HPC (COMS4040A) PROJECT

Parallel Image Segmentation

Author:
Zubair BULBULIA (1249593) &
Naeem DOCRAT (1322634)

Lecturer:
Dr. Hairong BAU

UNIVERSITY OF THE
WITWATERSRAND,
JOHANNESBURG

June 23, 2019

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Image segmentation is the application of image processing techniques to alter an image such that meaningful information can be obtained. It is commonly applied in many computer vision algorithms. As computing has advanced over the years, so has the field of computer vision, which is used on images/videos for the purposes of object detection and in self-driving cars/drones amongst other uses. Image segmentation can be used to analyse images and extract features from image data by finding shapes, lines or contours in an image, or by grouping pixels with shared or similar characteristics.

This project aims to exploit the highly parallelizable aspect of image processing by parallellizing several image segmentation algorithms. These are namely the K-Means Clustering algorithm, the Nick Thresholding algorithm and the Canny Edge Detection algorithm. These are discussed in the background in chapter 2. Thereafter necessary system specifications and details related to the project are given in chapter 3. The various algorithms were parallelized in both CUDA and MPI, which is expounded upon in chapter 4. The implementations were timed and the results were recorded for various images and parameters, and these are displayed and evaluated in chapter 5. Lastly, chapter 6 provides a conclusion which summarises the report.

Chapter 2

Background

This chapter provides some background on each of the image segmentation algorithms used, as well as some high-level insight as to how they work.

2.1 K-Means Clustering

The K-Means clustering algorithm is an unsupervised machine learning algorithm that is often used on unlabelled data to find patterns or some structure in the data. This makes it apt to use on images, which contain only pixel values and no labels or other information. Here it is used to segment a colour image into a specified number (K) of colours. This is useful for image compression, as a finite number of colours can be used to represent an image with little information loss. This is known as vector quantization.

The K-Means algorithm works by first creating K random cluster centres. There are two main steps in this algorithm, namely assigning pixels to clusters and updating cluster centers. The former is carried out by looping through each pixel in an image, so pixels are assigned to clusters that they are 'closest' to (in terms of having similar colour). After this is done for all pixels, each cluster centre has its value is updated to the mean value of all the pixels in the particular cluster. This process is repeated until convergence, and thereafter the pixels in the output image are assigned the same value as the cluster centre value of their respective cluster.

2.2 Nick Thresholding

Nick thresholding is an adaptive thresholding algorithm which makes use of a given window size and k. The window size and k should be image specific. It works as follows: For each pixel intensity calculate the sum and square sum of all the neighbouring pixels which fall within the given window size. Then calculate the mean of all the pixels in the window using the sum. Then use the following formula to calculate the threshold:

$$Threshold(x,y) = M(x,y) + K * \sqrt{(E(x,y) - M(x,y)^2)/WS}$$

Where M(x,y) is the mean of all the neighbouring pixels of x and y, K is given and image specific, E(x,y) is the sum of the square of each neighbouring pixel and WS is the Window Size given. $Threshold(x,y)$ is then compared to the pixel intensity at (x,y), if $Threshold(x,y)$ is less than (x,y) then the pixel is set to 1, else its set to 0.

2.3 Canny Edge Detection

The Canny Edge detection algorithm is often used in image segmentation to accurately detect edges, and consists of four main steps. These are as follows:

- **Gaussian smoothing** - This is a simple step done to reduce noise in the image, by using convolution and a 3x3 gaussian filter.
- **Find Gradient** - Sobel filters (size 3x3) are applied to get the vertical and horizontal gradients, and thereafter the gradient magnitude and direction is calculated.
- **Non-Maximum Suppression** - Used to find thin edges in image. The pixels from the magnitude image are used to find the local maximum in the direction of the gradient(ridge pixels), and sets non maximum values to 0.
- **Hysteresis thresholding** - Thresholding is done using two values T1 and T2, where ridge pixels are classified as 'strong' or 'weak' edge pixels. Thereafter, the algorithm attempts to link edges by using the weak pixels that are connected to the strong edge pixels.

Chapter 3

Specifications

3.1 System specifications

3.1.1 Hardware specifications

All the implementation were run on one machine to avoid any unforeseen issues, that come with using different machines ie. one machine may be faster than the other. The Specifications of the Machine are as follows:

- Intel© Core i7-7700 CPU @ 3.60GHz x 8 cores,
- 16GB DDR4 ram and
- Nvidia© GeForce GTX 1060 GPU with 1280 CUDA cores @ 1506 MHz base clock

3.1.2 Software specifications

- Operation system: Ubuntu 18.04.1
- CUDA 10.0
- GCC 7.4.0

3.1.3 Images' details

The K means algorithm used its own colour images while the Nick thresholding and canny edge detection algorithms used their own gray scale images. The image sizes are discussed in the subsections below.

K means

The algorithm was tested on the following 3 colour images:

- storm.ppm, size 1910x1000 = 1 910 000 pixels
- cat.ppm, size 800x800 = 640 000 pixels
- suburbs.ppm, size 702x336 = 235 872 pixels

FIGURE 3.1: Original images for storm, cat and suburb respectively

Nick Thresholding and Canny Edge Detection

Nick thresholding and Canny Edge detection were tested on the same images, however nick thresholding was tested on one extra image that has different gray gradients on different parts of the image, this is the image stated below as paper.pgm. Nick thresholding and Canny edge were tested on 3 gray scale Images:

- Paper.pgm, size 1216x1632 = 1 984 512 pixels
- Newspaper.pgm, size 1080x1080 = 1 166 400 pixels
- Lena.pgm, size 512x512 = 262 144 pixels
- MRI.pgm, size 225x225 = 50 625 pixels

Based on the time complexity analysis, image sizes would greatly affect performance of all three algorithms. To verify the analysis as well as to create a decent measure of performance on the algorithm, three different image sizes were chosen. These three image sizes are chosen specifically so that the total pixels in the image vary from less to more pixels, this will allow us to accurately benchmark the implementations of our algorithms as they scale up to bigger image sizes.

FIGURE 3.2: Original images for Paper, Newspaper, MRI and Lena respectively

Chapter 4

Methodology and Experiment Setup

4.1 Introduction

This chapter provides more details as to how each implementation is parallelized, as well as the manner in which testing was carried out. Each algorithm has a serial implementation, as well as CUDA and MPI parallelizations. These are as follows:

4.2 K-Means Clustering

The serial algorithm works as mentioned previously in the background. After the K random clusters are first created, serially assigns each pixel value to the cluster it is most similar to. This involves a double for loop going through each pixel, where each pixel also loops through the K clusters to find out which one it is most similar to. After all assignments, there is a for loop through the clusters to update their centers. As this process is very serial and is also repeated till convergence, there is overall a lot of room for parellelization.

4.2.1 CUDA

The CUDA implementation was done by making use of a kernel function, where each thread is responsible for assigning a pixel to its respective cluster. This was then run using the same number of threads as there are pixels in the image. This very effectively parallelizes the assignment part of the algorithm. Updating the cluster centres could only be done after all of the threads had finished assigning pixels to clusters, so it is necessary to wait. Therefore the assignment was done serially. This is effective only for smaller sized images and for small values of K as there is less overhead.
The assignment and update processes are repeated until convergence; the former was done on the gpu and the latter serially. After convergence, the new image was created in parallel by assigning pixel values based on the cluster centres values on the gpu.

4.2.2 MPI

The MPI Implementation splits up the work done for assigning the pixels to clusters similarly to how it is done in CUDA, however the cluster updates are performed differently. An MPI barrier is called after the assignment process, and reduction and broadcast operations are then needed to do the cluster updates. After much experimentation, this was done using MPI AllReduce. This function does a reduction

operation, and then broadcasts it to the other processes. Therefore most of the code is parallelised in the MPI implementation, at the expense of increased overhead.

