

To aid in the development of this optimised version a simple C# PPM viewer application was written to view PPM files. The provided source code for this assignment reads and writes a PPM file, a file type which isn't supported by most image viewing applications. So, I wrote this viewer myself which simply loads and shows a PPM. Once loaded you can switch the loaded images with the arrow keys on the keyboard. It was optimised to enable the fast loading of these files as in some of our test cases we outputted an 8k image which some other software I used previously for displaying PPM files didn't always work well.

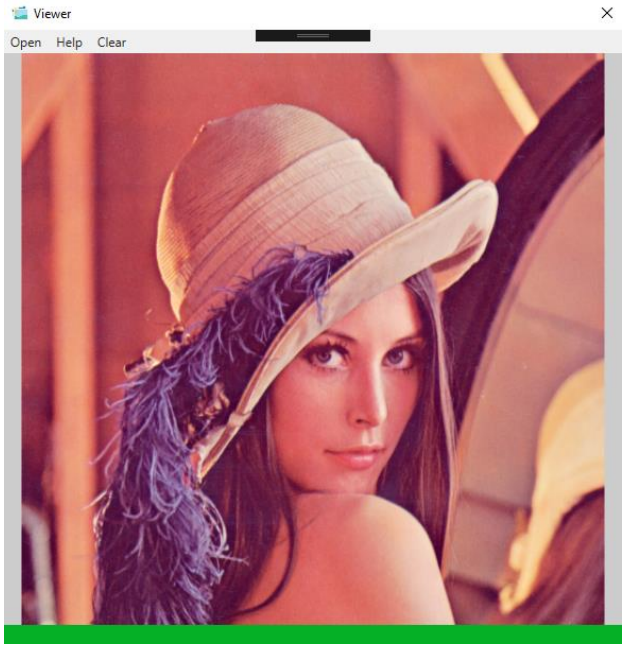


Figure 5: Viewer with a loaded image

Figure 6: Viewer without a loaded image

VISUALISER DEVELOPMENT

For extra marks a Visualiser was developed. This uses OpenCL to fill an OpenGL texture that is then rendered on screen in real time. This wasn't too complex as the standard Image2D objects were just replaced by the ImageGL type.

```
cl::Image2D inputImage(context, flags, format, w, h, 0, nullptr);
cl::Image2D passImage(context, flags, format, w, h);
cl::Image2D outImage(context, flags, format, w, h);

// Simple conversion

cl::ImageGL inputImage(context, flags, GL_TEXTURE_2D, 0, ID1);
cl::ImageGL passImage(context, flags, GL_TEXTURE_2D, 0, ID2);
cl::ImageGL outImage(context, flags, GL_TEXTURE_2D, 0, ID3);
```

Figure 7: Binding examples showing how easy it was to make the images OpenGL compatible

Add in some additional parameters when creating an OpenCL context and OpenCL is able and ready to be used in tandem with OpenGL. When creating the context and windowing code GLFW was used. We use it to open a window and setup the OpenGL context. I then subscribe to keyboard events which handle increasing the blur radius and switching the texture to display. All the parameters are put in the windows title so you know what the current value of the radius is and the name of the image that was loaded.

