Multicore Programming: Project Report

Paweł Jusięga

1 Introduction

Looking at a computer monitor very closely one may see that each pixel is composed out of three subpixels of the colours red, green and blue. Apparently [1] this has been the case back in the days of CRT displays as well. It should be of no surprise then that the average image file can be decomposed to three tables, each of the size equal to the product of the image's height and width in pixels and holding red, green and blue values ofr each subpixel. Perhaps more surprising is the fact that each cell of these tables hold values from 0 to 255. This is naturally an amount which is equal to a certain power of two, namely 2^8.

Having an image decomposed like this allows for its manipulation in a number of ways. Intuitively we might see that to blur an image for instance one needs to change the colour of each pixel into some sort of an average of its neighbourhood. This is called a discrete convolution and may be described in rather fancy mathematical terms, which will not be done here. It will be sufficient to say that it means that every particular pixel in the image is lined up with the centre of a normalised matrix of odd size called a kernel, like the one below which is a 3x3 box blur (a type of low pass filter, as all blur filters are) kernel:

$$K = \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix} \tag{1}$$

and what follows is what happens next: define a subpixel neighbourhood $N_r(x, y)$ which is a matrix composed of red subpixel values and identical in spatial localisation as the kernel K. The resulting red subpixel value shall then be a sum of element-wise products of K and N_r:

$$r(x, y) = \sum_{i,j} K_{i,j} N_{r\,i,j}(x, y) \equiv K * N_r(x, y). \tag{2}$$

The symbol $*$ denotes a convolution operation. For other subpixel colours the same reasoning shall be used and the same procedure applied in order to achieve a coloured, filtered image.

Naturally the image does not have to be blurred or otherwise filtered; there are other, more elementary operations that can be done on images, such as adding or multiplying two images together in a way that is more or less straightforward. For instance it is possible to extract a geometric mean of two images which is at one point used in the programs which this report is describing. These operations however do not require a massive number of calculations and thus it is not exactly tempting to attempt speeding them up using a GPU and an API such as CUDA which can lay over some of the work to the GPU. The operation subject to acceleration should, in essence, have quite a large kernel in order for the speedup gained when computing on a GPU to be truly significant. And it is quite obvious from the name *graphical processing unit* that a GPU should have a massive advantage over anything else when it comes to processing graphics.

In order to reduce the computational complexity of performing a convolution one may separate a square kernel into two vectors, the convolution of which is identical to the kernel. For instance the box blur kernel in (1) may be decomposed like this:

$$K = \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix} * \begin{bmatrix} 1/3 & 1/3 & 1/3 \end{bmatrix} \tag{3}$$

and one of the Sobel operators which may be used for edge detection decomposes like this:

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}. \tag{4}$$

Owing to the fact that the convolution operation is associative [2]

$$f * (v * h) = (f * v) * h \tag{5}$$

A separable filter $s = v * h$ may be applied by convolving the image f with the vertical portion first and the horizontal portion later. For some reason this significantly decreases the computational cost of the operation; in practice it means descending to one less level of for loops in the filtering function. Not all filters are separable; an example of this would be the circular box blur filter which simulates an image from an out of focus camera lens, the effect known as bokeh, although there exists a (literally) complex way to approximate such a filter using separable filters. [3]

The object of this report is to describe a program which was created and is more or less able to do 8 types of image manipulation: a gaussian blur, a box blur, a bokeh filter, sobel edge detection, difference of gaussians edge detection, an identity transformation, a basic unsharp mask and a slow, unseparated box blur. This program was then accelerated using CUDA with some success; the details of this success in terms of execution times and failures in the area of implemented features will be described below. The source code is available at `https://github.com/jusiega/multicore_project`.

2 Importing an image to C++

There exists no built-in feature of C/C++ which allows for easy loading and exporting of images. Fortunately some easy to use libraries serving this purpose may be found on the Internet [4]. To utilise them one needs to input the following:

```
#define STB_IMAGE_IMPLEMENTATION
#include "../Common/stb_image.h"
#define STB_IMAGE_WRITE_IMPLEMENTATION
#include "../Common/stb_image_write.h"

...

    unsigned char *data=stbi_load(source.c_str(), &width, &height, &nchannels, 0);
    //load image as a lengthy array containing r, g and b values sequenitally

    for(int i=0; i<height; i++) //initialise r, g and b
    {
        for(int j=0; j<width; j++)
        {
            r[i*width+j]=float(data[i*width*nchannels+j*nchannels]);
            g[i*width+j]=float(data[i*width*nchannels+j*nchannels+1]);
            b[i*width+j]=float(data[i*width*nchannels+j*nchannels+2]);
        }
    }

...

    stbi_write_bmp(destination.c_str(),width,height,nchannels, data);
    //write data as .bmp (a conversion inverse to the loop above has been performed)
```

The .bmp format was chosen as it is uncompressed.

