Parallelization of Convolution Using MPI, Pthreads and CUDA

COMP5055: Final Project Report

Roy F. Cruz

June 22, 2023

1 Introduction

Convolution of images forms an ever-increasing part in many areas of research in computer science and physics. As an example of the latter, in the CMS collaboration at CERN convolution of detector data is a fundamental part of the application of machine learning in the attempt to automate the process of data quality management [1]. Given the high amounts of data involved in these efforts, it is absolutely essential to optimize every part of the process [2]. One useful approach to do this is with the parallelization of the code used. In this report, we study such an approach towards the convolution of data. The convolution algorithm considered here is edge detection.

Edge detection is a fundamental convolution algorithm frequently implemented in image processing. As its name suggests, its aim is to highlight the boundaries of objects in an image. A common way of detecting edges is with the use of gradient estimating convolution filters which, when applied to an image, gives an estimate of the gradient magnitude at each point of the image. The Prewitt operators [3] are a set of two 3×3 convolution kernels which, when applied to an image, can do just that. These operators are shown below.

$$P_x = \begin{pmatrix} 1.0 & 0.0 & -1.0 \\ 1.0 & 0.0 & -1.0 \\ 1.0 & 0.0 & -1.0 \end{pmatrix}, \qquad P_y = \begin{pmatrix} 1.0 & 1.0 & 1.0 \\ 0.0 & 0.0 & 0.0 \\ -1.0 & -1.0 & -1.0 \end{pmatrix} \tag{1}$$

In this report, we present the implementation of an edge detection convolution using the aforementioned operators[1] on a set of TIFF images. For testing purposes, the images were sourced from the Dstl Satellite Imagery Feature Detection Competition data set[2]. Out of these images, a subset of 25 images were selected in order to obtain an average convolution time for each of the implementations, of which there were four: serially and then parallelized using MPI, Pthreads and then CUDA. The performance results and comparisons, as well as drawbacks and other details about each of these implementations, are discussed in this report.

2 Serial Implementation

In this implementation, the convolution of each TIFF image was employed in series, meaning that the pixels were convoluted sequentially or one after the other. Below we show the snippet of code in this implementation which forms the heart of the convolution algorithm.

[1]Various attempts were made to instead use a different gradient filter called Laplacian of Gaussian (LoG) which is given by the matrix

$$H(\delta) = \begin{pmatrix} 1 & 2 & 1 \\ 2 & -16 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

However, these attempts failed as the processed images did not highlight the edges present in the original image. It was then decided that the Prewitt operators would be used, given that they serve the same function and that, although they would not be easier to implement, it is certainly more immediately understandable how they work.

[2]Hosted on Kaggle. Link: `https://www.kaggle.com/competitions/dstl-satellite-imagery-feature-detection`

Listing 1: Convolution function parallelized using MPI

```
// Loop over input image pixels
for (int i = 0; i < height; i++) {
    for (int j = 0; j < width; j++) {
        float grad_x = 0.0f;
        float grad_y = 0.0f;

        // Loop over kernel elements
        for (int k = -1; k <= 1; k++) {
            for (int l = -1; l <= 1; l++){

                // Get pixel coodinates in the input image
                int x = j + l;
                int y = i + k;

                // Clamp the coordinates to the image boundaries
                x = std::max(0, std::min(x, (int)width - 1));
                y = std::max(0, std::min(y, (int)height - 1));

                // Get pixel value in the input image and convert to gray
                    scale
                uint32_t pixel = inbuff[y * width + x];
                uint8_t r = TIFFGetR(pixel);
                uint8_t g = TIFFGetG(pixel);
                uint8_t b = TIFFGetB(pixel);

                // Compute luminance
                float gray = 0.2126f * r + 0.7152f * g + 0.0722f * b;

                // Multiply pixel value with corresponding kernel element
                    and add to the gradient values
                grad_x += gray * kernel_x[(k + 1) * 3 + (l + 1)];
                grad_y += gray * kernel_y[(k + 1) * 3 + (l + 1)];
            }
        }

        // Compute and store conv pixel value
        float grad = sqrt(grad_x * grad_x + grad_y * grad_y);
        grad = 255 * sigmoid(grad);
        uint8_t edge = (uint8_t)grad;
        outbuff[i * width + j] = (edge << 24) | (edge << 16) | (edge <<
           8) | (255);
    }
}
```

Before the loop presented in the above code, the image data was loaded into the memory buffer inbuff, which is (a pointer to) an array of type uint32_t of size width $*$ height. During the course of the loop, the luminance of each of the pixel was computed, and the result was multiplied with the corresponding element in both kernels. The result of both of these multiplications was then added to accumulators grad_x and grad_y. After this process, the gradient magnitude was estimated and, given that the resulting number ought to be between 0 and 255, the result was fed into the sigmoid function and multiplied by 255. This final number was then stored in outbuff.