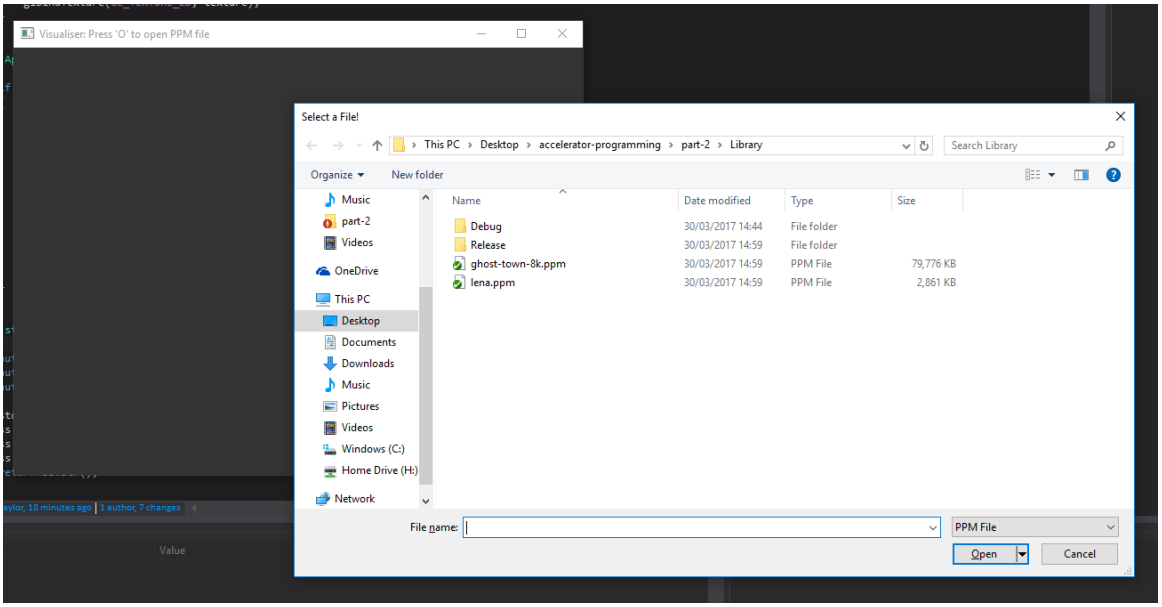


Figure 7: Dialog to open a PPM file

DEVELOPMENT

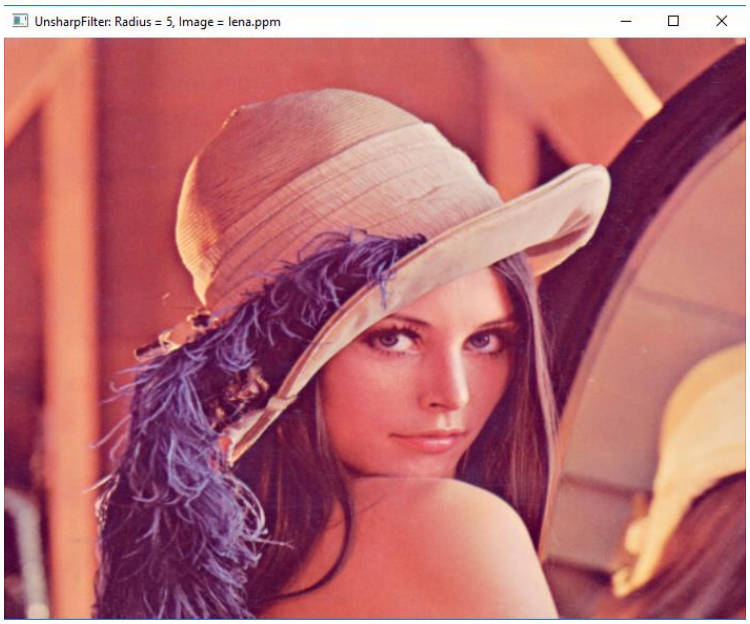


