Gaussian Blur on Images using Parallel Computation Techniques

I. Introduction

Gaussian blurring is a way of filtering an image, and is widely used in image processing. This makes use of a Gaussian function in order to process an image. The resulting image is a version of the original image, but is smoothed by reducing noise and reducing detail. One could say that the effect of a Gaussian blur resembles the effect of viewing an image through a translucent screen [1], an example of this is shown in Fig. 1. Gaussian blurring is also useful for enhancing visual structures while pre-processing an image for a computer vision task. The resulting algorithms of this project will be used as part of a medical imaging project, in order to pre-process the image before running a edge detection algorithm. However, as the image being processed increases in resolution (as is the case in digital mammography), the efficiency and speed of a normal CPU implementation would decrease. Therefore, this implementation is simply not good enough when efficiency and speed is required, especially when one is processing many images. In this report we will be exploring two different ways of making this algorithm more efficient using CPU and GPU based parallel computation methods, CUDA and MPI. We will compare the two methods based on out observations such as time and speed-up.

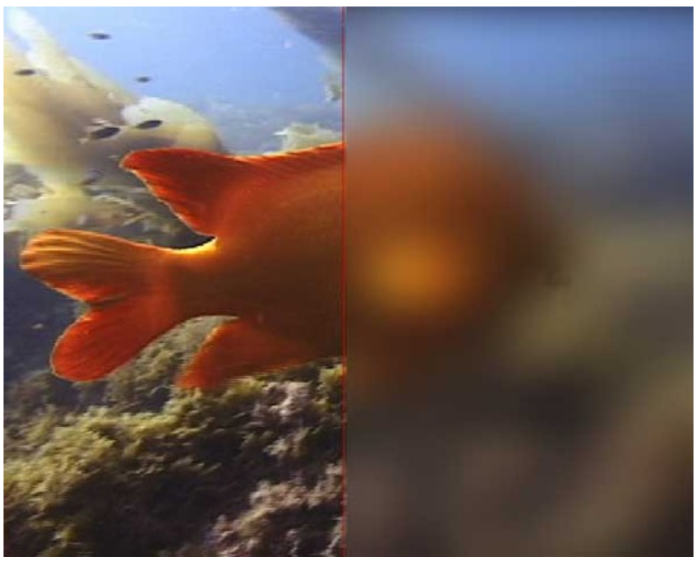
Fig. 1: example of a gaussian blur. Source: [2].

II. Gaussian Blurring

The Gaussian filter is used to reduce noise and detail, in order to blur an image. The main concept of a Gaussian blur is to update a pixel by calculating a weighted average of it's neighbouring pixels, and repeat this for all the pixels in the image. The weights of the neighbouring pixels are calculated by generating a Gaussian kernel. A Gaussian distribution is sometimes referred to as bell-shaped, the closer we are to the centre, the greater the value. Therefore, the pixel in question will have a greater value of the ones surrounding it. As an image is two dimensional, we will use a two dimensional gaussian function. The one dimensional and two dimensional functions are shown in the Fig. 2 and 3, respectively.

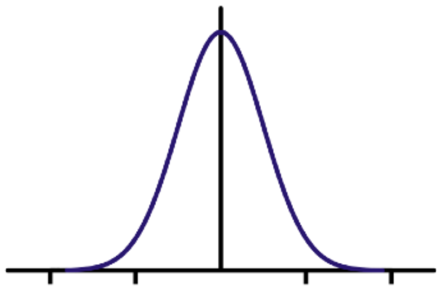

Fig 2. 1-D Gaussian Function.

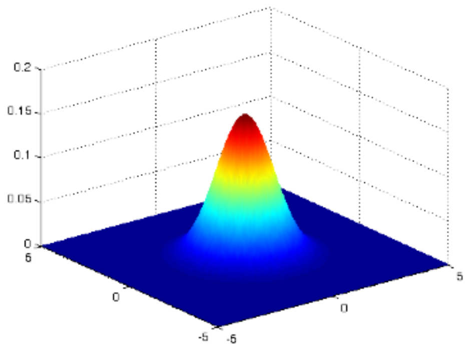

Fig 3. 2-D Gaussian Function.

The mathematical function for a two dimensional Gaussian distribution is defined in Fig. 3. The values of *x* and *y* refer to the horizontal and vertical distance, in pixels, from the origin centre pixel, and *g* gives the value at the weight of the kernel coordinate, which will be multiplied by the pixel value. The sigma, σ, value is the width of the gaussian kernel, which in statistics is referred to the standard deviation. Naturally, the origin pixel will receive the heaviest weight. This two dimensional distribution is used to generate a convolution matrix that will be applied to every pixel in the image.

$$g(x, y) = \frac{1}{2\pi\sigma^2} \cdot e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Fig. 3. Function of a Gaussian distribution. [3]

If the image resolution is 512x512, the amount of times the kernel will be applied is 512^2 times, and if the kernel has dimensions of 5x5, the amount of calculations that will be made is $512^2 * 5^2$ which is 6553600 calculations. This means that the amount calculations increases drastically as the image size increases. The expression for the time complexity of this algorithm is:

$$O = (image_w * image_h * kernel_w * kernel_h)$$

III. Implementation

In order to import an image, the LodePNG C++ library [4] was used. This imports the image into a one dimensional unsigned char vector filled with the RGBA values of every pixel, as shown below.