3 Timing tool

The same timing tool as described in a previous report [5] was used. For mysterious reasons this time around it started to exhibit odd behaviour, namely returning the total execution time of the program as a value smaller than the execution time of one of its parts. The issue has been thought to be mitigated but it still has appeared

recently in some circumstances in the accelerated program. Perhaps this was a result of the rest of the program not working properly. This is not good. Frankly though the tool has been tested before and turned out not to be bad either. One might say that this is middling.

```
#ifndef _POSIX_C_SOURCE
#define _POSIX_C_SOURCE 199309L
#endif
#include <sys/time.h>
#include <time.h>
#include <stdlib.h>
#include <unistd.h>

//invoke wtime() from main to get a monotonic clock reading
//God knows when does the reading start so a starting point needds to be set
double wtime()
{
    static int sec = -1;
    struct timespec tv;
    clock_gettime(CLOCK_MONOTONIC, &tv);
    if (sec < 0) sec = tv.tv_sec;
    return (tv.tv_sec - sec) + 1.0e-9*tv.tv_nsec;
}
```

4 Clamp output

Some filters which are not simple low pass filters may result in the resulting output pixels exceeding the 0–255 RGB range. It is therefore necessary to implement a function which limits the range before the output gets passed on too far and invoke it soon after performing the filtering.

```
void clamp(float* r, float* g, float* b, int width, int height) /////CLAMP TO RGB RANGE
{
    for(int i=0; i<height; i++)
    {
        for(int j=0; j<width; j++)
        {
            if (r[i*width+j]>255.0){
                r[i*width+j]=255.0;
            }
            else if (r[i*width+j]<0){
                r[i*width+j]=0;
            }
            if (g[i*width+j]>255.0){
                g[i*width+j]=255.0;
            }
            else if (g[i*width+j]<0){
                g[i*width+j]=0;
            }
            if (b[i*width+j]>255.0){
                b[i*width+j]=255.0;
            }
            else if (b[i*width+j]<0){
                b[i*width+j]=0;
            }
        }
    }
}
```

5 Gaussian blur and the separable filter functions, including acceleration

As the two-dimensional Gaussian is a product of two one-dimensional Gaussians it is of no surprise that the Gaussian blur filter is separable. In fact it is so separable that the programs do not include an unseparated Gaussian blur filter because a vector kernel representing a Gaussian is easier to obtain than a matrix kernel with a 2D Gaussian. This does however mean that the blur kernel is centrally symmetric and so is the effect. The program chooses the operation to carry out using a switch sensitive to the variable *sw* which is input using std::cin at a prompt displayed soon after the launch of the program; so is *p*1 which determines the size of the blur (side dimension of a square matrix).

```
switch (sw)
{
case 0:
{    ////////GAUS BLUR/////////////
    float stdev=2;//default
    if (p1>0)
    {
        stdev=p1;
    }
    int gaussize=(2*int(stdev)-1)*3;// 3 sigma wide blur
    if (int(stdev)==0)
    {
        gaussize=3;
    }
    float* gausblur=(float*)malloc(gaussize*sizeof(float));
    float norm=0;
    for (int i=0;i<gaussize;i++)
    {
        gausblur[i]=1/sqrt(2*3.14159)*exp(-(float)pow(i-(gaussize-1)/2,2)/(2*stdev*stdev));
        norm+=gausblur[i];
    }
    for (int i=0;i<gaussize;i++)
    {
        gausblur[i]/=norm;
        //std::cout<<gausblur[i]<<' ';
    }
    elapsedtime=wtime()-elapsedtime;
    totaltime+=elapsedtime;
    std::cout<<"time elapsed from starting program to beginning of filtering: ";
    std::cout<<elapsedtime<<" s."<<std::endl;
    sepfilter(r,g,b,rnew,gnew,bnew,width,height,gausblur,gausblur,gaussize);
    elapsedtime=wtime()-elapsedtime;
    totaltime+=elapsedtime;
    std::cout<<"time elapsed during filtering: "<<elapsedtime<<" s."<<std::endl;
    free(gausblur);
}
break;
...
```