As expected, this implementation was one of the slowest of them all, coming in with an average convolution time per image of $t_{serial} = 2.48829$ s. The reason for this is simple: each pixel is convoluted one at a time (i.e., sequentially). This was the biggest drawback to this implementation, and thus a motivating factor towards parallelizing the code. Given this, we took t_{serial} to be the baseline time against which all other results shown in the sections to come would be compared.

3 MPI Implementation

For the parallelization of the convolution algorithm in MPI, the starting idea was to somehow distribute parts of the image data to each process, each of which would then apply a convolution on the part of the data it received before sending it back to the 0'th process. However, given that calls to data sharing functions such as MPI_Send and MPI_Recv are labor-intensive for the CPU, specially when it involves such a large amount of data, this idea was put into question. Moreover, the data transfer problem would have been worsened by the fact that the convolution of one pixel requires access to all the surrounding pixels. These observations, paired with the tediousness involved in implementing it, led to the conclusion that this approach was not feasible. To get around all of this, a shared data approach was selected, in which all the processes would have full access to a singular copy of the image data. Although this would create the risk of one process overextending into a domain of the picture it was not assigned, this problem that was worth tackling given the aforementioned alternative.

In order to apply all of this, the pointer to the tiff data was shared to each of the processes and then a memory window was created which contained the input buffer (unto which the image data was loaded) and the output buffer. Moreover, each process, using the size of the communicator as well as its rank and the dimensions of the image, computed the range of rows it would perform the convolution on. The logic applied to do this was simple: the image is divided vertically into an amount of sections that is divisible by the number of processes, and if this is not possible, any leftover rows were given to the last process. Although this is simple, because the last process can get an amount of rows assigned to it higher than the other processes, this introduces a bottleneck in performance given that it can cause the convolution to stall while the final process finished processing its additional rows.

Another glaring issue with this implementation is that, given that you can only assign as many cores as the CPU has, it is very hardware dependent[3]. This issue does not only affect general performance enhancements, but it also impacted the data taking ability of this analysis as the system used for testing was an M1 MacBook Pro which had just 8 cores.

Below we show a snippet of code which consists of the MPI parallelization of the code shown in Listing 1. Note that in the following code, what we will usually call inbuff will be shmptr_in (i.e. "shared memory pointer input") and what we usually call outbuff will be shmptr_out (i.e. "shared memory pointer output"). The difference in names is just a result of the memory window creation process. Moreover, to avoid redundancy, some parts that have no substantial differences with the serial code are omitted.

Listing 2: Convolution loop for serial impementation

```
// Process computes the rows its will apply convolution to.
int my_numrows = height/comm_sz;
int my_startrow = my_numrows * my_rank;
int rowsleft = height % comm_sz;
if ((my_rank == (comm_sz - 1)) & (rowsleft != 0))
    my_numrows += rowsleft;
int my_endrow = my_startrow + my_numrows;

// Process applies convolution to its assigned domain of the image
for (int i = my_startrow; i < my_endrow; i++) {
    for (int j = 0; j < width; j++) {
        // Gradient value initialization.
        ...
        // Loop over kernel elements
        for (int k = -1; k <= 1; k++) {
            for (int l = -1; l <= 1; l++){
                // Get pixel coodinates in the input image
```

[3]Curiously enough, through testing in one of the systems, it was found that some implementations of MPI allow you to use a number of processes which is higher as the amount of cores actually available. However, after the amount of processes used was higher than the amount of cores in the system used, there was no appreciable change in performance.

```
                    ...
                    // Clamp the coordinates to the image boundaries
                    ...
                    // Get pixel value in the input image
                    uint32_t pixel = shmptr_in[y * width + x];
                    ...
                    // Compute luminance
                    ...
                    // Multiply pixel value with corresponding kernel element
                        and add to the gradient values
                    ...
                }
            }
            // Compute magnitude of gradient
            ...
            //Restrict value to [0, 255] using sigmoid
            ...
            //Convert magnitude to integer value
            ...
            // Set output pixel value to the edge value in all channels;
            shmptr_out[i * width + j] = (edge << 24) | (edge << 16) | (edge
                << 8) | (255);
        }
    }
MPI_Barrier(nodecomm);
```

4 Pthreads Implementation

Using Pthreads, we were also able to parallelize the process of applying the Prewitt operators to the set of test TIFF images. Given how Pthreads works, however, the implementation had substantial differences compared to the MPI implementation, although it was based on the same logic of distributing the rows among the threads and, if not possible, giving the last thread all the remaining rows.

In order to use Pthreads, unlike with MPI, conv_tiff was not invoked directly given that we first needed to create the threads. Therefore, the function pthread_create was called first which created a given number of threads which would then run a function called thread_proc in which each thread would initialize all the necessary variable to perform the convolution (if it was thread 0, it would also allocate the input buffer). Note, however, that in this implementation all necessary variables for the convolution were stored in a **struct**. This included the inbuff and outbuff pointers, the image width and height, the number of threads, the Prewitt operators as well as other supplementary variables. The definition of this **struct** is shown below.