$$image_v = [R_1, G_1, B_1, A_1, R_2, ..., A_n]$$

As mammogram scans images are usually grayscale, we need only one value for each pixel. In order to make sure the image is grayscale, the mean of the three RGB values are stored in a new array, and the alpha channel is discarded as it is no longer needed. The convolution will be done on these grey level values. In order to use less memory and for an increase in speed, both the generated kernel and image matrices are stored as dynamic one dimensional arrays. A simple utility function was created and used every time we needed to convert a two dimensional index, x and y, into a one dimensional index, x. To get the one dimensional index for x and y, we can simply do:

$$x_1 = y_2 * width + x_2$$

CUDA Implementation

The CUDA C programming language extends standard C, and adds functions useful for GPU programming. These functions are called kernels. When a kernel is called, it runs N times in parallel in N channels [5]. This creates CUDA threads.

As an example, let's consider that we have launched a CUDA kernel with the following grid and block dimensions:

```
dim3 blockDim(5, 3, 1);
dim3 gridDim(3, 2);
```

The thread organisation for this kernel would look like Fig. 4, there are 3*2 blocks per grid, and 5*3 threads per block.

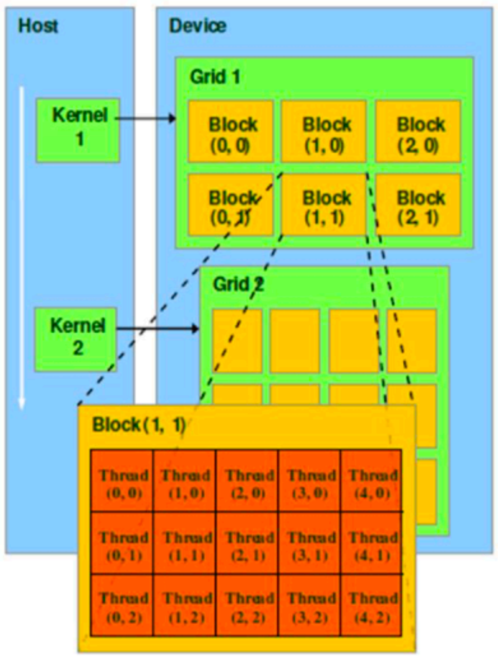

Fig. 4 Thread organisation. Source: [5]

The steps taken in the CUDA implementation of a Gaussian blur are as follows:

1. Import image using LodePNG.
2. Generate Gaussian kernel.
3. Allocate memory on the host for the input and output image arrays using cudaMallocHost.
4. Allocate memory on the device for the input and output arrays using cudaMalloc, and copy the contents of the host arrays into the allocated memory on the device using cudaMemcpy and cudaMemcpyToSymbol.
5. Start timer.
6. Run the CUDA kernel using defined block and grid dimensions, allocating each pixel to it' own CUDA thread.
7. Run the Gaussian filter algorithm on each pixel:
    i. If pixel coordinates are within the image dimensions, do:
        i. Iterate through the whole kernel matrix and do:
            i. new_val = new_val + kernel_weight*pixel_val.
        ii. Set the new centre pixel's value = new_val.
8. Synchronise the device using cudaDeviceSynchronize.
9. Copy the contents of the filtered array from the device back to the host using cudaMemcpy.

10. End timer
11. Save the image to disk using LodePNG again

In our algorithm, the number of blocks per grid is reliant on the number of threads per block and the dimensions of the image. This is given by the width or height of the image divided by the number of rows or columns of threads in a block. The x and y dimensions of the grid are given by:

$$grid_x = \frac{width}{block_x}, \ grid_y = \frac{height}{block_y}$$

MPI Implementation

The steps taken in the MPI implementation of the Gaussian blur are as follows:

1. Initialise MPI using MPI_Init(), and declare the size and rank using MPI_Comm_size() and MPI_Comm_rank() with the MPI_COMM_WORLD communicator.
2. In process 0:
   2.1.Import the image using LodePNG.
   2.2.Generate the Gaussian kernel by calling the getGaussianKernel() function.
3. Broadcast the kernel and image dimensions, previously calculated in process 0, to all processes in order to make sure every process has access to these vital variables. Using MPI_Bcast().
4. Allocate memory for the two arrays that would store the original pixel values and the filtered pixel values of the image, using the C function malloc().
5. In process 0:
   5.1.Process the raw RGBA vector generated by LodePNG by only storing one grey level value per pixel.
   5.2.Start the timer.
6. Allocate memory in each process for it's respective portion of the image and and output, using the C malloc() function.
7. Using MPI_Scatter(), divide the array and send an equal portion of the data to each process.
8. Get each process to call the runFilter() function by passing through it's own portion of the data.
9. Wait for all processes to finish working until continuing, by using MPI_Barrier().

10. Using MPI_Gather(), collect all portions of data from every process back into one array handled by process 0.
11. In process 0:
    11.1. End the timer.
    11.2. Save the image to disk using LodePNG again

MPI_Bcast

The MPI broadcast function is a standard collective communication technique. By using MPI_Bcast, one process sends the same piece of data to every other process in order to make sure that every process has a copy of the exact data [6]. We are using this to send some vital variables from where they were calculated, in process 0, to all other processes, however many there may be. Fig. 5 shows an example of this. If process 0 has data A, after running MPI_Bcast, all processes will also have data A.