The arrays *r*, *g* and *b* are initialised as shown before and arrays *rnew*, *gnew* and *bnew* are allocated prior and passed on so that this process needn't wait until the filter functions are invoked (and all of them need an additional set of arrays like this). The usage of the timing tool is seen here as well. The function sepfilter() allows for asymmetrical kernels, however passing two of the same vectors here means that the resulting kernel is symmetrical. The entire body of this function is shown below and it is the single most important thing in the entire program.

```
void sepfilter(float* r, float* g, float* b, float* rnew, float* gnew, float* bnew,
        int width, int height, float*  blurh, float* blurv, int blursize)
```

```
{
    float red_tmp=0;
    float green_tmp=0;
    float blu_tmp=0;
    int shift=(blursize-1)/2;
    //MAIN BLUR VERTICAL
    for(int i=shift; i<height-shift; i++)
    {
        for(int j=0; j<width; j++)
        {
            for (int k=0; k<blursize; k++)
            {
                red_tmp+=blurv[k]*(r[(i-shift)*width+k*width + j]);
                green_tmp+=blurv[k]*(g[(i-shift)*width+k*width + j]);
                blu_tmp+=blurv[k]*(b[(i-shift)*width+k*width + j]);
            }
            rnew[i*width+j]=red_tmp;
            gnew[i*width+j]=green_tmp;
            bnew[i*width+j]=blu_tmp;
            red_tmp=0;
            green_tmp=0;
            blu_tmp=0;
        }
    }
    //EDGE UP USING MIRROR & VERTICAL BLUR
    //SOMEBODY ONCE TOLD ME THAT IF STATEMENTS ARE SLOW
    for(int i=0; i<shift; i++)
    {
        for(int j=0; j<width; j++)
        {
            for (int k=0; k<blursize; k++)
            {
                red_tmp+=blurv[k]*(r[abs(((i-shift)+k))*width + j]);
                green_tmp+=blurv[k]*(g[abs(((i-shift)+k))*width + j]);
                blu_tmp+=blurv[k]*(b[abs(((i-shift)+k))*width + j]);
            }
            rnew[i*width+j]=red_tmp;
            gnew[i*width+j]=green_tmp;
            bnew[i*width+j]=blu_tmp;
            red_tmp=0;
            green_tmp=0;
            blu_tmp=0;
        }
    }
    //EDGE DOWN
    for(int i=height-shift; i<height; i++)
    {
        for(int j=0; j<width; j++)
        {
            for (int k=0; k<blursize; k++)
            {
                red_tmp+=blurv[k]*(r[(-1+height-abs(height-1-(i-shift+k)))*width + j]);
                green_tmp+=blurv[k]*(g[(-1+height-abs(height-1-(i-shift+k)))*width + j]);
                blu_tmp+=blurv[k]*(b[(-1+height-abs(height-1-(i-shift+k)))*width + j]);
            }
            rnew[i*width+j]=red_tmp;
```

```
            gnew[i*width+j]=green_tmp;
            bnew[i*width+j]=blu_tmp;
            red_tmp=0;
            green_tmp=0;
            blu_tmp=0;
        }
    }
    //MAIN BLUR HORIZONTAL, NO LEFT & RIGHT EDGES
    for(int i=0; i<height; i++)
    {
        for(int j=shift; j<width-shift; j++)
        {
            for (int l=0; l<blursize; l++)
            {
                red_tmp+=blurh[l]*(rnew[i*width + j-shift+l]);
                green_tmp+=blurh[l]*(gnew[i*width + j-shift+l]);
                blu_tmp+=blurh[l]*(bnew[i*width + j-shift+l]);
            }
            r[i*width+j]=red_tmp;
            g[i*width+j]=green_tmp;
            b[i*width+j]=blu_tmp;
            red_tmp=0;
            green_tmp=0;
            blu_tmp=0;
        }
    }
    //EDGE LEFT HORIZONTAL PART
    //VERTICAL PART WAS DONE IN MAIN BLUR VERTICAL
    for(int i=0; i<height; i++)
    {
        for(int j=0; j<shift; j++)
        {
            for (int l=0; l<blursize; l++)
            {
                red_tmp+=blurh[l]*(rnew[i*width + abs((j-shift+l))]);
                green_tmp+=blurh[l]*(gnew[i*width + abs((j-shift+l))]);
                blu_tmp+=blurh[l]*(bnew[i*width + abs((j-shift+l))]);
            }
            r[i*width+j]=red_tmp;
            g[i*width+j]=green_tmp;
            b[i*width+j]=blu_tmp;
            red_tmp=0;
            green_tmp=0;
            blu_tmp=0;
        }
    }
    //EDG RIGHT
    for(int i=0; i<height; i++)
    {
        for(int j=width-shift; j<width; j++)
        {
            for (int l=0; l<blursize; l++)
            {
                red_tmp+=blurh[l]*(rnew[i*width + width-1-abs(width-1-(j-shift+l))]);
                green_tmp+=blurh[l]*(gnew[i*width + width-1-abs(width-1-(j-shift+l))]);
                blu_tmp+=blurh[l]*(bnew[i*width + width-1-abs(width-1-(j-shift+l))]);
```

```
            }
            r[i*width+j]=red_tmp;
            g[i*width+j]=green_tmp;
            b[i*width+j]=blu_tmp;
            red_tmp=0;
            green_tmp=0;
            blu_tmp=0;
        }
    }
}
```

One may see that there are many symbols in the indices which inevitably introduce some confusion. The very worst of these acrobatics are performed in order to achieve filtering up to the very edges of the image. This method depends on pretending that beyond every side of the image lies its mirror reflection. On the corners something similar happens but my understanding of what is it exactly boils down to the fact that it looks alright.

