GRAPHICAL PROCESSING UNIT (GPU) BASED 3D SEISMIC SOBEL EDGE DETECTION

BY

Abdulaziz AlSharikh

A Thesis Presented to the

DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

Computer Science

May 2010

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **ABDULAZIZ S. AL-SHARIKH** under the direction of his thesis adviser and approved by his thesis committee, has been presented and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER SCIENCE.**

Thesis Committee

Dr. Farag Azzedin (Adviser)

Dr. Muhammad Shafique (Member)

Dr. Saleh Al-Dossary (Member)

Dr. Kanaan Abed Faisal
Department Chairman

Dr. Salam A. Zummo
Dean of Graduate Studies

2/10/10
Date

DEDICATION

This thesis is dedicated to my parents, my loving wife, and my two wonderful children: Saud and Ali.

ACKNOWLEDGEMENTS

In the name of Allah, Most Gracious, Most Merciful

All praise and glory to Almighty Allah (SWT) who gave me courage and patience to carry out this work. Peace and blessing of Allah be upon last Prophet Muhammad (Peace Be upon Him).

Acknowledgement is due to King Fahd University of Petroleum and Minerals, and to Saudi ARAMCO for supporting this research.

My unrestrained appreciation goes to my advisor, Dr. Farag Azzedin, for his tremendous support, guidance and encouragement. I am thankful for his lenient nature and overlooking patience. I also wish to thank my thesis committee members, Dr. Mahammad Shafique, and Dr. Saleh Al-Dossary for their support and contribution. I would like also to thank Osama Nagib for his support and help in configuring and setup the environment.

Finally, I wish to express my gratitude to my family members for being patient and for their words of encouragement to spur my spirit at the moments of depression.

TABLE OF CONTENTS

LIST OF FIGURES

THESIS ABSTRACT

Name: Abdulaziz S. Al-Sharikh
Title: Graphical Processing Unit (GPU) Based 3D Seismic Sobel Edge Detection
Major: Computer Science
Date: May 2010

The size of 3D seismic data is increasing rapidly, and since it needs some filtering and feature extraction in order that it would make sense for the geologist and seismic data interpreters. Many algorithms were designed to do so, such as Sobel edge detection. However those algorithms consume a lot of resources such as CPU and memory, so running those algorithms on a single CPU may take days. CPUs cluster came to solve the slowness problem, however it came with high cost in terms of money, manpower, electricity, and space that makes implementing this solution difficult for small organizations. In this proposal, we introduce the utilization of multi-core NVIDIA graphics card (GPU) to detect edges in 3D seismic data via Sobel algorithm and CUDA language. The implementation requires modification in data distribution and modification in Sobel edge detection algorithm to be able to run on the new structure of NVIDIA graphics card.

ملخص الرسالة

الاسم: عبدالعزيز الشارخ

عنوان الرسالة: استخدام المعالجة الرسومية و فلتر السوبل الاستخراج الحواف الجيولوجية

التخصص: علوم الحاسب الآلي

تاريخ التخرج: أيار ٢٠١٠

حجم البيانات السيزمية ثلاثي الابعاد في تزايد سريع ، ونظرا لأنه يحتاج إلى بعض الفلاتر لأستخراج سمة من أجل أن يكون له معنى بالنسبة للجيولوجي والمترجمين الفوريين للبيانات. تم تصميم خوارزميات عديدة للقيام بذلك ، مثل الكشف عن حافة السوبيل. لكن تلك الخوارزميات تستهلك الكثير من الموارد مثل وحدة المعالجة المركزية والذاكرة ، و قد تستغرق ايام لكي تكتمل النتيجة. جاءت كتلة وحدة المعالجات المركزية لحل مشكلة بطء ، غير أنها جاءت مع ارتفاع تكلفة من حيث المال والقوى العاملة ، والكهرباء ، وكبر مساحة المكان الذي يجعل من الصعب تنفيذ هذا الحل بالنسبة للشركات الصغيرة. نقدم في هذا الاقتراح استخدام وبطاقة الرسومات نفيديا (الجرافيك) متعددة النواة للكشف عن حواف في البيانات السيزمية عبر الخوارزمية سوبيل واللغة كودا. التنفيذ يتطلب تعديل في توزيع البيانات وتعديلها في حافة سوبيل خوارزمية الكشف عن أن تكون قادرة على تشغيل على الهيكل الجديد لبطاقة الرسومات نفيديا.

CHAPTER 1

INTRODUCTION

Earth exploration is considered to be one of the first activities in oil industry and it has been rapidly expanded during the last decades. Oil and gas exploration is an extensive process that starts with 2D/3D seismic survey and ends up with oil final products such as gasoline and diesel. The seismic data survey is one of the initial and crucail activities when looking for oil, the results of these surveys consist of huge volumes of data that are gathered to be inspected and filtered to extract some geological features that help locate the oil and gas reservoir.

Seismic data consists of millions of traces each of which consists of hundreds of amplitude values. The data is gathered by sending continuous wave signals into the ground and at the same time the located sensors (geophones) on the surface record the reflection of those wave and then convert them to electrical signals. Those waves are of two types, some get reflected back to the surface, and some propagate deeper. Each receiver generates a trace of amplitudes which represent the strength of the wave

reflection at a specific depth .Those traces are all grouped together to construct a seismic volume that contains millions of traces of a few thousand amplitude values each as shown in figure 1.1. After recording the data of a particular wave, all sender and receiver devices are moved to a new location and the whole process of sending/receiving waves is repeated again [10] [28] [18].

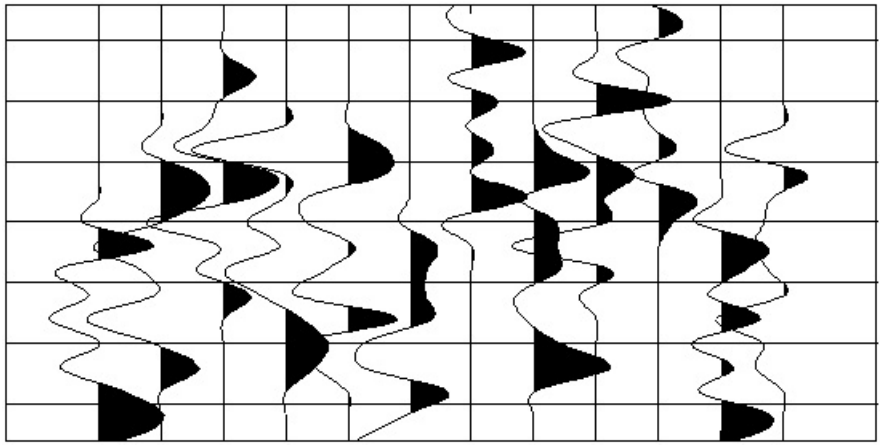


Figure 1.1: Seismic traces. (adapted from (www.bki.net/ricc/xtra/oppgavermatlab.htm))

The method of gathering seismic data has evolved during the last 2 decades due to change in technology and techniques. The digital recording method started during the 1960s. The interpretation of the seismic data that was mainly depending on the some features were manually extracted by the interpreters themselves, but as the technology advanced during the 1970s, several two dimensional (2D) attributes of the seismic data such as texture analysis and pervasive use of color were introduced which helped better and accurate interpretation [8]. However, the 2D view has a disadvantage of being restricted to a vertical cross-section view only. This technology was overtaken by the 3D seismic acquisition in the 1990s as shown in figure 1.2 which allowed the

interpreters to view the data in a horizontal view or time slice section view. The 3D acquisition is considered to be the most successful method of data acquisition as it provided a clearer and more obvious reflection of the subsurface layers [3].

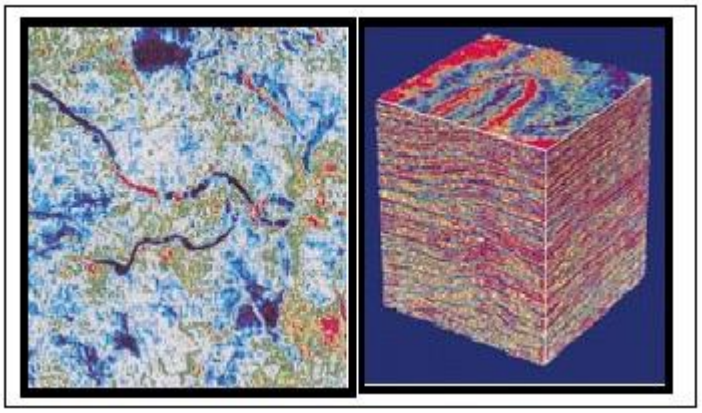


Figure 1.2: 2D images and 3D volume. (adapted from (www.ges.hu/e_szodp.htm))

However, those huge data volumes have several problems. The first of the problems is the noisy and unclear data due to some interruption during the wave recording process. The second problem is the time required to process the data to extract features such as channels and faults, which in some cases may take hours or days to finish. This long processing time is due to two factors: the first is the huge size of the 3D seismic data which for example can reach several Gigabytes, the second factor is the speed limitations of the processor. With regards to the first problem, there are several techniques used to make the data meaningful for the geologist [2] [24]. With respect to the second problem, there are several successful solutions to speed up processing; however this speed up comes at a high cost in terms of money, space, and electricity that for small or medium size organizations cannot afford [1] [17].

Edge detection is one of the solutions used to clarify the visualization of the data so it makes sense for the geologist during the data interpretation process [2]. Edge detection filter is simply an operator with certain values that cover the whole 3D data volume doing mathematical operations. This operator uses the convolution as the main operation which is basically applying a 3D mask on the data thus resulting in better extraction of the seismic features [19][12][26].

In this proposal, I will concentrate only on two categories of edge detection, the Gaussian, and the Gradient. The basic idea behind all the Gaussian operators is to work in symmetric with edge and applying a smoothing filter as a pre processing step. An example of this category is the Shen-Castan operator and the Canny operator which are considered two of the best edge detection operators. However it was excluded from this research since it works with 2D data only. On the other hand, the Gradient operator works by calculating the degree of the seismic data slop by computing the first and second derivative of the gray color of the seismic data [24][6]. The Sobel edge detection is an example of the Gradient operators that is considered to be fast and suitable for 3D data. or this reason it will be considered as the edge detection algorithm in this research.

Performing the edge detection process on multi-CPU is the solution to speed up the seismic data processing step, however this solution is costly and not easy for small organizations to implement. Therefore, some proposed to utilize the multi-core GPU

to take the load of the CPU using stream processing technology and programming the code using OpenGL [17] [23]. The multi-core GPU solution seems to better fit individuals and small companies since it does not required a lot of resources in terms or money, manpower and space.

Usually the acquired seismic data is noisy and needs a lot of pre-processing filtering to have clear and meaningful information. This problem would lead to another problem, which is the time taken to do the pre-processing step, which is considered to be long especially with a big volume of data. Figure 1.3 shows the problem of seismic data associated with each step.

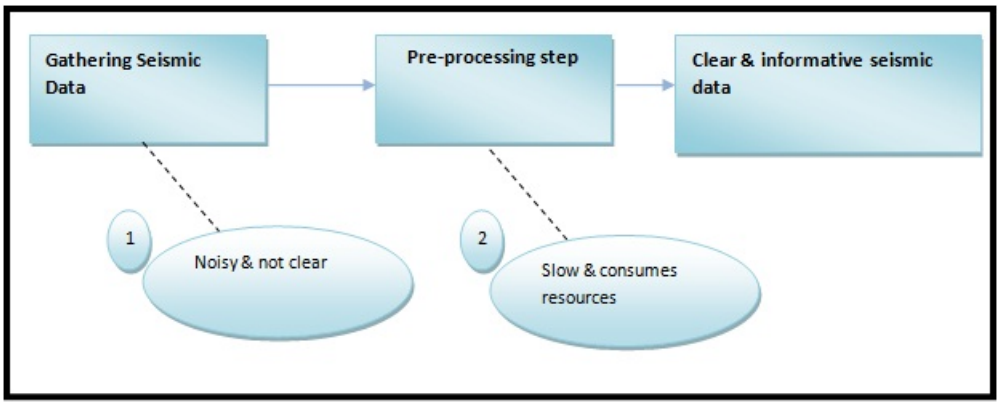


Figure 1.3: Problems with seismic data processing.

The noisy data results from the distraction during recording the wave signals [28], and as a result, it is hard for the seismic interpreters to obtain any geological features. For this reason, some techniques have to be applied on the seismic data to get better quality data and to make it easier for the interpreters to detect geological features.

As mentioned earlier, there are techniques to make reading seismic data easy for interpreters. Those techniques are a group of filters that can be applied to detect some features in seismic data. For example, it takes about two minutes to apply an edge detection algorithm on a very small amount of data that consists of only 256 traces [7]. This processing time will extend rapidly, such that might take not only hours but days as the size of the data gets larger. The reason for this is that those filters consist of 3D masks that cover all the 3D seismic data. Those 3D filters cover the whole seismic volume while doing calculations to highlight the seismic features that they are designed for. For large seismic volume, running those filters in a single CPU takes a lot of time which is considered a major problem.

1.1 Summary

Edge detection is an important tool that helps seismic interpreters to have better look at the seismic data to locate seismic features. However the cost of applying edge detection on seismic data would be high if we consider the large size of the 3D seismic data which sometimes reaches to several gigabytes. Throughout several decades, several techniques were suggested and implemented to make edge detection more accurate and faster.

In this research, we propose the use of new NVIDIA multi-core GPU to perform the edge detection process on 3D seismic data using CUDA language. The Contributions

are expected to be as follows:

- Utilizing the GPUs multi-core functionality to take the load off the CPU in doing the mathematical calculation during the Sobel edge detection process.
- Modifying the Sobel algorithm in data decomposition .
- Modifying the Sobel algorithm to map the 3D data into the 2D thread structure of the GPU.

The thesis is organized as follows: chapter 2 covers details of edge detection techniques, chapter 3 is a literature review that covers both solutions of noisy, unclear seismic data, and slow seismic data processing, chapter 4 talks about the motivation of writing this proposal and contributions, chapter 5 covers the proposed solution of performing Sobel edge detection on the GPUs, chapter 6 covers the performance evolutions, chapter 7 presents results and discussion, and finally chapter 8 talks about conclusion and future work.

CHAPTER 2

EDGE DETECTION

2.1 Overview

Edge detection is one of the most important tools used in image processing applications to extract information from the images as pre-processing step of feature extracting. The edge detections process works by detecting outlines of an object and boundaries between objects and the background of the image. In addition, the edge detection filter can enhance the appearance of noisy and unclear images. Most of the Edge detection algorithms are a matrix area gradient operation that determines the level of variance between the different pixels [19].

2.2 Problems with edge detection

There are three significant problems that come along with edge detection [9]. The first problem is detecting some edges with a noisy image. For example, there might be some intensity changes in the pixel but no edge exists and it is possible to completely

ignore an existing edge. The second problem is edge localization such that with a noisy image, the edge might get shifted to another location. The third problem is the difficulty to distinguish between a high frequency edge that appears very clearly on the original image and some noise that has a high frequency values in a way when applying some noise elimination filter to smooth the image, it is hard to tell whether this is an edge or a noise.

2.3 Convolution

Convolution is considered the main fundamental algorithm to many common local image processing operators, such as edge detectors. Typically, convolution is performed in a way by which two arrays of numbers, of different sizes, but of the same dimension, multiply together to produce a third array of numbers of the same dimension. In terms of image processing, convolution is the summation of pixels in the neighborhood of the source pixel.

If the first array, e.g. 2D seismic time-slice, has M rows and N columns, and the second array, known as kernel, has m rows and n columns, then the size of the output array (O) will have $M-m+1$ rows, and $N-n+1$ columns [12], as shown in Figure 2.1.

Mathematically, we can express the convolution as:

$$O(i,j) = \sum_{k=1}^{m} \sum_{l=1}^{n} I(i+k-1, j+l-1) K(k,l) \tag{2.1}$$

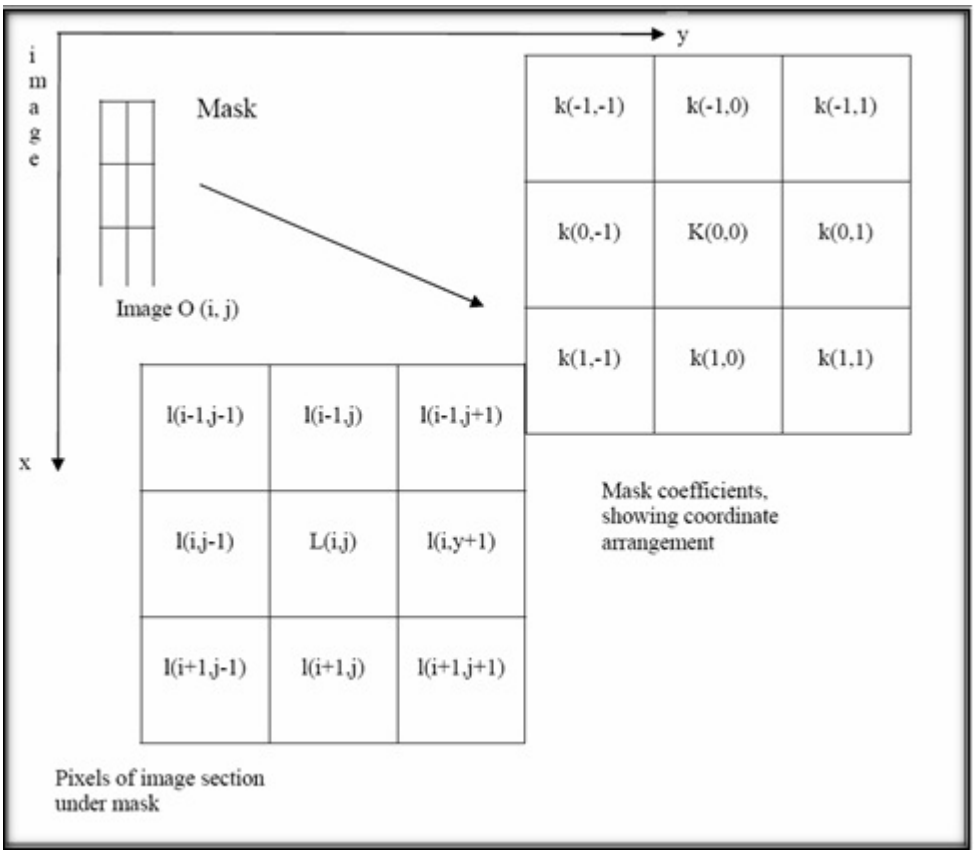


Figure 2.1: Convolution mechanism.(adapted from [12])

where $K()$ is the convolution function that is being applied on the input image $I()$.

The equation must be applied on the values of $i = 0, 1, 2$ to $M - 1$ and $j = 0, 1, 2$ to $N - 1$, to obtain a complete filtered image O [12]. By doing this we make sure that the mask processes all pixels in the image. It is implied that this repetitive procedure of masking can be a very time-consuming task and needs heavy computational power when $M * N$ is large. 3D convolution was discussed in [26] on a three-dimensional grid with initial function F on Domain $D1$ and three-dimensional kernel K defined on domain $D2$, the result G of the convolution at any point (u, v, w) would be:

$$\forall(u,v,w) \in D1, G(u,v,w) = \sum_{(i,j,k)\in D2} K(i,j,k)F(u-i,v-j,w-k) \tag{2.2}$$

The mechanism that the 3D convolution is to move the 3D kernel to cover the whole 3D data volume starting for example from the upper left corner where the coordinates are $(0,0,0)$ as in the Figure 2.2 below and compute the convolution product for each point in the volume.

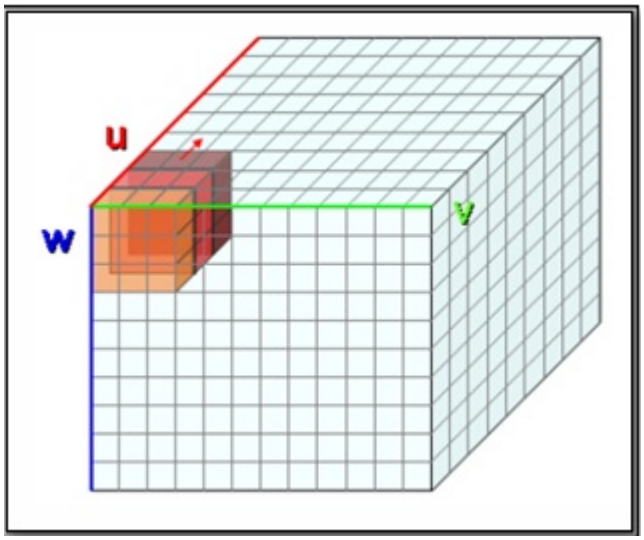


Figure 2.2: Three-dimensional convolution. (adapted from [26])

2.4 Summary

Edge detection is an important tool used to enhance looking at the seismic data; this technique comes with some problems that were overcome during the past decades. Convolution is considered to be basic operation used in edge detection. It consists

of a 2D/3D operator that contains values used to change the outcome of the original data values. Applying that operator would change the way that the data look and would help to better enhance the quality of the data.

CHAPTER 3

LITERATURE REVIEW

In this section, we are going to cover several papers and researches conducted on the two main parallel methods of detecting edges, by using a cluster of CPUs and by using GPUs. However, before explaining both methods, we are going to talk about the importance of edge detection in solving the problems of unclear data and covering the two categories that most of the edge detection operators fall in.

3.1 Solutions for unclear data using edge detection

The problem of unclear data is that it is hard for the geologists to interpret the seismic features [28]. Those features such as natural fractures, faults, channels, etc would be presented as edges in the seismic data. As such detecting edges in seismic data is very crucial and interpreters are paying great attention to correctly locating those features in order to increase productivity of their exploration efforts such that millions of dol-

lars and time would be saved if they got the right tool and mechanism of detecting those features.

However, detecting fine features cannot be done easily, and many filters have to be applied to the data to have those features recognized correctly. According to [2], there are several edge detection algorithms such as Canny, Haralick, Sobel, and Nalwa-Binford which are considered as a core tool which should be used in the exploration process to speed up and enhance the exploration efforts of detecting geological features.

All the edge detection algorithms consist of the edge operator. This operator is the kernel where all the mathematical operations are computed. It is simply a neighborhood operation which tries to find for each pixel to what extent this particular pixel neighborhood can be portioned by a divider path such that all the pixels on one side would have a range of common values and the pixels on the other side of that divider would have different range of common values. [24]. There are two categories that most of the edge detection operators fall in which are the Gaussian operators and the gradient operators.

3.1.1 Gaussian operators

According to [24], there are several algorithms of extracting edge operators that fall into the Gaussian edge detectors category. Those operators work in symmetric with respect to the edge and reduce the noise by doing smoothing filters.

One of the famous operators that use the Gaussian method is the Canny and Shen-Castan. The Canny is based on specific mathematical models for edges and is considered to be the best edge detection algorithm because it meets all the three performance criteria of edge detection. The first, is minimizing the saturations of detecting false edges and missing actual edges. The second is minimizing the distance between the detected edges and actual edges. The last criterion is minimizing multiple responses to an actual edge. the first step of Canny algorithm is smoothing the image by convolving with a Gaussian filter. The second step is passing the smoothed image to convolution operation with the derivative of the Gaussian in both the vertical and horizontal directions.

Another algorithm that belongs to the Gaussian family is the Shen-Castan algorithm that gives better results than the Canny at finding the precise location of the edge pixel. The operator provides a filter with exponential impulse response in order to detect step edges using a second derivative filter. To implement this filter, the filter was designed based on a recursive form (a first-order filter), and results obtained showed a good resistance to noise and a good localization of contours, but the optimality was not clearly defined [24] and [6].

Although the Canny edge detection is considered one of the most accurate edge detection algorithem, however it cannot be applied on 3D seismic data since it operates only on 2D images. For this reason, it was not selected as the edge detection

filter in this research.

3.1.2 Gradient operators

The gradient operator works by computing the degree of a slope of the seismic data as discussed in [24] and [2]. In other words, it is using the first or second derivative of the amplitude values of the seismic data so that the first derivative marks edge points and the second derivative calculates two impulses on either side of the edge. The benefit of this is that the line that would be drawn between the two impulses is the location where this line crosses the zero axis of the center of the edge.

One of the basic gradient operators is Laplacian operator. It looks for the correct places of the edge so it would test the neighborhood pixels but does not take in consideration the values at corners and curves. it works as a second derivative operator that is used to detect the zero-crossings of image intensity and often resulting more exact edge detecting result. Unfortunately, because of its operator involves two derivatives, it is highly affected by noise [29]. Since the seismic data is usually noisy by nature, the Laplacian operator cannot be applied in this case and that left the Sobel as the edge detection algorithm to be chosen for this research.

Another well-known basic gradient operator is the Sobel [5] which employs local gradient that detects only that edge that has certain orientations however it performs poorly when the edges are blurred and noisy. For this reason, and to have better

results, Sobel operator can be combined with directional operators to approximate a rotationally constant operator.

In summary, the Figure 7.1 shows the taxonomy of the solutions for unclear seismic data using edge detection.

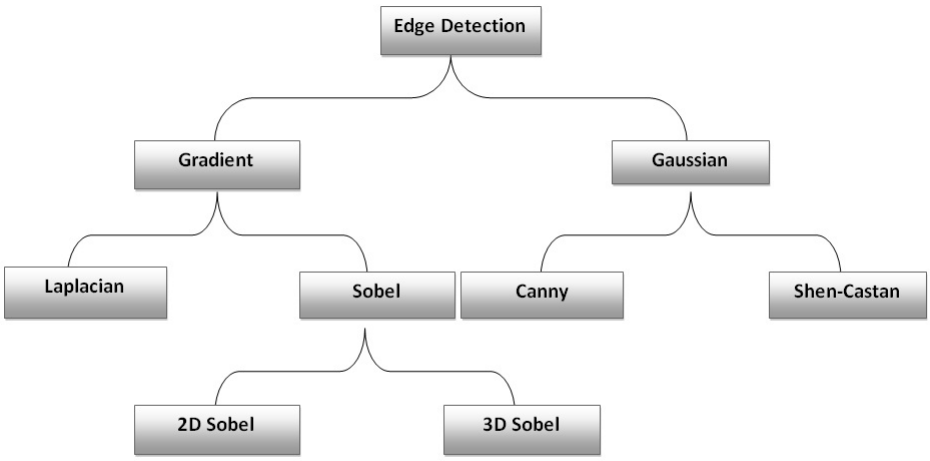


Figure 3.1: Taxonomy of the solutions for unclear seismic data using edge detection.

The Sobel edge detection as discussed in [27], perform edge detection on the assumption that an edge exists if there is a discontinuity in the intensity function, or where there is a sudden intensity gradient in the image. Therefore, by taking the first derivative of the intensity function and then finding the points where this derivative is at its maximum value, the edge could be located. The gradient in Sobel is a vector that consists of values that represent how rapid pixel value are changing along the x and y directions of the 2D image.

The gradient values can be calculated using the following equations

$$\frac{\partial f(x,y)}{\partial x} = \triangle x = \frac{f(x+dx,y) - f(x,y)}{dx} \tag{3.1}$$

and

$$\frac{\partial f(x,y)}{\partial x} = \triangle y = \frac{f(x,y+dy) - f(x,y)}{dy} \tag{3.2}$$

The dx and dy values represent the number of pixels between two points. So assigning the value 1 to each of dx and dy (pixel spacing) is the point which pixel coordinates are (i, j) so $\triangle x$ and $\triangle y$ will be calculated as follows.

$$\triangle x = f(i+1, j) - f(i, j) \tag{3.3}$$

$$\triangle y = f(i, j+1) - f(i, j) \tag{3.4}$$

The magnitude measure which is change of the gradient at the point (i, j) can be calculated as follows.

$$M = \sqrt{\triangle x^2 + \triangle y^2} \tag{3.5}$$

And the gradient direction (theta) can be calculated using

$$\theta = \begin{bmatrix} \frac{\triangle x}{\triangle y} \end{bmatrix}$$

Since the Sobel operator is an example of the gradient method, the operator itself is a discrete differentiation operator which computes an approximation of the gradient of the image intensity function. The $\triangle x$ and $\triangle y$ masks can be represented as following.

$$\triangle x = \begin{bmatrix} -1 & 1 \\ \\ 0 & 0 \end{bmatrix}$$

$$\triangle y = \begin{bmatrix} -1 & 0 \\ \\ 1 & 0 \end{bmatrix}$$

By applying these masks, the upper left value of the mask is forced over each pixel of the image, also the $\triangle x$ and $\triangle y$ values are calculated by applying the mask coefficients in a weighted sum of the value pixel (i, j) and it is neighbors.

However, some Sobel operators come with a larger mask size, for example $3X3$ or $5X5$ matrix , to eliminate errors resulting from the noise effect. Also the advantage of using an odd number of elements in the operator size is that the operator is centered so it can provide an estimate that is based on a center pixel.

$$\triangle x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$\triangle y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

The mechanism of the operator works as follows, first, it calculates the gradient of the image intensity at each point of the image by providing the direction of largest possible increase from light to dark and the rate of change in the direction. The outcome of performing this step is to show how easily the image changes at that point and how likely for that point to be part of the edge. In addition , it would calculate how the edge would be oriented. In other words, the gradient at each point can be represented as a 2D vector with the components given by the derivatives in the horizontal and vertical directions so that at each point the gradient vector points to the largest possible intensity increase and the length of the gradient vector would represent the rate of change in the direction.

Seismic data consists of many layers that construct the seismic volume, the 3D volume needs a different method of applying edge detection regardless of the type of the edge detection. For this reason Sobel edge detection was modified to handle 3D

data. Tertois and Frank in [26] discussed the 3D Sobel operator or what is sometimes called a kernel. Figure 3.2 shows an example of 3D Sobel kernel which computes the gradient in w direction.

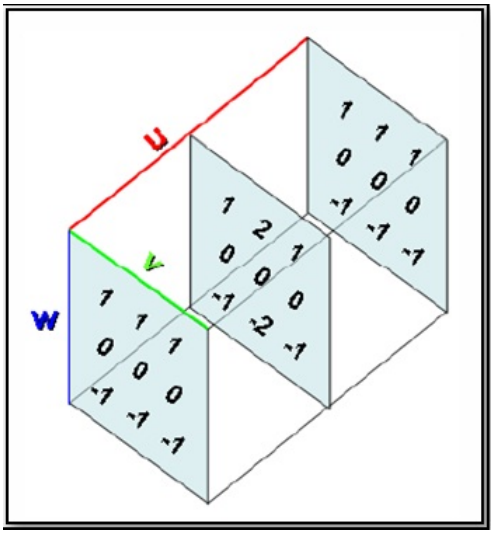


Figure 3.2: Sobel 3D convolution kernal.(adapted from [26])

One of the advantages of using Sobel edge detection [27] is edge orientation in which the kernel can decide the direction that is most sensitive to edges whether it is horizontal, vertical, a diagonal edge. Sobel operator would produce good results in noisy data, since the edges and noise would have the same property of the attributes' values which are usually at high frequency, the operator would normalize enough data to extract noisy pixels and as a result the edges would be clear enough to be detected. Some edges are not constructed in a direct change in intensity as previously mentioned but following gradual change in intensity due to poor focus, in such case, Sobel kernel can be sensitive to this gradual change.

3.2 Solutions for slow seismic data processing

Section 2.2 mentioned the main reason for the slowness in processing seismic data when running on a single CPU. Even with a fast processors as mentioned in [25], there is always a physical limitation. There are billions of transistors that are contained in single processors now-a-days, and the number is expected to increase rapidly, however it will reach to a state when there is no more spaces for any additional transistors. For this reason the idea of processing the data on multiple CPU using Message Passing Interface (MPI) during the 1990s got a lot of attention when it was applied, such that a great performance was obtained. Parallel processing has proven to be a viable solution to improve performance in seismic industry [25], and that was proven correctly by running the Phase Shift Plus Interpolation (PSPI) modeling algorithm on a 3D data with $(97 * 401 * 350) = 13.6$ millions amplitude values. The running time on a single processor was about 225 minutes while it took only 16 minutes when running on 32 processors. With 28 times faster seismic data processing, this solution showed great performance. However, there are different techniques of data partioning that are associated with different seismic data processing algorithms. Another technique in [9] that makes use of Prestack Kirchhoff Time Migration algorithms to process seismic data, was used on multi CPU cluster and there was a big performance, such that while it took 15 hours to process the data on a single CPU, it took only 1 hour to do the same job using 15 CPU.

3.2.1 Edge detection using cluster of CPUs

Detecting edges in 3D seismic data using cluster of CPUs was implemented in oil companies several years ago using a network of inexpensive computers (a cluster) using Linux operating System [1]. There are two types of clusters, the first is the clusters that are located in the IT (cold room) as racks of identical computers. The second type is the ad-hoc cluster which is a network of personal computers that are connected. The ad-hoc cluster has an advantage of its existing in most sites, however it is hard to administer and manage.

Each computer in a cluster is called a node of a cluster and in each node there are multiple CPUs and every CPU consists multiple cores, for example the dual core CPU. Inside the cluster, there are two CPUs per node, and each node consists of 4 cores, which means there are 8 computational cores per node. There are two types of memory in CPUs cluster; the shared and the distributed. With regards to the former, it is where al the cores of a single node read and write, so it is shared between multi cores that belong to a certain node. With respect to the later, it is the memory where nodes communicate and share data. The shared memory uses the OpenMP toolkit software to program where Massage Passing Interface (MPI) is used to program the distributed memory application. The edge detection algorithm was written using the MPI software toolkit.

The programming model that is being used is called Data parallel, in which the

same program is executed on every core of each cluster on a different data set but with two different data flow patterns or topologies, the master node topology and the peer topology. In the master node pattern, a master node is dedicated to partitioning the data and distributeing it among the rest of the nodes and then gathering the data to write it back on the disk, so it is only serial input/output operation from and to the storage disk by the master node. In the peer pattern, the only role of the master node is to point which part of the data that each node is responsible for, However reading and writing the data from and to storage disk is the node responsibility not the master node responsibility as shown in figure 3.3. The implementation of either topology depends mostly on the specifications of the storage disk in supporting parallel I/O and the capability of the network in handling the traffic of huge amount of data at the same time.

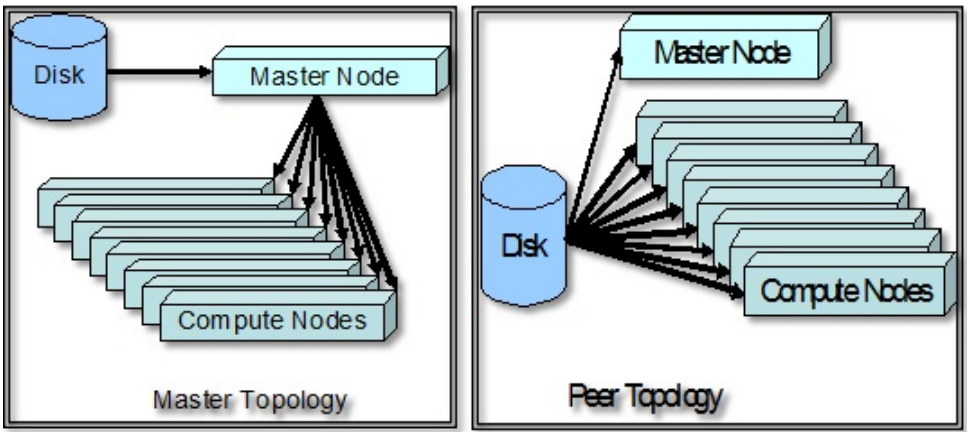


Figure 3.3: Master node topology Vs peer topology.(adapted from [1])

A single set of 3D seismic data in Saudi Aramco is very huge and sometimes reaches to 1 TB of size. This huge amount of data takes a long time to process when running

in a single CPU, so parallelizing the data to run on several nodes was a solution to get faster execution time. The 3D seismic data is stored in disk as a series of vertical slices where each slice is called a seismic line. The horizontal slice in the 3D volume is the slice where the waves are reflected and it is called the time slice. Those time slices in seismic data are the main criteria of parallel data decomposition. As show in figure 3.4 a group of those time slices is called a slab. Those slabs are being distributed among the nodes when processing the data while having one time slice overlap between every two adjacent slabs.

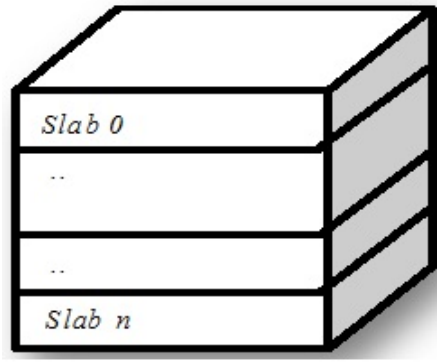


Figure 3.4: 3D seismic data model.(adapted from [1])

The data is decomposed as follows: The number of time slice in each slab is:

$numSlicePerSlab = numTimeSlices/NumCores$ where

$numSlabs = numCores$

if $numTimeSlices/NumCores$ is not an integer, the last node receives a different size of slab. If the slab size is greater than the RAM size of a node, the slab size is recalculated to fit the size of the RAM. For example, if there is 260 time slices to be distributed among 64 cores, and since 260/64 is not an integer, the first 63 cores are

going to receive four slices which is a slab, while the last core receives 8 cores. Figure 3.5 shows the flowchart of the whole process starting from reading data from the disk and ending by writing the resulting data back on the disk.

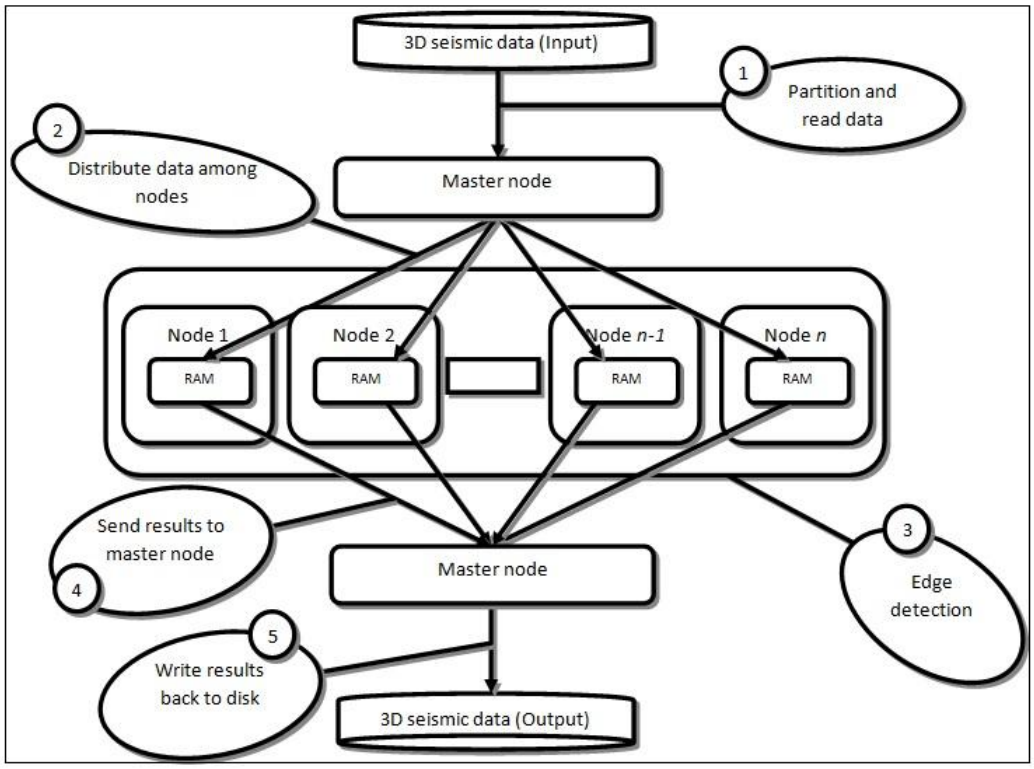


Figure 3.5: 3D seismic volume edge detection on cluster of CPUs (nodes).

3.2.2 Edge detection using GPU

One of the first attempts to accelerate seismic features extraction using GPUs was proposed in [17]. That attempt was based on stream processing technology which provides a software abstraction that allows multiple independent threads to be automatically generated during the execution time using the Brook language that targets the ATI graphics cards (HD4850 and HD4870) series.

Stream processing is the direct implementation of vector processing which is the core of the graphics card. The GPU is designed for single precision floating point operation that supports and optimized for three and four dimensional vectors. There are three processing phases in the stream processor, each of which is programmed separately using shader programs. The first phase is vertex processor which allows the programming to adjust and manipulate the incoming vertices to be positioned in a certain place on the grid. The second phase is the geometry processor which deals with the whole structure or shape instead of individual vertices. The third phase is the fragment processor which takes care of the rendering process such as assigning colors to different pixels. To allow these processes to be executed simultaneously, the GPU runs the same set of Shader programs on different vertices on a dedicated processor [23]. Figure 3.6 shows how GPU would handle multiprocessor activities.

Because of the hardware optimization, the vector implementation allows the computational intensity of the kernel to increase with little impact on the memory bandwidth. Both single stream and multiple streams were implemented in [17], the single stream which is the simplest implementation of algorithms that operate on one pixel only of the image with a single input/output. On the other hand, a more complicated version of the algorithm was modified to work on multiple streams by dividing the input and the output into multiple streams as shown in Figure 3.7.

The multi stream algorithms showed some improvement when simulated on the CPU with a speed up factor of 1.6 compared to a single stream. On the other hand, when executing the algorithm on the GPU, there is a speed up of 10 compared to

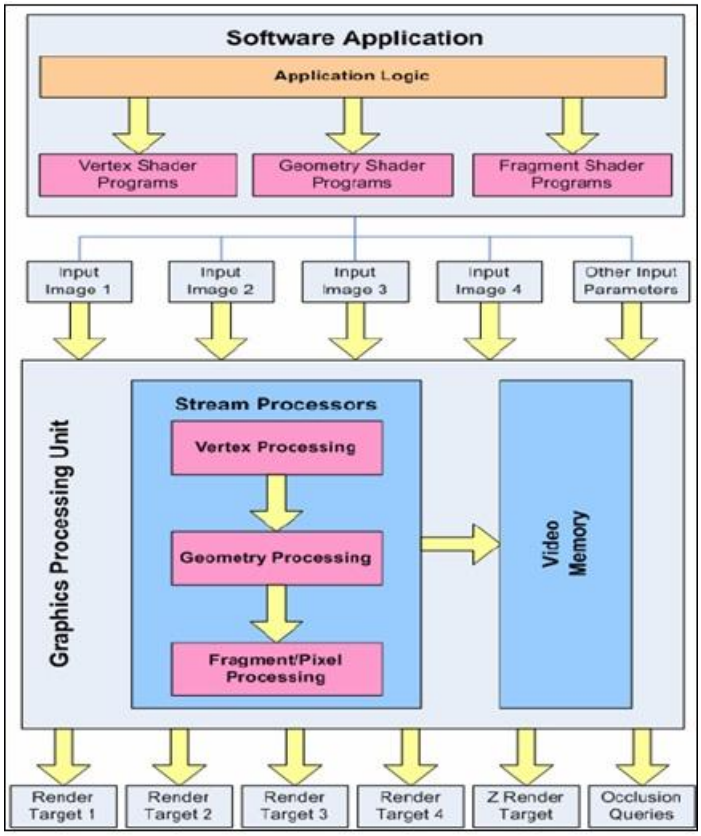


Figure 3.6: Architecture overview of the GPU.(adapted from [23])

the CPU performance with one stream on the GPU due to hardware optimization. However when it comes to using 4 and 8 streams, the performance remained the same. The worst was with 2 streams. The reasons of this strange result are from two factors; the first is the bandwidth limitation of exporting the output streams on the GPUs that were tested, the second factor is that the algorithm itself is not computationally optimized and any additional stream would result in an overload to the input/output processes.

In [7], they proposed extract faults (edges) in 3D seismic data using stream processing. Since the 3D texturing at that time was not supported in GPUs, they used a

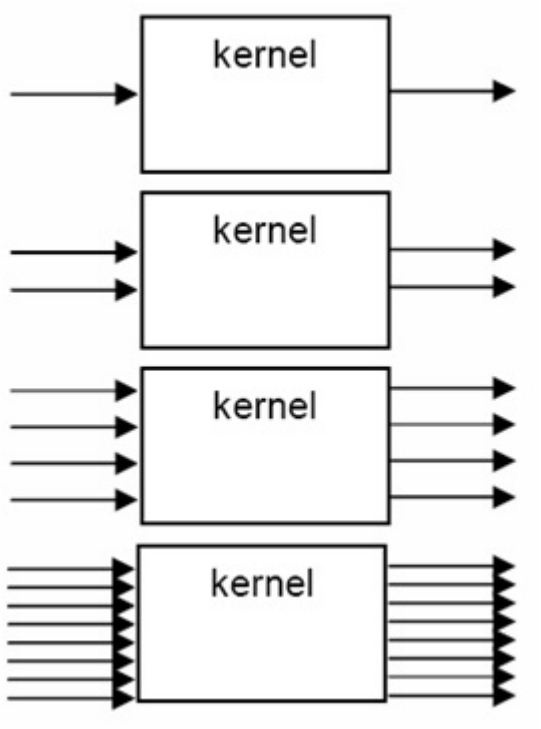


Figure 3.7: Kernels with 1, 2, 4 and 8 input streams.(adapted from [17])

set of 2D textures to represent a 3D volume where there are multiple rendering passes corresponding to a one 2D slice of the 3D seismic volume. The way that the output is computed is by simply using the rendering process which is defined as binding the sources textures. Then rendering a screen-size onto a target texture using vertex/fragment programs which run directly on the GPU without the need to render it back to/from CPU memory. Running the program on the Nvidia 7800 GTX graphic card using the Cg toolkit and Glew library for the GPU and the OpenGL extension, the system performed about 20-times faster than Pentium 4 3.6 GHz CPU. For example detecting edges of 3D seismic data with 256 slices took 2 minutes on the CPU while it took only 5.5 seconds on GPU.

The Canny edge detection algorithm in [23] was implemented using the Gaussian filtering method. The GPU implementation does not have a loop to process every pixel in the image, instead, each fragment program is executed for every pixel. To process a pixel coordinate, it needs to be enabled in the vertex shader program. The vertex fragment program enables the 1st texture coordinate channel and transforms each individual vertex that was queued for rendering to the fragment or pixel that the processing takes place in. The current pixel color is extracted from the input image and then converted to a luminance value. This value is compared to the threshold and the appropriate output fragment which is chosen and rendered. Using the OpenGL language for programming, the results showed that moving the intensive calculation from CPU to GPU came up with approximately 80 frames per second edge detection with an image of size $2048X2028$ in real-time. However, the problem with Canny edge detection algorithm is that it works on only 2D images and not suitable for 3D images as in seismic data.

In [16], seismic attributes were calculated using the NVIDIA 8800 graphics card that contains 128 processors. A program was developed using CUDA language to calculate three different types of attributes which are; frequency, reflection, and phase. For each calculated attribute, there are several pre-steps involved which are; creating complex number from the seismic data, convert the data from time domain to frequency domain using Fast Fourier Transform (FFT), removing the negative frequencies, and the last pre-step is to convert the values back from frequency domain to time domain.

Just after those pre-steps are performed, the three different seismic attributes can be calculated. Since the SIMD architecture is applicable for processing seismic data. Therefore, the 128 processors in the GPU can work in parallel to process different data at the same time. However there are extra steps that are involved in data copying when it comes to GPU; those steps are copying data from CPU to GPU memory, and vice versa.

Results showed a huge improvement on the performance when utilizing the GPU multi-core functionality. Using 512 MB of 3D seismic data, it took about 206 second to calculate the reflection attribute on single CPU; on the other hand, it took only 3.6 seconds on GPU. The speedup that was gained is about 100 times faster with calculating the reflection attribute, and 60 times faster with calculating frequency and phase attributes. Copying data from CPU to GPU and vice versa is considered an overhead; however the utilization of the 128 processors on the GPU rules the transferring of the data.

In [14], nine different seismic core attributes algorithms were tested on 3D seismic data. The data first loaded from CPU memory to GPU global memory, and all the variables that are expected to be accessed frequently were initialized in the shared memory since it has faster access to the global memory. Those nine attributes were tested in a machine with two AMD Opteron 280 dual-cores at 2.4 GHz each (four cores total). The GPU that was used is the NVIDIA Quadro FX 4800 that contains

192 cores. The operators that used to calculate those attributes on 3D (2048^3) of seismic data were design to work on 2D data, and the operators were applied on each separate slice of the 2048 slices.

Results showed tremendous improvement in the performance i.e. the spreadable convulsion got (66.1) times faster, and matrix transpose got (51.1) times faster, however the performance that was gained to calculate the histogram was the lowest (2.1 times faster) since the nature of histogram calculation is sequential because of the accumulation of the histogram values.

Also [14], went to explore the ability of applying GPU in an interactive seismic applications since the response time was decreased from minutes on CPU to seconds on GPU. Using the Fast Iterative Method (FIM) in [13] that uses a computational scheme which takes advantage of the high parallel architecture of GPU by updating independent nodes iteratively till the convergence is approached. By calculating seismic tomography with FIM, seismic data can be interactively explored.

3.3 Summary

In this chapter, two different problems and their solutions were discussed in details. The first problem is the unclear seismic data and it was solved using edge detection. However different type of edge detection can be applied to different structure of seis-

mic data to come up with better results. Those edge detection operators fall into two main categories; Gaussian operators such as Canny operator and Gradient operators such as Laplacian and Sobel operator.

The second problem is the slow seismic data processing which was solved using multi-CPU clusters. Each CPU contains several cores that can communicate using OpenMP to distribute the data among those cores; also MPI is used to do the communication between the CPUs. Good results were achieved using cluster of CPUs however with a high cost. Another solution for slow seismic data processing is using GPUs during early 2000s. This technology was based on ATI graphics cards using the stream processing technology. Not all the results showed improvement over the CPU, however some algorithms were 10 times faster. Nvidia came up with advanced graphics cards that contain many cores (240 cores in advanced models). Those cards are well integrated with the CUDA language that can utilize the hardware architecture of those cards to archive the best performance.

CHAPTER 4

MOTIVATION AND CONTRIBUTIONS

Since the seismic data is now being gathered in 3D values, processing those data started consuming more CPU and memory recourses, for this reason, many oil exploration institutes started to speed up this processes by distributing the data between many CPUs to get the job done faster. However Parallel CPUs clusters is considered to be a costly technology in more than one aspect. First and the most important factor is the high prices of not only acquiring multi CPUs clusters , but also the room that should be used to install them, the cooling system that would cost a lot by itself, and not to forget the maintenance annual fees. All of those costs would seem too high for small or medium institutes to tolerate. The second factor is the time that takes a batch job to be done on those clusters. If for example the cluster is shared by many users who are trying to submit their jobs at the same time, a job might wait for a long time in the queue to start processing, so in this case, it is not only the time needed

to processes the job, but also the time that the job takes to wait in the queue to be processed.

Looking at all of these factors, it is time to move to a new technology that would overcome the identified problems. For this reason, GPU was introduced to take the load off the CPU and then speed up the process with the multi-cores that are built-in inside the GPU devices. By utilizing this feature in GPUs most of the problems with CPU clusters would be solved. Figure 4.1 gives an idea of number of Giga operations performed per second (GFLOP/sec) by NVIDIA GPUs compared to CPUs.

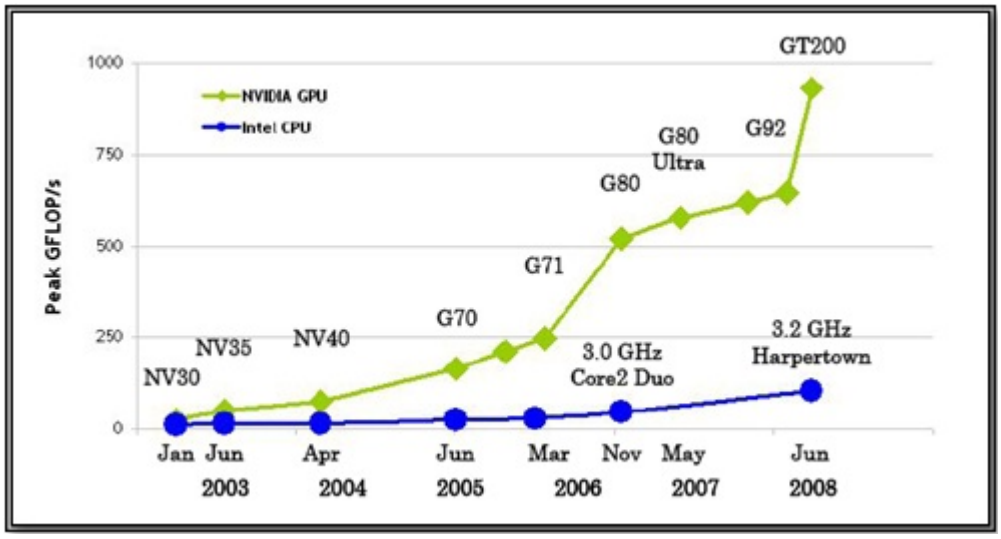


Figure 4.1: Comparing GPU to CPU in terms of GFLOP/sec.(adapted from [21])

With regards to the financial aspect, a single GPU costs about 400 USD [4] and it does not require installation and maintenance fees, and it does not require alot of space since it is installed inside the machine. With respect to time needed to finish a job, each user is going to have his own cluster with 128 built-in cores inside his ma-

chine which would deduct the queuing time from the total processing time and leave only the time required to performe the processing part alone. In addition, it would ease the way of resubmitting the job if there is a change in a program since there is no queuing time and the job would immediately star running.

we believe that by utilizing the GPUs optimized feature of parallelism, a remarkable speed up in performance will be gained, and would provide an easy way to implement in those institutes that cannot afford buying CPU clusters.

The expected contributions of this research work are:

1. Modification in data decomposition of Sobel edge detection algorithm on Nvidia graphic processing unit (GPU).

   The data partitioning methodology on a cluster of CPUs can occupy a big amount of memory of the CPU since they come with a large Random Access Memory (RAM). However, the situation with GPU is different since GPUs come with a limited size of memory. So handling this limited size of memory will be the first contribution of the research paper.

2. Modification in Sobel edge detection algorithm on Nvidia GPU card.

   Nvidia GPU is divided into grids and blocks. Those distributions of grids and blocks are 2D, on the other hand, seismic data is 3D, so mapping the 3D seismic data onto the GPU structure and modifying the Sobel edge detection algorithm to run on the thread level inside the blocks will be the second contribution of

this research work.

3. Utilizing the multi-core feature of Nvidia FX5600 GPU card to detect edges of 3D seismic data using Sobel edge detection.

   We plan to implement the modified Sobel edge detection algorithm to run on Nvidia GPU using the Compute Unified Device Architecture (CUDA) language and compare the performance with Sobel algorithm running on a cluster of CPUs.

4.1 Summary

The current solution of processing seismic data is considered to be costly in many aspects which make it more than what small company can afford. For this reason, many researches were looking at a cheaper technology that can do the same job, as a result GPU came as one of the solution that is capable of performing seismic and image processing with the same quality as CPUs cluster do but with lower cost. This thesis suggests apply Sobel edge detection on 3D seismic data using GPU. However to achieve this goal two main modifications has to be done on the CPU version of Sobel edge detection; the first is modification on the data decomposition and the second is the modification on the Sobel edge detection to work on the GPU. By applying these two modification, a low cost hardware can be used to achive the CPUs cluster performance with the same output quality.

CHAPTER 5

PROPOSED SOLUTION

In this thesis, we proposed to utilize a multi-cores NVIDIA graphics card in detecting edges for seismic data using Sobel algorithm that will be implemented using CUDA language. The results will be compared wwith the results to the already working solution of CPUs clusters. This chapter covers in detail the NVIDIA graphics card architecture, and explains the modifications that are going to be applied on the Sobel edge detection algorithm to be able to work on GPU architecture.

5.1 NVIDIA Graphics card Architecture

In this thesis, we are proposing to perform Sobel edge detection using the multi cores NVIDIA graphics card. Graphical processing units (GPUs) were originally designed to take the load of the CPU from processing and outputting graphics and that was mostly for gaming, however as those GPUs advanced in terms of processing power

and memory size, computer programmers started thinking to make use of the GPUs. Recent GPU devices are dedicated processing devices that perform a fast parallel complex algorithms, the parallelism capability was introduced in [23] due to the pipelines technique which gives the ability to override the fixed functionality of the hardware.

The pipeline concept appeared to be very suitable for GPUs as it would allow the GPU to work as a stream processor. This is appropriate especially when processing images. Image data is broke down into vertices that can be processed independently at the same time. Figure 5.1 shows how GPU structure would differ from CPU structure.

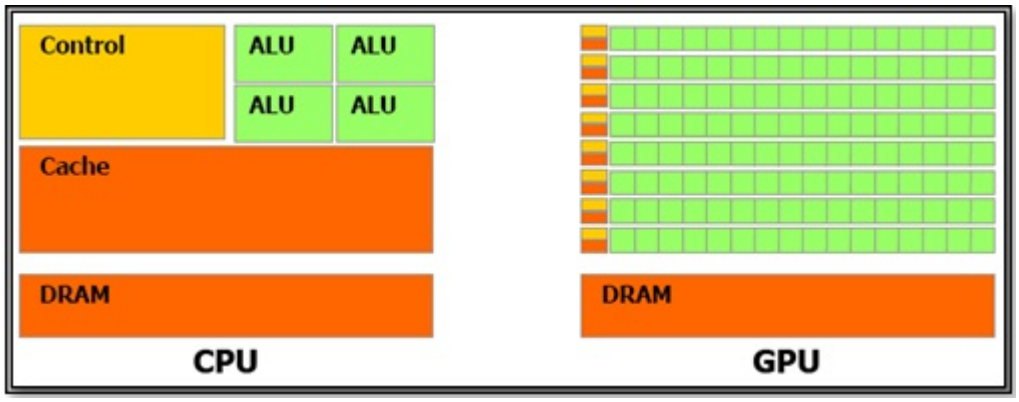


Figure 5.1: More Transistors in GPU than CPU for data processing.(adapted from [21])

According to [11] , recent NVIDIA GPUs have the power to not only work as a graphics hardware, but also to perform a parallel complicated arithmetic operation and that is because of the high-end specifications were implemented into those GPUs. High parallel floating point computation is one of the features that were newly implemented into GPUs where 128 processors can perform parallel floating point operations which

can beat the performance of the CPU by many times. One of the new features is the increase of the memory bandwidth, which is between the cores and the built in memory in the device, to be faster than the CPU accessing the RAM. New GPUs are designed with inboard Graphics Double Data Rate (GDDR) memory of 1.5 GB with an average 70 GB/sec bandwidth speed. This amount of memory and fast accessing rate would enhance the performance of the GPUs to perform operations especially with huge amount of data. Multi-GPU Computation is one of the new features as well. It allows having multiple GPUs installed next to each other in one board and data can be partitioned and sent to different GPU devices which would help scale up the capability of the GPUs to perform much faster.

5.2 Compute Unified Device Architecture (CUDA)

Compute Unified Device Architecture (CUDA), which is the language that is going to be used to program GPUs, is a C-like parallel programming language and software environment introduced in 2006 to solve the challenges of multi core CPU with minimum set of extensions to C language [21]. The main parallelism concept of CUDA is to split up the problem into sub problems that can be solved individually and then break those sub-problems into smaller problems that can be solved concurrently. By doing this, transparent scalability is allowed since any group of sub-problems (threads) can be communicated then be assigned to the free processors.

The CUDA code as described in [15] runs in two phases, the first phase is the CPU

which is called the host, and the second phase is the device which is the GPU. The NVIDIA C compiler (NVCC) makes sure where each part of the code runs. The part of the code that needs the least amount of parallelism would run on the host with ANSI C code and compiled with standard C compiler, on the other hand the part of the code that needs a lot of parallelism runs on the device using extension of C language with the keyword 'Kernels' that is followed by the part of the code that will be running in parallel. However, in case there is no device, the NVCC compiler is capable of emulating the code to run on the host as if it were running on the device. The main functionality of the Kernels is to generate a large number of threads that run concurrently. For example, in the matrix multiplications, each thread is responsible for calculating each output of the resulting matrix. Compared to CPU threads, the GPU threads run much faster because of the few number of cycles needed to generate and to schedule them due to the hardware support.

CUDA has access to different device memory types with different privileges. The host code can read and write data through only the global and constant memory of the device, however the device code can read and write to any other memory on the device except the constant memory.

The thread structure described in [20] consists of several components as shown in Figure 5.2. A grid is a set of blocks which are themselves a set of threads that can run simultaneously and cooperate with each other through barrier synchronization and a shared access to local memory inside each block. However during the programming

stage, the number of threads and number of blocks must be specified in the code to give each thread a unique ID of a block that is given a unique block ID as well with taking into consideration that the maximum number of threads on all the blocks that CUDA can handle is 512 threads.

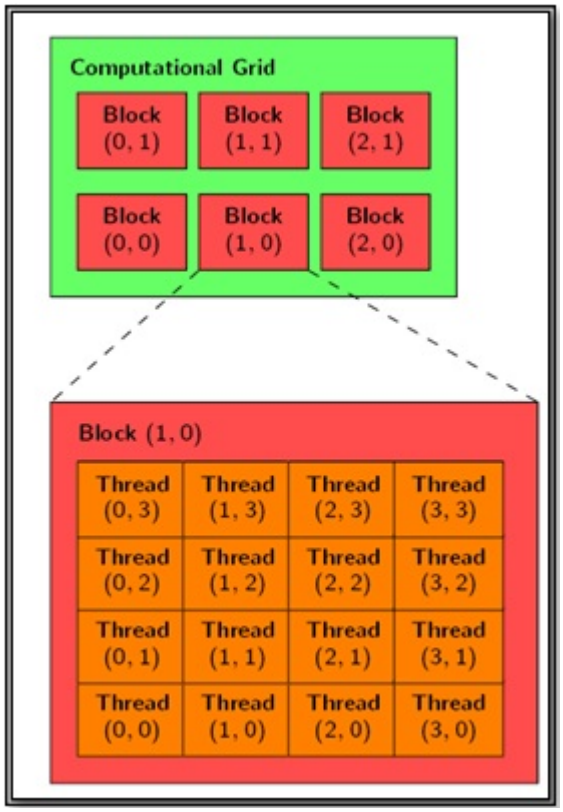


Figure 5.2: Threads structure.(adapted from [21])

The method that threads are executed and scheduled is automated by the device itself where all the threads of a block can be executed at the same time and may synchronize at a barrier by calling the *_syncthreads()* function that makes sure that no thread belongs to the same barrier proceeds until all the threads have reached the barrier. This will allow all the thread at the same time to communicate by reading and writing to the same block memory (shared memory) at a synchronization barrier.

The same situation can run also across group of grids, however a group a of grids can run dependently or independently based on the way of coding.

Memory in CUDA has four main types. The first, is the register memory that is attached to each generated threads. This register memory would contain variable and data that are only accessed via this thread and has thread scope only. Threads can communicate through local memory, the second type, which can have a scope of block where all the threads belonging to a certain block can have an access to that local memory. The third type of memory is the shared memory where all the blocks share the same memory to communicate. The fourth type of memory is the global memory where all the thread can have an access to and this is to communicate across the grids. Figure 5.3 would give a clear picture of the main types of CUDA memories. To summaries, the GPU has the following types of memory with the associated Read/Write privileges:

1. Register per thread (read/write).
2. Local memory per thread (read/write).
3. Shared memory per block (read/write).
4. Global memory per grid (read/write).
5. Constant memory per grid (read only).
6. Texture memory per grid (read only).

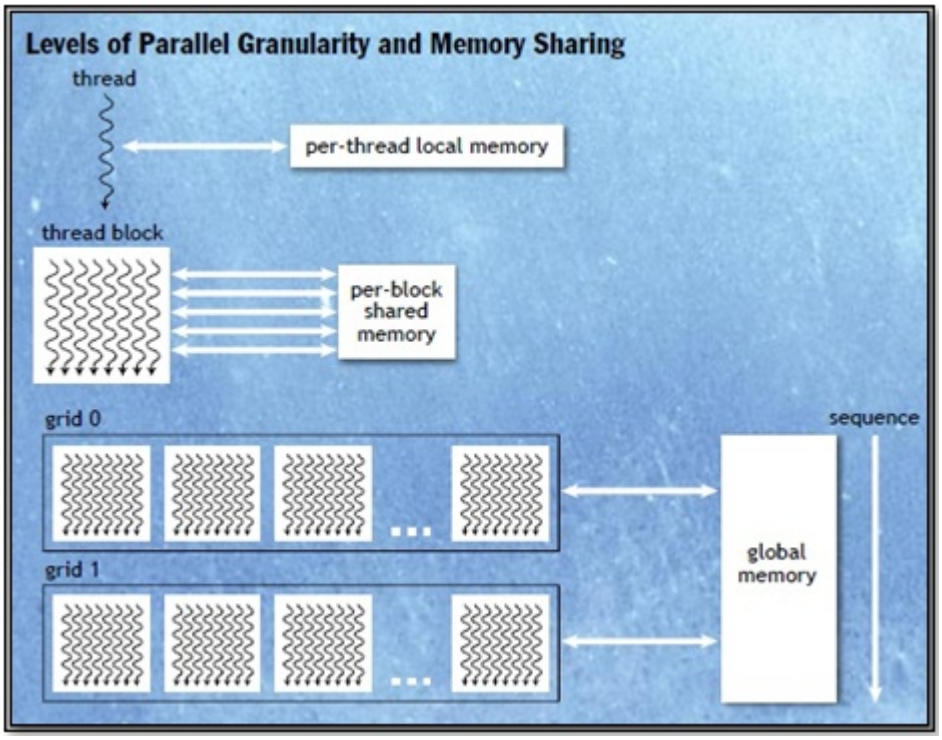


Figure 5.3: CUDA main memories.(adapted from [20])

Two important architectures were discussed in [20] and [22] which are the Single-Instruction, Multiple-Data (SIMD) and the Single-Instruction, Multiple-Thread (SIMT). The SIMD architecture controls the multiprocessor in the device. Each of those multiprocessors executes the same instruction but on a different data at the same time. The relation between the SIMT and a block would be as follows: each multiprocessor can execute more than one block using time slicing such that each block split into SIMD groups of threads which are called (warp) which leads to the SIMT structure. The SIMT creates, manage schedule the activities of the whole threads in the program which are 32 per-warp. All the warps are controlled by the Streaming Multiprocessor (SM) that consists of eight Scalar SP cores as shown in Figure 5.4 where each thread is assigned to single SP scalar core which is executed independently with its own in-

struction address and register state.

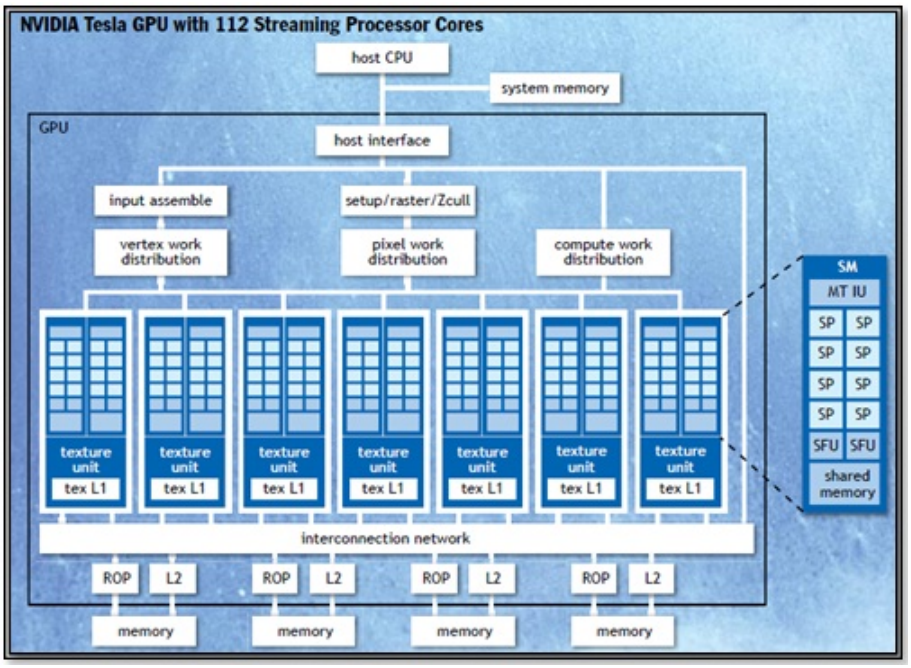


Figure 5.4: Stream processing structure in CUDA.(adapted from [20])

5.3 Sobel algorithm modifications

The original Sobel detection algorithm consists of three simple steps which are reading the data from disk, apply the Sobel filter, and then write the data back to disk. However there are two more additional steps added when running the algorithm on a cluster of CPUs which are partitioning the data after reading from disk and then gathering the data back before writing back to disk as a final result. But when dealing with GPU we have to put in consideration that we have to make additional step of copying from CPU RAM to GPU memory as shown in figure 5.5, and during this process we must handle the limited size of the GPU memory. For this reason, the

Sobel algorithm has to be modified to be able to run in the GPU. These modifications will take place in data partitioning and the Sobel algorithm itself.

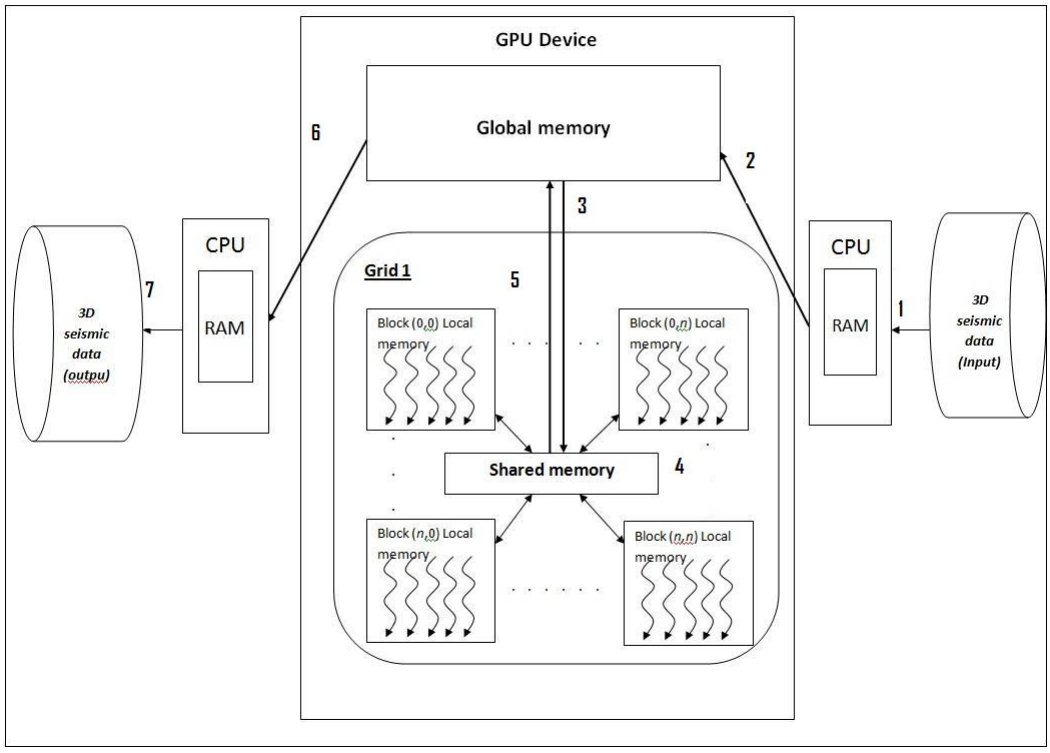


Figure 5.5: The Proposed CPU-GPU architecture.

5.3.1 Programming efforts

Programming on GPUs would add more steps to the execution process. For example, in Sobel edge detection modification in this proposal, there are two extra steps involving copying data from/to CPU, which is going to add more programming efforts. In addition, the code has to manage various programming aspects in CUDA which are:

1. Allocating memory for GPU internal memories such as the global and the shared memories.

2. Copying the data from CPU to GPU and vice versa.

3. Mapping the 3D seismic data into the 2D GPU(grid/blocks/threads) distribution.

4. Allocate space for all the variables in memory that has a fast access (shared and local), and allocate space for 3D semsic data in the memory with a large capacity (global).

5. Freeing the GPU memories for more data to process.

On the other hand, programming in MPI less effort comparing to programming in CUDA, so most of the extra steps that involved in programming in CUDA are not required in MPI programming. For example, there is no extra step of copying the data, so in MPI it is all between the data storage and the memory of the node, and as a result, there is no extra memory allocation other than the one that is done on the node memory. Programming with MPI deals only with one type of memory which is the RAM. However on GPU there are various types of memory with different size and speed access, such that the programmer has to make sure where to allocate memories for data, variables depending on the size of the data and how frequent the variables are being called or used.

5.3.2 Modification in data decomposition

The data partition method for the parallel Sobel algorithm with a cluster of CPUs is considered to be direct since each node copies the data that is going to br processed from the hard disk directly without being concerned about data size as long as it fits the node memory size as shown in Figure 3.5, on the other hand, decomposing the data is always an issue with the GPU and more attention has to be given when it comes to the GPU memory since it is smaller than the CPU RAM. For this reason, a large amount of data like 3D seismic data cannot be copied at once to the GPU device and has to partitioned within the size of the of the GPU memory. In the thesis we will use the Quadro FX 5600 NVIDIA graphics card that has 1.5 GB of memory and this is what the global memory is as shown in Figure 5.5.

The main goal of data decomposition is to process as much as possible data with minimum data copy since the connection between the RAM and the GPU is considered to be slow. For this reason the 3D seismic volume is going to be divided into smaller 3D volume to fin into the global memory. However we have to take in consideration that half of the global memory will be dedicated for the input and the other half is for the output, for this reason the maximum amount of the seismic data that can be processed at a time is almost 750 MB. The way that the data will be divided is according to the Z direction, or what is called the time slices, such that in each copy call, as many as possible group of seismic layers as shown in Figure 3.4 or slabs that have the size of almost 750 MB is copied to the GPU.

The reverse method will be applied to the output data such that after the GPU is done with processing the 750 MB of data, it will be copied back to the RAM, and according to the time slices index, the output 3D volume will be built.

5.3.3 Modification in Sobel algorithm

When programming the Sobel for cluster of CPUs, the whole code will be compiled and run in one place such that the entire variables will be allocated and assigned values in the RAM. However when programming with GPU using the CUDA language, the program will be divided into two parts, the first part is the code that is going to run on the CPU and the second part is code that is going to run on the GPU.

Since the GPUs have more calculation power than the CPU because of the multi core functionality, all the Sobel core calculation will be performed inside the GPUs, on the other hand, the CPU handles the Input/output operations and data partitioning. When it comes to programming with CUDA, two types of memoires will be utilized, the shared memory and the global memory. The shared memory is where the three $3X3$ Sobel masks will be initialized and assigned values, since it has a faster access than the global memory. However since the shared memory has a small memory size, the data cannot be copied to it , instead it is copied to the global memory that has a size of 1.5 GB.

Mapping the 3D seismic data volume into the GPU thread structure shown in Figure 5.2 is one of the most complicated issues. The seismic data are in 3D volume, where on the other hand the thread structure in the GPU is 2D such that each grid consists of a number of blocks in each dimension, and for each block there is a defined number of threads in each dimension as shown in Figure 5.2. Mapping the seismic data should be reflected into 2D map of group of blocks and threads. The following steps show how to reach to the optimum number of 2D blocks and 2D threads.

1. For a defined $threadsNum$ in each direction of the block, the number of values to be processed

   $valuesNum =$

   $NumOfRows{*}NumOfColumns{*}NumOfSlabs/(NumOfThreads{*}NumOfThreads)$

   In case the total size of the data is greater than the half size of the GPUs memory, the 3D volume will be divided into sub volumes

   $NumOfValues = NumOfValues/NumOfSubVolumes$

2. The number of blocks in each grid dimension is obtained by the following:

   $NumOfBlocks = \left\lfloor \sqrt{NumberOfValues} \right\rfloor + 1$

   The reason of addition of one to the number of blocks is to make sure that the allocated number of blocks is more than what is needed by data.

3. The total number of threads allocated will be:

   $TotalNumOfThreads = NumOfBlocks * NumOfBlocks * NumOfThreads$

For example, if the 3D seismic of size 8 rows and 8 columns and 5 slices, and if the

number of threads in each direction in a block is 2, then the total number of thread per block is $2X2$ which is 4, the number of valuesr will be calculated to be $320/4 = 80$. The number of blocks in each direction of the grid will be calculated to be the floor value of the square root of 80 which equals 8 and with the addition of 1, the value is 9. As a result of those numbers, there will be 9 blocks in each direction of the grid with 4 threads each which has a total of $(9X9X4) = 324$ threads, however only 320 threads are needed to do the calculation.

As shown in Figure 5.5, there are seven steps to accomplish the Sobel edge detection process on the GPU. The CPU part consists of reading the data from the disk to RAM and then copy the data again to GPU within the GPU global memory size. The GPU part consists of applying the three 3D masks in the x, y, and z direction and saves the output values that is going to be copied back to RAM by the CPU and then at the end will be copied to disk.

5.4 Summary

In this chapter, the architecture of the NVIDIA graphics card and CUDA which is the language used to program on NVIDIA cards platform were discussed in details. Threads, blocks, and grids layout were discussed in addition to the memory types and memory distribution on the GPU. On the programming aspect, CUDA was discussed in details including the structure of the language, the compiler, and the memory ac-

cess privileges.

As proposed solution, there are two main modifications were done to the CPU version of Sobel edge detection to run on GPU, the first is the modification on the data decomposition, and the second is the modification on Sobel edge detection. The challenges were on converting the C code to CUDA code, and also on applying the extra steps of copying data from CPU memory to GPU memory. These challenges were discussed as programming efforts that would help to reach acceptable results in terms of performance and output quality.

CHAPTER 6

PERFORMANCE EVALUATION

The objective of this evaluation is to test the performance of running 3D Sobel edge detection on NVIDIA GPU card using CUDA language and to compare the results with those resulting from running 3D Sobel edge detection on cluster of CPUs using messege passing interface (MPI). This section, looks into details of the steps required to run Sobel edge detection on GPU and on cluster of CPUs.

6.1 Performance metrics

The performance metrices that are going to be used to evaluate the performance are going to be follow: On GPU, For non-partitioned data the total time is calculated as follows:

$$T_{GPU} = RAM_{GPU} + Sobel_{GPU} + GPU_{RAM} \tag{6.1}$$

For partitioned data, the total time is calculated as follows:

$$T_{GPU} = (RAM_{GPU} * p) + (Sobel_{GPU} * p) + (GPU_{RAM} * p) \tag{6.2}$$

where RAM_{GPU} is time taken to copy from RAM to GPU memory, GPU_{RAM} is the time taken to copy from GPU memory to RAM, and p is the number of partitions. With respects to CPU and non-partitioned data, the total time is calculated as follow:

$$T_{CPU} = Sobel_{CPU} \tag{6.3}$$

For partitioned data, the total time can be calculated by

$$T_{CPU} = Sobel_{CPU} * p \tag{6.4}$$

1. The detailed steps on GPU include:

   i Reading data from storage to RAM and data partitioning.

   ii Copying each partition to GPU global memory.

   iii Performing Sobel edge detection.

   iv Copying the partition back to RAM.

   v Gathering the partitioned data back and writing the output to disk.

2. The detailed steps on CPU include:

   i Reading the data from storage to RAM.

ii Performing Sobel edge detection.

iii Gathering the partitioned data back and writing the output to storage.

To have a fair comparison between the GPU and the CPU performance, the GPU performance have to be optimized to have the fastest execution time without effecting the quality of the output data. This evaluation consists of six groups of experiments; the first four groups are related to GPU and deal with step *ii*, *iii* and *iv*. These groups of tests focus on enhancing the GPU performance in two aspects, the first is the Sobel edge detection performance (step *iii*) and the second is the reading and writing performance from and to GPU (step *ii* and *iv*). The first and the last step were eliminated because of inconsistent results of reading and writing data that are stored on the network. The fifth group is related to CPU cluster to measure the performance on step *ii*.

There are four main factors that play a big role in affecting the GPU performance, the first is the number of threads that will be assigned to each block, the second is the threads layout in each block (1D *vs* 2D) . Both of these factors deal with step *iii* on GPU in optimizing the Sobel edge detection performance. The third factor is the read and the write performance from and to GPU which is related to step *ii* and *iv* on GPU. The fourth factor is the number of data partitioning in case of the data size is more than the GPU memory size, this factor is also related to step *ii* and *iv* on GPU

The results will be verified by comparing the output edge detected seismic images that run on GPU with the edge detected seismic images that run on CPU cluster. The

output that comes from CPU cluster is already verified and confirmed by geophysics groups and is seriously considered in real oil exploration projects in Saudi Aramco.

6.2 Optimizing Sobel on GPU

6.2.1 Objectives

The first group of experiments is running Sobel with a different number of threads in each block. As the number of threads increase, the number of blocks decrease. The experiment is going to measure the effect of increasing the number of blocks on the performance. The results are going to be studied to look for the best number of blocks that the Sobel edge detection required to have the best performance. This group of experiments is going to be performed with a fixed number of partitions and a fixed threads layout on blocks. The second group of experiments is running Sobel edge detection on GPU with different threads layout but with a fixed number of threads. According to [21], threads layout can be in 1D or 2D layout, and what determines the best layout is the data structure and whether there was any data dependency. The tests will be performed with a number of threads that give the best performance in the first group of experiments with a fixed number of data partitioning. The third group of experiments is running Sobel edge detection on GPU with enhanced memory allocation. Instead of allowing the CPU to allocate memory by itself, the GPU will be given the privilege to allocate the memory on the CPU.

6.2.2 Environment setup

The Nvidia GPU card was installed on 8 cores machine that has dual CPU, each of which contains quad core Intel Xeon with speed of 3 .0 GHz. The machine is equipped with 16 GB of memory and connected to Network-attached storage (NAS) where the data is located.

The initial plan was to use the Quadro FX 5600 NVIDIA card which comes with 128 cores and 1.5 GB of global memory, However during the implementation phase, we received a new high end Tesla C1060 card that we decided to run all the experiments on. The Tesla C1060 contains 240 cores with 1.4 GB clock rate. The maximum number of threads that the GPU can handle is 512 per block that can be of 1D, 2D or 3D layout. The GPU has 4 GB of global memory which have a bandwidth of 102 GB/sec. this fast bandwidth allows to have a fast access between CPU and GPU.

6.2.3 Increasing number of threads per block

The purpose of this group of experiments is to measure the performance in different number of threads in each block. With a fixed data size, and a fixed threads layout in each block, the number of threads in each block affects the total number of blocks that is going to be generated when performing the Sobel edge detection with 3D seismic data of size 1.8 GB.

As the number of threads increases, the access to the shared memory in the block increases. In addition, multiple blocks are going to access the global memory where the data is located. The number of threads to be tested are 4, 8, 8, 16, 256, and 512, as in [22] the maximum number of threads per block is 512. These experiments will show the impact of increasing the number of threads per block on the performance, and also the impact on the performance of increasing the number of blocks on accessing the global memory to read data from.

6.2.4 Changing threads layout from 2D to 1D

The goal of this group of experiments is to measure the performance of different layouts of threads inside a block. The total number of threads that is going to be tested in each block will be obtained from the best performance result obtained in the first group of experiments. The Sobel will be tested on different 2D layout such that the total number of threads which is calculated by multiplying the 2 dimensions is always going to be the same number.

However, the 2D layout is going to be a multiple of 32 as each 32 threads are grouped into a wrap and executed at once in stream processing [21]. For example, if the best performance is with 128 threads, the 2D layout that is going to be tested is $(32 * 4)$, $(64 * 2)$, and $(128 * 1)$ which is the 1D layout.

6.2.5 Enhancing the I/O processes

The goal of this group of experiments is to observe the effects of locking the RAM by the GPU to be accessed faster. By using (*malloc*) function on C language which enables the program to reserve part of the RAM to be occupied by data. While using an advanced function from CUDA (*cudaHostAlloc*), it allocates memory that is page-locked and accessible to the GPU.

The GPU tracks the virtual memory ranges allocated with this function and automatically accelerates calls to other functions such as (*cudaMemCpy*) that copies data from RAM to GPU memory. Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth. On the other hand, the disadvantage of using the locked-page is that the system performance will decrease in case there is not enough RAM [21]. The reading and writing rate is expected to increase by using the locked-page memory allocation which helps decrease the total running time.

6.3 Comparing GPU to CPU performance

6.3.1 Objectives

The first experiment of comparison is running Sobel edge detection on GPU with different data size to have the data partitioned into sub-volumes which are going to be

processed in serial. The results will be studied to notice the effect of data partitioning on the GPU performance. The second experiment of comparison is running the CPU cluster version of Sobel edge detection. The data size that will be tested is of the same sizes as in the previous groups of experiments. The number of nodes that Sobel edge detection will start from one node that has dual CPU with quad core each. However, if the data size exceeds the node memory, the data will be partitioned using MPI between different nodes. The results will be studied to observe the best performance that Sobel can get and compare the results with the performance of Sobel on GPU. The third experiment of comparison will focus in establishing a guide line of making the decision to go with either the GPU or the CPU. These experiments will use dot product which contains 5 instructions to compare the GPU and the CPU performance.

6.3.2 Comparing the GPU to multi-cores CPU

The goal of this group of experiments is to observe the effects of increasing the number of CPUs in Sobel edge detection using OpenMP to distribute the data between the cores and then using MPI to distribute the data between different CPUs and compare the results with best GPU obtained result. The data size that will be tested is 1.8 GB of 3D seismic data which is going to be partitioned among the number of cores that are going to be used.

6.3.3 Comparing GPU to CPU with different data sizes

The goal of the group of experiments is to test the GPU and the CPU performance on different data sizes. All the previous experiments were performed on fix data size which is 1.8 GB, however in this group of experiments, the data will be divided into three groups, the first group is the data that is less than the GPU memory, the second groups is the data that is more than half of the GPU memory (2 GB) and less than half of the RAM (16 GB), the third groups is the data that is more than half size of the RAM. This group of data will be processed in serial in the CPU side such that the CPU copies the maximum data it can handle to process and then the GPU copies the maximum data it can handle of this portion. The GPU code that is going to be tested is based on the code of the fastest performance in the previous group of experiments.

Both the GPU and the CPU will run with these three data groups, the results will be studied to observe the effect of data that is less than and more than half the memory size of the GPU and the CPU. The result should also show the impact of increasing the data size in general on the CPU and the GPU performance.

6.3.4 Comparing the GPU to CPU on base of number of instructions

The purpose of this experiment is to establish a guide for when to move to GPU technology by doing multiples of the same number of instructions on the GPU and

the CPU. During the experiment, the same hardware specifications of the previous experiments will be applied with the same data set. In this experiment there will be two different codes that will be written for GPU using CUDA language and for CPU using C to do simple 1D convolution.

1. The CPU version works as follows:

   i Copy data from disk to RAM.

   ii Loop on N from 1 to 100

   iii For each N loop on M from 1 to N.

   iv Run the convolution.

   v Write output to disk.

   The total time of the inner loop (step *iii*) will be taken in consideration in the CPU version.

2. The GPU version works as follows:

   i Copy data from disk to RAM.

   ii Copy data from RAM to GPU memory.

   iii Loop on N from 1 to 100.

   iv For each N, loop on M from 1 to N

   v Run the convolution M times.

   vi Copy data from GPU memory to RAM.

vii Write back to disk.

the maximum number of N is chosen to be 100 to give more number of instructions to be calculated. The timing of steps *ii* and *vi* will be taken once, while the total time of step *iv* will be taken on each loop on N.

The 1D dot product convolution contains 5 instructions and that is what it takes to complete the first inner loop. The second loop will deal with $2 * 5$ instructions. This test will provide a figure of the CPU and the GPU performance starting with 5 instructions and then going to multiples of 5 until it reaches a point where total time is the same. This point will be the breakeven which would tell when to implement in CPU and GPU depending on the performance that is before and after the breakeven point. The GPU code will be compiled with the default compilation argument as in the previous experiments. On the other hand, the CPU code will be compiled into two different versions; the first is using Intel icc compiler version 11.0 then with optimized compilation arguments as in the production environment to achieve the best performance. The other version will be compiled using the default C compiler arguments. The reason for having two different compilations is first to compare the GPU performance to the default compiled CPU version that is used by some programmers in the non-industrial environment which is considered by not utilizing the machine capability . The second reason is to compare the GPU with the optimized CPU version which is used in the optimized production environment.

6.4 Summary

In this chapter, we discussed the objective and the environment setup of all the experiments. Also we discussed the performance matrices that will be used and explained how GPU performance is compared to CPU performance. in this chapter, The experiments were grouped into two groups. The first group is enhancing the performance of the GPU and the second group is comparing the GPU performance with CPU performance. At the end of this chapter, an experiment was introduced that should help to establish a guide for deciding whether to program using GPU or CPU given the environment setup this is used in these experiments.

CHAPTER 7

RESULTS AND DISCUSSION

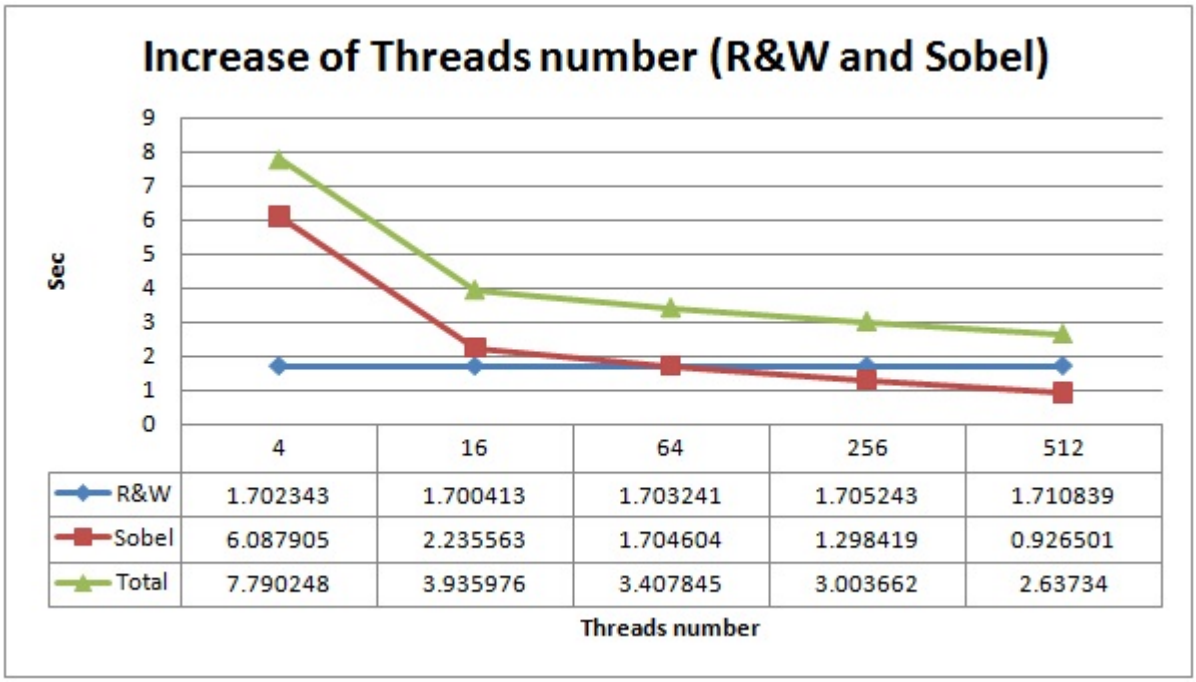
7.1 overview

This chapter shows the results of six experiments were conducted for two reasons. The first reason is enhancing the performance running Soebel edge detection on of GPU and the second reason is comparing the GPU performance to the CPU performance. The last experiments shows a guild of whither to choose running Sobel on GPU or CPU based on the algorithm used and the environment setup.

7.1.1 Results and Discussion of optimizing Sobel on GPU

In this group of experiments, 1.8 GB of data was tested on 3D Sobel edge detection. Since the data was read from network storage, the reading and writing time from and to the storage was not accurate and different results were reached at different executions of the program. therefore, this time was excluded from the performance

calculation, and what included is only the time of reading and writing to RAM and the Sobel edge detection time. The program was executed 10 times on each number of threads and the average results were taken. In Figure 7.1, as the number of threads increases, the total execution time dropped from 7.78 sec on 4 threads to 2.63 sec on 512 threads (300% faster). The increase in the performance resulted from the increase of the Sobel performance that dropped from 6.08 sec on 4 threads to 0.92 sec on 512 threads (660% faster). However, since the reading and the writing time remains constant when running with different number of threads (1.70 sec) as expected, the total time did not gain performance as the Sobel edge detection did.



	4	16	64	256	512
R&W	1.702343	1.700413	1.703241	1.705243	1.710839
Sobel	6.087905	2.235563	1.704604	1.298419	0.926501
Total	7.790248	3.935976	3.407845	3.003662	2.63734

Figure 7.1: The effects of increasing the threads number

To conclude, there is a great impact resulting from increasing the number of threads on the Sobel edge detection, (6.6 times faster), and zero effects on reading and writing

process which become a bottleneck since no matter how much the Sobel performance increases, the total time will never get less than the reading and writing time as shown in Figure 7.1. The verification of the results is shown in Figure 7.2 where a slide of the input data and of the output data were taken and compared with Sobel edge detecion using CPU.

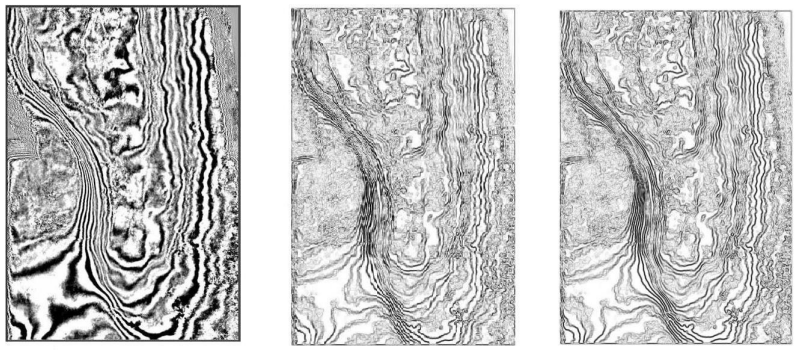
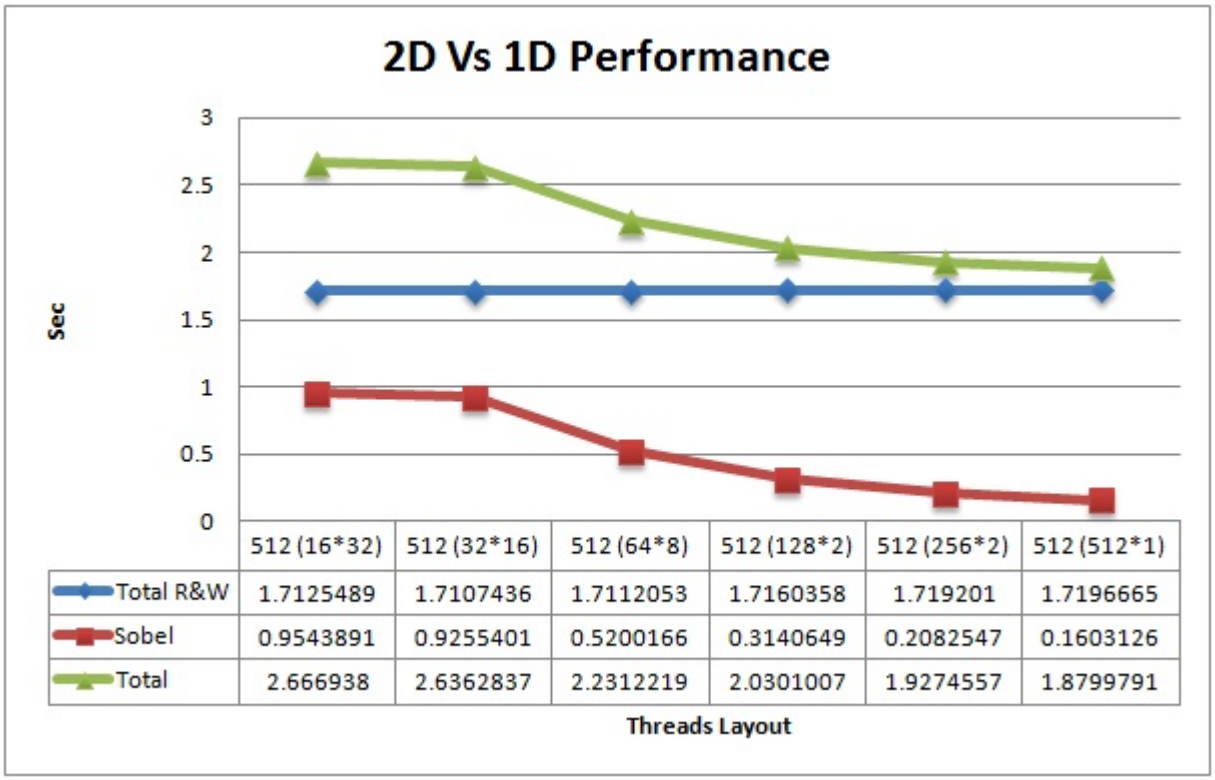


Figure 7.2: The input data on the left, the GPU edge detection using the Sobel in the center, the CPU edge detection on the right.

As shown in the previous experiments, the best performance was obtained with 512 threads. As a result, the second group of experiments will use this number of threads to start testing with different layouts. The initial threads layout was (32 in x direction and 16 in y direction), so this group of experiments was performed with $(16 * 32)$, $(32X16)$, $(64X8)$, $(128X4)$, $(256X2)$, and $(512X1)$. As in [Cuda tutorial], every wrap ,which contains 32 threads, is executed at once. Since the first layout tested $(16 * 32)$ contains only half a wrap, the second half will be taken from the other block and that would cause some latency as shown in Figure 7.3. As the layout increases in x direction, to be a multiple of 32, and decreases in y direction, the performance increases as in figure 7.3.



	512 (16*32)	512 (32*16)	512 (64*8)	512 (128*2)	512 (256*2)	512 (512*1)
Total R&W	1.7125489	1.7107436	1.7112053	1.7160358	1.719201	1.7196665
Sobel	0.9543891	0.9255401	0.5200166	0.3140649	0.2082547	0.1603126
Total	2.666938	2.6362837	2.2312219	2.0301007	1.9274557	1.8799791

Figure 7.3: Threads layout

Another reason for increase in the performance when moving from 2D layout to 1D is the calculation needed to figure out the thread index. For a 1-D, the Index of a thread and its thread ID are the same, and For a 2D, which is of size (Dx, Dy), thread ID of a thread of index (x, y) is calculated with $(x + yXDx)$, and for 3D of size (Dx, Dy, Dz), the thread ID of a thread of index (x, y, z) is calculated with $(x + yXDx + zXDxXDy)$.

The reading and writing did not gain any performance since the threads layout affects the calculation only and not the Input/output process. On the other hand, the Sobel speed dropped from 0.925 sec with $(32X16)$ layout to 0.160 sec on 1D layout

(512X1) and this is almost 5.7 times faster. The decrease in the Sobel time decreased the total GPU time to run 1.4 times faster. The verification of the results is shown in Figure 7.4 where a slide of the output data was taken and compared to a slide of an output from a CPU cluster.

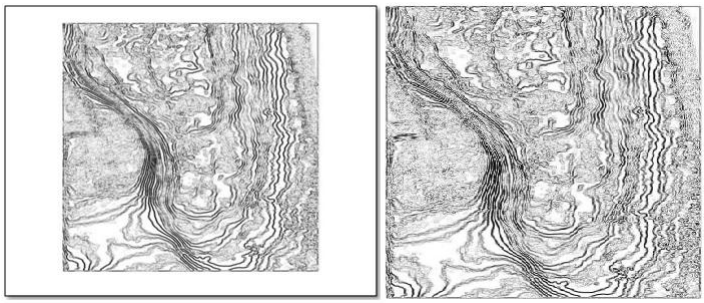


Figure 7.4: Sobel edge detection using the 1D threads layout on GPU on the left, the CPU edge detection on the right.

The tests were performed in the same environment setup for the second experiment, however the way of allocate memory was changed in the programming level. By allowing the GPU to allocate memory on the CPU RAM, a big improvement was gained in the reading and writing rate as shown in Figure 7.5.

The effect on the performance was huge as the performance of reading and writing dropped from 1.71 sec to 0.65 sec (2.6 times faster). The gained performance affected the total time to speed up about 240% as it dropped from 1.87 sec to 0.78 sec. The verification of the results is shown in Figure 7.6 where a slide of the output data was taken and compared to a slide of an output from CPU cluster.

The initial tests showed a 300% increase in the performance when running on 512

	512 (16*32)	512 (32*16)	512 (64*8)	512 (128*2)	512 (256*2)	512 (512*1)
Total R&W-Enh.	0.655727	0.6558562	0.6568488	0.657117	0.6571824	0.658763
Total-Enh.	1.4789256	1.4663526	1.1034534	0.9232712	0.8229352	0.7830016
Total R&W	1.7125489	1.7107436	1.7112053	1.7160358	1.719201	1.7196665
Total	2.666938	2.6362837	2.2312219	2.0301007	1.9274557	1.8799791

Figure 7.5: Regular Read&Write Vs enhanced Read&Write

threads compared to 4 threads. According to 7.7, most of the total time was consumed by performing the Sobel process (78%) and only 22% was taken by the I/O operation. For this reason, any enhancement of the Sobel would affect 78% of the total time as shown in Figure 7.7. For example, by reducing the Sobel time by 650% when increasing the number of threads from 4 to 512, the total time was reduced by 300% to be 2.63 sec.

Figure 7.8 left chart shows that most of the time was consumed by the R&W process (64%) and only (34%) was consumed by the Sobel operation, so as a result of reducing the Sobel time by 600% , the total time was just reduced by 30% to be 1.8 sec as shown in Figure 7.8 right chart.

Figure 7.6: Sobel edge detection using 1D threads layout and enhanced I/O running on GPU

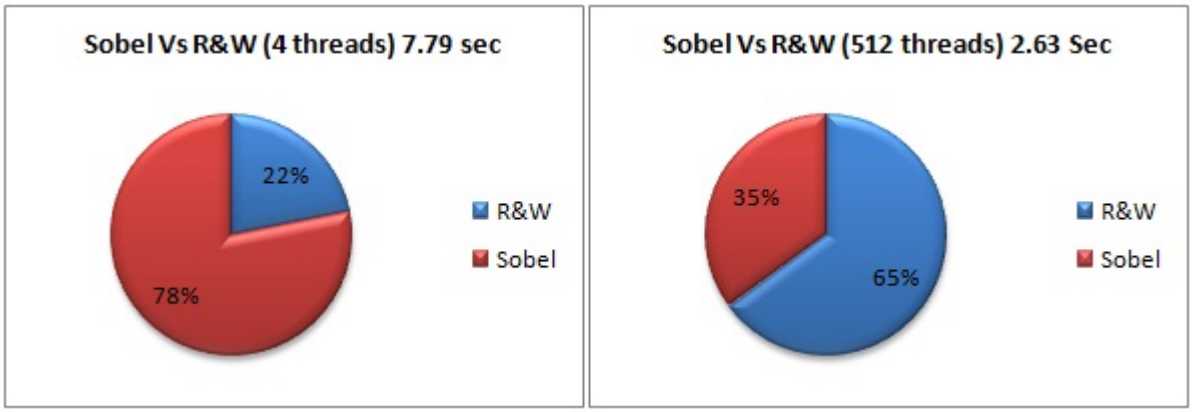


Figure 7.7: Sobel Vs R&W (4 threads) Sobel vs R&W (512 threads)

The I/O process is consuming the majority of the total time (91%) as shown in Figure 7.8, that means any enhancement in the I/O process would have an improvement on the total time. So by enhancing the memory allocation of the CPU, the read and write processes were reduced by 260%, and as a result, the total time was reduced by 240% to be 0.124 sec as shown in Figure 7.9.

In 7.9, still most of the total time is being consumed by the I/O process (84%),

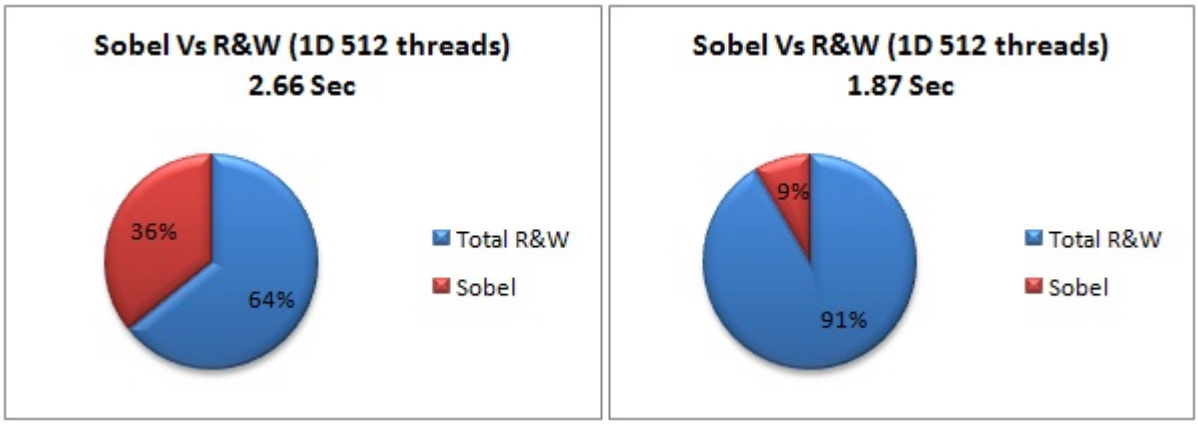


Figure 7.8: 2D Sobel to R&W ratio 1D Sobel to R&W ratio

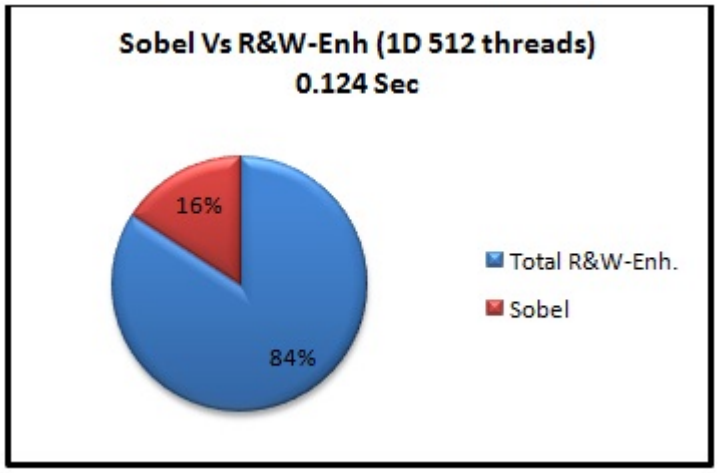


Figure 7.9: 1D Sobel to Enhanced R&W ratio

which opens the door for more enhancement to be performed in the future to have a big effect in reducing the total time.

7.1.2 Results and Discussion of Comparing GPU to CPU performance

Since this group of experiments was performed on a single node that contains 32 GB of RAM, and the 3D seismic data is only 1.8 GB, there was no need to go for multi

nodes and to use MPI. Instead, OpenMP was used to share the data among the dual CPUs which each CPU contains quad cores.

Figure 7.10 compares the best GPU performance, obtained from enhancing GPU performance which is (0.78 sec), with CPU performance running on 1 to 8 cores. Figure 7.10 shows that the best GPU performance is performing 360% faster than CPU when running on a single core. However, as the number of CPU cores increases, both performances are getting close such that when running on 4 cores, the performance is the same. As the number of cores increase to 8 cores, the CPU performance exceeds the GPU performance by 53%.

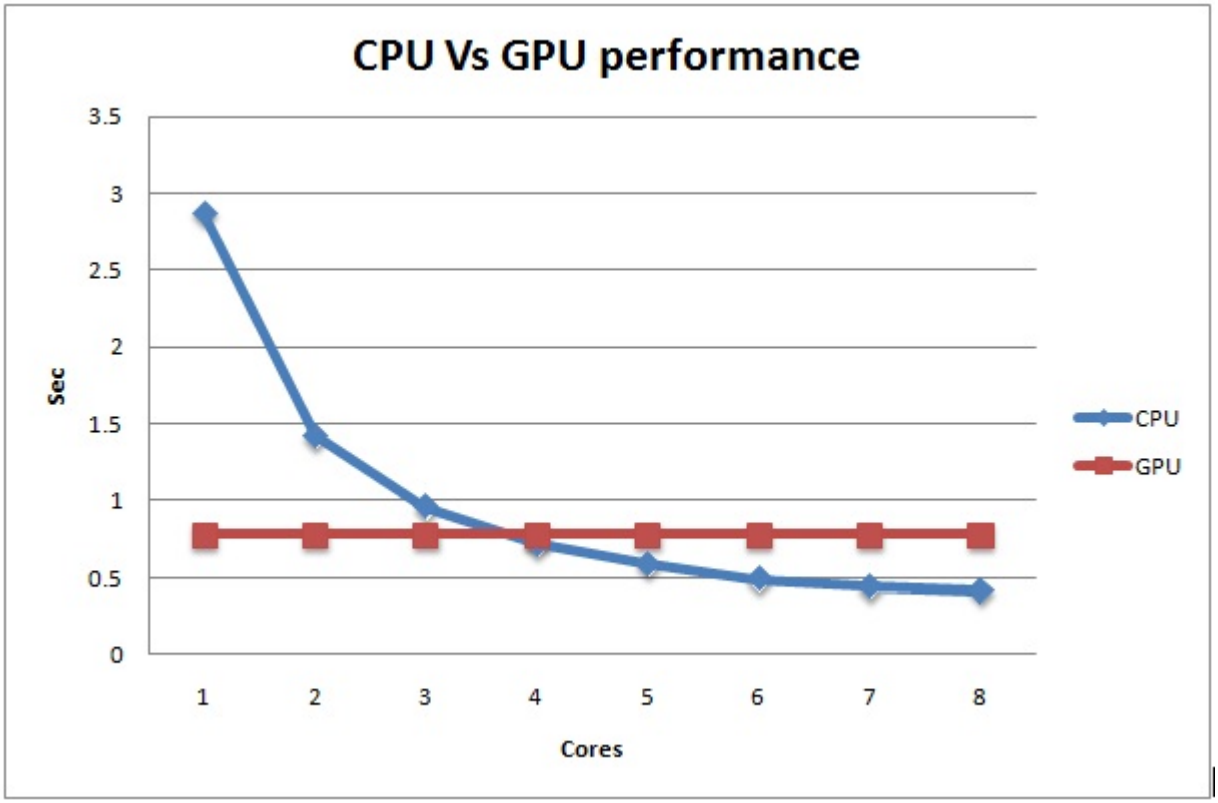
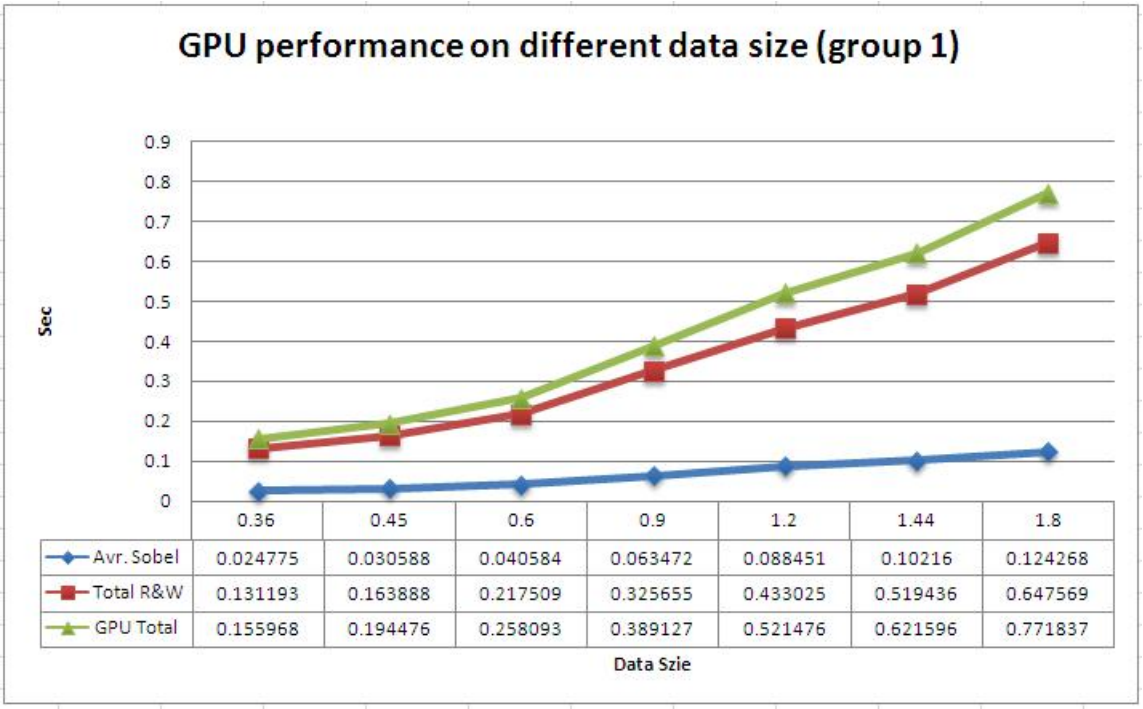


Figure 7.10: CPU vs GPU performance

When it comes to different data sizes comparison, on the GPU side, the first group

of data shows a gradual increase on the total time as the data size is getting larger. As shown in Figure 7.11, the main reason of increasing the total time is the reading and writing processes which reflects the correctness of the results in the previous groups experiments.



