Parallel Image Segmentation

HPC - Project

Uvir Bhagirathi : 1141886

Storm Menges : 1102107

May 22, 2018

Chapter 1

Introduction

This report will describe how the problem of parallel image segmentation through various segmentation algorithms was approached. The three chosen algorithms; Edge Detection, Otsu Thresholding and K-Means Clustering, were parallelised in CUDA and MPI. Thereafter each implementation was timed and compared to one another. This is done by first programming the above algorithms in serial, then coding up a CUDA implementation and finally implement a MPI version.

The output of the implementations of each algorithms is compared against the serial implementation of that algorithm to ensure that the program is producing the correct, accurate output. Once each implementation has been optimized in the best known way and produces the correct output, each implementation is then timed and compared. The report will then conclude with the analysis of the results as well as the optimal parallelism platform for the problem that is image segmentation.

Chapter 2

Specifications

2.1 System Specifications

CPU : Intel Core i7-7700 CPU @ 3.60GHz x 8

RAM : 16Gb

GPU : GeForce GTX 1060

CUDA Cores : 1280

GPU RAM : 6Gb

Operating System : Ubuntu 16.04 LTS 64-bit

2.2 Images

The Lena image will be used in the Edge Detection and Otsu Thresholding methods. The 1024x1024 pixel image is shown below.

Figure 2.1: Lena

The following 3008x1960 pixel image will be used in the processing of the K-means clustering algorithm.

Figure 2.2: Porchlicht

Chapter 3

Background

3.1 Edge Detection

Edge detection is an image processing technique used to find boundaries of objects within an image. This is done by identifying points in an image at which the pixel intensities change sharply. A filter is applied to the image using a form of convolution. Convolution is the addition of the elements of an image to it's relative neighbours, weighted by a mask/filter. Edge detection is a fundamental tool used in image processing, machine learning and computer vision as well as areas of feature detection and extraction.

3.2 Otsu Thresholding

Otsu thresholding is a thresholding method used to binarize a bi-modal gray-scale image. This is done by finding a suitable threshold that maximizes the variance between the intensities before and after the selected threshold.

3.3 K-Means Clustering

K-Means Clustering is a machine learning algorithm that is used in image segmentation by clustering similar pixels together and assigning the pixels that are similar to each other to one value. The similarity measure in our algorithm was chosen to be the color of the pixel, essentially similarly colored pixels would be clustered together and the value of each pixel would be assigned to the mean color of all the pixels in the cluster. This process is done until the algorithm converges.

Chapter 4

Methodology

4.1 Edge Detection

To extract the edges of objects in an image, a filter needs to be applied to that image. For the sake of this project a Sobel filter will be used on the Lena image.

4.1.1 CUDA

In order to implement an efficient version of edge detection in CUDA all memory types need to be utilized as effectively as possible. This is done by firstly reading in the image into texture memory. This allows all values of the image to be stored in coalesced memory. This then leads to fast reads of the image whenever it needs to be used. Secondly, the filter being used as well as the width and height of the image is stored in constant memory. This is beneficial as the filter is called for every pixel in the image and the width and height are used by every thread being executed. Before the image can be processed it needs to be padded. This is done in serial on the host. The convolution is done on the padded image.

4.1.2 MPI

The MPI implementation of edge detection is fairly straight forward. The image being processed is divided up into sections depending on the number of nodes being used. This allows for a somewhat equal distribution of the workload amongst the nodes. Each node will have a copy of the image but only process a single section. After each node completes its dedicated section the root node will perform a reduction on the processed values to obtain an output image. The root node will then write out the output image.

4.2 Otsu Thresholding

A histogram of an image is needed to obtain a suitable threshold. This threshold can be used to binarize the image, setting values with intensities lower than the threshold to black and intensities higher than the threshold to white.

4.2.1 CUDA

The CUDA implementation of this program is done in parts. Firstly, the histogram of the image is calculated. Only once that is done can the rest of the algorithm can continue. This is because the rest of the algorithm is dependent on the histogram. Afterwards a scan is done using the histogram to calculate the background weights and the summation at those weight levels. These variables will be used in the next step where each thread on the device will be used to calculate the variance at each possible threshold value of the image. i.e. There will be 256 threads running in parallel, one thread per intensity value. Once this is then the maximum variance is chosen which will provide the best threshold. Lastly, the threshold is applied to the image resulting in a binarized, segmented image.

4.2.2 MPI

The image histogram and relevant variables are calculated in serial. This is done in serial rather than with MPI as doing such computations in serial has a lower overhead than with MPI. These computations are done on the root node and then broadcasted to the rest of the nodes. The image is then divided somewhat equally amongst the nodes. Each node will then find the max variance of the portion of the image that it is given. Thereafter, a gather is called to the root node to get all the max variances from each node. The max variances are then compared and the highest max variance is chosen which provides the best threshold. The threshold is then applied to the image and the root nodes writes out the output image.

4.3 K-Means Clustering

To segment the image we need to cluster all of the pixels in an image to various clusters, then the cluster centroids will be set to the mean of all the pixels in the cluster and this process is repeated until convergence, thereafter we assign all the pixels in each cluster to one RGB color value.

4.3.1 CUDA

The CUDA implementation of this algorithm posed a few issues such as the updating of the cluster centroids which required the synchronization of all the various threads doing the per pixel computation of assigning the pixels to clusters. The reason the threads needed to be synchronized is to complete the update of the centroids process, this requires all of the pixels to be assigned to each of the clusters and then the centroids are updated by assigning them to the mean of all the pixels assigned to them, thus if each pixel is assigned to a cluster by a different thread then they need to come together to achieve the desired task. To solve this problem in CUDA we ran the assignment process in parallel and then exited out of the kernel and updated the clusters in serial, we then repeat the assignment process and continue until convergence.

4.3.2 MPI

The MPI Implementation was a bit easier to implement because the synchronization problem expressed with CUDA Implementation is alleviated by a simple function in MPI called MPI All Reduce, this function performs a reduction on the root node and then the result is broadcasted to the rest of the threads. Using the before mentioned function in conjunction with the MPI barrier we were able to synchronize the threads allowing most of the computation to be in parallel. After the new centroids have been broadcasted to all of the nodes then the algorithm goes through the cluster assignment process again until convergence.

Chapter 5

Experiment Setup

The timing of each algorithm will be compared against a serial implementation of itself. This will provide a base to obtain the actual speed up of the parallel implementations of each algorithm. The standard Lena image (1024x1024) will be used in both the Edge Detection and Otsu Thresholding algorithms. This will allow for a fair comparison between the CUDA and MPI implementations of each algorithm. The Porchlicht image (3008x1960) will be used in implementations of the K-Means clustering algorithm for the same reason. It should be noted that if other images were used there would be time changes due to the various sizes of the image.

5.1 CUDA

The time taken up by the actual processing of the algorithm on the device will be recorded as Processing Time. The processing time will show the actual time taken up by the device. Any other time taken executing serial code in the implementation will recorded as Overhead. This is basically all the time taken up by the program that is not run on the device.

5.2 MPI

Once again the time taken up by the processing of the algorithm on the cluster will be recorded as Processing Time. The processing time shows the actual time taken up by the implementation of the algorithm. The overhead time of the implementation is the total time taken up by each MPI call. This overhead is due to the communication costs between nodes. All testing was done with 8 nodes and a single process per node.

Chapter 6

Results and Analysis

All testing was done on the machine with the specifications mentioned before. Each of the algorithms below will contain a table representing the recorded times of each implementation and an explanation as to what influenced the time obtained.

6.1 Edge Detection

Implementation	Serial	CUDA	MPI
Processing Time (ms)	47.49	0.23	5.992
Overhead (ms)	82.125	172.11	13.439

Table 6.1: Edge Detection Results

It can be seen table 6.1 above that the processing time in both the CUDA and MPI implementations were faster than the serial implementation. This is due to the CUDA and MPI implementations having a portion of the workload per thread/node which execute in parallel.

6.1.1 Processing Time

The MPI implementation distributes sections of the image amongst the 8 nodes which then execute in parallel to process the image. The Lena image used is a 1024x1024 pixel image which is a total of 1048576 pixels. This means that each node is processing at least 131072 pixels in serial. This suggests that 8 pixels are being processed in parallel 131072 times. The device used for the CUDA implementation contains 1280 cores which means 1280 computations are executed in parallel. This suggests that 1280 computations are run in parallel roughly 819 times.

Therefore, it can be seen that the CUDA implementation is theoretically supposed to be much faster than the MPI implementation with regards to processing time, this is evident when observing the results above. Additionally, the image being processed is stored in texture memory, and the filter as well as relevant variables are stored in constant memory. This allows for faster computation times as texture memory stores the image data as coalesced memory which then reduces the time needed to read the data, and other variables stored in constant memory provide read speeds as fast as reading from the register which then lowers the processing time even more.

6.1.2 Overhead

In the MPI version of Edge Detection the root node reads in the image and broadcasts the image data, and the width and height of the image to all other nodes. Each node will then process it's portion of the image which resulted in a much faster processing time than serial. At the end a reduction is done on the output of each node. The root node then writes out the output image file. These communications between nodes lead to the overhead costs as can be seen in the table above.

In the CUDA implementation of edge detection different memory types are used. The image is stored in texture memory and the filter being applied and other necessary variables are stored in constant memory. Copying to these memory types is a once of execution and should hardly effect the overhead time. However, the padding of the image as well as the memory allocation for data being used in the program all effect the overhead of the implementation as it is all done in serial which then explains the overhead time recorded in the table.

The following image is the output of the edge detection algorithm. All 3 implementations produced the exact same output image.

Figure 6.1: Lena Edge Detection

6.2 Otsu Thresholding

Implementation	Serial	CUDA	MPI
Processing Time (ms)	12.26	2.342	3.7
Overhead (ms)	56.276	157.59	0.35

Table 6.2: Otsu Thresholding Results

It can be seen in table 6.2 that the MPI version of the algorithm has a much faster processing time as compared to the serial implementation. The same can be said for the CUDA implementation. It can be sen that CUDA has the fastest processing time, however, it also has the highest overhead.

6.2.1 Processing Time

Again, in the MPI version the image is divided into sections and is distributed to the 8 nodes. Each node then executes in parallel and processes it's portion of the image. The Lena image used has a total of 1048576 pixels. This means that each node is processing at least 131072 pixels in serial. This suggests that 8 pixels are being processed in parallel 131072 times. As per the device specifications previously mentioned, it can be said that 1280 computations are run in parallel roughly 819 times.

Therefore, the CUDA implementation is supposed to be faster than the MPI version with regards to processing time. Additionally, the image being processed is stored in texture memory, and the histogram of the image is then stored in constant memory after it has been calculated. This allows for faster computation times as texture memory stores the image data as coalesced memory which then reduces the time needed to read the data, and the histogram stored in constant memory provides faster read speeds than global memory.

6.2.2 Overhead

In the MPI version of Otsu Thresholding the root node reads in the image and broadcasts the image data, and the width and height of the image to all other nodes. The histogram of the image is then calculated in parallel and the relevant variables such as the partial sum of the weights and summations are calculated in serial. Each node will then process it's portion of the image. At the end a gather is done on the output of each node which returns the maximum variance of that section of the image. The root node then compares the variances of each section and selects the highest one which provides the best threshold. The root node then writes out the output image file which is binarized using the calculated threshold.

In the CUDA implementation of Otsu Thresholding different memory types are used. The image is stored in texture memory and the histogram calculated is stored in constant memory. The partial sum calculated for the weight and sum variable are done in parallel, storing the histogram data in shared memory. Copying to these memory types is a once of execution and should hardly effect the overhead time. The calculation of the histogram is done in parallel, however, each of the calculations afterwards are dependent on the histogram. This means that the histogram calculation needs to be done in it's own kernel. This leads to several kernel launches needing to be made in order to implement this algorithm which increases the overhead and processing times of the implementation.

The following image is the output of the edge detection algorithm. All 3 implementations produced the exact same output image.

Figure 6.2: Lena Otsu Thresholding

6.3 K-Means Clustering

Implementation	Serial	CUDA	MPI
No. of Clusters	4	4	4
Processing Time (ms)	1557.543	73.777	462.322
Overhead (ms)	519.0981	537.41	93.058
Serial Computation(MPI)	-	-	306.85
No. of Clusters	6	6	6
Processing Time (ms)	1478.05	73.199	553.744
Overhead (ms)	503.15	548.71	91.77
Serial Computation(MPI)	-	-	302.96
No. of Clusters	8	8	8
Processing Time (ms)	1882.838	67.39	567.193
Overhead (ms)	553.867	554.447	92.15
Serial Computation(MPI)	-	-	308.51
No. of Clusters	16	16	16
Processing Time (ms)	3504.69	54.17	695.315
Overhead (ms)	512.338	538.158	93.141
Serial Computation(MPI)	-	-	320.023
No. of Clusters	32	32	32
Processing Time (ms)	6714.514	55.772	1112.971
Overhead (ms)	515.613	538.88	126
Serial Computation(MPI)	-	-	280.67
No. of Clusters	64	64	64
Processing Time (ms)	13002.209	53.31	2091.96
Overhead (ms)	516.549	555.036	131.73
Serial Computation(MPI)	-	-	279.23
No. of Clusters	256	256	256
Processing Time (ms)	25031.26	78.73	4163.67
Overhead (ms)	511.875	519.95	247.699
Serial Computation(MPI)	-	-	270.68

Table 6.3: K-Means Clustering Results

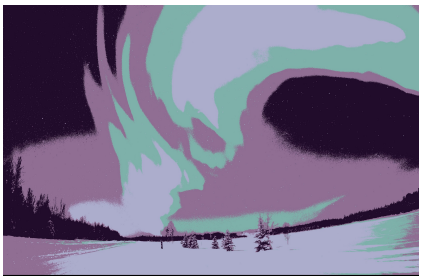
Above is the table of results pertaining to the K-means algorithm with the various implementations listed. As we can see the serial algorithm is always the slowest, this is to be expected as it runs the entire process on only one thread , when we compare this to MPI , which was run on 8 nodes , we see that there is a distinct speed up , this is due to us partitioning the iteration space in order to divide the workload evenly among nodes. Another section that was added to the table is the serial computation for MPI which refers to the time spent doing serial computation where only the root node is actively working.

6.3.1 Processing Time

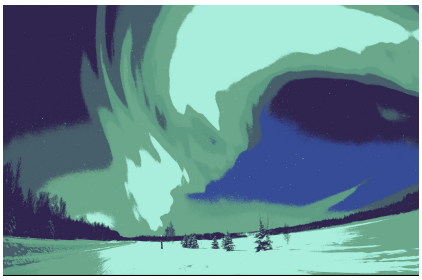
Next we consider the CUDA implementation which is clearly the fastest platform of the listed three. The presumed reason that the CUDA implementation is so fast is due to the number of threads that can be run concurrently , which is far more than we dared to test on the cluster. Due to the high number of threads that CUDA supports, we were able to run over 6000000 threads concurrently on the Porchlicht image as we ran one thread per pixel in the 3008x1960 pixel image, this would naturally speed up the processing time significantly which resulted in extremely fast and similar processing times regardless of the increase in clusters. I believe that the similarity in the processing times is due to the serial cluster centroid update which falls as part of the processing time.

6.3.2 Overhead

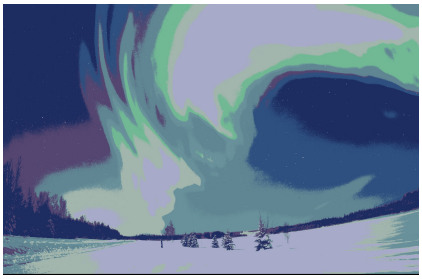
Next we will discuss the overhead , of which there is not much to say. The overhead for all methods stays consistent despite the number of clusters considered in each implementation. With regards to what overhead means, we see that in CUDA overhead is the time in which the process is not performing the core function of the algorithm, this includes time for memory allocation as well as the time to read and write the images. For MPI , overhead mainly refers to the communication overhead that is present in the use of various MPI Broadcast related functions.



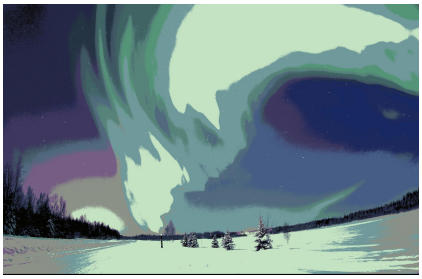
(a) 4 Clusters



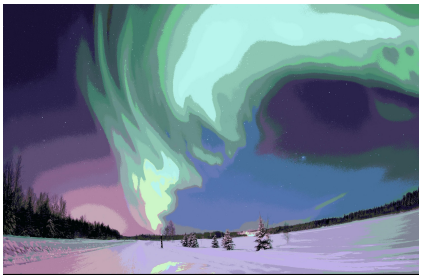
(b) 6 Clusters



(a) 8 Clusters



(b) 16 Clusters



(a) 32 Clusters

(b) 64 Clusters

(a) 256 Clusters

(b) Original image

Above are the results of the clustering algorithm when run with different amounts of clusters.

Chapter 7

Conclusion

In conclusion we can clearly see that among all of the implementations of the all of the algorithms, the CUDA implementation was always the fastest, we believe this to be because of the ability to split the images up in such a way that the algorithm does per pixel computation, this allowed us to divide the iteration space up in the most optimal way as well as having the greatest amount of threads running concurrently. When we compare this result to the MPI implementation we realize that MPI would only be comparable if the cluster was big enough to compete with the immense amount of threads that can be run concurrently in CUDA. This leads us to believe that CUDA is the best platform for general high performance computing as it is easier to get a graphics card than to setup a cluster that is able to run over 10000+ threads, but if one had the resources to set up a massive cluster then the MPI implementation should be faster but would require further testing to confirm this suspicion.

Chapter 8

References

[1] Andrea Trevino. 2018. Introduction to K-means Clustering. [ONLINE] Available at: https://www.datascience.com/blog/k-means-clustering.

[2] CS221. 2018. CS221. [ONLINE] Available at: http://stanford.edu/ cpiech/cs221/handouts/kmean

[3] Otsu Thresholding - The Lab Book Pages. 2018. Otsu Thresholding - The Lab Book Pages. [ONLINE] Available at: http://www.labbookpages.co.uk/software/imgProc/otsuThresh

[4] Automatic Thresholding. Available at: http://www.math.tau.ac.il/ turkel/notes/otsu.pdf

[5] Gupta, S. and Mazumdar, S.G., 2013. Sobel edge detection algorithm. International journal of computer science and management Research, 2(2), pp.1578-1583.