Figure 8: Press 1 to show the original image loaded

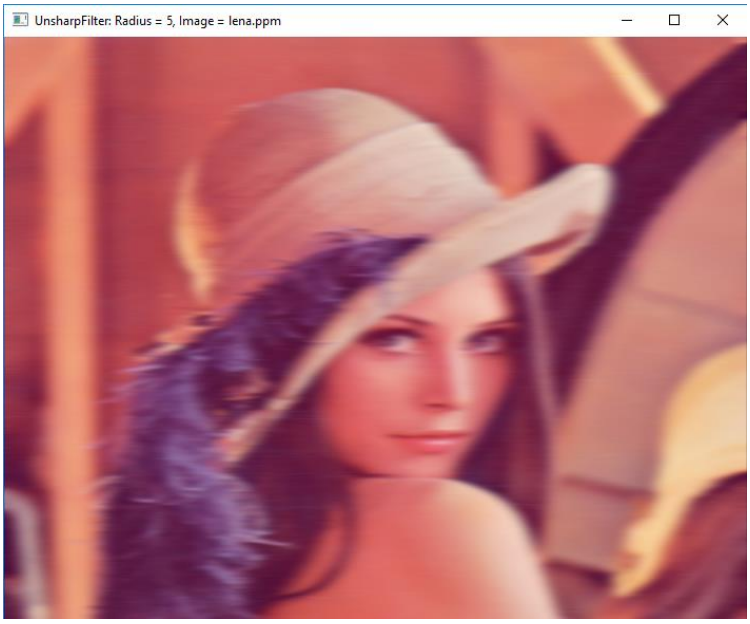


Figure 9: Press 2 to see the image after the first pass (Horizontal Blur)