Listing 3: Struct used in the Pthreads implementation

```
struct thread_args {
    uint32_t *inbuff;
    uint32_t *outbuff;
    uint32_t width;
    uint32_t height;

    TIFF* tiff_point;

    int thread;
    long thread_count;

    // Prewitt operators (i.e. kernels) for edge detection (i.e. gradient
        magnitude calculation)
    const float edge_x[9] = {
        1.0, 0.0, -1.0,
        1.0, 0.0, -1.0,
        1.0, 0.0, -1.0
    };

    const float edge_y[9] = {
        1.0, 1.0, 1.0,
        0.0, 0.0, 0.0,
        -1.0, -1.0, -1.0
    };

    int done_creating_flag = 0;
    int done_conv_num = 0;

    std::chrono::time_point<std::chrono::high_resolution_clock> start,
      end;
    std::chrono::nanoseconds duration;
    float time_avg;
};
```

The reason this was found to be a convenient way to handle these variables was because the function thread_proc only accepts one variable, which has to be a pointer to a **void**. So, because the initialization and convolution process involved multiple variables, it was decided that the most straightforward way of handling all of these variables was to place them in the **struct** shown above. However, note that this decision may not have been ideal. This is because once a thread finished initializing in the function thread_proc, then and only then would the next thread be able to start initializing its variables. This introduces a bottleneck to the overall performance of the program, given that the threads are initialized in a serial fashion. This, however, was of no relevance to this analysis given that the focus was on the performance of the convolution loop, not on the program as a whole.

Once all the threads were initialized, they all had access to the **struct** thread_args type variable named ta. The following snippet of code shows the part of the program where the actual convolution takes place. Note that, equivalently to how it was done for MPI, all the threads had complete access to the input buffer, but each would only do the convolution on a particular set of rows of the image. As was mentioned before, this set of rows was determined in the exact same way as it was done for the MPI implementation.

Listing 4: Convolution function made for the Pthreads implementation

```
void conv_tiff(
        struct thread_args *ta, /* in  */
        int my_startrow,        /* in  */
        int my_endrow           /* in  */) {

    for (int i = my_startrow; i < my_endrow; i++) {
        for (int j = 0; j < ta->width; j++) {
            // Gradient value initialization.
            ...
            // Loop over kernel elements
            for (int k = -1; k <= 1; k++) {
                for (int l = -1; l <= 1; l++){
                    // Get pixel coodinates in the input image
                    ...
                    // Clamp the coordinates to the image boundaries
                    ...

                    // Get pixel value in the input image
                    uint32_t pixel = ta->inbuff[y * ta->width + x];
                    ...
                    // Compute luminance
                    ...
                    // Multiply pixel value with corresponding kernel
                        element and add to the gradient values
                    grad_x += gray * ta->edge_x[(k + 1) * 3 + (l + 1)];
                    grad_y += gray * ta->edge_y[(k + 1) * 3 + (l + 1)];
                }
            }
            // Compute magnitude of gradient
            ...
            // Restrict value to [0, 255] using sigmoid
            ...
            // Convert magnitud to integer value
            ...
            // Set output pixel value to the edge value in all channels;
            ta->outbuff[i * ta->width + j] = (edge << 24) | (edge << 16)
                | (edge << 8) | (255);
        }
    }
} /* conv_tiff */
```

5 CUDA Implementation

In many ways, the CUDA implementation was identical to the MPI implementation in terms of its basic functionality. However, there were some important differences which originated from the structure of Nvidia GPUs and from how CUDA worked in general. As a primary example of the similarities, similar to when the MPI implementation was being developed, the option of sending sections of the image data to each streaming multiprocessor (SM) for each of them to work on separately was considered, but was eventually discarded. However, it can be argued for this implementation that this decision was less than ideal. The reason for this is that, if we are trying to maximize the speed-up of just the actual convolution process, it might be best for the data to already be in the GPU before the convolution starts. This way, the streaming processors would not have to wait to receive the data from the host memory before doing the convolution. In this implementation, to keep things simple, the choice was made to allocate a unified memory which the CPU and GPU both had access to. To do this, the CUDA function cudaMallocManaged was used, which allowed the passing of the input buffer and output buffer pointers to the kernel as if it were a regular function performed by the host. This is illustrated in the snippet of code shown below in Listing 6.