In the accelerated program the corresponding part of the switch looks like this:

```
switch (sw)
{
case 0:
{    ////////GAUS BLUR////////////
    float stdev=2;//default
    if (p1>0)
    {
        stdev=p1;
    }
    int gaussize=(2*int(stdev)-1)*3;// 3 sigma wide blur
    if (int(stdev)==0)
    {
        gaussize=3;
    }
    float* gausblur;
    cudaMallocManaged(&gausblur, gaussize*sizeof(float));
    float norm=0;
    for (int i=0;i<gaussize;i++)
    {
        gausblur[i]=1/sqrt(2*3.14159)*exp(-(float)pow(i-(gaussize-1)/2,2)/(2*stdev*stdev));
        norm+=gausblur[i];
    }
    for (int i=0;i<gaussize;i++)
    {
        gausblur[i]/=norm;
        //std::cout<<gausblur[i]<<' ';
    }
    cudaMemPrefetchAsync(gausblur, gaussize*sizeof(float), deviceId);
    cudaDeviceSynchronize();
    elapsedtime=wtime()-elapsedtime;
    totaltime+=elapsedtime;
    std::cout<<"time elapsed from starting program to beginning of filtering: "<<elapsedtime<<" s."
    sepfilterV<<<numberofblocks,threadsperblock>>>(r,g,b,rnew,gnew,bnew,width,height,gausblur,gauss
    cudaDeviceSynchronize();
    sepfilterH<<<numberofblocks,threadsperblock>>>(r,g,b,rnew,gnew,bnew,width,height,gausblur,gauss
    cudaDeviceSynchronize();
    elapsedtime=wtime()-elapsedtime;
    totaltime+=elapsedtime;
```

```
    std::cout<<"time elapsed during filtering: "<<elapsedtime<<" s."<<std::endl;
    cudaFree(gausblur);
}
break;
...
```