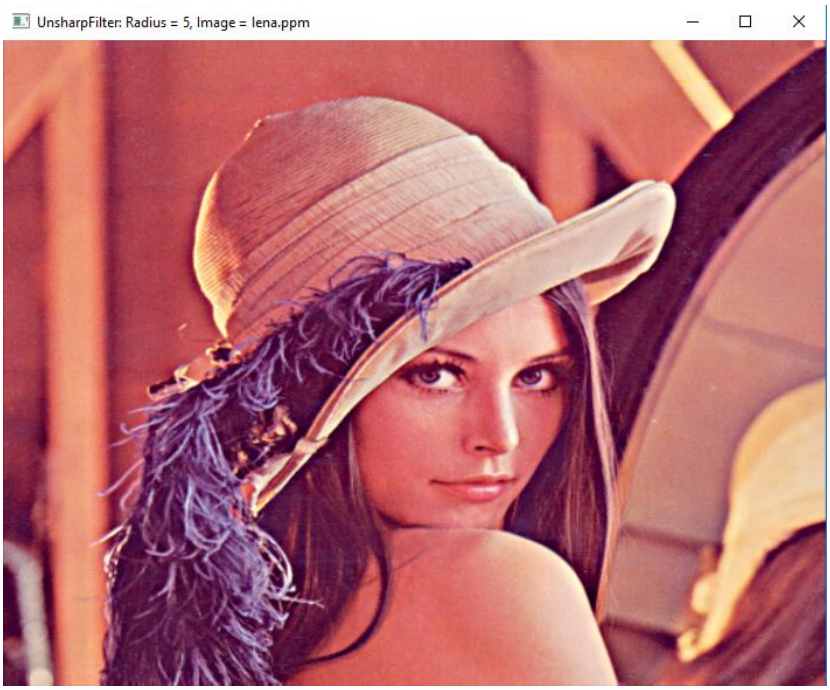


Figure 10: Press 3 to show the final image which is the sharpened image

Performance

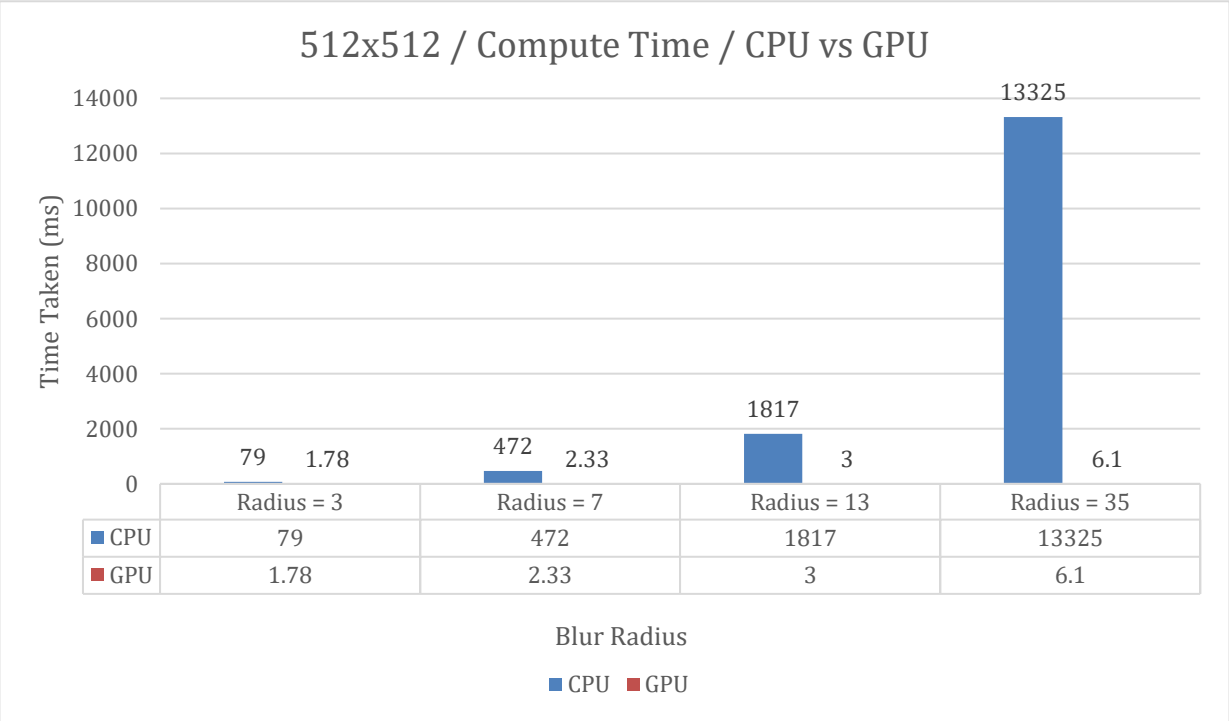
METHOD

For measuring performance the time to load and write the PPM file was taken out of measuring entirely as it was common to both programs and measuring this would have added noise to the performance measurements taken. *Compute Time* is just the time taken to calculate the output. It doesn't take into consideration any initialising costs or memory movement costs to and from the GPU for example. *Real time* is the total time required to put the output in memory ready to write to file. For the OpenCL version of the application this means that allocations for buffers and transferring data back from the GPU will be included as part of the timings.