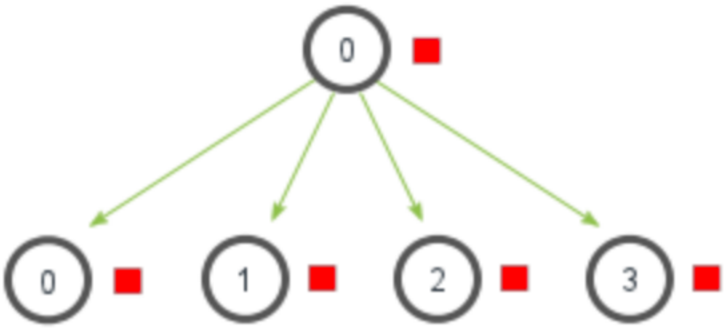

Fig. 5 MPI_Bcast diagram. Source: [7].

MPI_Scatter

The scatter function in MPI is similar to the broadcast function as it also sends data from one process to all other processes. The difference is that, instead of sending the same exact data, MPI_Scatter sends a chunk of an array to each process. For example, if we have an array X=[A,B,C,D], MPI_Scatter will send X[0] to process 1, X[1] to process 2, and so on. This is illustrated in Fig. 6. For this project, we are using MPI_Scatter to split the image array, into n chunks, where n is the number of processes, and send each chunk to it's respective process.

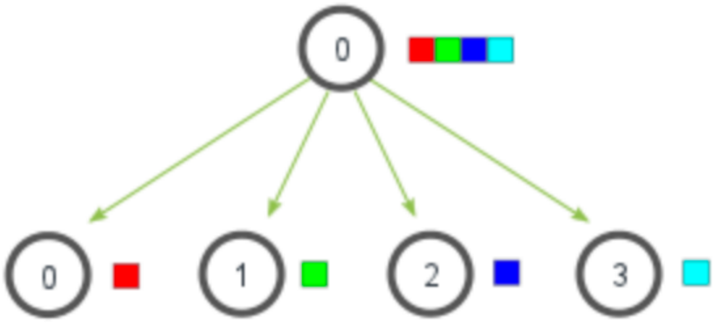

Fig. 6 MPI_Scatter diagram. Source: [7].

MPI_Gather

The gather function in MPI, is simply a inverse of the MPI_Scatter function shown above. Instead of sending chunks of data from one process to many processes, it collects chunks from all processes into one array in one process [7]. For this project, MPI_Scatter is used after all processes are done running the Gaussian filter on it's chunk of data. It collects all the filtered data chunks into an new output array that is handled by process 0. Fig. 7 shows an illustration of this function.

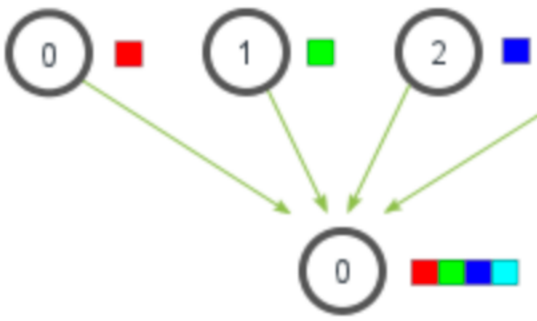

Fig. 7 MPI_Gather diagram. Source: [7].

MPI_Barrier

The final MPI method of collective communication that we use is called MPI_Barrier. This function forms a barrier where it is placed and does not allow any processes to pass until all other processes have caught up [6]. This makes sure that all processes are at the same point. This is particularly useful for us as we have to wait for all processes to finish running the Gaussian filter on their chunk of data before sending all the data back to process 0. By doing this, we make sure that the filtered image is complete and isn't missing any data due to a process not being able to complete in time. An illustration of MPI_Barrier is shown in Fig. 8, with T indicating the progression of time.

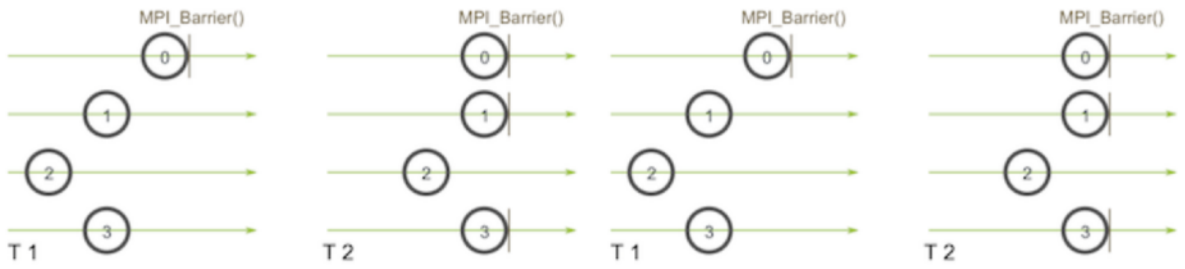

Fig. 8 MPI_Barrier diagram. Source: [7].

IV. Evaluation

Results

Due to time constraints, it was not possible to get the MPI algorithm to output a desired blurred image, therefore, the output image isn't accurate. However, this does not in any way affect the performance of the algorithm so we can still compare it to the CUDA solution. Fig 9 shows the input and output of the CUDA filter when using a generic image, using a standard deviation of 3 and kernel size of 15. This is included in order to make it easier to demonstrate the blur effect as this could sometimes be difficult to notice on mammogram scans.

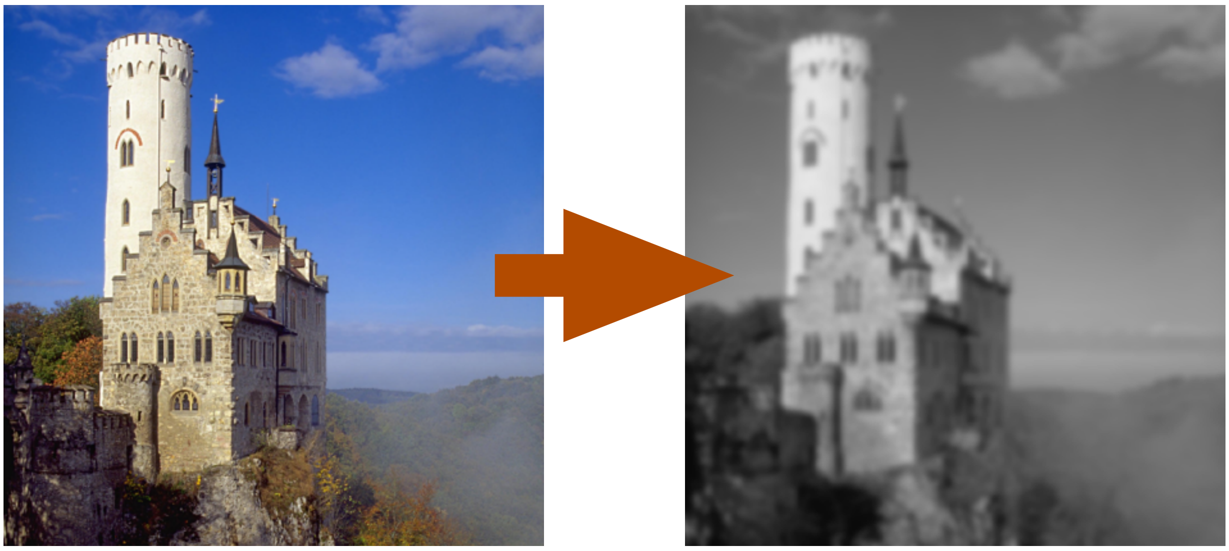
Fig. 9 CUDA Gaussian blur output.

Fig. 10 shows the output of the filter when using a digital Mammogram scan image as it's input. It shows the effects of different standard deviation values on the output image. As can be seen, this filter is very successful at reducing the noise of the image, making it more effective when running an edge detection algorithm. This is because unnecessary bright part of the image, such as noise, etc. are reduced, and the most important vessel and artery lines are more visible. This is crucial for classifying things such as calcifications in the breast as these rely on visibility of the different blood vessels and their condition. As the standard deviation increases, the images becomes more blurred, however, if the standard deviation is too high, the image starts to become too dark.