Immediately it is visible that the sepfilter() function was broken up into two parts, separated by a synchronisation barrier. This had to be done to avoid errors. The array *gausblur* was allocated using cudaMallocManaged(), as were all the other arrays passed to the sepfilterV() and sepfilterH() functions. The execution configuration was chosen with 64 threads per block, as this appeared to be a reasonable value if having in mind the conclusions from one of the previous reports [6], at least for the machines that were utilised when taking those measurements, and the number of blocks large enough to make each thread work on exactly one pixel. CUDA limitations in total block size seem large enough to be far beyond what the image import library allows so a grid-stride loop was not implemented in the function bodies. These are shown below fully, which is perhaps not exactly necessary but it could be worse than that.

```
__global__ void sepfilterH(float* r, float* g, float* b, float* rnew, float* gnew, float* bnew,
    int width, int height, float*  blurh, int blursize)
{   __syncthreads();
    long long int idx= threadIdx.x + blockIdx.x * blockDim.x;
    int j=int(idx%width);
    int i=int(idx/width);

    float red_tmp=0;
    float green_tmp=0;
    float blu_tmp=0;
    __syncthreads();
    int shift=(blursize-1)/2;
    //MAIN BLUR HORIZONTAL, NO LEFT & RIGHT EDGES
    if(i>=0 && i<height && j>=shift && j<(width-shift))
    {
            for (int l=0; l<blursize; l++)
            {
                red_tmp+=blurh[l]*(rnew[i*width + j-shift+l]);
                green_tmp+=blurh[l]*(gnew[i*width + j-shift+l]);
                blu_tmp+=blurh[l]*(bnew[i*width + j-shift+l]);
            }

            r[i*width+j]=red_tmp;
            g[i*width+j]=green_tmp;
            b[i*width+j]=blu_tmp;

            red_tmp=0;
            green_tmp=0;
            blu_tmp=0;
    }
    __syncthreads();
    //EDGE LEFT HORIZONTAL PART
    //VERTICAL PART WAS DONE IN MAIN BLUR VERTICAL
    if(i>=0 && i<height && j>=0 && j<shift)
    {
            for (int l=0; l<blursize; l++)
            {
                red_tmp+=blurh[l]*(rnew[i*width + abs((j-shift+l))]);
                green_tmp+=blurh[l]*(gnew[i*width + abs((j-shift+l))]);
                blu_tmp+=blurh[l]*(bnew[i*width + abs((j-shift+l))]);
            }
```

```
            r[i*width+j]=red_tmp;
            g[i*width+j]=green_tmp;
            b[i*width+j]=blu_tmp;
            red_tmp=0;
            green_tmp=0;
            blu_tmp=0;
    }
    __syncthreads();
    //EDG RIGHT
    if(i>=0 && i<height && j>=(width-shift) && j<width)
    {
            for (int l=0; l<blursize; l++)
            {
                red_tmp+=blurh[l]*(rnew[i*width + width-1-abs(width-1-(j-shift+l))]);
                green_tmp+=blurh[l]*(gnew[i*width + width-1-abs(width-1-(j-shift+l))]);
                blu_tmp+=blurh[l]*(bnew[i*width + width-1-abs(width-1-(j-shift+l))]);
            }
            r[i*width+j]=red_tmp;
            g[i*width+j]=green_tmp;
            b[i*width+j]=blu_tmp;
            red_tmp=0;
            green_tmp=0;
            blu_tmp=0;
    }
}

__global__ void sepfilterV(float* r, float* g, float* b, float* rnew, float* gnew, float* bnew,
    int width, int height, float* blurv, int blursize)
{
    __syncthreads();
    long long int idx= threadIdx.x + blockIdx.x * blockDim.x;
    int j=int(idx%width);
    int i=int(idx/width);

    //int i=blockIdx.x*blockDim.x+threadIdx.x;
    //int j=blockIdx.y*blockDim.y+threadIdx.y;

    float red_tmp=0;
    float green_tmp=0;
    float blu_tmp=0;

    int shift=(blursize-1)/2;
    __syncthreads();
    //MAIN BLUR VERTICAL
    if(i>=shift && i<(height-shift) && j>=0 && j<width)
    {
            for (int k=0; k<blursize; k++)
            {
                red_tmp+=blurv[k]*(r[(i-shift)*width+k*width + j]);
                green_tmp+=blurv[k]*(g[(i-shift)*width+k*width + j]);
                blu_tmp+=blurv[k]*(b[(i-shift)*width+k*width + j]);
            }

            rnew[i*width+j]=red_tmp;
            gnew[i*width+j]=green_tmp;
            bnew[i*width+j]=blu_tmp;
```

```
            red_tmp=0;
            green_tmp=0;
            blu_tmp=0;

    }

    __syncthreads();
    //EDGE UP USING MIRROR & VERTICAL BLUR
    //SOMEBODY ONCE TOLD ME THAT IF STATEMENTS ARE SLOW
    if(i>=0 && i<shift && j>=0 && j<width)
    {
            for (int k=0; k<blursize; k++)
            {
                red_tmp+=blurv[k]*(r[abs(((i-shift)+k))*width + j]);
                green_tmp+=blurv[k]*(g[abs(((i-shift)+k))*width + j]);
                blu_tmp+=blurv[k]*(b[abs(((i-shift)+k))*width + j]);
            }
            rnew[i*width+j]=red_tmp;
            gnew[i*width+j]=green_tmp;
            bnew[i*width+j]=blu_tmp;
            red_tmp=0;
            green_tmp=0;
            blu_tmp=0;
    }
    __syncthreads();
    //EDGE DOWN
    if(i>=(height-shift) && i<height && j>=0 && j<width)
    {
            for (int k=0; k<blursize; k++)
            {
                red_tmp+=blurv[k]*(r[(-1+height-abs(height-1-(i-shift+k)))*width + j]);
                green_tmp+=blurv[k]*(g[(-1+height-abs(height-1-(i-shift+k)))*width + j]);
                blu_tmp+=blurv[k]*(b[(-1+height-abs(height-1-(i-shift+k)))*width + j]);
            }
            rnew[i*width+j]=red_tmp;
            gnew[i*width+j]=green_tmp;
            bnew[i*width+j]=blu_tmp;
            red_tmp=0;
            green_tmp=0;
            blu_tmp=0;
    }
}
```

Most of the for() loops were replaced with if statements and indices i and j were calculated from the total thread index expression. Incredibly, all this works perfectly:

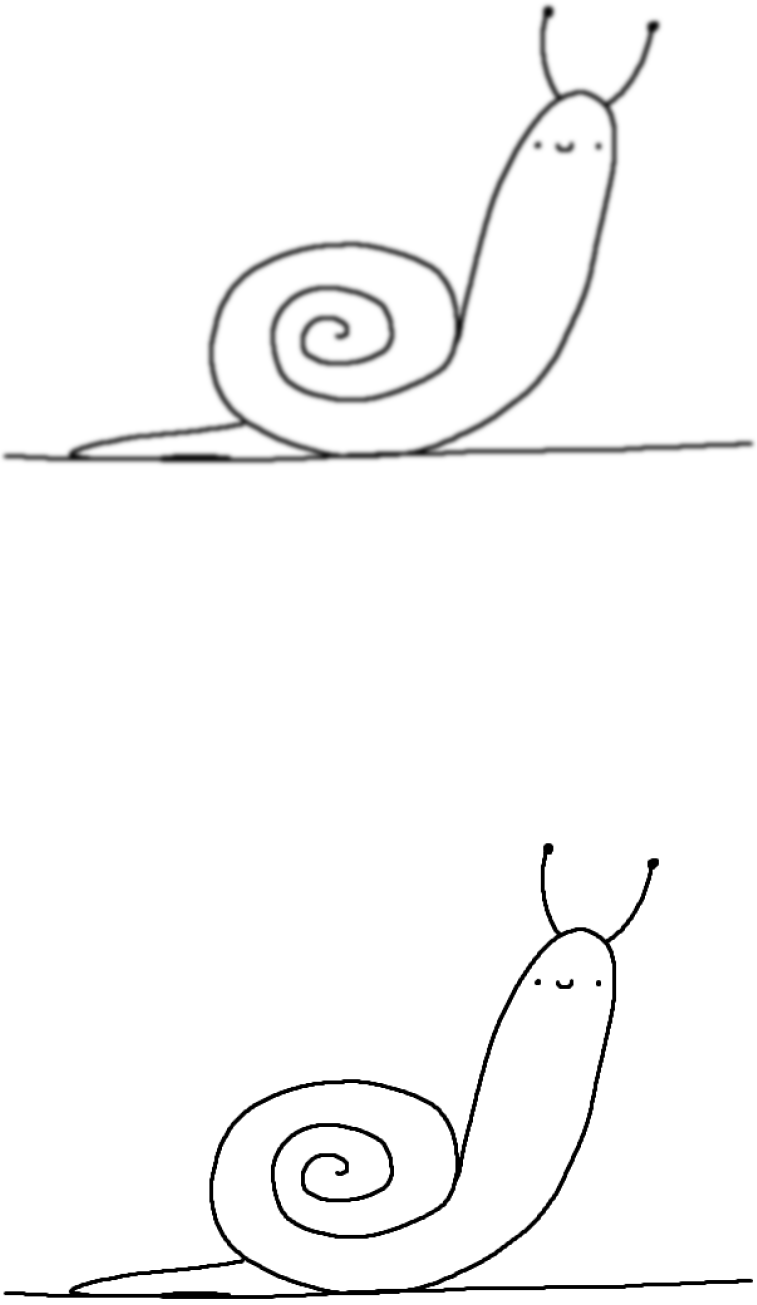

Figure 1: Image treated with the Gaussian blur of standard deviation equal to seven pixels (top) made using the accelerated program and the original image (bottom).

The result of the less basic algorithm of difference of Gaussians edge detection is presented on the next page.