Listing 5: Part of the code in which the unified memory was allocated and the kernel called

```
for (int j = 0; j < tiff_points.size(); j++) {
        int width, height;
        uint32_t* inbuff;
        uint32_t* outbuff;

        // Read image parameters from tiff
        TIFFGetField(tiff_points[j], TIFFTAG_IMAGELENGTH, &height);
        TIFFGetField(tiff_points[j], TIFFTAG_IMAGEWIDTH, &width);

        // Allocate memory in host and GPU
        cudaMallocManaged(&inbuff, height * width * sizeof(uint32_t));
        cudaMallocManaged(&outbuff, height * width * sizeof(uint32_t));

        // Read image data from tiff
        TIFFReadRGBAImage(tiff_points[j], width, height, inbuff, 0);

        ...

        // Start convolution in GPU
        conv_tiff<<<blk_ct, th_per_blk>>>(inbuff, outbuff, width, height)
            ;
        cudaDeviceSynchronize();

        ...
```

With this set up, the matter of having the SP convolve the different parts of the image was simple. As before, the logic of dividing the image into equal amount of rows was done, with any leftover rows assigned to the last SP. In this case, the implementation of this logic had to be adjusted somewhat given that the threads are organized into blocks, each of which is separate from each other. In the following piece of code, the initialization of the start and end rows are shown, as well as the loop in which the SP did the actual convolution of its assigned domain of the data is shown.

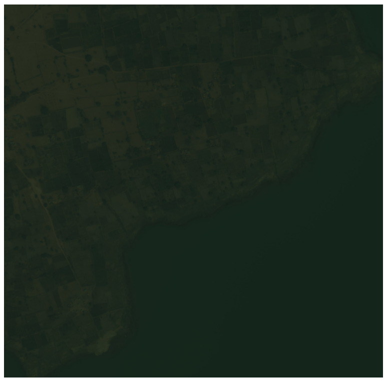
Listing 6: Part of the code in which the unified memory was allocated and the kernel called

```
int my_index = blockIdx.x * blockDim.x + threadIdx.x;
    int thread_count = gridDim.x * blockDim.x;
    int my_numrows = height / thread_count;
    int my_startrow = my_numrows * my_index;
    int rowsleft = height % thread_count;
    if ((my_index == (thread_count - 1)) & (rowsleft != 0))
        my_numrows += rowsleft;
    int my_endrow = my_startrow + my_numrows;

    for (int i = my_startrow; i < my_endrow; i++) {
        for (int j = 0; j < width; j++) {
            // Gradient value initialization.
            ...
            // Loop over kernel elements
            for (int k = -1; k <= 1; k++) {
                for (int l = -1; l <= 1; l++){
                    // Get pixel coodinates in the input image
                    ...
                    // Clamp the coordinates to the image boundaries
                    ...
                    // Get pixel value in the input image
                    ...
                    // Convert pixel value to grayscale
                    ...
                    // Multiply pixel value with corresponding kernel
                        element and add to the gradient values
                    ...
                }
            }
            // Compute magnitude of gradient
            ...
            //Restrict value to [0, 255] using sigmoid
            ...
            //Convert magnitud to integer value
            ...
            // Set output pixel value to the edge value in all channels;
            ...
        }
    }
```

Figure 1: Example of result from convolution

(a) **6030_3_4.tif**



(b) **6030_3_4_conv.tiff**

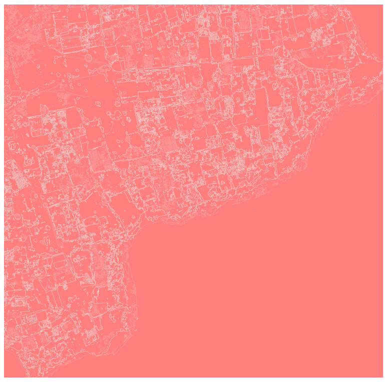
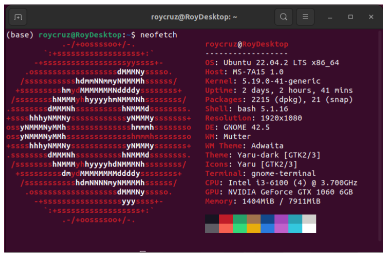


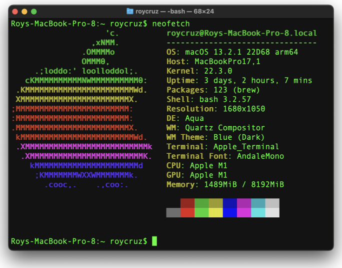
Figure 2: Specifications of the systems used during testing

(a) **6030_3_4.tif**



```
(base) roycruz@RoyDesktop:~$ neofetch
            .-/+oossssoo+/-.               roycruz@RoyDesktop
        `:+ssssssssssssssssss+:`           ------------------
      -+ssssssssssssssssssyyssss+-         OS: Ubuntu 22.04.2 LTS x86_64
    .ossssssssssssssssssdMMMNysssso.       Host: MS-7A15 1.0
   /ssssssssssshdmmNNmmyNMMMMhssssss/      Kernel: 5.19.0-41-generic
  +ssssssssshmydMMMMMMMNddddyssssssss+     Uptime: 2 days, 2 hours, 41 mins
 /sssssssshNMMMyhhyyyyhmNMMMNhssssssss/    Packages: 2215 (dpkg), 21 (snap)
.ssssssssdMMMNhsssssssssshNMMMdssssssss.   Shell: bash 5.1.16
+sssshhhyNMMNyssssssssssssyNMMMysssssss+   Resolution: 1920x1080
ossyNMMMNyMMhsssssssssssssshmmmhssssssso   DE: GNOME 42.5
ossyNMMMNyMMhsssssssssssssshmmmhssssssso   WM: Mutter
+sssshhhyNMMNyssssssssssssyNMMMysssssss+   WM Theme: Adwaita
.ssssssssdMMMNhsssssssssshNMMMdssssssss.   Theme: Yaru-dark [GTK2/3]
 /sssssssshNMMMyhhyyyyhdNMMMNhssssssss/    Icons: Yaru [GTK2/3]
  +sssssssssdmydMMMMMMMMddddyssssssss+     Terminal: gnome-terminal
   /ssssssssssshdmNNNNmyNMMMMhssssss/      CPU: Intel i3-6100 (4) @ 3.700GHz
    .ossssssssssssssssssdMMMNysssso.       GPU: NVIDIA GeForce GTX 1060 6GB
      -+sssssssssssssssssyyyssss+-         Memory: 1404MiB / 7911MiB
        `:+ssssssssssssssssss+:`
            .-/+oossssoo+/-.
```

(b) **6030_3_4_conv.tiff**



```
Roys-MacBook-Pro-8:~ roycruz$ neofetch
                    'c.          roycruz@Roys-MacBook-Pro-8.local
                 ,xNMM.          --------------------------------
               .OMMMMo           OS: macOS 13.2.1 22D68 arm64
               OMMM0,            Host: MacBookPro17,1
     .;loddo:' loolloddol;.      Kernel: 22.3.0
   cKMMMMMMMMMMNWMMMMMMMMMM0:    Uptime: 3 days, 2 hours, 7 mins
 .KMMMMMMMMMMMMMMMMMMMMMMMWd.    Packages: 123 (brew)
 XMMMMMMMMMMMMMMMMMMMMMMMX.      Shell: bash 3.2.57
;MMMMMMMMMMMMMMMMMMMMMMMM:       Resolution: 1680x1050
:MMMMMMMMMMMMMMMMMMMMMMMM:       DE: Aqua
.MMMMMMMMMMMMMMMMMMMMMMMMX.      WM: Quartz Compositor
 kMMMMMMMMMMMMMMMMMMMMMMMMWd.    WM Theme: Blue (Dark)
 .XMMMMMMMMMMMMMMMMMMMMMMMMMMk   Terminal: Apple_Terminal
  .XMMMMMMMMMMMMMMMMMMMMMMMMK.   Terminal Font: AndaleMono
    kMMMMMMMMMMMMMMMMMMMMMMd     CPU: Apple M1
     ;KMMMMMMMWXXWMMMMMMMk.      GPU: Apple M1
       .cooc,.    .,coo:.        Memory: 1489MiB / 8192MiB


Roys-MacBook-Pro-8:~ roycruz$
```

6 Results & Performance Comparisons

The implementation discussed in the preceding sections were tested on a subset of 25 images of the data set. As a demonstration of the effectiveness of these implementations in edge detection, Figure 1 shows side by side the original image and the convoluted image. In this example, the convolution was done by the Pthreads implementation, but each implementation produced identical results. Note that, as intended, the convoluted image highlights the edges of the original image.

In order to ascertain the performance of these implementations, they were tested using a subset of the original data [4]. Using these images with the serial implementation, an average convolution time per image of $T_{\text{serial}} = 2.48829$ was obtained. All the times obtained for the rest of the implementations are shown in Table A.1. Because the focus of this report is the speedup ($S(p)$) and efficiency ($E(p)$) of the parallel implementations, these quantities were computed and visualized using the Matplotlib Python package. The definition of speedup and efficiency are

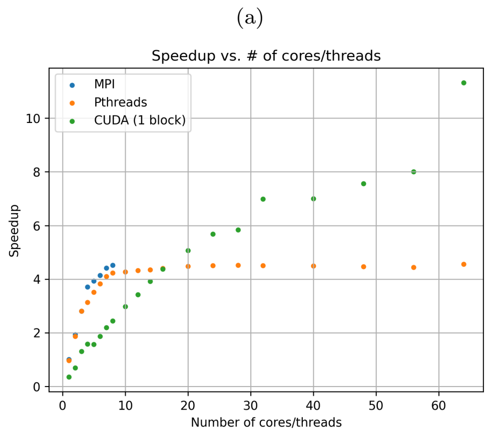
$$S(p) = \frac{T_{\text{serial}}}{T_{\text{parallel}}} \qquad E(p) = \frac{S(p)}{p}, \tag{2}$$