4.3 Nick Thresholding

The serial approach is coded as mentioned in the background, however this approach is highly parallelizable and only serves as a minimum benchmark for the parallelized approaches that follow.

Cuda Approach

In the cuda approach the image was loaded into global memory, then each thread from its respective block loaded its pixel intensity of the image into its respective shared memory of that block. This effectively parallelizes the algorithm since multiple threads can execute simultaneously, while also reducing the global memory reads by using the blocks shared memory to get its neighbours pixels intensity, thus reducing execution time.

The limitation with this approach is that since each thread loaded its pixel intensity into shared memory for its respective block the maximum window size is 32 as the maximum amount of threads allowed in a block is 32*32 = 1024.

MPI Approach

In the MPI approach the image is equally divided and scattered across all nodes, each node then computes the convolution on its sub image, the sub image is then sent back to the root node where it is combined to form the final processed image once again.

4.4 Canny Edge Detection

In the Serial approach of the Canny Edge Detection algorithm, a convolution of the image with a blur kernel is done for smoothing. Thereafter, two more convolutions of the image with edge detector kernels are performed. Then computation of the gradient magnitude and direction is done. The next step performed on the resultant image is non-maximum suppression, and lastly hysteresis thresholding (with edge linking).

Cuda Approach

The CUDA approach parallelizes the convolution operation, as this happens thrice. The kernels/filters for each convolution are also stored in constant memory to reduce memory access costs. Storing the image in texture/shared memory can be done as well.

MPI Approach

In the MPI Approach The image was scattered in Multiple nodes Each Node performed its respective calculations on each sub-image getting its neighbours from previous nodes and next nodes(Unless its the root or last node) then the image was gathered back to the root node. In this approach if the the node is the root or last

node than it would only need to get its neighbours from the next node or the previous node respectively.

4.5 Experiment Setup

The experiments will be run on the various images of which the data descriptions are given in more detail in Chapter 3 above. Testing will be done to measure the processing time (this is what is parallelised), the overhead time, the throughput and the total time for each algorithm as follows:

4.5.1 K-Means Clustering

When running the K-Means algorithm, the values of K used are 2, 8, 64 and 128 for each of the three images used for testing and for all parallelizations.

4.5.2 Nick Thresholding and Canny Edge Detection

All the approaches used in nick thresholding and canny edge detection were tested on the images mentioned 3.2(besides paper.pgm for canny edge detection)as well as 4 different filters of size 20x20, 40x40, 60x60 and 80x80 . This allowed us to get an empirical analysis on the scalability of the image.

Chapter 5

Results & Evaluation

This chapter shows the results that were obtained for all implementations, as well as discussions on the obtained results.

5.1 K-Means Clustering

The K-Means Clustering algorithm was run on each of the colour images for various values of K. The following shows image storm.ppm of resolution 1910*1000, as well as the 4 output images after clustering when K = 2,8,64 and 128 respectively. It can be seen that with increasing K values, the output starts resembling the original image. There are many different colours in this image, so more clusters may be needed to obtain better results.

FIGURE 5.1: Original storm image, K=2, K=8, K=64 & K=128

Figure 5.2 below shows the input image cat.ppm of resolution 800*800, as well as the 4 output images after clustering, again when K = 2,8,64 and 128 respectively. When K = 128, the output image is almost identical to the input image. Clustering works very well in this image as it has relatively few colours.

FIGURE 5.2: Original cat image, K=2, K=8, K=64 & K=128

Lastly, Figure 5.3 below shows the input image suburb.ppm of resolution 702*336, as well as the 4 output images after clustering when K = 2,8,64 and 128 respectively. The end result with 128 cluster centers again resembles the original image very closely.

FIGURE 5.3: Original suburb image, K=2, K=8, K=64 & K=128

5.1.1 Metrics: K-Means

The graph below shows run times obtained when running the serial K-Means clustering algorithm on various colour images for multiple K values.

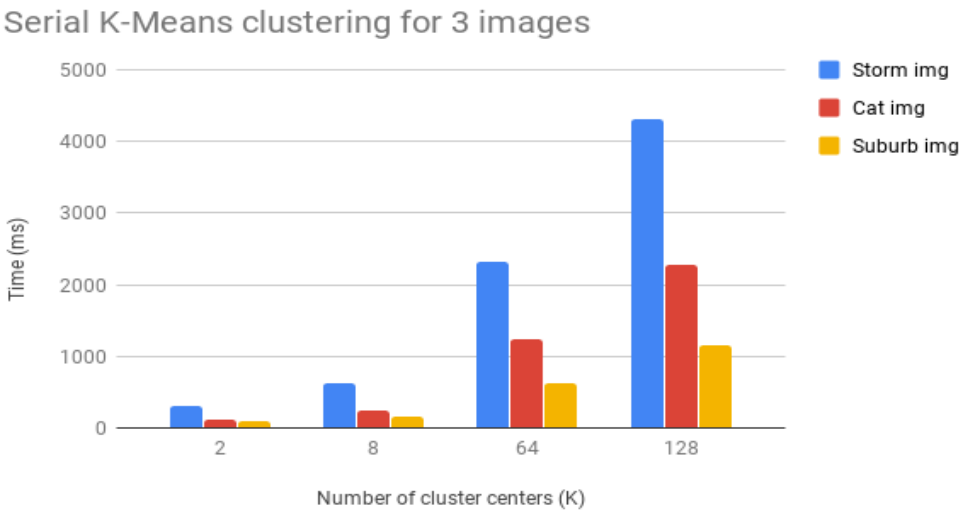

FIGURE 5.4: Graph of Total Time (ms) for Serial K-Means Clustering

The following depicts the results of running the CUDA implementation of the K-Means clustering algorithm on various colour images for multiple K values.

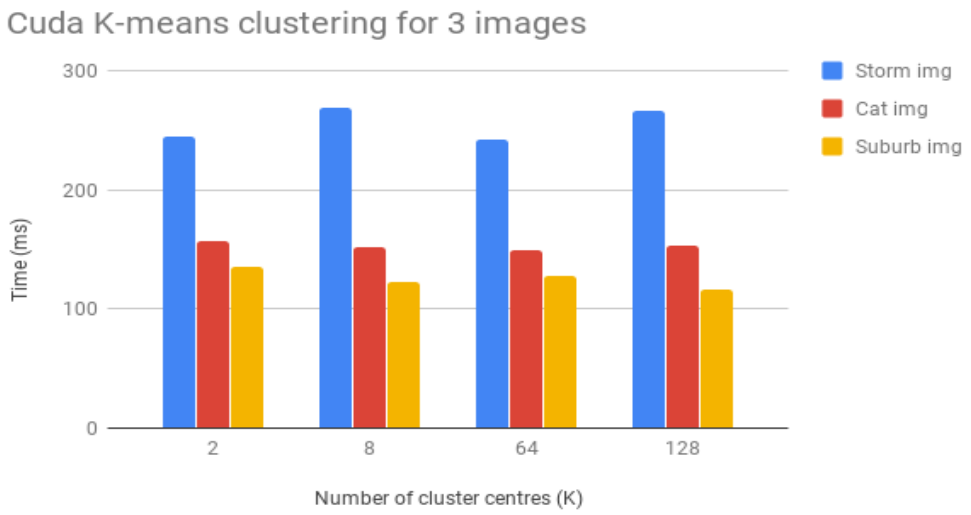

FIGURE 5.5: Graph of Total Time (ms) for CUDA K-Means Clustering

This graph shows the run times when running the MPI implementation of the K-Means clustering algorithm with 4 processes for multiple K values.