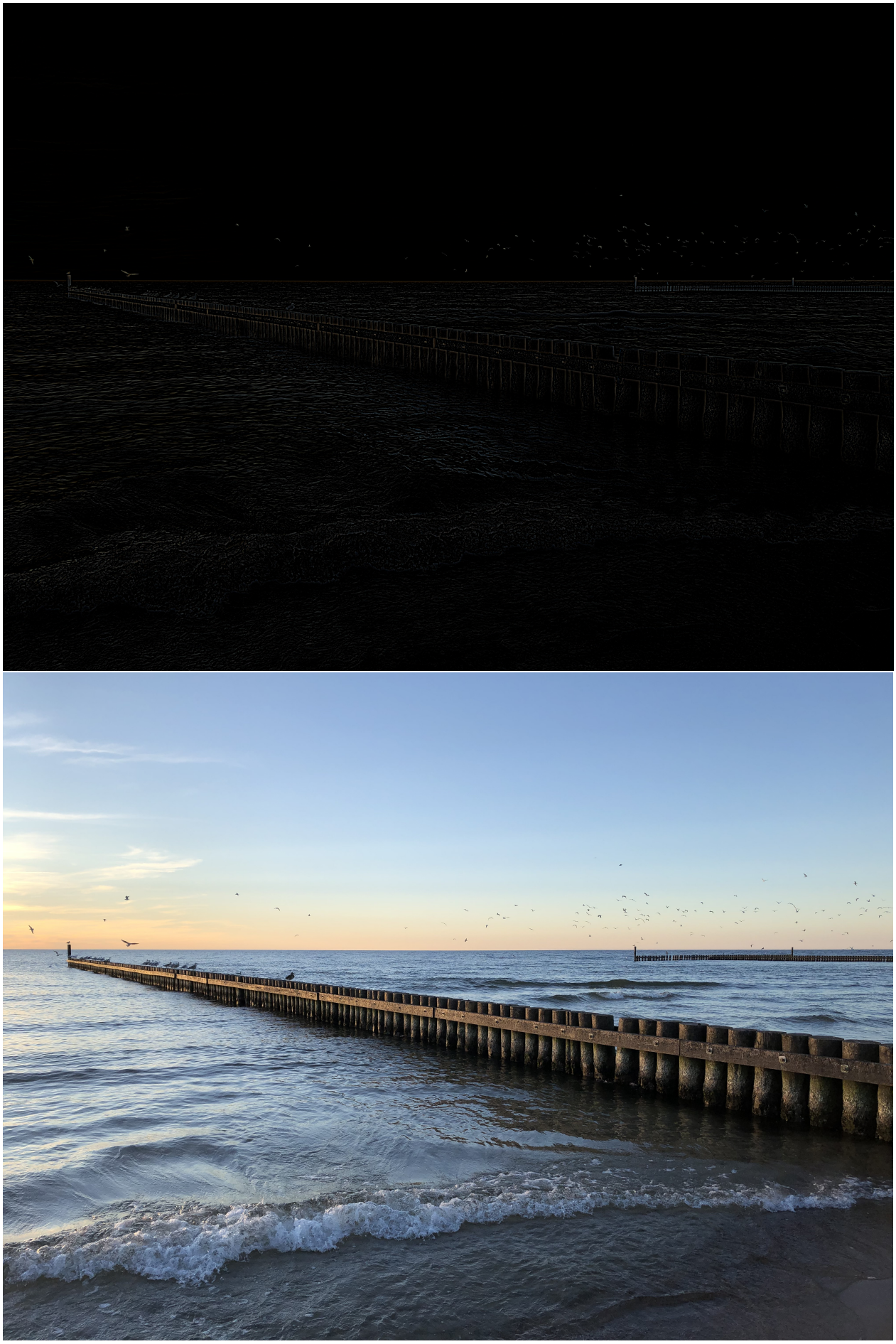

Figure 2: Difference of Gaussians edge detection (top) and the original image (bottom). It turned out quite dark and probably the sample image was not chosen very wisely.

To be precise, the *perfection* mentioned before works up to, in the case of the Gaussian blur, a certain standard deviation, beyond which the *blursize* becomes greater than one of the image dimensions. The resulting image is then not blurred at all, which is also an outcome much more welcome than a segmentation fault or something similar. Other filters utilising the sepblurV() and sepblurH() also seem to work fine. Now, onto what does not...

6 Non-separated blur

The easier conceptually and seemingly easier to implement non-separated blur function, or fblur() as it is called in the programs' codes, seemed to be working correctly in the standard program but leaks memory in the accelerated program. Namely it is the edge blurring that does that. The memory leak might be present in the unaccelerated version of the program, perhaps due to one of the stupidly complicated index expressions being written wrong and the mistake being stupidly hard to spot. Unfortunately the (quite relaxed) time constraints have proven still too tight for possessed time management skills and/or programming skills to fix this on time. Not to oversaturate this report with lines of code, an interested reader is referred to the repository of the project [7].