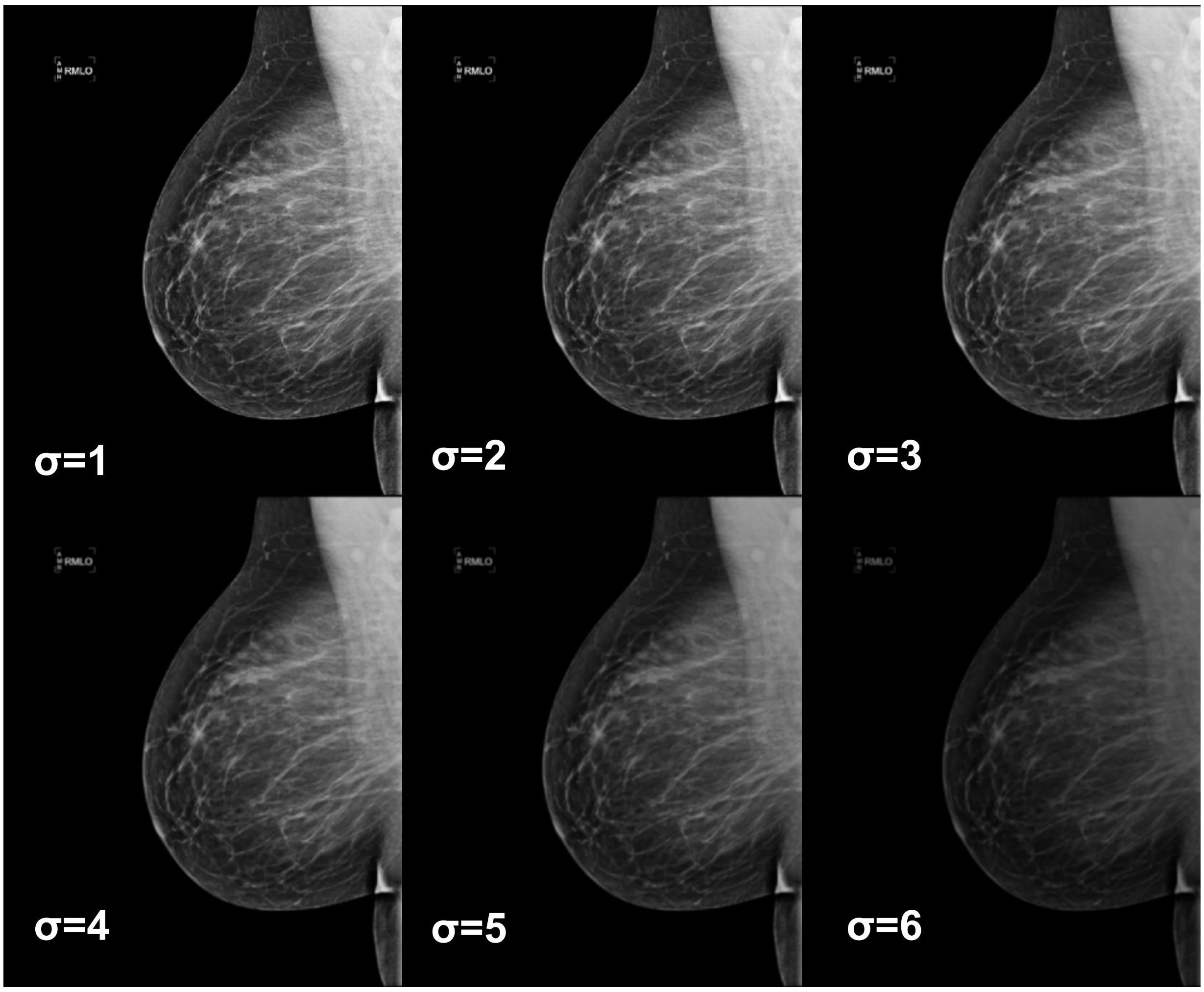

Fig. 10 CUDA Gaussian blur output.

Testing, Comparison and Evaluation

The image that was used for the experiments has an original resolution of 3328x4096px. In order to test the performance of different image resolutions, it was decided to create copies of the image with 80%, 60%, 40% and 20% of the original resolution. Each image resolution will be tested exactly the same way and it's results recorded. For CUDA, the experiment was repeated for different block and grid dimensions, for each image resolutions. The block dimensions that were tested are 32x32x1, 16x16x1, 8x8x1, 4x4x1 and 2x2x1. The greater bock size was chosen as any block size greater than this would show minimal difference in performance. Table 1 shows the CUDA experiment result, showing the performance of the algorithm in milliseconds. The timer was started just before the CUDA kernel launch and was ended just after it returns. Therefore the performance does not consider any overheads such as importing the data, as this is not relevant to the performance in CUDA. The table shows the grid dimensions relative to the block and image dimensions.

Table 1: CUDA Performance

Image resolution	grid size	Thread block size	Time (ms)
3328x4096	104x128	32x32x1	115
3328x4096	208x256	16x16x1	114
3328x4096	416x512	8x8x1	188
3328x4096	832x1023	4x4x1	580
3328x4096	1664x2046	2x2x1	2110
2662x3277	83x103	32x32x1	75
2662x3277	166x205	16x16x1	74
2662x3277	333x410	8x8x1	121
2662x3277	666x820	4x4x1	388
2662x3277	1331x1639	2x2x1	1362
1997x2458	63x77	32x32x1	42
1997x2458	125x154	16x16x1	42
1997x2458	250x308	8x8x1	69
1997x2458	500x615	4x4x1	223
1997x2458	999x1229	2x2x1	768
799x983	25x31	32x32x1	9
799x983	50x62	16x16x1	8
799x983	100x123	8x8x1	12
799x983	200x246	4x4x1	38
799x983	400x492	2x2x1	134
144x177	5x6	32x32x1	2
144x177	9x12	16x16x1	2
144x177	18x23	8x8x1	2
144x177	36x45	4x4x1	2
144x177	72x89	2x2x1	5

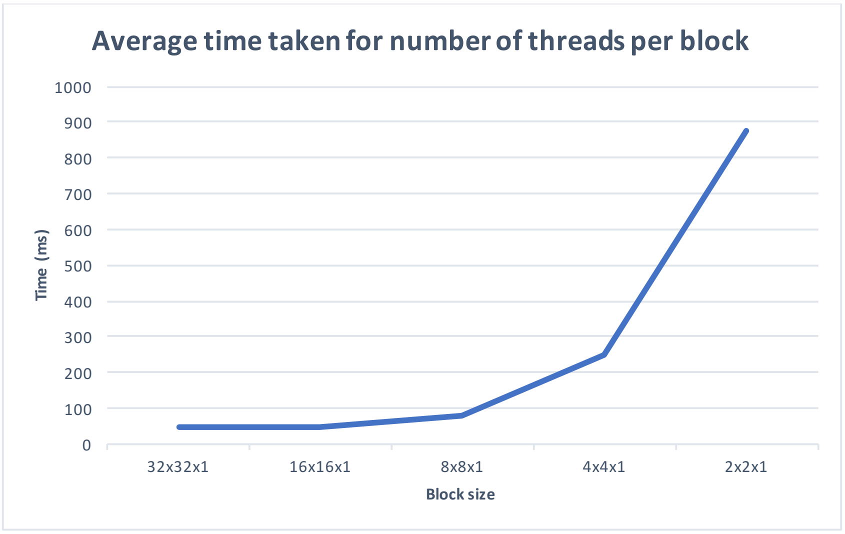

Fig. 11. Effect of increase in block size on performance.