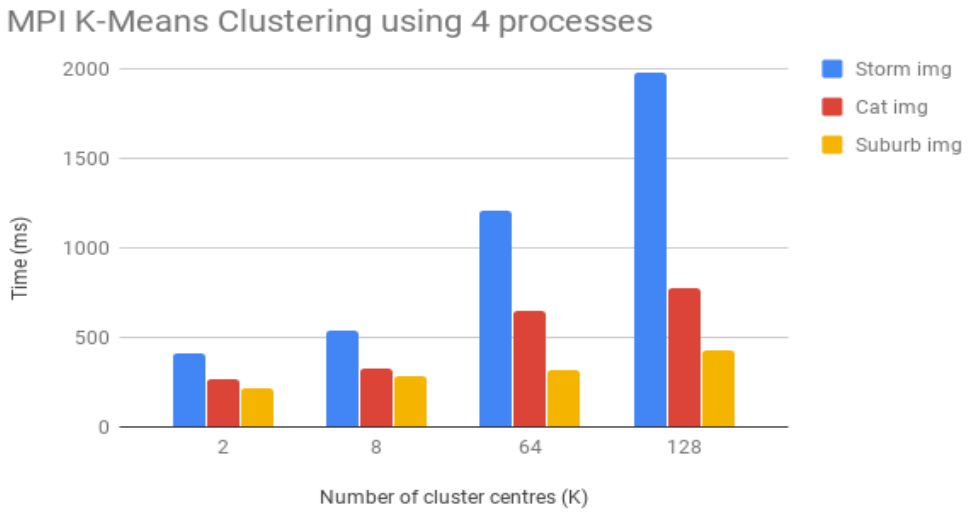

FIGURE 5.6: Total Time (ms) of MPI K-Means Clustering with 4 processes

The following shows run times when running the MPI implementation of the K-Means clustering algorithm with 8 processes for multiple K values.

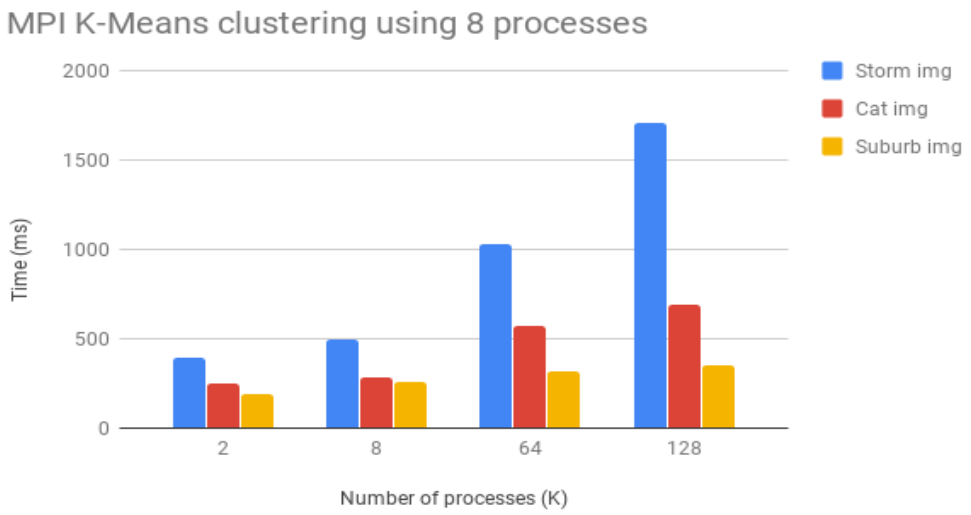

FIGURE 5.7: Total Time (ms) of MPI K-Means Clustering with 8 processes

The next graph shows run processing time, overhead time, throughput and total run time for all implementations of the K-Means clustering algorithm on the storm image.

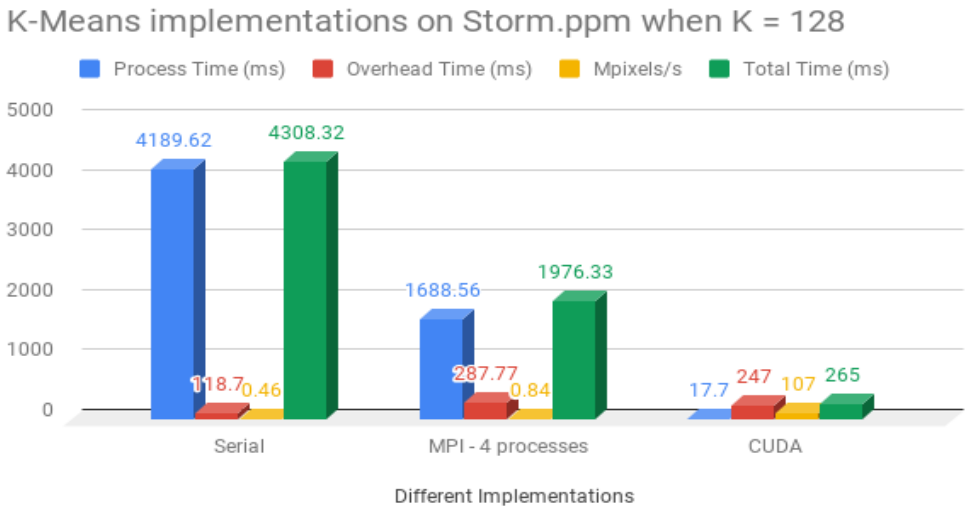

FIGURE 5.8: Metrics of K-Means clustering

This last graph in 5.9 shows the throughput for K-Means clustering on the storm image. It is clear that CUDA has the greatest throughput.

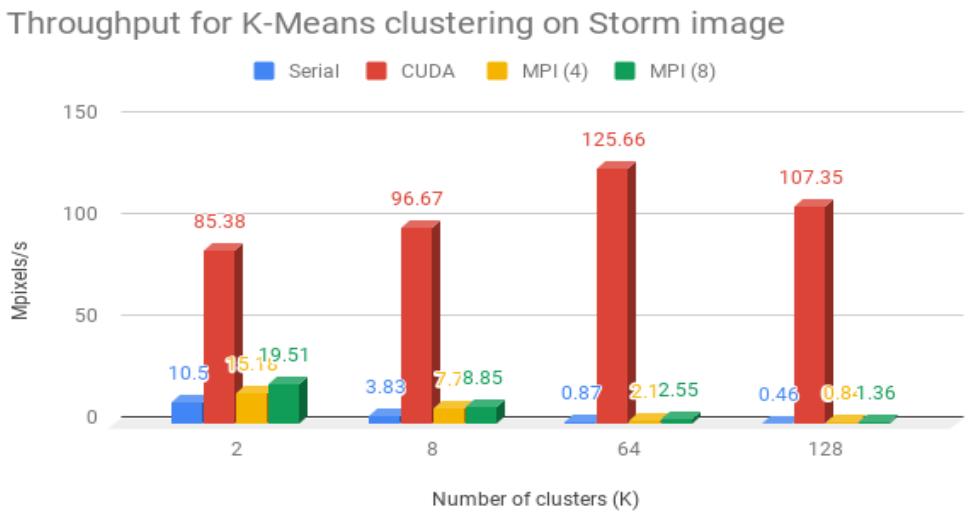

FIGURE 5.9: Throughput of K-Means clustering for storm image

The results obtained by parallelizing the K-Means algorithm vs the results for the serial version show a speedup as can be seen in the table that follows below. It is again clear that CUDA gives the greatest speedup, which is expected after seeing the throughput previously.

Image	CUDA Speedup	MPI (4 processes) Speedup	MPI (8 processes) Speedup
Storm	16.21	2.18	2.53
Cat	14.88	2.96	3.31
Suburb	9.88	2.69	3.28

TABLE 5.1: Table of speedups for K-Means algorithm implementations

5.1.2 Evaluation: K-Means Clustering

The results for the serial K-Means clustering implementation shows that as the number of cluster centres increases for a given image, the run time increases as well. As expected, run times scale proportionately with increasing image size for the same K values.

Similar trends hold for the MPI implementations. The overall MPI run times decreased when 8 processes were used instead of 4. This decrease in total run time occurs despite an increase in the overhead costs as number of processes increases due to communication. MPI vs serial on the storm, cat and suburb images yield a speedup of 2.18, 2.96 and 2.69 respectively at K=128. When 8 processes are used, the speedup increases to 2.53, 3.31 and 3.28 respectively at K = 128 as well, which shows increasing the number of processes provides an increase that is non-linear.

However with the CUDA implementation, for increasing K values the run times remain similar as can be seen in Figure 5.5. The changes in run time are negligible for the K values as GPUs are very efficient for processing images in parallel. Run times do scale with image resolution (size of the input data), which is apparent from the graph. Most of the run time cost comes from the overhead for this implementation, as can be seen in Figure 5.9. Using a better form of memory access could decrease these overheads. The CUDA implementation on storm, cat and suburb images yield a speedup of 16.21, 14.88 and 9.88 respectively at K=128.

Figure 5.5, which show processing, overhead, and total run time, also shows throughput in terms of Mega-pixels processed per second. This shows how much processing is done. It is clear to see that the CUDA implementation has the highest throughput and the lowest total run-time, and is therefore the most suitable for performing K-means Clustering.
Figure 5.9 reinforces this notion, with CUDA having the highest throughput and the serial implementation always having the lowest throughput for large images like *storm.ppm*. This holds true for smaller images as well, although the margins are closer. This is because a thread is used for each pixel, which means 1 910 000 threads are used for the large storm.ppm image, whereas for MPI the scaling cannot be that high. It can be seen from the graphs above that MPI implementations always fare better than the serial one as well, and that increasing the number of processes increases throughput.

5.1.3 Time Complexity Analysis: K-Means Clustering

The serial K-means algorithm has an asymptotic time complexity of $O(K * N^2)$ for an image of size N(N = height x width). The CUDA implementation reduces this by a factor of at least $\frac{Num_of_threads}{2}$, due to splitting up the assignment and update processes, and without taking into account overhead costs, which decreases ideal speedup.
For the MPI implementation, the complexity is $O(\frac{K*N^2}{Num_of_Nodes})$ plus additional communication costs. By increasing the number of nodes the total run time will decrease and throughput will increase, however overhead costs will increase as well.

5.2 Nick Thresholding

The nick thresholding algorithm was run on each of the gray-scale images for various filter sizes. The following shows Newspaper.pgm of size 1080x1080, as well as its respective output from the nick thresholding algorithm.

FIGURE 5.10: Nick Thresholding on Newspaper.pgm

Figure 5.11 below shows the image Lena.pgm image of size 512x512, the following is the original image as well as its output after performing nick thresholding on the image.

FIGURE 5.11: Nick Thresholding on Lena.pgm

Figure 5.12 below shows the image MRI.pgm image of size 225x225, the following is the original image as well as its output after performing nick thresholding on the image.

FIGURE 5.12: Nick Thresholding on MRI.pgm

Lastly, as can be seen in Figure 5.13 below, Nick Thresholding was tested on the paper.pgm image of size 1216x1632, which has a noticeable gradient. The following is the original image as well as its output after performing the thresholding on the image.

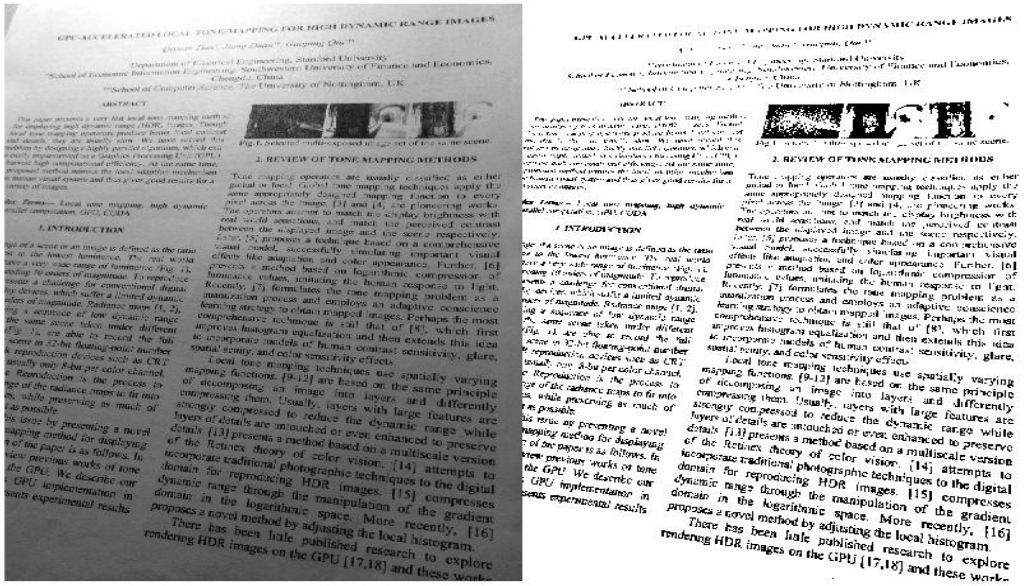

FIGURE 5.13: Nick Thresholding on Paper.pgm

5.2.1 Metrics and Evaluation: Nick Thresholding

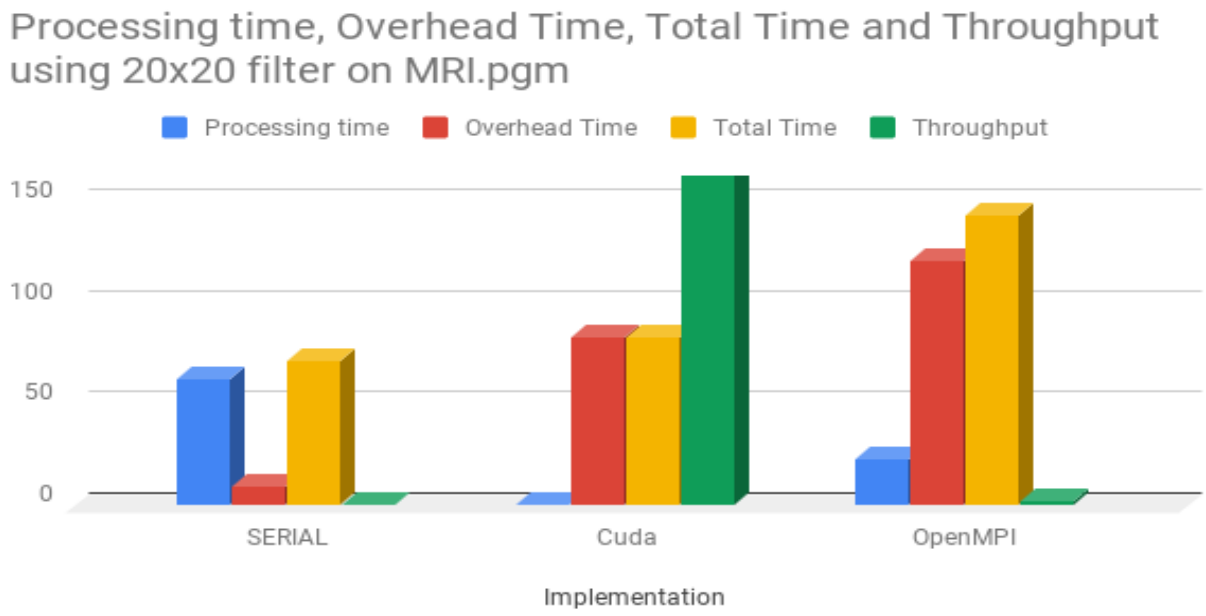

FIGURE 5.14: Results of Nick Thresholding using 20x20 filter on MRI.pgm

The graph 5.14 above is a result of running MRI.pgm with a 20x20 filter. The next graph 5.15 below is a result of running newspaper.pgm using a 20x20 filter.