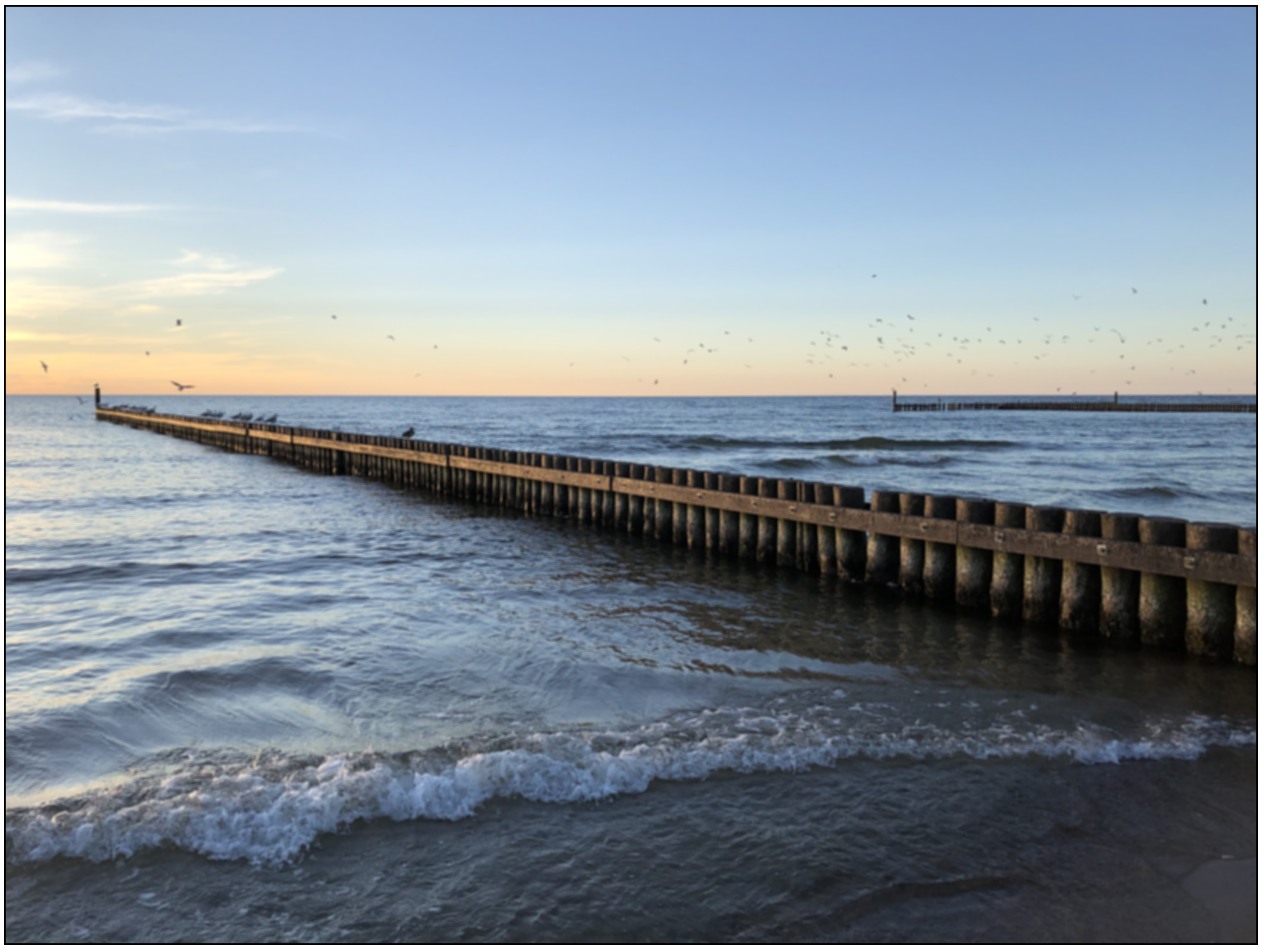

Figure 3: Output of the bokeh filter with a diameter of 13 pixels, from the unaccelerated program.

7 Execution time measurements

Execution times of the filter functions were measured for three types of operations, one of which was the bokeh and represented the non-separated filter function. As the edge handling seemed broken it was turned off for both the unaccelerated (slow) and accelerated (fast) program. The image filtered was the beach image visible in figures 2 and 3, of dimensions 4032x3024.

Table 1: Table of results obtained in the measuring process. One of the filters turned out too difficult for the CPU. Times are in seconds. Numbers of measurements were reduced out of impatience, however it seems that the relative error decreases as the execution time increases which perhaps makes this practice not so bad.

	Gaussian blur, $\sigma = 7$		**Gaussian blur**, $\sigma = 100$		**bokeh**, $d = 13$	
	fast	slow	fast	slow	fast	slow
	0.672	5.35	0.568	no	0.278	11.7
	0.472	5.49	1.099	output	0.511	11.9
	0.455	5.31	0.745	in	0.336	12.2
	0.048	5.54	1.179	any	0.538	11.6
	0.793	5.47	0.481	reasonable	0.888	12.2
	0.138	5.84	0.447	time	0.264	
	0.475	5.59	1.225		0.128	
	0.962		0.492		0.195	
	0.808		0.640		0.242	
	0.594		0.397		0.091	
	0.156		1.252		0.271	
average	0.507	5.511	0.77		0.340	11.93
SE	0.090	0.066	0.10		0.069	0.13
speedup	1088.06%		∞		3507.98%	

It is obvious that the GPU thrives in this kind of a workload. Accelerating the program has also allowed for a bigger range of parameters that the filters may be launched with and thus increased the usefulness of the program undeniably. The speedup on the bokeh was larger than on the Gaussian blur likely due to differences in computational complexity of the two processes.

8 Future development

If any future development is to be commenced there are some areas easily noted as having potential to improve. One of the functions that is not implemented very well is the unsharp masking as it is very lacking in control and very basic and thus the images produced by this filter are not exactly inspiring. Probably an unsharp mask based on a Gaussian blur would give results more satisfying, with the implementation being similar to that of the difference of Gaussians edge detection filter. Definitely a fix is needed for the broken edge handling for non-separated filters. An approximate antialiasing filter based on the FXAA algorithm [8] could definitely be implemented in a fairly straightforward way basing upon what has been already done within the code, but the issues with accelerating the code unfortunately have prevented this from happening at this time.

9 Conclusions

The program has become fast, however the code writing has become slow. Perhaps it has never been fast, but still. The accelerated code lost some functionality as well. Probably a mistake was made when deciding to implement this many types of image manipulation as that in itself was time consuming and has made the code difficult and tedious to modify and debug. It is very much possible, and in some cases certain (a clear image with high Gaussian blur settings at first felt like a complete mystery), that the issues exposed by the accelerated code were also present in the standard version of the program but were not noticeable until the astounding speed of using the GPU allowed for running the filters with sufficiently extreme settings. This possibility has in turn improved the program vastly and thus it has to be concluded that the decision to accelerate the program was rather reasonable.

10 Literature

[1] https://en.wikipedia.org/wiki/RGB_color_model

[2] https://blogs.mathworks.com/steve/2006/10/04/separable-convolution/

[3] https://www.ea.com/frostbite/news/circular-separable-convolution-depth-of-field

[4] https://github.com/nothings/stb

[5] *Multicore Programming: Lab Report 2*, of which I am the author

[6] *Multicore Programming: Lab Report 1*, of which I am the author

[7] https://github.com/jusiega/multicore_project

[8] https://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf