Word Count (excluding code and headers) [ 2825 ]

Module Code: **COMP10065**

Unsharp Mask Report

University of the west of scotland

GPGPU

B00308927

2018

Table of Contents

[1. Introduction 2](#_Toc532131595)

[2. Design approach 2](#_Toc532131596)

[2.1 OpenCL Implementation 2](#_Toc532131597)

[2.2 Kernel Parallelisation 7](#_Toc532131598)

[2.2.1 Blur Kernel 7](#_Toc532131599)

[2.2.2 Add Weighted Kernel 9](#_Toc532131600)

[2.3 Development Observations 11](#_Toc532131601)

[2.4 Blur Radius Comparison 12](#_Toc532131602)

[3. Benchmarking 13](#_Toc532131603)

[3.1 Serial vs Serial with MSCV++ Optimisations 13](#_Toc532131604)

[3.3.Serial vs CPU 14](#_Toc532131605)

[3.3.1 Impact of MSCV++ Optimisation on CPU 17](#_Toc532131606)

[3.2 Serial vs GPU 19](#_Toc532131607)

[3.2.1 Impact of MSCV++ On GPU 21](#_Toc532131608)

[3.4 CPU vs GPU 23](#_Toc532131609)

[3.5 Full Comparison 25](#_Toc532131610)

[4. Conclusion 26](#_Toc532131611)

# 1. Introduction

This report lays out the steps involved in parallelising an unsharp mask operation on an image. **Parallelising** refers to redesigning and reimplementing existing code that originally ran on a single thread for use in multiple threads. This enables code to take advantage of the multiple cores in modern processing devices such as CPUs and GPUs with the expected outcome of greater processing speed.

An **Unsharp Mask** is an image sharpening technique that predates image processing on computers. Its purpose is to improve the apparent resolution of an image, it does not actually give the image a larger resolution but manipulates it to make it appear as such. This is achieved by subtracting a blurred version of the image from the original. The result is that the image appears to have sharper edges and it looks more defined because the noise of the blurry image has been removed.

**Serial** refers to a program that performs one instruction at a time.

# 2. Design approach

In order to parallelise the unsharp mask the approach used was to parallelise the individual serial steps. This made sense because it would keep the code close to the logic of the original and allow a user to see the similarity between the two.

## 2.1 OpenCL Implementation

The API chosen was OpenCL. OpenCL is open source which makes easy to work with as opposed to APIs like Cuda or Thrust which are more confined. OpenCL was also a good choice because it will run with any OpenCL compatible device whether it is a CPU, GPU or other unlike the Cuda and Thrust which will commonly not run on non-NVIDIA graphics hardware and is not designed for use on a CPU on all but the oldest versions of Cuda. The specific version of OpenCL being used however is OpenCL version 1.2. This version is now quite old but was suitable because of its availability on the system that the program was benchmarked on.

The first step was to setup OpenCL. That required a bit of preamble in comparison to the traditional serial way. The main steps consisted of:

1. Setting up appropriate buffers for OpenCL to gather data to be processed in kernels into. The OpenCL and vectors were collected together for usability. The two blur buffers will be alternated during the blurring process. They will be alternated in the blurring process instead of using 3, this saves processing time.

struct Buffers

{

std::vector<unsigned char> h\_original\_image, h\_blurred\_image, h\_sharpened\_image;

cl::Buffer d\_original\_image, d\_sharpened\_image;

cl::Buffer d\_blurred\_image1, d\_blurred\_image2;

}buffers;

Figure 1 - Buffer setup.

1. Detect which device would be used to run the kernels. The code implemented will give preference to NVIDIA GPUs if one is available. If a GPU cannot be found it will default to a CPU. OpenCL leaves the device choice to the implementation, so function was implemented to handle it.

// Device Selection - Prefers GPUs

int deviceSelection;

// Defaults to CPU in case there are no GPUs.

deviceSelection = CL\_DEVICE\_TYPE\_CPU;

for (std::vector<cl::Platform>::iterator plat = platforms.begin();

plat != platforms.end(); plat++)

{

std::string s;

plat->getInfo(CL\_PLATFORM\_VENDOR, &s);

if (s == "NVIDIA Corporation")

{

deviceSelection = CL\_DEVICE\_TYPE\_GPU;

std::cout << "NVIDIA GPU Selected." << std::endl;

break;

}

else if(s == "Advanced Micro Devices, Inc." && plat == platforms.end())

{

deviceSelection = CL\_DEVICE\_TYPE\_GPU;

std::cout << "AMD GPU Selected." << std::endl;

break;

}

}

if (deviceSelection == CL\_DEVICE\_TYPE\_CPU)

{

std::cout << "CPU Selected." << std::endl;

}

Figure 2 - Device selection.

1. The kernels would then be loaded using a command queue. Rich error checking was implemented for the kernels. The parallel code is wrapped in a try -> catch block. This proved extremely useful and sped up development time as the error checking in kernel was detailed. The error checking will respond when the program.build() call is made and compiling the kernel is initiated.

// Get the command queue

cl::CommandQueue queue(context);

// Create Blur Kernel – Load Kernel Source

program = cl::Program(context, util::loadProgram("../../blur.cl"));

// Build the cl file to check for errors.

program.build(context.getInfo<CL\_CONTEXT\_DEVICES>());

// Create the kernel

auto blur = cl::make\_kernel<cl::Buffer,

cl::Buffer,

const int,

const unsigned,

const unsigned,

const unsigned>(program, "blur");

// Create Add\_Weighted Kernel – Load Kernel Source

program = cl::Program(context, util::loadProgram("../../add\_weighted.cl"));

// Build the cl file to check for errors.

program.build(context.getInfo<CL\_CONTEXT\_DEVICES>());

// Create the kernel

auto add\_weighted = cl::make\_kernel<cl::Buffer,

cl::Buffer,

const float,

cl::Buffer,

const float,

const float,

const unsigned int

const unsigned int,

const unsigned int>(program, "add\_weighted");

Figure 3 - Kernel setup.

The catching block is quite dense and will catch general OpenCL errors as well as kernel errors.

catch (cl::Error err ) {

if (err.err() == CL\_BUILD\_PROGRAM\_FAILURE)

{

for (cl::Device dev : deviceList)

{

// Check the build status

cl\_build\_status status = program.getBuildInfo<CL\_PROGRAM\_BUILD\_STATUS>(dev);

if (status != CL\_BUILD\_ERROR)

continue;

// Get the build log

std::string name = dev.getInfo<CL\_DEVICE\_NAME>();

std::string buildlog = program.getBuildInfo<CL\_PROGRAM\_BUILD\_LOG>(dev);

std::cerr << "Build log for " << name << ":" << std::endl

<< buildlog << std::endl;

}

}

Figure 4 - Kernel error catching.

If the error is a general OpenCL error the rest of the block handles it.

else

{

std::cout << "Exception\n";

std::cerr

<< "ERROR: "

<< err.what()

<< "("

<< err\_code(err.err())

<< ")"

<< std::endl;

}

}

Figure 5 - Generic OpenCL error catching.

1. Once the kernels were successfully built, the device buffers would be assigned the same information that is processed on the serial implementation. The original image is assigned to the corresponding device buffer and space is allocated for the blurred and sharpened image on the device buffers with a combined call. This simultaneously handles the transfer of data and memory allocation for the original image buffer and performs the function of memory allocation for the blurred and sharpened image buffers.

Note that the original image buffer is marked as Read Only as the information in this buffer will only ever be read to be manipulated when the blurred image is created in the blur kernel and when it is applied to the sharpened image in combination with the blurred image in the add\_weighted kernel. The blurred and sharpened images however are written to, the blurred images are written to inside the blur kernel as this is the kernel’s output buffer and the sharpened image is written to inside the add\_weighted kernel as its output buffer.

The last parameter is particularly crucial. This parameter controls whether the buffer will use Host Pointers. This is when the device will use pointers to data on the host instead of copying the information to the device. The memory specified by the iterators must be contiguous which in this case it is because an array is being used as the buffer. This one parameter being set to true causes a noticeable speed increase as the device does not need to transfer memory. In testing it was also noted that this effect was true for the CPU as well. This is confusing as the assumption is that the CPU is the host device but when using multiple threads information is still transferred to them away from the single thread which takes up valuable time. So, this factor should also be considered when using CPUs as there is still a noticeable speed increase. The sharp and original image buffers are setup outside of the timed code as this is the case in the serial version where they are externally loaded and then passed in.

//Assign buffers

buffers.d\_sharpened\_image = cl::Buffer(context,

buffers.h\_sharpened\_image.begin(),

buffers.h\_sharpened\_image.end(),

CL\_MEM\_READ\_WRITE, true);

buffers.d\_original\_image = cl::Buffer(context,

buffers.h\_original\_image.begin(),

buffers.h\_original\_image.end(),

CL\_MEM\_READ\_ONLY, true);

buffers.d\_blurred\_image1 = cl::Buffer(context,

buffers.h\_blurred\_image.begin(),

buffers.h\_blurred\_image.end(),

CL\_MEM\_READ\_WRITE, true);

buffers.d\_blurred\_image2 = cl::Buffer(context,

buffers.h\_blurred\_image.begin(),

buffers.h\_blurred\_image.end(),

CL\_MEM\_READ\_WRITE, true);

Figure 6 - Buffer assignment.

1. Once the kernels and buffers are setup all that needs to be done is to pass the arguments to them. The Blur kernel runs 3 times, producing same effect as the triple blurring in the serial version of the program and importantly it stays with the C++ API. In Figure 7, we can see the buffers alternating between the kernel calls. This is a small way of saving processing time.

// Execute Blur Kernels

blur(

cl::EnqueueArgs(

queue,

cl::NDRange(img.w, img.h)),

buffers.d\_blurred\_image1,

buffers.d\_original\_image,

blur\_radius,

img.w,

img.h,

img.nchannels);

blur(

cl::EnqueueArgs(

queue,

cl::NDRange(img.w, img.h)),

buffers.d\_blurred\_image2,

buffers.d\_blurred\_image1,

blur\_radius,

img.w,

img.h,

img.nchannels);

blur(

cl::EnqueueArgs(

queue,

cl::NDRange(img.w, img.h)),

buffers.d\_blurred\_image1,

buffers.d\_blurred\_image2,

blur\_radius,

img.w,

img.h,

img.nchannels);

// Execute Add\_Weigted Kernel

add\_weighted(

cl::EnqueueArgs(

queue,

cl::NDRange(img.w, img.h)),

buffers.d\_sharpened\_image,

buffers.d\_original\_image,

imgval.alpha,

buffers.d\_blurred\_image1,

imgval.beta,

imgval.gamma,

img.w,

img.h,

img.nchannels);

Figure 7 - Kernel execution.

1. Finally, copy the data back to the host for use in writing the image. The implementation approach used allows the device buffers to be worked on by both kernels one after the other without copying them back to the host. If this was required, the use of host pointers would still speed up the process. The data is copied back to the host buffer with a single call to **cl::copy**. This method of creating the kernel and enqueuing arguments takes advantage of the OpenCL C++ API. This can be seen in Figure 8.

cl::copy(queue, buffers.d\_sharpened\_image, buffers.h\_sharpened\_image.begin(), buffers.h\_sharpened\_image.end());

Figure 8 - Device data being copied back to host.

The data on the host is then used to write the output image.

## 2.2 Kernel Parallelisation

### 2.2.1 Blur Kernel

The kernels are where the real parallelisation is done. The level of parallelisation could be defined as naïve parallelisation as it makes use of global memory in the device to divide the tasks between threads. The task division is defined by the **NDRange** that is passed into the kernel – this becomes the global id (**index**) of each thread to hand a task to. This high-level parameter has a strong effect on the efficiency of the kernel as it is defining how much work is going to be given to a single thread, the capacity of which will vary by device. The NULL value preceding the call to **NDRange** must be left NULL in the version of OpenCL being used here but is planned to control an offset to the global ids to allow them to begin operating at a value other than 0. The NULL value succeeding the call to NDRange controls the local id of each thread in a work group. This can be used to create an independent work group wherein the threads can “cache” data which speed up operation. The kernel takes a 2-dimensional approach with **NDRange** and uses the image width and image height as the global ids in order to remove the nested loop inside the serial code. This means that one task is equal to one pixel of the image which a thread will process. The parallelisation process is shown in Figure 9, Figure 10 and Figure 11.

void pixel\_average( unsigned char \*out,

const unsigned char \*in,

const int x, const int y, const int blur\_radius,

const unsigned w, const unsigned h, const unsigned nchannels)

{

float red\_total = 0, green\_total = 0, blue\_total = 0;

for (int j = y-blur\_radius+1; j < y+blur\_radius; ++j) {

for (int i = x-blur\_radius+1; i < x+blur\_radius; ++i) {

const unsigned r\_i = i < 0 ? 0 : i >= w ? w-1 : i;

const unsigned r\_j = j < 0 ? 0 : j >= h ? h-1 : j;

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];

}

}

const unsigned nsamples = (blur\_radius\*2-1) \* (blur\_radius\*2-1);

unsigned byte\_offset = (y\*w+x)\*nchannels;

out[byte\_offset+0] = red\_total/nsamples;

out[byte\_offset+1] = green\_total/nsamples;

out[byte\_offset+2] = blue\_total/nsamples;

}

void blur(unsigned char \*out, const unsigned char \*in,

const int blur\_radius,

const unsigned w, const unsigned h, const unsigned nchannels)

{

for (int y = 0; y < h; ++y) {

for (int x = 0; x < w; ++x) {

pixel\_average(out,in,x,y,blur\_radius,w,h,nchannels);

}

}

}

Figure 9 - Serial blur class.

The blur method was prime for parallelisation as it contained a nested loop. The pixel average method would need to stay serial as each thread would still need to perform the averaging process on the pixels around it to produce an accurate average. Attempts at parallelising the pixel average method were made but the image would emerge sharpened and brightened which was not the intended effect.

The parallelising of the blur method was solved by first creating a kernel that mirrored the process in the blur.hpp file, called blur.cl. It was then reparametrized to handle kernel type arguments in the method definitions and made to fetch the global ids to be passed into the pixel average method.

void pixel\_average(

\_\_global unsigned char \*out,

\_\_global const unsigned char \*in,

const int x,

const int y,

const int blur\_radius,

const unsigned w,

const unsigned h,

const unsigned nchannels)

{

float red\_total = 0, green\_total = 0, blue\_total = 0;

for (int j = y - blur\_radius + 1; j < y + blur\_radius; ++j) {

for (int i = x - blur\_radius + 1; i < x + blur\_radius; ++i) {

const unsigned r\_i = i < 0 ? 0 : i >= w ? w - 1 : i;

const unsigned r\_j = j < 0 ? 0 : j >= h ? h - 1 : j;

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];

}

}

const unsigned nsamples = (blur\_radius \* 2 - 1) \* (blur\_radius \* 2 - 1);

unsigned byte\_offset = (y\*w + x)\*nchannels;

out[byte\_offset + 0] = red\_total / nsamples;

out[byte\_offset + 1] = green\_total / nsamples;

out[byte\_offset + 2] = blue\_total / 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)

{

~~for (int y = 0; y < h; ++y) {~~

~~for (int x = 0; x < w; ++x) {~~

~~pixel\_average(out, in, x, y, blur\_radius, w, h, nchannels);~~

~~}  
 }~~

int x = get\_global\_id(0);

int y = get\_global\_id(1);

pixel\_average(out, in, x, y, blur\_radius, w, h, nchannels);

}

Figure 10 - Blur class transitioning to kernel.

The x and y parameters proved a convenient place to assign the thread ids to since they were previously governing which pixel was being processed by counting from zero up to the width and height limit of the image.

void pixel\_average(

\_\_global unsigned char \*out,

\_\_global const unsigned char \*in,

const int x,

const int y,

const int blur\_radius,

const unsigned w,

const unsigned h,

const unsigned nchannels)

{

float red\_total = 0, green\_total = 0, blue\_total = 0;

for (int j = y - blur\_radius + 1; j < y + blur\_radius; ++j) {

for (int i = x - blur\_radius + 1; i < x + blur\_radius; ++i) {

const unsigned r\_i = i < 0 ? 0 : i >= w ? w - 1 : i;

const unsigned r\_j = j < 0 ? 0 : j >= h ? h - 1 : j;

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];

}

}

const unsigned nsamples = (blur\_radius \* 2 - 1) \* (blur\_radius \* 2 - 1);

unsigned byte\_offset = (y\*w + x)\*nchannels;

out[byte\_offset + 0] = red\_total / nsamples;

out[byte\_offset + 1] = green\_total / nsamples;

out[byte\_offset + 2] = blue\_total / 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);

pixel\_average(out, in, x, y, blur\_radius, w, h, nchannels);

}

Figure 11 - Final blur kernel

### 2.2.2 Add Weighted Kernel

The next stage was to parallelise the add\_weighted class. This would be done using the same approach as was used on the blur class. The process in seen in Figure 12, Figure 13 and Figure 14.

template <typename T>

void add\_weighted(unsigned char \*out,

const unsigned char \*in1, const T alpha,

const unsigned char \*in2, const T beta, const T gamma,

const unsigned w, const unsigned h, const unsigned nchannels)

{

for (int y = 0; y < h; ++y) {

for (int x = 0; x < w; ++x) {

unsigned byte\_offset = (y\*w+x)\*nchannels;

T tmp = in1[byte\_offset+0] \* alpha + in2[byte\_offset+0] \* beta + gamma;

out[byte\_offset+0] = tmp < 0 ? 0 : tmp > UCHAR\_MAX ? UCHAR\_MAX : tmp;

tmp = in1[byte\_offset+1] \* alpha + in2[byte\_offset+1] \* beta + gamma;

out[byte\_offset+1] = tmp < 0 ? 0 : tmp > UCHAR\_MAX ? UCHAR\_MAX : tmp;

tmp = in1[byte\_offset+2] \* alpha + in2[byte\_offset+2] \* beta + gamma;

out[byte\_offset+2] = tmp < 0 ? 0 : tmp > UCHAR\_MAX ? UCHAR\_MAX : tmp;

}

}

}

Figure 12 - Serial add\_weighted class.

There was however a caveat this time, the original class contained a template which OpenCL version 1.2 does not support in kernels so this had to be removed and replaced with float values. In this scenario’s case float values were enough to allow the program to function, but in future versions of OpenCL where templates are supported this kernel can be updated to allow the Generics to return.

As with the original blur class the parameters of the method were changed to support kernel syntax and the nested for loop was removed and global ids are now responsible for assigning individual threads a pixel from the original image to add the weighted pixel to.

~~template <typename T>~~

~~void add\_weighted(unsigned char \*out,~~

~~const unsigned char \*in1, const T alpha,~~

~~const unsigned char \*in2, const T beta, const T gamma,~~

~~const unsigned w, const unsigned h, const unsigned nchannels)~~

\_\_kernel void add\_weighted(

\_\_global unsigned char \*out,

\_\_global const unsigned char \*in1,

const float alpha,

\_\_global const unsigned char \*in2,

const float beta,

const float gamma,

const unsigned w,

const unsigned h,

const unsigned nchannels)

{

~~for (int y = 0; y < h; ++y) {~~

~~for (int x = 0; x < w; ++x) {~~

~~unsigned byte\_offset = (y\*w+x)\*nchannels;~~

~~T tmp = in1[byte\_offset+0] \* alpha + in2[byte\_offset+0] \* beta + gamma;~~

~~out[byte\_offset+0] = tmp < 0 ? 0 : tmp > UCHAR\_MAX ? UCHAR\_MAX : tmp;~~

~~tmp = in1[byte\_offset+1] \* alpha + in2[byte\_offset+1] \* beta + gamma;~~

~~out[byte\_offset+1] = tmp < 0 ? 0 : tmp > UCHAR\_MAX ? UCHAR\_MAX : tmp;~~

~~tmp = in1[byte\_offset+2] \* alpha + in2[byte\_offset+2] \* beta + gamma;~~

~~out[byte\_offset+2] = tmp < 0 ? 0 : tmp > UCHAR\_MAX ? UCHAR\_MAX : tmp;~~

~~}~~

~~}~~

int x = get\_global\_id(0);

int y = get\_global\_id(1);

unsigned byte\_offset = (y\*w + x)\*nchannels;

float tmp = in1[byte\_offset + 0] \* alpha + in2[byte\_offset + 0] \* beta + gamma;

out[byte\_offset + 0] = tmp < 0 ? 0 : tmp > UCHAR\_MAX ? UCHAR\_MAX : tmp;

tmp = in1[byte\_offset + 1] \* alpha + in2[byte\_offset + 1] \* beta + gamma;

out[byte\_offset + 1] = tmp < 0 ? 0 : tmp > UCHAR\_MAX ? UCHAR\_MAX : tmp;

tmp = in1[byte\_offset + 2] \* alpha + in2[byte\_offset + 2] \* beta + gamma;

out[byte\_offset + 2] = tmp < 0 ? 0 : tmp > UCHAR\_MAX ? UCHAR\_MAX : tmp;

}

Figure 13 - add\_weighted class transitioning to kernel.

\_\_kernel void add\_weighted(

\_\_global unsigned char \*out,

\_\_global const unsigned char \*in1,

const float alpha,

\_\_global const unsigned char \*in2,

const float beta,

const float gamma,

const unsigned w,

const unsigned h,

const unsigned nchannels)

{

int x = get\_global\_id(0);

int y = get\_global\_id(1);

unsigned byte\_offset = (y\*w + x)\*nchannels;

float tmp = in1[byte\_offset + 0] \* alpha + in2[byte\_offset + 0] \* beta + gamma;

out[byte\_offset + 0] = tmp < 0 ? 0 : tmp > UCHAR\_MAX ? UCHAR\_MAX : tmp;

tmp = in1[byte\_offset + 1] \* alpha + in2[byte\_offset + 1] \* beta + gamma;

out[byte\_offset + 1] = tmp < 0 ? 0 : tmp > UCHAR\_MAX ? UCHAR\_MAX : tmp;

tmp = in1[byte\_offset + 2] \* alpha + in2[byte\_offset + 2] \* beta + gamma;

out[byte\_offset + 2] = tmp < 0 ? 0 : tmp > UCHAR\_MAX ? UCHAR\_MAX : tmp;

}

Figure 14 - Final add\_weighted kernel.

The combined parallelisation of these two classes into kernels results in a large increase in speed over the serial implementation when running the program. Precise statistics on this are discussed in the [Benchmarking](#_Benchmarking) section.

## 2.3 Development Observations

During testing it was apparent that MSVC++ optimization settings could be manipulated to increase the speed of the program. In this project, this is handled automatically for the user. When the project is generated using the CMake file provided, optimization settings automatically set. These settings are enabled for Release mode. It was observed that MSCV++ optimization settings improved the serial implementation significantly, however the parallel implementation didn’t seem affected.

The use of constants in the kernels was explored. The \_\_constant would be used in the kernel for values that would remain the same throughout the kernel execution. There are additional restrictions on this address space qualifier however, it may only be used on a pointer type that will remain constant and it must be initialised. The implementation route chosen was incompatible with these restrictions, so it was excluded from use. \_\_constant values have a capacity that is dependent on the device, this may slow operation of a kernel if used on a device that had a small cache.

Local memory was also considered but not implemented due to its complexity. The kernels would have needed redesigned to accommodate the local memory accesses to the adjacent pixels so that the blurring operation wouldn’t be entirely refreshed on every iteration and within the blur radius the information would be maintained, enabling other pixels could access to it. Despite this short coming, code that pertains to its implementation has been kept within the project and marked with the letter **L** to identify it as local memory implementation.

The Blur operation takes place 3 times because it is attempting to mimic the Gaussian blur image processing technique where an image is blurred using a function which generates pixel values in a grid that correspond to the values of the Gaussian curve as seen in Figure 15. This grid “sweeps” over the image in a serial design and blurs is according those values and creates a very uniform blurring effect. This is then used as a weight to subtract from each pixel in the same way as this project behaves. The reason Gaussian blurring is now so popular is because by its very nature (a grid where multiple calculations will happen at the same time) it is very parallelisable. There would then be no need for the sweep as a device could handle the entire problem size all at once. In contrast, this project in its initial state was a serial unsharp mask, which is a digital adaptation of a photo development technique, not something natively synergetic with parallelism.

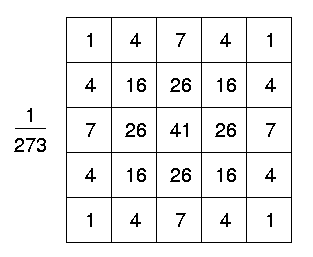


Figure 15 - Gaussian grid.

It should be noted that the image being processed in this project was a 24-bit image. It contained 3 channels Red, Green, Blue (RGB) for each pixel. This could be easily altered to 4 channels which would allow the image to have an alpha channel, turning the image into a 32-bit Red, Green, Blue, Alpha (RGBA) image. This would however, mean allocating memory for an additional 8 bits per pixel which could negatively impact performance. For this project it wasn’t included as the image is opaque and thus an alpha channel is not needed and the negative impact on performance is undesirable.

## 2.4 Blur Radius Comparison

Below is a comparison of the processed image after having a blur radius, of 3, 6 and 9 applied as seen in Figure 16.



Figure 16 - Blur radius comparison.   
3 -> 6 -> 9

As the blur radius increases, the wheel becomes more focused and defined.

# 3. Benchmarking

When benchmarking both the serial and parallel processes, an average was taken over 6 iterations of processing. The first two were ignored as they would return highly varied values. The number of iterations and the amount to ignore are controlled with variable values.

The kernels were timed together as during testing they were timed separately and would generate incorrect results due to asynchronous processing.

For detailed tables please refer to the [**Benchmarking Excel**](Benchmarking.xlsx) document that accompanies this report.

## 3.1 Serial vs Serial with MSCV++ Optimisations

The serial process benefitted greatly from MSCV++ optimisation. It improved the performance consistently by a factor of 4. This is illustrated in Figure 17, Figure 18 and Figure 19.

Figure 17 - Serial vs Serial Execution w/MSCV++ Optimisations on 720 x 576 pixels.

Figure 18 - Serial vs Serial Execution w/MSCV++ Optimisations on 3000 x 2000 pixels.

Figure 19 - Serial vs Serial Execution w/MSCV++ Optimisations on 3840 x 2160 pixels.

## 3.3.Serial vs CPU

The CPU parallel process was faster than the serial process. The CPU process did not experience any significant difference in speed when MSCV++ optimisation was applied but was still faster than the serial process by a factor of 2 with MSCV++ optimisation and a factor or 8 without. Seen in Figure 20 - Figure 25.

Figure 20 - Serial Vs Parallel CPU Execution 720 x 576 pixels.

Figure 21 - Serial Vs Parallel CPU Execution w/MSCV++ Optimisations 720 x 576 pixels.

Figure 22 - Serial Vs Parallel CPU Execution 3000 x 2000 pixels.

Figure 23 - Serial Vs Parallel CPU Execution w/MSCV++ Optimisations 3000 x 2000 pixels.

Figure 24 - Serial Vs Parallel CPU Execution 3840 x 2160 pixels.

Figure 25 - Serial Vs Parallel CPU Execution w/MSCV++ Optimisations 3840 x 2160 pixels.

### 3.3.1 Impact of MSCV++ Optimisation on CPU

When MSCV++ optimisation was enabled, the CPU didn’t experience significant difference in speed as seen in Figure 26, Figure 27 and Figure 28.

Figure 26 - Parallel CPU vs Parallel CPU Execution w/MSCV++ Optimisations 720 x 576 pixels.

Figure 27 - Parallel CPU vs Parallel CPU Execution w/MSCV++ Optimisations 3000 x 2000 pixels.

Figure 28 - Parallel CPU vs Parallel CPU Execution w/MSCV++ Optimisations 3840 x 2160 pixels.

## 3.2 Serial vs GPU

The parallel GPU process was faster than the serial process, this advantage expanded as the work size grew. The serial process experiences a trend of exponential growth as the blur size is increased while the parallel process increases at a much slower rate. Eventually it may become exponential as the capacity of a thread is reached but this problem isn’t intensive enough to reach that point. In the values tested the parallel process on the GPU was up to 130 times faster than the serial process without MSCV++ optimisation and up to 31 times faster with it. This is illustrated in Figure 29 - Figure 34.

Figure 29 - Serial Vs Parallel GPU Execution 720 x 576 pixels.

Figure 30 - Serial Vs Parallel GPU Execution w/MSCV++ Optimisations 720 x 576 pixels.

Figure 31 - Serial Vs Parallel GPU Optimisations 3000 x 2000 pixels.

Figure 32 - Serial Vs Parallel GPU Execution w/MSCV++ Optimisations 3000 x 2000 pixels.

Figure 33 - Serial Vs Parallel GPU Execution 3840 x 2160 pixels.

Figure 34 - Serial Vs Parallel GPU Execution w/MSCV++ Optimisations 3840 x 2160 pixels.

### 3.2.1 Impact of MSCV++ On GPU

Much like the CPU, the GPU doesn’t benefit from MSCV++ optimisation as seen in Figure 35, Figure 36 and Figure 37.

Figure 35 - Parallel GPU vs Parallel GPU Execution w/MSCV++ Optimisations 720 x 576 pixels.

Figure 36 - Parallel GPU vs Parallel GPU Execution w/MSCV++ Optimisations 3000 x 2000 pixels.

Figure 37 - Parallel GPU vs Parallel GPU Execution w/MSCV++ Optimisations 3840 x 2160 pixels.

## 3.4 CPU vs GPU

The GPU was faster than the CPU in this scenario. This becomes more apparent as the blur size increases, much like the serial process the CPU experiences an exponential growth when measured against the GPU. In the tested problem size the GPU was faster than the CPU by up to a factor of 16. This is illustrated across Figure 38 - Figure 43.

Figure 38 - Parallel CPU vs Parallel GPU - 720 x 576 pixels.

Figure 39 - Parallel CPU vs Parallel GPU Both w/ Optimisations - 720 x 576 pixels.

Figure 40 - Parallel CPU vs Parallel GPU - 3000 x 2000 pixels.

Figure 41 - Parallel CPU vs Parallel GPU Both w/ Optimisations - 3000 x 2000 pixels.

Figure 42 - Parallel CPU vs Parallel GPU - 3840 x 2160 pixels.

Figure 43 - Parallel CPU vs Parallel GPU Both w/ Optimisations - 3840 x 2160 pixels.

## 3.5 Full Comparison

Here we compare the serial, the CPU and the GPU together. We have seen the effects of MSCV++ Optimization on all three previously and on different resolutions, so here we compare them based on one resolution. The serial process is significantly slower than the CPU or GPU but does close the gap somewhat with MSCV++ optimisation, but not enough to justify forgoing the parallel process on the CPU. This is seen in Figure 44 and Figure 45.

Figure 44 - CPU vs GPU vs Serial - 3840 x 2160 pixels.

Figure 45 - CPU vs Parallel GPU Both w/ Optimisations - 3840 x 2160 pixels.

# 4. Conclusion

In conclusion, the serial program was adapted to function in a naïve parallel manner using global memory that was processed in parallel in multiple threads. This made it more effective than the serial version by up to a factor of 130. The more the blur size increased, the larger the gains the parallelised system had over the serial one. The GPU performed best on parallelism against the CPU in this case, it was able to complete faster than the CPU in all cases by up to a factor of 16. This advantage came through its larger thread capacity which meant more pixels could be processed at once. With careful use of the OpenCL API this program has been accelerated and the amount of acceleration will only improve as the problem size expands. As time passes and multithreaded hardware gets faster, OpenCL is an advantage as it will run on multiple device platforms and for the foreseeable future.