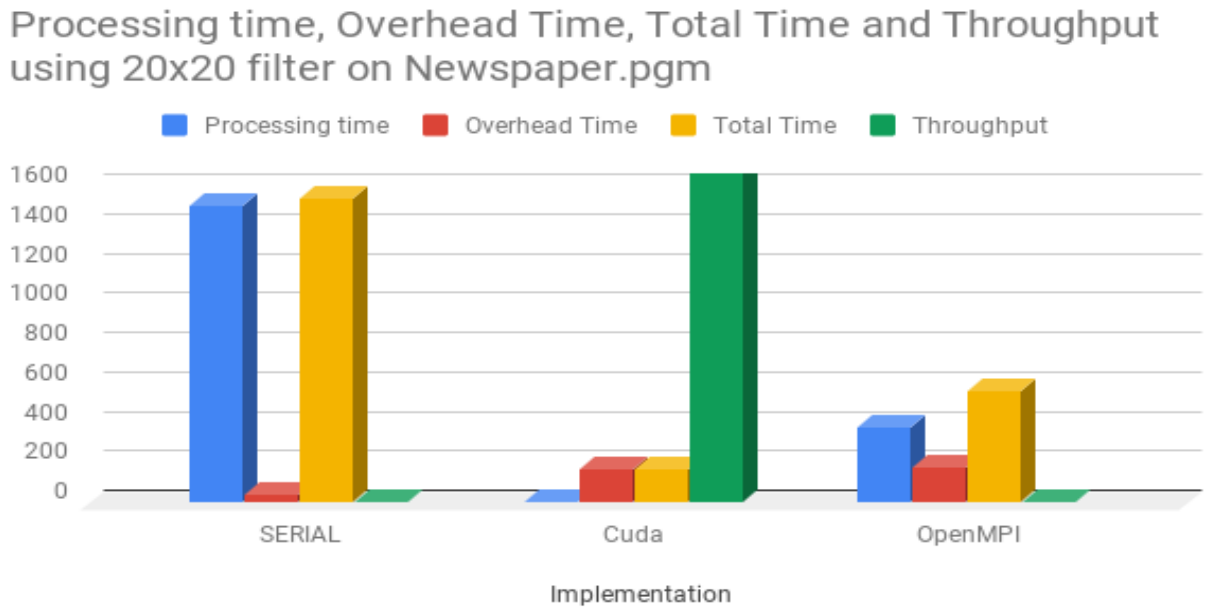

FIGURE 5.15: Results of Nick Thresholding using 20x20 filter on Newspaper.pgm

As seen from 5.15 the Cuda implementation is the fastest for larger images, while MPI is the second fastest and serial takes the longest. However, for smaller images

the overhead costs associated with Cuda and MPI significantly increase the total time such that the serial implementation time is much faster. In order to reduce overhead costs for the MPI implementation we will have to reduce the amount of communication between nodes or when gathering sub images we can gather them into sub nodes and then from these sub nodes gather them into the root node. The next graph below 5.16 is the throughput from running Nick thresholding on MRI.pgm using 20x20 filter size.

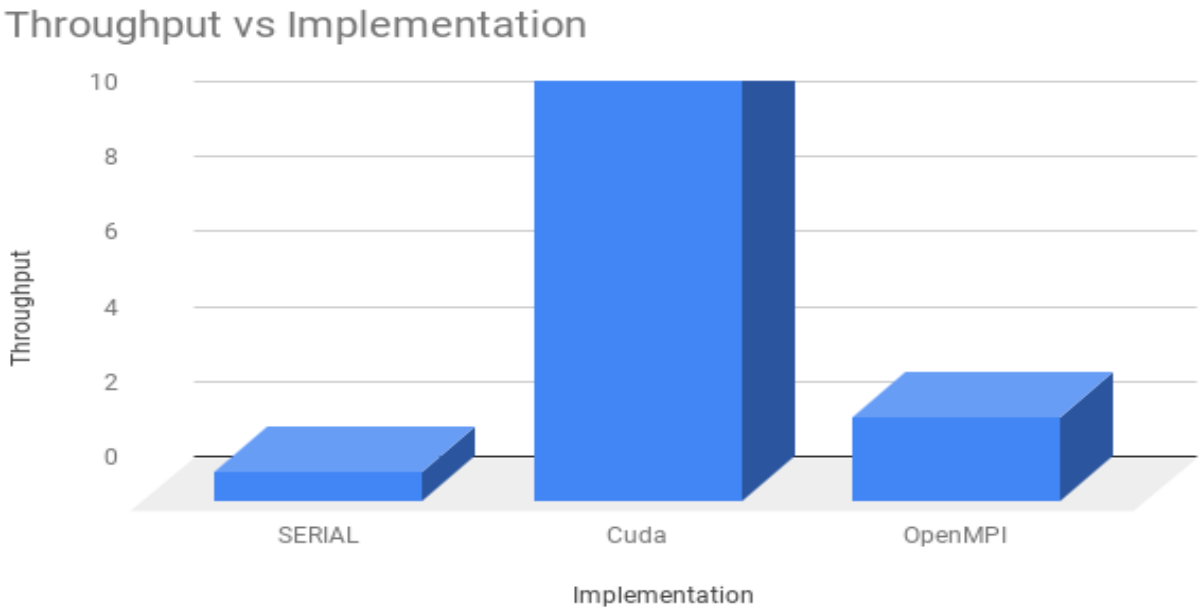

FIGURE 5.16: Througput of Nick Thresholding using 20x20 filter on MRI.pgm

As we can see above in 5.16 the throughput vs the respective implementations. The throughput only takes into account the parallelized sections of code in order to get a measure of the speed of execution of these section in order to determine if the parallelization or the overhead cost needs to be optimized. The throughput for cuda is extremely fast, this is due to the fact that the cuda kernel is using a shared memory approach which significantly reduces latency from global memory reads. The next fastest is MPI, which is faster than serial but the implementation can we optimized by using more nodes and reduce comunication. As the image scales the trend of the throughput remains the same.

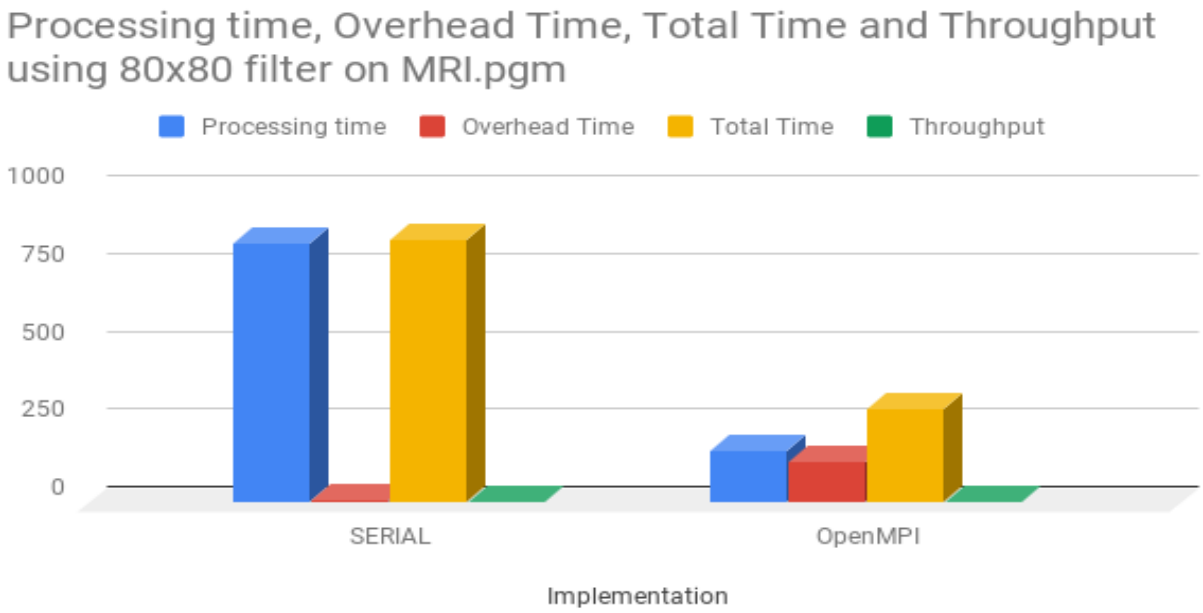

FIGURE 5.17: Results of Nick Thresholding using 80x80 filter on MRI.pgm

The graph 5.17 above is a result of running MRI.pgm with a 80x80 filter. The next graph 5.18 below is a result of running newspaper.pgm using a 80x80 filter.

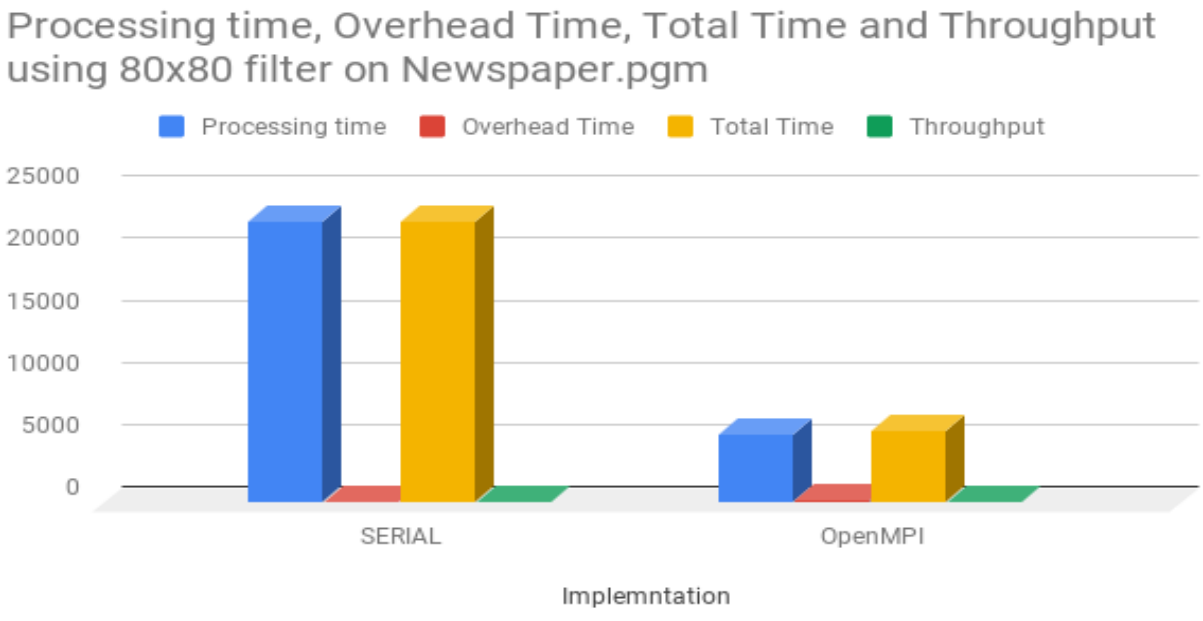

FIGURE 5.18: Results of Nick Thresholding using 80x80 filter on Newspaper.pgm

Using a 80x80 filter on the smaller image (MRI.pgm) and larger image (Newspaper.pgm) in the graphs above reveal that MPI is much faster on the smaller image using a larger filter, unlike in 5.14 where the serial implementation was faster on a smaller image. This is due to the fact that a much larger filter size (80x80) is

used, thus the overhead cost - while still significant for the smaller image - does not increase the total time to the point where it's slower than the serial implementation. In 5.18 we see that for larger images and bigger filter sizes, the overhead time becomes much more insignificant; this means that the implementation is much faster and optimized for larger image sizes with large filters.

Speedup: Nick Thresholding

	Speedup on Newspaper.png
Cuda	2.75x
MPI	9.07

TABLE 5.2: Speedup of CUDA and MPI compared to Serial using 20x20 filter on Newspaper.png

5.2.2 Time Complexity Analysis: Nick Thresholding

Serial Implementation

In the serial approach, assuming a square image of size $N * N$ and a filter of size $W * W$, the complexity is:

$$O(W^2N^2)$$

This type of complexity scales with larger sizes of images as well as larger filter sizes and this can be easily seen in 5.14 and 5.6 as the image and filter size gets bigger the scale of the vertical axis is much larger and the serial implementation takes a lot longer.

CUDA Implementation

In the CUDA approach, assuming a square image of size $N * N$ and filter of size$W * W$, the complexity is:

$$O(\frac{W^2N^2}{NumOfThreads}) + Overhead$$

The time using this CUDA implementation does scale with larger image sizes, but at a much slower rate than the serial one since the work load is spread amongst multiple threads. However, the overhead costs can greatly affect the performance of the implementation and for smaller images make it slower than even the serial implementation as seen in 5.14.

MPI Implemnetatiom

In the MPI approach, assuming a square image of size $N * N$ and a filter of size $W * W$, the complexity is:

$$O(\frac{W^2N^2}{NumOfNodes}) + Communication$$

Again in this implementation by increasing the number of nodes we can decrease the time. However, increasing the number of nodes will increase communication costs therefore, for smaller images we should use less nodes and larger images should use more nodes.

5.3 Canny Edge Detection

The Canny Edge Detection algorithm was run on the 3 grayscale images of different sizes for each implementation. The following shows image mri.pgm of resolution 225x225, as well as the output images after gaussian blurring, finding the gradient, after non-maximum suppression and hysteresis thresholding.

FIGURE 5.19: Canny Edge Detection on image MRI, from left, Original,Gaussian,Gradient Magnitude,NMS,hysteresis

Figure 5.20 below shows the input image Lena.pgm of resolution 512x512, as well as the 4 output images after blurring, finding the gradient, non-maximum suppression and hysteresis thresholding as before.

FIGURE 5.20: Canny Edge Detection on image Lena, from left, Original,Gaussian,Gradient Magnitude,NMS,hysteresis

Lastly, Figure 5.21 below shows the input image Newspaper.pgm of resolution 1024x1024, as well as the 4 output images after each step of the canny edge detection algorithm as before.

FIGURE 5.21: Canny Edge Detection on image Newspaper, from left, Original,Gaussian,Gradient Magnitude,NMS,hysteresis

5.3.1 Metrics and Evaluation: Canny Edge Detection

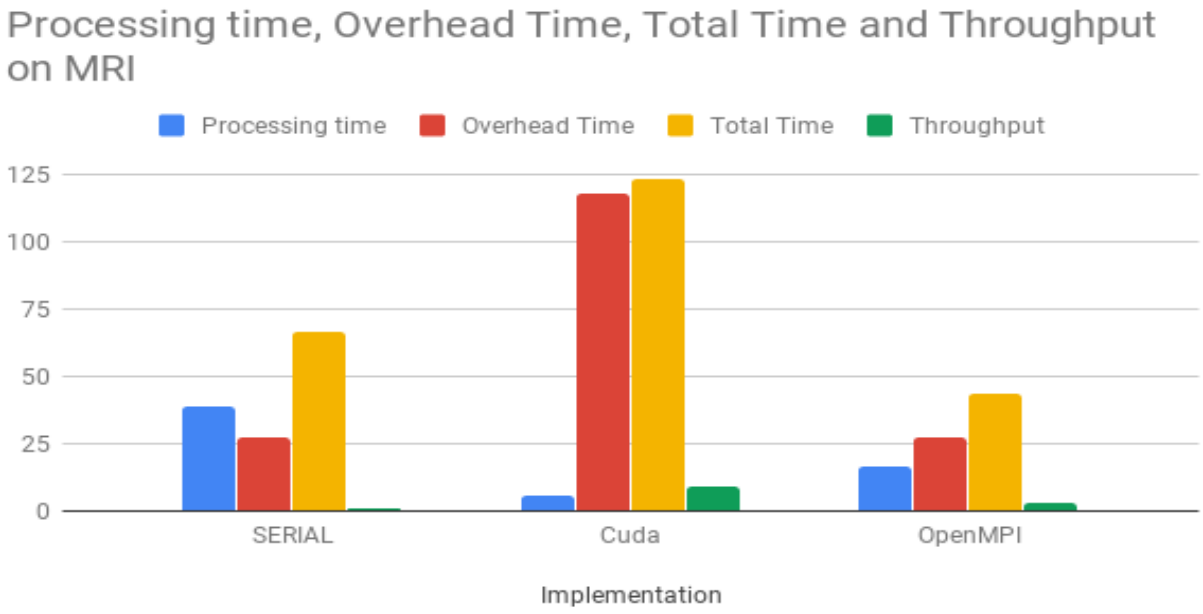

FIGURE 5.22: Results of Canny Edge Detection on MRI.pgm

The graph 5.22 above is a result of running MRI.pgm with a 3x3 filter. The next graph 5.23 below is a result of running Lena.pgm using a 3x3 filter.

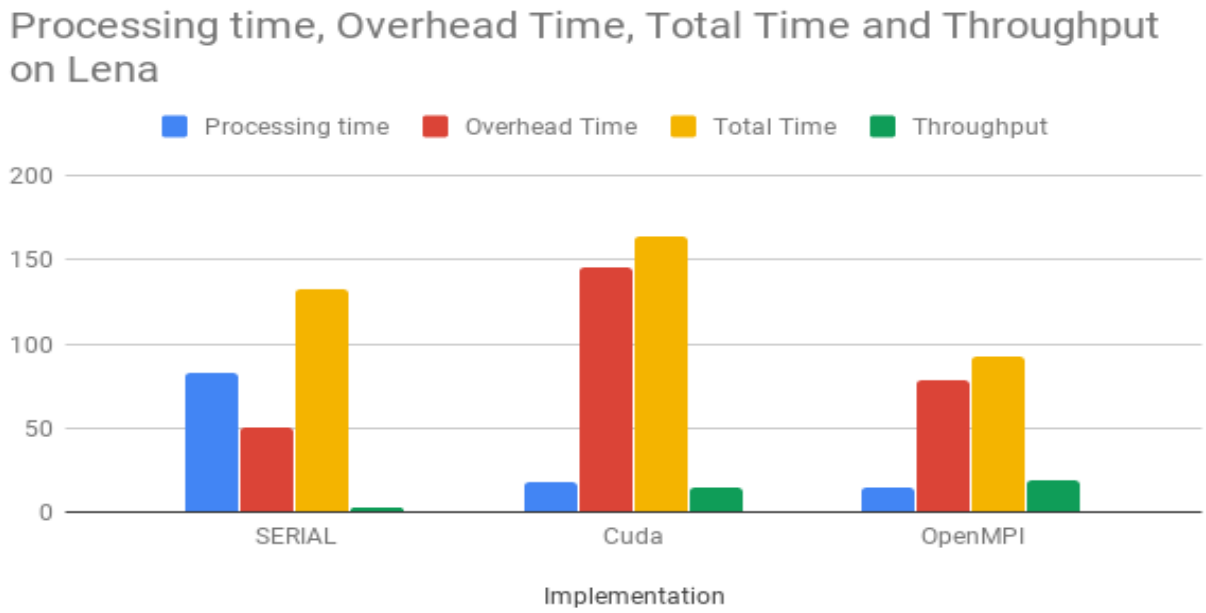

FIGURE 5.23: Results of Canny Edge Detection on Lena.pgm

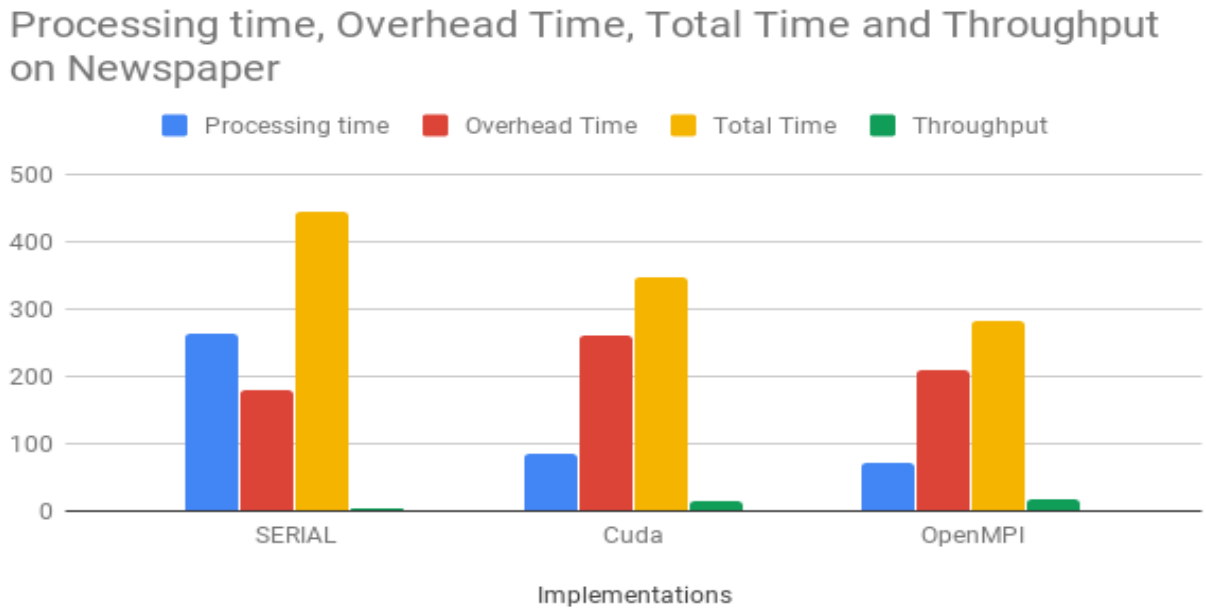

FIGURE 5.24: Results of Canny Edge Detectiong on Newspaper.pgm

The graph 5.24 above is a result of running Newspaper.pgm with a 3x3 filter. Lastly, the graph 5.25 above is a the threshold vs implementation of running mri.pgm using a 3x3 filter.

In the 3 graphs above MPI is the fastest implementation, however, when comparing just throughput in 5.25 above, it's observed that CUDA in fact has the fastest parallel approach and rather the overhead costs are extremely high. These overhead costs are due to the nature of the canny edge detection algorithm, which

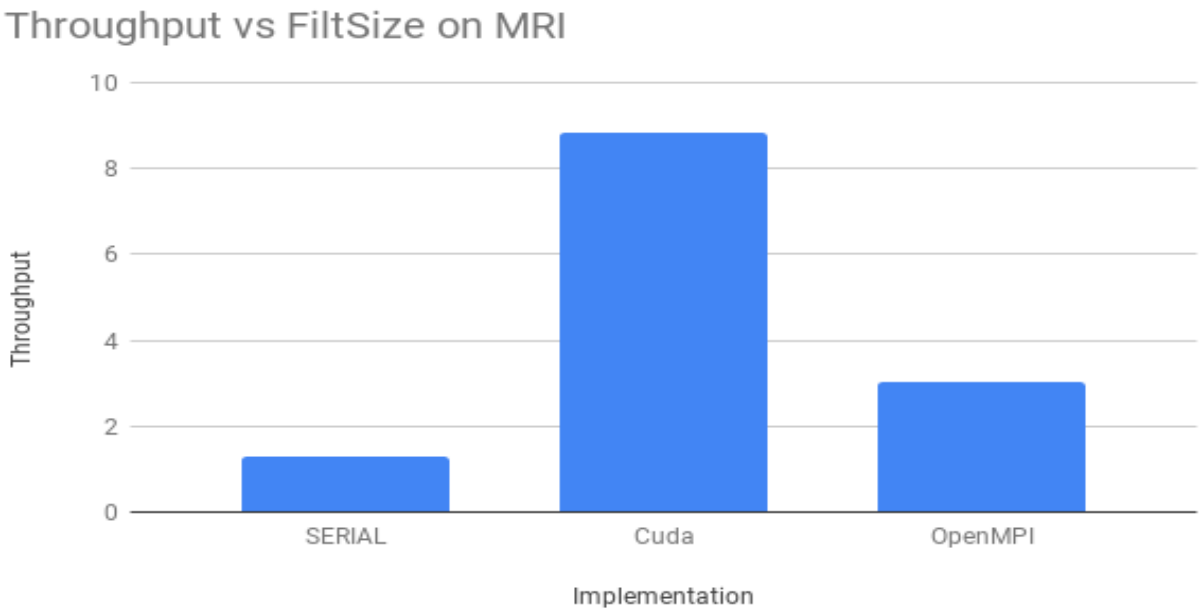

FIGURE 5.25: Throughput Results of Canny Edge Detection on MRI.pgm

requires 3 image convolutions, because of the three image convolution, the implementation will always perform 6 host to device and device to host memory copies of the entire image (2 for each image convolution).

In the CUDA Implementation only the image convolutions were paralized as parallelizing the calculation of gradient direction and magnitude, non-max supression and hysteresis will just increase the already high overhead cost, while not decreasing processing time by much since it's minimized already by a significant amount compared to the serial implementation. In order to minimize the overhead cost in the CUDA implementation, we can consider using pinned shared memory, since each block can load its part of the image into its respective shared memory block ie. it does not need to wait for the entire image to load into global memory before executing the convolution.

In the MPI Implementation, the overhead cost are also significantly high as seen in 5.22 and this trend is observed regardless of image size, increasing the overall time of the program. To minimize the overhead cost of the MPI implementation we need to reduce the overall communication cost, one approach is to decrease the send and receive requests which the MPI implementation does to get the neighbouring pixels it does not have when performing the convolution. This can be achieved by simply increasing each nodes image height by the size of the filter on either side and allocating it those extra pixel intensities, thus when it requires its neighbours it will be able to access it from it's local cache instead of getting it from its neighbouring node. Another approach is to reduce the communication when gathering the final images, instead of gathering the final images in the root node we can gather 2 halves of the final images in 2 different nodes and then combining the 2 final half images into one image in the root node. Lastly, the program outputs 4 images, instead of gathering all 4 images into the root node, we can gather the 4 images into

4 different nodes and then send it to the root node, thus reducing the wait time since other gather calls have to wait until the previous gather call is done executing.

Speedup: Canny Edge Detection

	Speedup on Newspaper.png
Cuda	1.28x
MPI	1.58x

TABLE 5.3: Speedup of CUDA and MPI compared to Serial on Newspaper.png

5.3.2 Time Complexity Analysis: Canny Edge Detection

Serial Implementation

In the serial approach, assuming a square image of size $N * N$ and a filter of size $W * W$, the complexity is:

$$O(W^2N^2)$$

This type of complexity scales again like in the Nick thresholding scales with larger sizes of images and this can be easily seen in 5.22 and 5.24 as the image gets bigger the scale of the vertical axis is much larger for all implementations.

CUDA Implementation

In the CUDA approach, assuming a square image of size $N * N$ and filter of size$W * W$, the complexity is:

$$O(\frac{W^2N^2}{NumOfThreads}) + Overhead$$

The time using this CUDA implementation does scale with larger image sizes, however this is not due to actual processing time, rather due to high overhead costs. In the case of this canny edge detection the overhead costs need to be reduced significantly before any performance increase can be seen.

MPI Implemnetatiom

In the MPI approach, assuming a square image of size $N * N$ and a filter of size $W * W$, the complexity is:

$$O(\frac{W^2N^2}{NumOfNodes}) + Communication$$

Again in this implementation by increasing the number of nodes we can decrease the time. However the communication costs in MPI are again driving up the total execution time. while faster than serial in every image size tested above, the MPI communication costs should also be minimized.

Chapter 6

Proof of cluster run

The following screenshots serve as proof of algorithm in the clusters squeue as well as an image of the output file (attached in the .zip).

6.1 Queue

```
zbulbulia@mscluster0:~/mpitry$ sbatch k.slurm
Submitted batch job 11556
zbulbulia@mscluster0:~/mpitry$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
             11554     batch   m_main kpaupama  R       1:34      2 mscluster[11-12]
             11556     batch    neeem zbulbuli  R       0:01      4 mscluster[13-16]
zbulbulia@mscluster0:~/mpitry$
```

FIGURE 6.1: Algorithm submitted and running in squeue

6.2 Output

```
-----------------------------------------------------
Job is running on node  mscluster[13-16]----------------------------------------
SLURM: sbatch is running on mscluster0.ms.wits.ac.za
SLURM: job ID is 11553
SLURM: submit directory is /home-mscluster/zbulbulia/mpitry
SLURM: number of nodes allocated is 4
SLURM: number of cores is 4
SLURM: job name is neeem
-----------------------------------------------------
EHREEHREEHREHERHEJDHERHEJDEHREHERHEJDHERHEJD291600  dfawfa  0100
291600  dfawfa  0300
291600  dfawfa  0200
291600  dfawfa  02200
291600  dfawfa  03300
Processing time: 5.527522 (ms)
0.21 Mpixels/sec
Overhead TIme: 0.055525 (ms)
TOTAL TIME: 5.583047 (ms)
```

FIGURE 6.2: Output file 'slurm.mscluster13.11553.out'

Chapter 7

Conclusion

To aim of this project was to develop several parallel image segmentation solutions. The results obtained showed that for the K-Means algorithm, CUDA was the best approach. It had the maximum throughput and was the fastest. CUDA is extremely well suited to straight-forward image processing tasks as it boasts a large number of cores and can be extremely multi-threaded, effectively splitting up computation per image pixel.
For the Nick Thresholding algorithm, there were several cases. Small images performed best in serial as other implementations had large overheads. CUDA was by far the best for large images and with small filter sizes (< 32x32). For large images and large filters, MPI was the fastest but only as CUDA could not utilize large filters due to the shared memory approach.
For the Canny Edge Detection, MPI was the fastest. This is because MPI had the most parallelization and partly because the CUDA algorithm wasn't completely parallelized. In addition to this, the CUDA overhead costs were quite large. If it was properly parallelized, the overhead costs would be larger. With a larger cluster to test on, the results for MPI would likely be better for larger images.

In conclusion, the chosen algorithms were parallelized and a suitable method of parallelization was found in each case. CUDA is very effective at dealing with images due to the sheer number of threading that can be achieved. MPI is a useful approach when the algorithm is pipelined and there is too much communication/synchronization for CUDA to be used, or the image size is not optimal.

Bibliography

[1] Najafi, M. Hassan Murali, Anirudh Lilja, David Sartori, John. (2015). GPU-Accelerated Nick Local Image Thresholding Algorithm. 576-584.10.1109/ICPADS.2015.78.

[2] K. Ogawa, Y. Ito and K. Nakano, "Efficient Canny Edge Detection Using a GPU," 2010 First International Conference on Networking and Computing, Higashi-Hiroshima, 2010, pp. 279-280. doi: 10.1109/IC-NC.2010.13, URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=arnumber=5695250isnumber=5695205

[3] Nadig M. G (2017, Feb 25) *Parallel Computing in C using OpenMP* Retrieved from http://madhugnadig.com/articles/parallel-processing/2017/02/25/parallel-computing-in-c-using-openMP.html

[4] Alyasseri, Zaid. (2014). Parallelize Bubble Sort Algorithm Using OpenMP. International Journal of Advanced Research in Computer Science and Software Engineering. 4. 103-110.

[5] Bau, Hairong (2019) *COMS4040A COMS7045A Assignment 1 Brief*

[6] Pandey, Pushkar. (2017, February 26) *Quicksort using OPENMP* Retrieved from https://pushkar2196.wordpress.com/2017/02/26/quicksort/