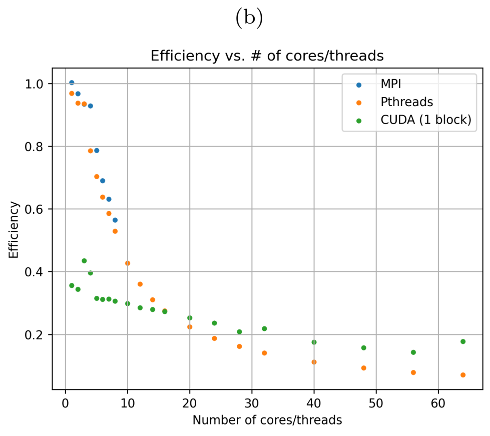
where p is the number of processes or threads used. The tables showing the numerical results of the speedup and efficiency can be seen in Table A.2 and Table A.3 respectively. Note that the serial, MPI and Pthreads implementations were tested on an M1 2020 MacBook Pro while the CUDA implementation was tested on a PC running Ubuntu. All the specifications of these systems are shown in the Figure 2.

With the test results, we began by comparing the speedup and efficiency of the MPI, Pthreads and CUDA (1 block) implementations. In Figure 3, the plots showing all of these cases' speedup and

[4]To be exact, the files used were **6030_i_j.tif**, where **i**, **j** $\in [0, 4]$.

Figure 3: Speedup and Efficiency Plots for MPI, Pthreads and CUDA (1 block)





efficiency are shown. In the case of speedup, we can clearly see that, for low p values, MPI and Pthreads surpass CUDA by a non-trivial amount. In fact, for values close to $p = 1$, CUDA took even longer to convolve each image on average compared to the serial implementation, as we can see from the efficiency values lower than 1. This was quite an unexpected result, but it can be easily attributed to the overhead contributed by the SPs accessing the unified memory where the input buffer was stored. For higher values of p, this lag becomes negligible, as the amount of time it takes for the SPs to do the convolution becomes less and less. Eventually, at $p > 16$, CUDA overtakes Pthreads in terms of speedup as the latter plateus at $S \approx 4.5$. This aforementioned behavior presented by Pthreads is something that MPI seems to exhibit as well: for higher values of p, MPI's speedup increase slows down. It seems to plateau as well, but this can not be said to happen with certainty given the limited amount of data available for this implementation.

In terms of the efficiency, we can see from Figure 3.b that for low values of p, MPI and Pthreads beat out CUDA by a substantial amount. For higher values, however, CUDA ends up being the most efficient implementation. This result is reasonable given that, for MPI and Pthreads, throwing higher amount of cores/threads to the problem leaves the speedup relatively unchanged for higher values of p, but for CUDA, more SPs results in ever-increasing speedup. This is not to say that the efficiencies of these implementations are particularly good. As we can see, they all end up at $E < 0.2$, which in itself is a low efficiency.

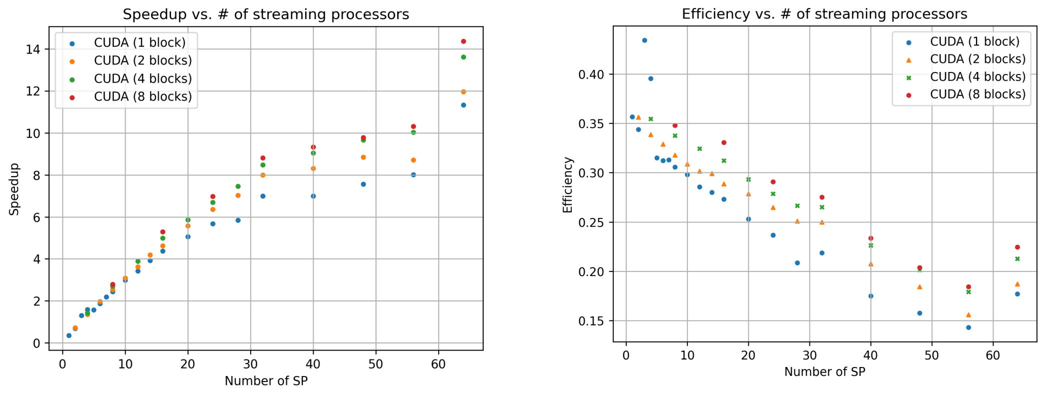
There are two noticeable details with the plots just discussed, both involving CUDA. Firstly, we can see in Figure 3.a that at $p \approx 4$ there is a sudden and non-negligible increase in speedup. A similar sudden increase can be seen for $p = 32$ and for $p = 64$. The reason for this sudden jump in S is not exactly known, but given that it happened at all but one values of p which are multiples of 4, it can be hypothesized that it has something to do with the way the rows are distributed among the SPs. It may be that at those values, the amount of rows that are left over minimizes and so the convolution does not stall while the last threat finishes processing its additional rows. This idea is not perfect, though, as it can be immediately refuted by the fact that this speedup boost did not happen for $p = 16$.

The second noticeable feature in these graphs can be seen in Figure 3.b, and it also involves CUDA. For this implementation, we can see a stark jump in efficiency for $p = 3$ and a subsequent drop at $p = 5$. This spike is clearly related to the boost in speedup discussed before, given that it happens at the same values of $p \approx 4$. A smaller, but still appreciable increase in efficiency can also be observed at $p = 32$ and $p = 64$, further supporting this claim.

Given how fundamentally different the CUDA implementation was because it involved an additional piece of specialized hardware (i.e., the GPU) and not just the CPU, it was decided that it would be worthwhile to compare the CUDA implementation applied to the same data, but with differing amounts of blocks (i.e., streaming multiprocessors or SMs). In the Figure 4 the speedup and efficiency graphs of these tests are shown. Looking first at the speedup plot, we see that for small amounts of SPs the difference in speedup provided by the selection of different amount of blocks is negligible. However, for larger amounts of SPs, the $S(p)$ of each selection of amount of blocks diverges, and we can see that for the higher the amount of blocks selected, the higher the speedup. It should be noted that

Figure 4: Speedup and Efficiency Plots for CUDA tests (1 to 8 blocks)

(a) (b)



the speedups involved here are vastly more significant than the ones observed for MPI and Pthreads. This is because the GPU is specialized in floating point operations [4] and can thus perform them at a much higher speed than the CPU can.

The observed difference in $S(p)$ for higher amount blocks is peculiar: if we are using the same amount of SPs, why would the amount of blocks has any effect on the speedup? It can be hypothesized that these discrepancies are fundamentally caused by limits in data transfer. Each SM has a limited amount of bandwidth, so if we use just one in our convolution, the amount of data transfer between the host device and this SM is much higher than if we were to use more than one SM. In this latter case, each SM would have to transfer less data to and from the host device, which would result in higher data transfer rates and thus a higher speed in total convolution time.

In Figure 4.b, we can see the efficiency plot of each selection of number of blocks. In this plot we can clearly see that, generally speaking, the higher the amount of blocks selected, the higher the efficiency will be. The only exception to this is for the 2 block selection graph, where we can see the same spike in efficiency we observed in Figure 3 around $p \approx 4$, the result of which is that this selection has higher efficiency than all the others for this value of p. This behavior is highly unusual and unexpected, and further testing and analysis is required in order to come up with a solid hypothesis of why this jump in efficiency was not observed with the other selections of blocks. Notably, however, all the selections of blocks showed roughly the same jump in efficiency at $p = 32$ and $p = 64$, leading some credence to the hypothesis postulated before, which involved the minimization of left over rows.

7 Conclusion

In this report, three different approaches were taken in the attempt to parallelize an edge detection convolution algorithm: MPI, Pthreads and CUDA. From the results presented in the preceding sections, we can conclude that the CUDA implementation showed the most promise in terms of speedup and efficiency, but only for higher values of p. For lower values of p, MPI and Pthreads work just as well if not better than CUDA. It can therefore be argued that, if the processing being done is of a small data set and is not too computationally heavy (i.e., not involving floating point numbers), MPI and Pthreads are better options compared to CUDA, which requires a potentially costly and possibly not readily available Nvidia GPU.

The same arguments presented above apply for the efficiency: for lower labor-intensive applications, MPI and Pthreads work better. It is important to note that, despite this, all the observed efficiencies dropped very rapidly as p increased, regardless of the implementation used. In the worst case, it even went as low as $E_{\text{Pthreads}}(p = 64) = 0.0712$.

All of this being said, it is essential to keep in mind that these results depend a lot on the specifics of how the code was parallelized. The attempts shown in this report were crude attempts at parallelization, which have quite a number of flaws. It is more than possible to do this parallelization in a more sophisticated fashion, which might give improved results for all the implementations. It should also be noted that, these results also heavily depend on the hardware being used: for the MPI and Pthreads, an M1 MacBook was used, while for CUDA a PC using an Intel i3-6100 and an Nvidia GeForce GTX 1060 (6 GB). These are two vastly different systems, and the CPUs they use are not even using the same architecture. In order to improve this analysis, it would be worthwhile to do all of these tests on the same system, using an Nvidia GPU and with a CPU that has a higher number of cores (which would give us more data for MPI)[5].

8 References

[1] A. A. Pol, G. Cerminara, C. Germain, M. Pierini, and A. Seth, "Detector monitoring with artificial neural networks at the cms experiment at the cern large hadron collider," 2018.

[2] P. Shanahan, K. Terao, and D. Whiteson, "Snowmass 2021 computational frontier compf03 topical group report: Machine learning," 2022.

[3] M. Lofroth and E. Avci, "Auto-focusing approach on multiple micro objects using the prewitt operator," *International Journal of Intelligent Robotics and Applications*, vol. 2, no. 4, p. 413–424, 2018.

[4] J. Dong, F. Zheng, W. Pan, J. Lin, J. Jing, and Y. Zhao, "Utilizing the double-precision floating-point computing power of gpus for rsa acceleration," *Security and Communication Networks*, vol. 2017, p. 1–15, 2017.

9 Appendix

Table A.1: Convolution times (in seconds)

# Threads	MPI	Pthreads	CUDA (1 blk)	CUDA (2 blk)	CUDA (4 blk)	CUDA (8 blk)
1	2.48058	2.56887	6.97722			
2	1.2866	1.32765	3.61733	3.48965		
3	0.887549	0.886716	1.90885			
4	0.669921	0.792072	1.57273	1.8363	1.75446	
5	0.632343	0.707099	1.58016			
6	0.600543	0.64986	1.32835	1.26104		
7	0.562903	0.606931	1.1355			
8	0.551143	0.587207	1.01708	0.979043	0.9209	0.893831
10		0.582136	0.834294	0.80609		
12		0.575142	0.726024	0.687501	0.638911	
14		0.572205	0.634959	0.594504		
16		0.564635	0.569288	0.53886	0.497739	0.470484
20		0.556015	0.491495	0.446354	0.424184	
24		0.551441	0.438333	0.391383	0.372041	0.356515
28		0.550463	0.42589	0.354445	0.333499	
32		0.551448	0.355881	0.311404	0.293246	0.282414
40		0.553087	0.355539	0.29968	0.275125	0.266536
48		0.556857	0.32934	0.281171	0.257648	0.254308
56		0.560787	0.310643	0.285489	0.248078	0.241209
64		0.545982	0.219716	0.207923	0.182693	0.173158

[5] An attempt was made to use the LHC Physics Center computing cluster, but the computer available had only 8 cores, the same amount as the M1 MacBook Pro. The idea was therefore discarded, as it was simpler to just run the programs locally.

Table A.2: Convolution speedups

# Threads	**MPI**	**Pthreads**	**CUDA (1 blk)**	**CUDA (2 blk)**	**CUDA (4 blk)**	**CUDA (8 blk)**
1	1.0031	0.9686	0.3566			
2	1.934	1.8742	0.6879	0.713		
3	2.8036	2.8062	1.3036			
4	3.7143	3.1415	1.5821	1.3551	1.4183	
5	3.935	3.519	1.5747			
6	4.1434	3.829	1.8732	1.9732		
7	4.4205	4.0998	2.1914			
8	4.5148	4.2375	2.4465	2.5416	2.702	2.7838
10		4.2744	2.9825	3.0869		
12		4.3264	3.4273	3.6193	3.8946	
14		4.3486	3.9188	4.1855		
16		4.4069	4.3709	4.6177	4.9992	5.2888
20		4.4752	5.0627	5.5747	5.8661	
24		4.5123	5.6767	6.3577	6.6882	6.9795
28		4.5204	5.8426	7.0202	7.4612	
32		4.5123	6.9919	7.9906	8.4853	8.8108
40		4.4989	6.9986	8.3032	9.0442	9.3357
48		4.4685	7.5554	8.8497	9.6577	9.7846
56		4.4371	8.0101	8.7159	10.0303	10.3159
64		4.5575	11.325	11.9674	13.6201	14.3701

Table A.3: Convolution efficiencies

# Threads	MPI	Pthreads	CUDA (1 blk)	CUDA (2 blk)	CUDA (4 blk)	CUDA (8 blk)
1	1.0031	0.9686	0.3566			
2	0.967	0.9371	0.3439	0.3565		
3	0.9345	0.9354	0.4345			
4	0.9286	0.7854	0.3955	0.3388	0.3546	
5	0.787	0.7038	0.3149			
6	0.6906	0.6382	0.3122	0.3289		
7	0.6315	0.5857	0.3131			
8	0.5643	0.5297	0.3058	0.3177	0.3378	0.348
10		0.4274	0.2983	0.3087		
12		0.3605	0.2856	0.3016	0.3245	
14		0.3106	0.2799	0.299		
16		0.2754	0.2732	0.2886	0.3124	0.3305
20		0.2238	0.2531	0.2787	0.2933	
24		0.188	0.2365	0.2649	0.2787	0.2908
28		0.1614	0.2087	0.2507	0.2665	
32		0.141	0.2185	0.2497	0.2652	0.2753
40		0.1125	0.175	0.2076	0.2261	0.2334
48		0.0931	0.1574	0.1844	0.2012	0.2038
56		0.0792	0.143	0.1556	0.1791	0.1842
64		0.0712	0.177	0.187	0.2128	0.2245