In order to make it easier to plot, The average performance for each thread dimension was calculated. Figure 11 shows the effect of increasing the number of threads per block. This shows that with a higher number of threads per block, the performance of the algorithm increases while the number of blocks per grid decreases. This is because the threads inside the same block can communicate more efficiently. Figure 12 shows the effect of increasing the amount of blocks per grid on the performance of the algorithm. This shows an exponential increase of performance due to an increase in the grid size.

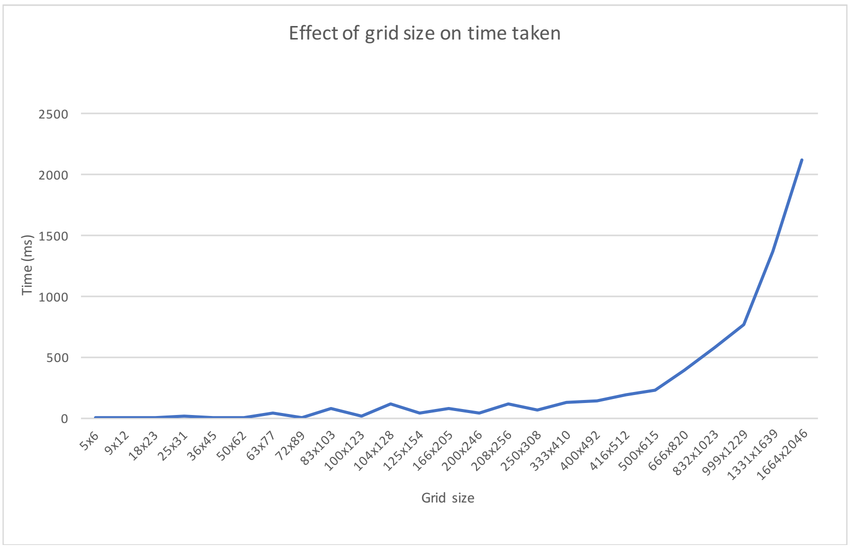

Fig. 12. Effect of grid size on the performance.

For the MPI algorithm, the experiment was repeated for different numbers of processes launched. It was decided to run the experiment with 4, 8, 16, 32 and 64 processes, for every image resolution. Table 2 contains the MPI experiment results, showing the performance of the algorithm in relation to the image resolution and number of processes that are running. The timer was started just before the MPI_Scatter() call and was ended just after the MPI_Gather() call. Therefore the performance does not consider any overheads such as importing the data, as this is not relevant to the performance of the MPI algorithm.

The average performance for each number of processes was calculated, in order to make it easier to visualise on a graph. Figure 13 shows the effect of the number of processes on the performance of the MPI algorithm. As can be seen, the performance of the algorithm peaks at around 8 processes, therefore, There is no need to launch more than around 10 processes, as the performance does not increase any further. The optimal number of processes is therefore ~8 processes.

Table 2: MPI Performance

image size	kernel size	No. Processes	Time (ms)
3328x4096	7	4	4106
3328x4096	7	8	3218
3328x4096	7	16	3337
3328x4096	7	32	3387
3328x4096	7	64	3466
2662x3277	7	4	2758
2662x3277	7	8	2054
2662x3277	7	16	2136
2662x3277	7	32	2153
2662x3277	7	64	2128
1997x2458	7	4	1575
1997x2458	7	8	1156
1997x2458	7	16	1226
1997x2458	7	32	1230
1997x2458	7	64	1195
799x983	7	4	249
799x983	7	8	184
799x983	7	16	184
799x983	7	32	195
799x983	7	64	207
144x177	7	4	8
144x177	7	8	6
144x177	7	16	8
144x177	7	32	9
144x177	7	64	14

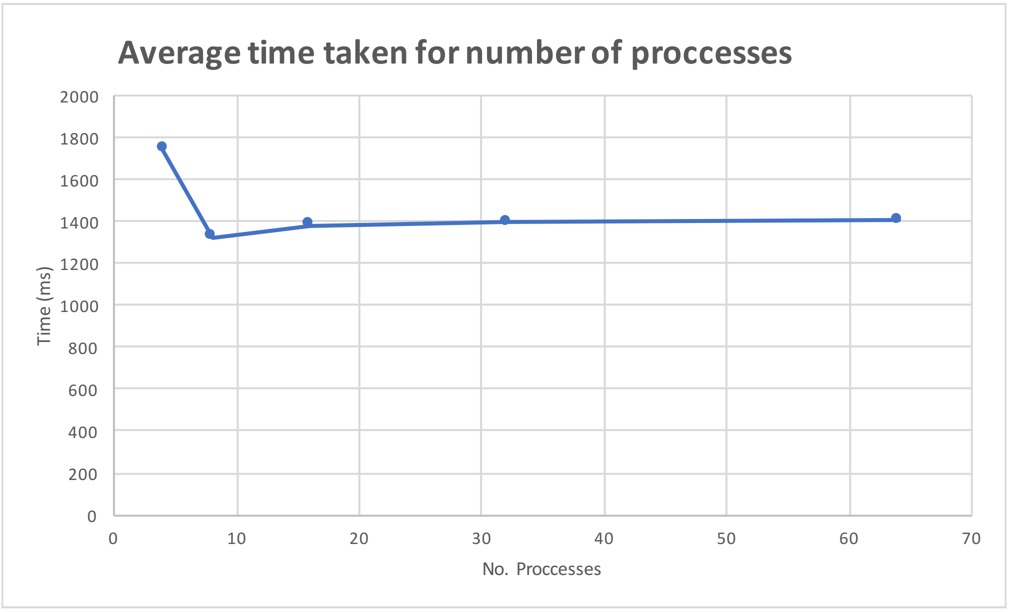

Fig. 13. Effect of number of processes on the performance.

In order to compare the MPI performance with the CUDA performance, the average and best performance for each image size was calculated, for both algorithms. The percentage increase in performance when using CUDA instead of MPI has also been included in the table to she the difference between the performance of the two algorithms. This can be seen in Table 3. We can see that the CUDA performance is always better than the MPI performance, with a 96% increase compared to MPI.

Table 3: CUDA and MPI comparison

Image resolution	Average CUDA time	Average MPI time	Best CUDA time	Best MPI time	CUDA % faster
3328x4096	621.4	3502.8	114	3218	96.45742697
2662x3277	404	2245.8	74	2054	96.39727361
1997x2458	228.8	1276.4	42	1156	96.36678201
799x983	40.2	203.8	8	184	95.65217391
144x177	2.6	9	2.6	6	56.66666667

The best performance for each image was plotted on the graph in Fig 14. As can be seen, when the image resolution is high, the performance of the CUDA algorithm is much better than the MPI algorithm. However, as the image resolution becomes smaller, the two algorithms begin to perform on a similar level, as the lines get closer together. This graph shows a convergence of the performance of the algorithms.

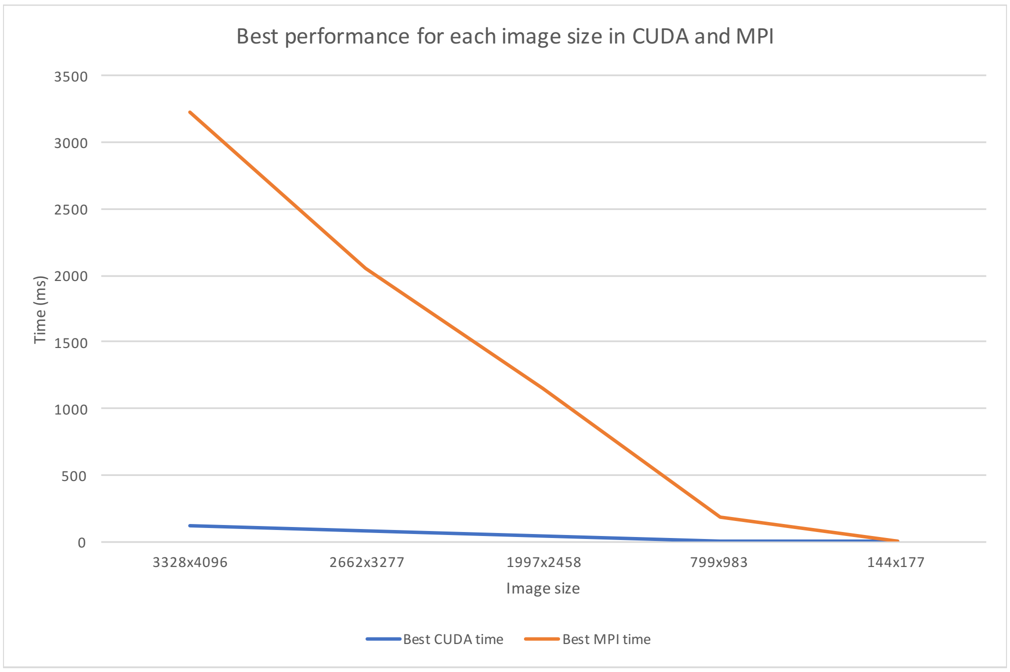

Fig. 13. Comparisons of best performance of CUDA and MPI.

We also need to compare the effect of the size of Gaussian kernel on the performance of each algorithm. In order to do this, the image resolution of 1997x2458 was selected. For CUDA, we select the best performing the block dimensions, which are 32x32x1. For MPI, we select the best performing number of processes, which was 8 processes. This was done in order to be able to compare the two algorithms at their peak performance. Seven experiments were run, each with a different kernel size value. Table 4 shows the results of these experiments.

Table 3: Effect of Gaussian kernel size on performance

Kernel size	MPI Time (ms)	CUDA Time (ms)	CUDA % faster
3	151	20	86.75496689
5	539	29	94.61966605
7	1156	42	96.36678201
9	2060	62	96.99029126
11	3142	87	97.23106302
13	4570	117	97.43982495
15	6188	152	97.54363284

There performance of the both algorithms in relation to the size of the Gaussian kernel were plotted on the graph in Fig. 14. This graph shows that the time taken for completion of the CUDA algorithm is very similar to the time taken by MPI. However, as the kernel size increases, the MPI algorithm becomes slower and it's performance decreases. Whereas the CUDA performance does not change as drastically.

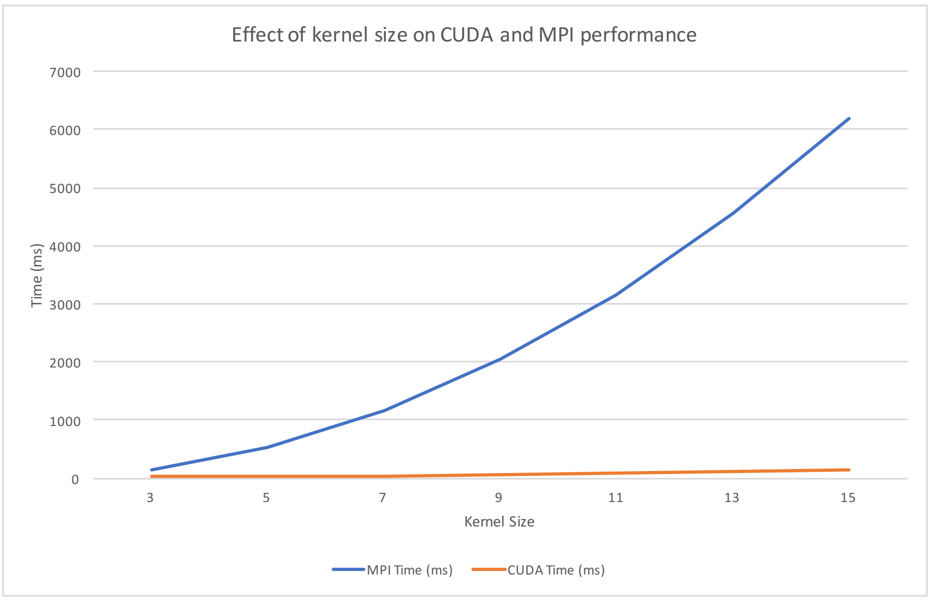

Fig. 13. Effect of kernel size on performance

V. Conclusion

This report has shown that Gaussian blurring can be carried out in a parallel way using both CUDA and MPI. In this project both algorithms were applied to different sized images and the results recorded. Different metrics were considered, such as the time taken to complete, block and grid dimensions for CUDA, number of processes for MPI, and the size of the Gaussian kernel. The first conclusion we can make is that the dimensions of the input image are in correlation with the CUDA grid and block dimensions.

Using MPI is always a cheaper and more portable solution, as it runs on CPU rather than having to own a specialised GPU to run CUDA. This means that it can be used by a wider range of users as most people do not own a GPU. Therefore, we will be looking to use MPI where possible, unless the performance of CUDA greatly exceeds the performance of MPI, which is mostly the case here.

As we can see in Fig. 12, when we have an image with a relatively small resolution (144x177), the performance of the two algorithms are very similar. However as we increase the image size, MPI begins to preform much worse than CUDA. A similar case can be seen on Fig. 13, where the performance of MPI and CUDA is very similar when we use a very small Gaussian kernel. However, using a small Gaussian kernel is not very effective at blurring the image, unless we have an image with a small resolution.

Although the CUDA algorithm is always faster than MPI by at least 56%, when the image resolution is small enough, a large percentage difference realistically would not make much of a difference in the real world. This is because the difference between 3ms and 6ms is much less noticeable than the difference between, say, 3000ms and 6000ms.

After taking all these findings above into consideration, we can conclude that the decision of whether to use MPI or CUDA for Gaussian blurring depends of the dimensions of the image we would like to filter. If the image is small enough, say smaller than 1000x1000px, it would be best to use MPI as, even if it is slower, the cost of running CUDA on a GPU is much greater. However, if we are filtering large high resolution images, which is usually the case for digital mammogram scan images, then CUDA greatly outperforms MPI. Therefore, the final conclusion can be made that, for this particular project of digital mammography, we should always use CUDA in order to perform a Gaussian blur using parallel computation.

References

1. En.wikipedia.org. (2018). Gaussian blur. [online] Available at: https://en.wikipedia.org/wiki/Gaussian_blur [Accessed 20 Jan. 2018].
2. Pixelstech.net. (2018). Gaussian Blur Algorithm | Pixelstech.net. [online] Available at: https://www.pixelstech.net/article/1353768112-Gaussian-Blur-Algorithm [Accessed 23 Jan. 2018].

3. Shapiro, L. and Stockman, G. (2001). Computer vision. Upper Saddle River,NJ: Prentice Hall.
4. Lodev.org. (2018). LodePNG. [online] Available at: http://lodev.org/lodepng/ [Accessed 17 Jan. 2018].
5. Bozkurt, F., Yağanoğlu, M. and Günay, F. (2015). Effective Gaussian Blurring Process on Graphics Processing Unit with CUDA. International Journal of Machine Learning and Computing, 5(1), pp.57-61.
6. Mpitutorial.com. (2018). MPI Broadcast and Collective Communication · MPI Tutorial. [online] Available at: http://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/ [Accessed 22 Jan. 2018].
7. Mpitutorial.com. (2018). MPI Scatter, Gather, and Allgather · MPI Tutorial. [online] Available at: http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/ [Accessed 26 Jan. 2018].