LENA IMAGE

Compute Time

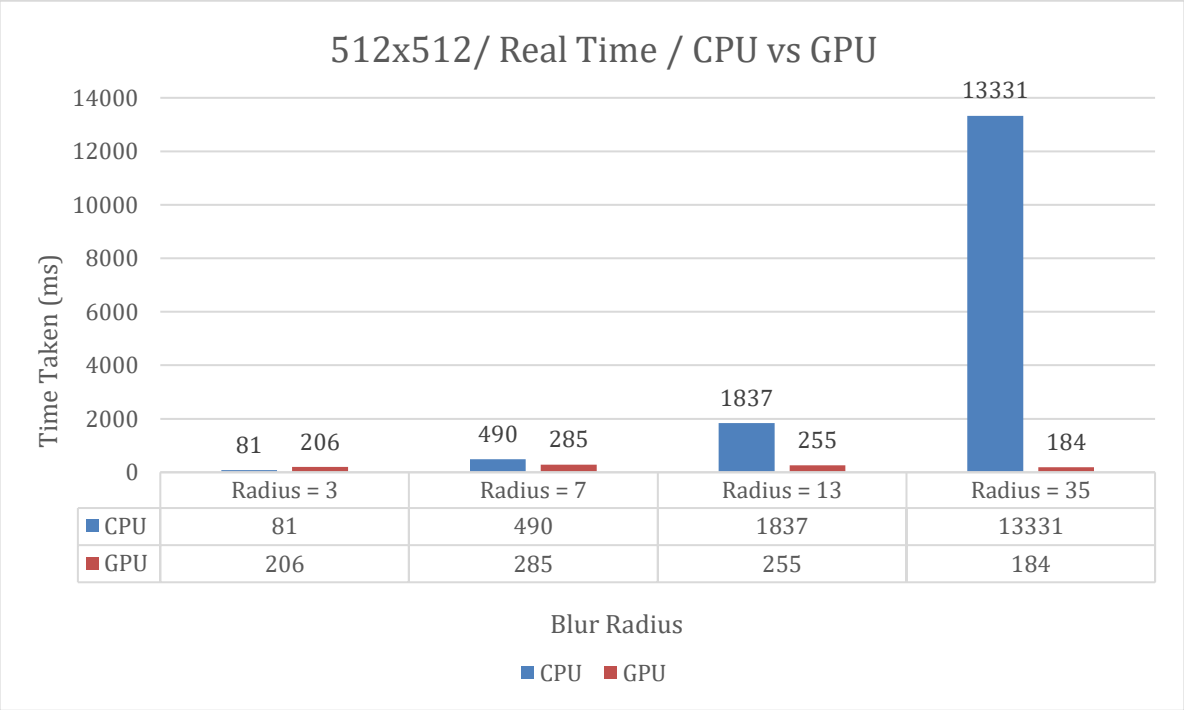
As we can see in the below graph the OpenCL version is significantly faster being 44x times faster with a blur radius of 3 and 2184x times faster using a blur radius of 35. However this benchmark only looked at the computation cost and didn't include memory transfers.



PERFORMANCE

Real Time

When we factor in memory transfer times and OpenCL program initialisation we see a different picture. Here the added cost of transferring memory stops the OpenCL version being faster with small blur sizes however it is still faster by a large margin with what seems to be any blur with a radius higher than 5. While the OpenCL version is only 1.7x times faster with a blur radius of 7 it does get significantly faster compared to the CPU version when dealing with larger blurs being 74x times faster than the given CPU version when a blur value of 35 is to be applied to the image.



Results

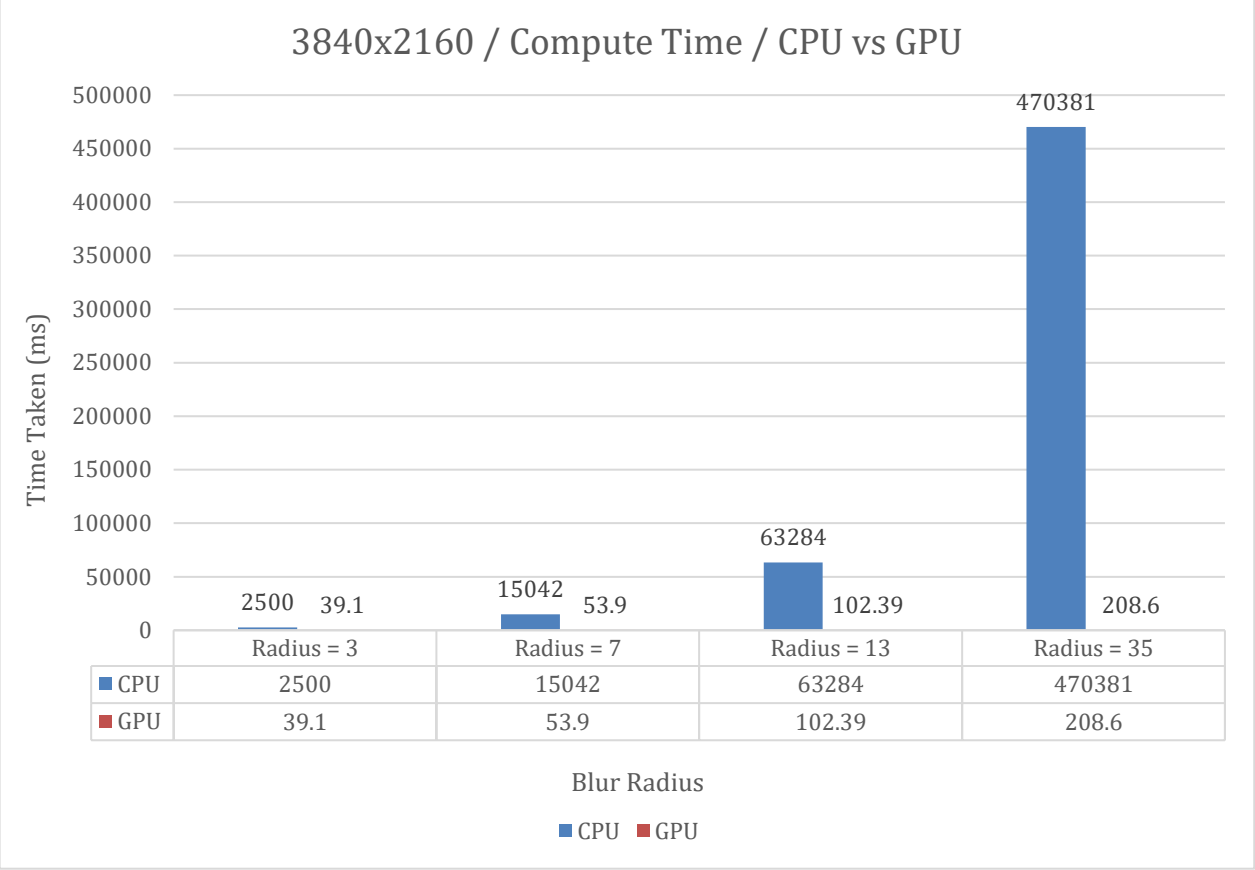
With the small given image, it is clear that with tiny blur radii values it will perform better than the OpenCL version. However that really is the only case where the given version wins. In all other cases the OpenCL version is faster by factors not percent. What is interesting given the OpenCL version is that using a blur radius of 13 gives better performance than using a blur radius of 7? What is going on here? Upon further investigation this was due to varying build times of the OpenCL kernel and varying initialising times for the buffers. It just so happens that when using a small image size that these factors become more noticeable and you will see this factor disappear in the following section where we deal with a larger image size.

PERFORMANCE

GHOST TOWN IMAGE

Compute Time

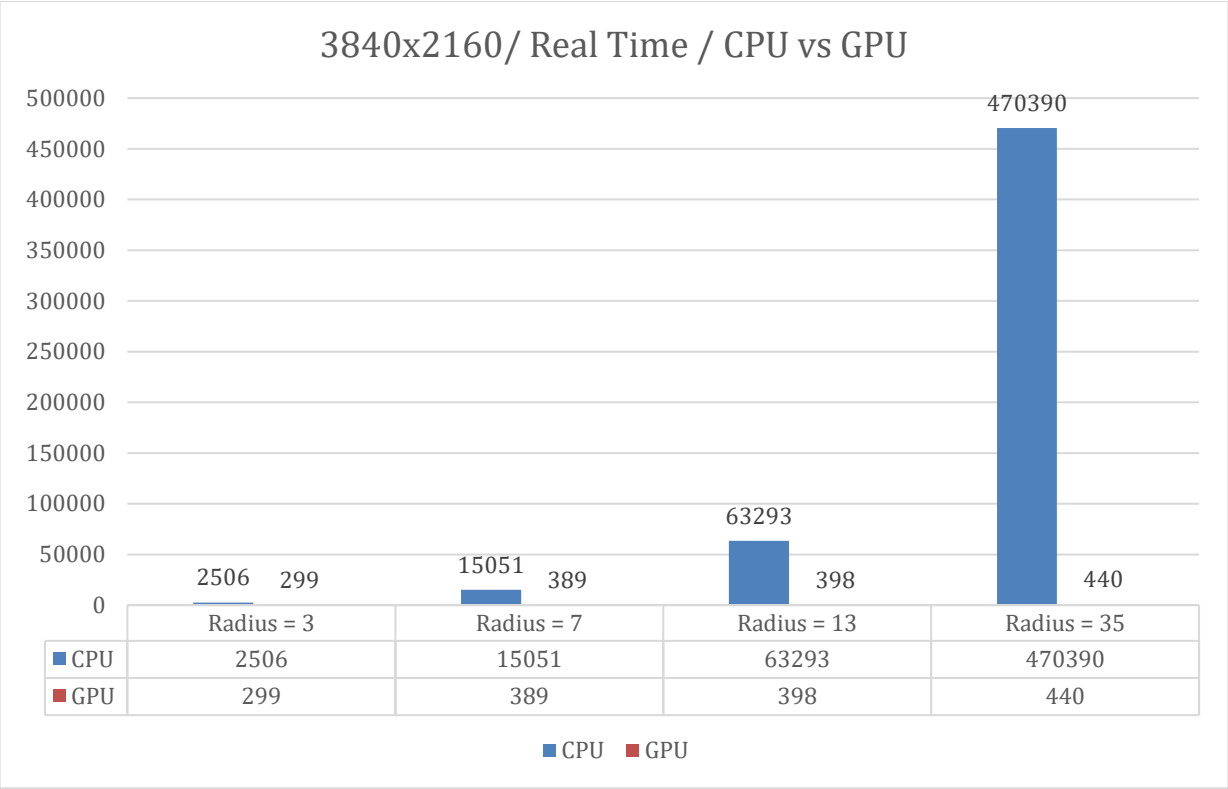
With this larger image, we can really see the advantages of the OpenCL version. I'm not going to quote factor increases because the chart explains it better than I can by quoting a single number. But it is abundantly clear that the OpenCL version is in a completely different league when dealing with larger image sizes.



Real Time

With the actual computation being so fast it is clear that the only bottleneck in the OpenGL version is actually the transfer of data to and from the GPU. However, with the larger image this bottleneck doesn't hamper the OpenCL version at all or at least it doesn't stop it from winning in every test set. With a blur radius of 3 and the larger image the OpenCL version is 8.3x times faster. With a blur radius of 35 its 1000x times faster.

PERFORMANCE



Results

So, with the larger image size the GPU wins all cases which is unsurprising. What is most intriguing is how performance scales linearly with the magnitude of the blur radius when it comes to the *compute time*. It doesn't increase exponentially like the given CPU version which is a great testament to the performance of the OpenCL version of the program.

CONCLUSION

Conclusion

This was a project that was bigger than expected and much can be said about it.

POSITIVES

There were many positives to the project. The objectives were met with a large increase in performance recorded as well as a more accurate output image. What this effectively demonstrates is that by using the GPU we need not take traditional shortcuts and sacrifice image quality when implementing traditional algorithms. Because not only is the image more accurate but it is generated faster making the traditional method redundant if you want a faster version with perfect results.

NEGATIVES

There were some negatives to this project as well. First a box blur was not implemented at all on the GPU which means a comparison was unable to be made. Additionally, the CPU was not sped up at all using threads and SIMD instructions meaning it's an unfair comparison between the two however the project does effective demonstrate how much faster operations can be when GPUs or parallel compute are leveraged.

SUMMARY

So to summarise, this project was a great success in my view with a large increase in performance and a more accurate result with various utilities developed to show the result and effects when varying blur radii. While more work could have been done by experimenting with a wider range of blur techniques and CPU multithreading the result in my opinion is still a great project which meets its objectives wholeheartedly and delivers an unprecedented speed up over the original code.