	0.36	0.45	0.6	0.9	1.2	1.44	1.8
Avr. Sobel	0.024775	0.030588	0.040584	0.063472	0.088451	0.10216	0.124268
Total R&W	0.131193	0.163888	0.217509	0.325655	0.433025	0.519436	0.647569
GPU Total	0.155968	0.194476	0.258093	0.389127	0.521476	0.621596	0.771837

Figure 7.11: GPU performance on data size less than half the GPU memory

The second group of data is taking the maximum performance of the GPU since it is more than the GPU memory as in each run, the GPU copies a new partitioned data to process and then to write back, therefore, the total time performance is linearly increasing as the data size increases as shown in Figure 7.12.

On the third group of data, the same behavior of the second group of data was performed as shown in Figure 7.13. In each run of this group of data, the CPU

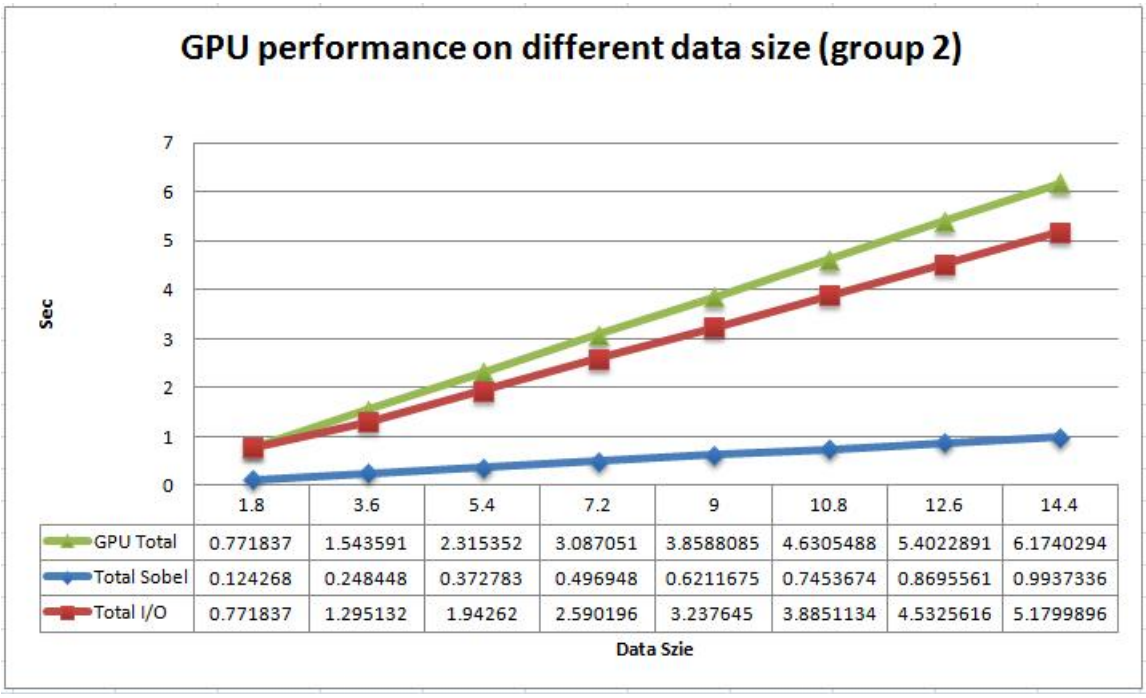


Figure 7.12: GPU performance on data size more than half the GPU memory but less than half the RAM

copies 16 GB of data, and only 2 GB of data was copied to GPU to be processed. This operation is going to take 8 runs on the GPU side for each single run on the CPU.

On the CPU side, as show in Figure 7.14, the total times reflects the increase on the data size. Since the reading time and writing time from and to storage is not calculated on all of the experiments, the total time increases linearly on the data size that more than 16 GB ,which is half the RAM size. Since the number of Sobel instructions is much lower than the GPU capability, as the data size is getting larger, most of the time is time consumed by performing the I/O operation, therefore, the gap between the GPU and the CPU increases as the data size is getting larger. The advantage in this case is going to the CPU side since there is less I/O operations which are 2 runs on the CPU side verses 8 runs on the GPU side with data size that is larger than half

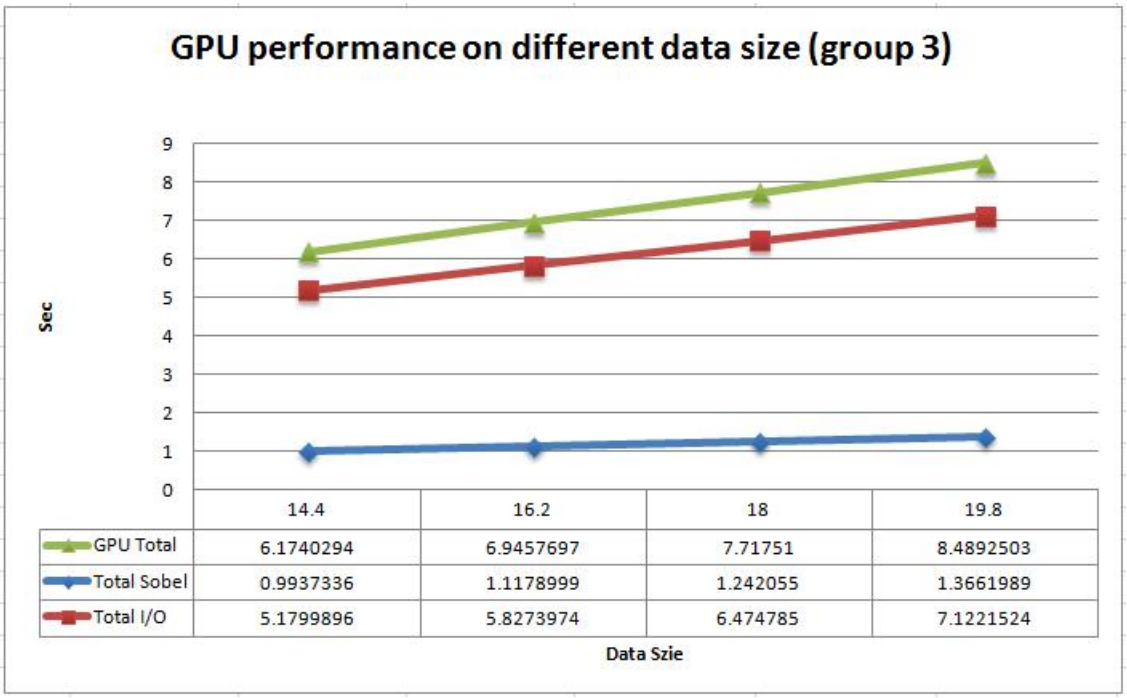


Figure 7.13: GPU performance on data size more than half the GPU memory and half the RAM

CPU RAM. Figure 7.15, shows the increase on the difference between the CPU and the GPU as the data size increases.

By using a simple 1D convolution that uses dot product which is a total of 5 instructions, Figure 7.16, which compares an optimized CPU version that runs on 8 cores, shows that the CPU version runs at 300% times faster than the GPU starting with 5 instructions. However as the number of instructions increases, the gap between the two versions is shrinking so that when they reach to 100 instructions, both have the same performance. As the number of instructions exceeds 100, the GPU performance starts to improve so that it performs 20% faster when the number of instructions reaches 500.

On the other hand, when comparing the default setting of C compiler version

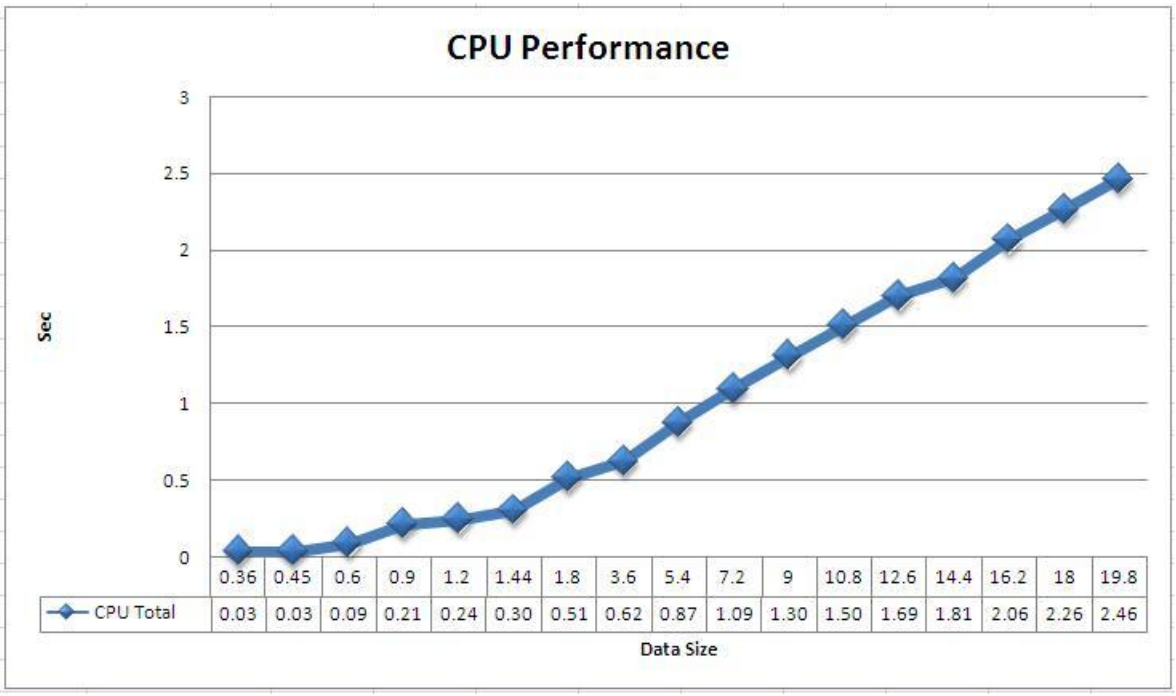


Figure 7.14: CPU performance on different data sizes

of the CPU version to the GPU version as shown in Figure 7.17, we notice a huge difference in performance just at number instructions that is equals to 5 such that the performance is 320% faster on the GPU side. As the number of operation increases, the gap is getting wider and the GPU becomes more powerful in handling a huge amount of instructions.

In conclusion, based on the previous results, when it comes to optimized compilation of C on CPU, and with a relatively small number of instructions, it is recommended to go for CPU, on the other hand, if the number of instructions is huge (500 instructions or more), it is recommended to go with GPU. When it comes to default C compiler setting that most of the compilers are setup with, the CPU version would run on a single core, and that would give the advantage to the GPU to have a noticeable better performance as the number of instructions increases.

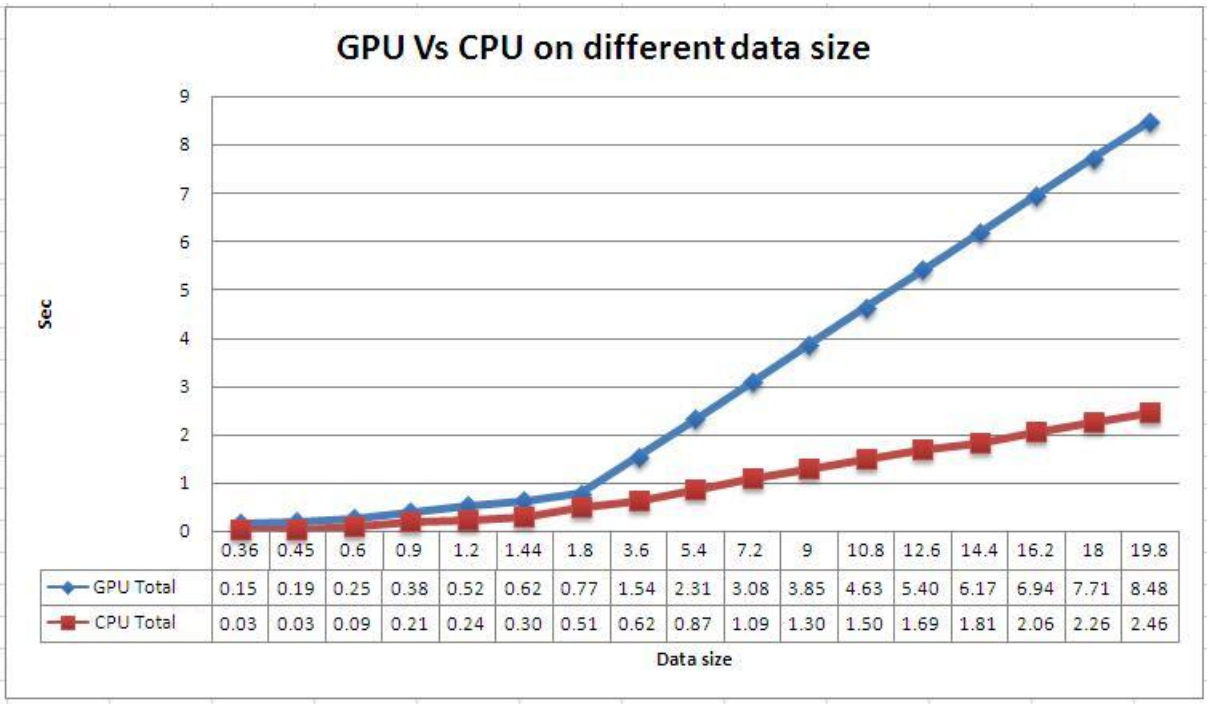


Figure 7.15: CPU Vs GPU performance on different data sizes

7.2 Summary

In this chapter, we showed the results of enhancing the GPU performance through three levels of improvement. these improvements were on the threads number, and thread layout on the block, and the enhancing the I/O processes. As a results, the GPU performance was decreased from 7.79 sec to 0.124 sec. The next section of the chapter talked about the compression between the GPU performance and the CPU performance in different data set. The last part of the discussed the results of comparing GPU to CPU based on instruction and showed a guild of when to move to GPU and when to move to CPU programming.

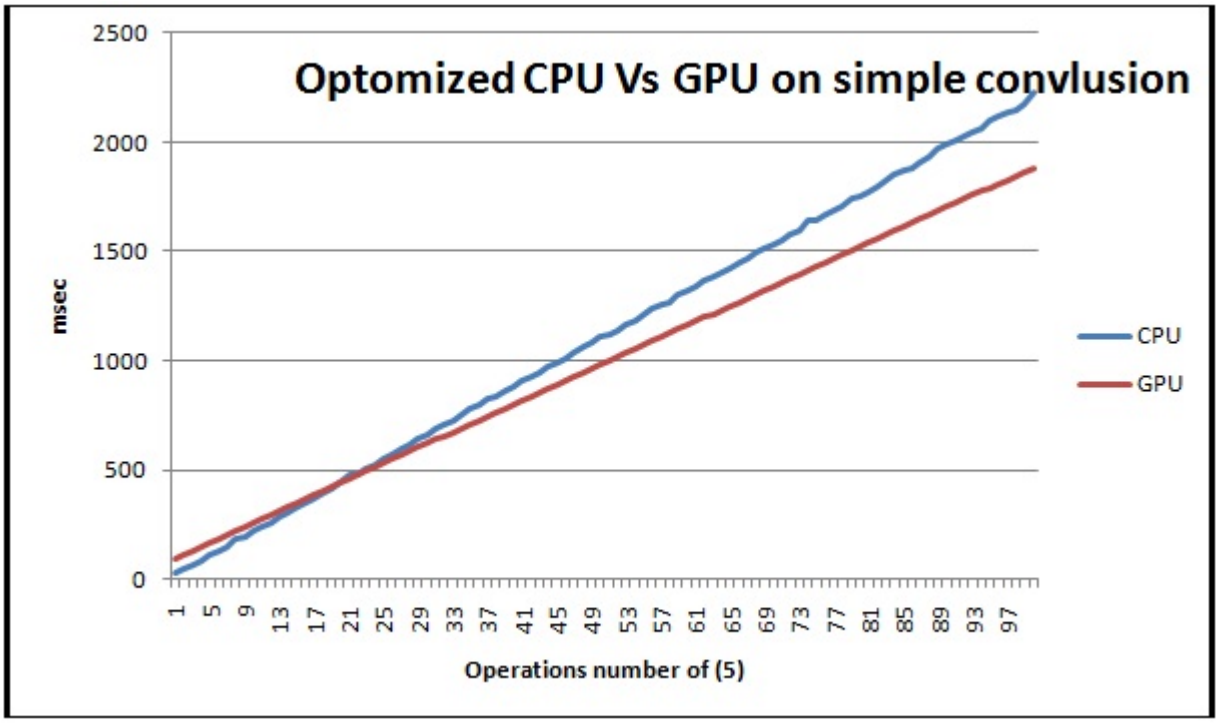


Figure 7.16: Optimized CPU Vs GPU on simple convolution

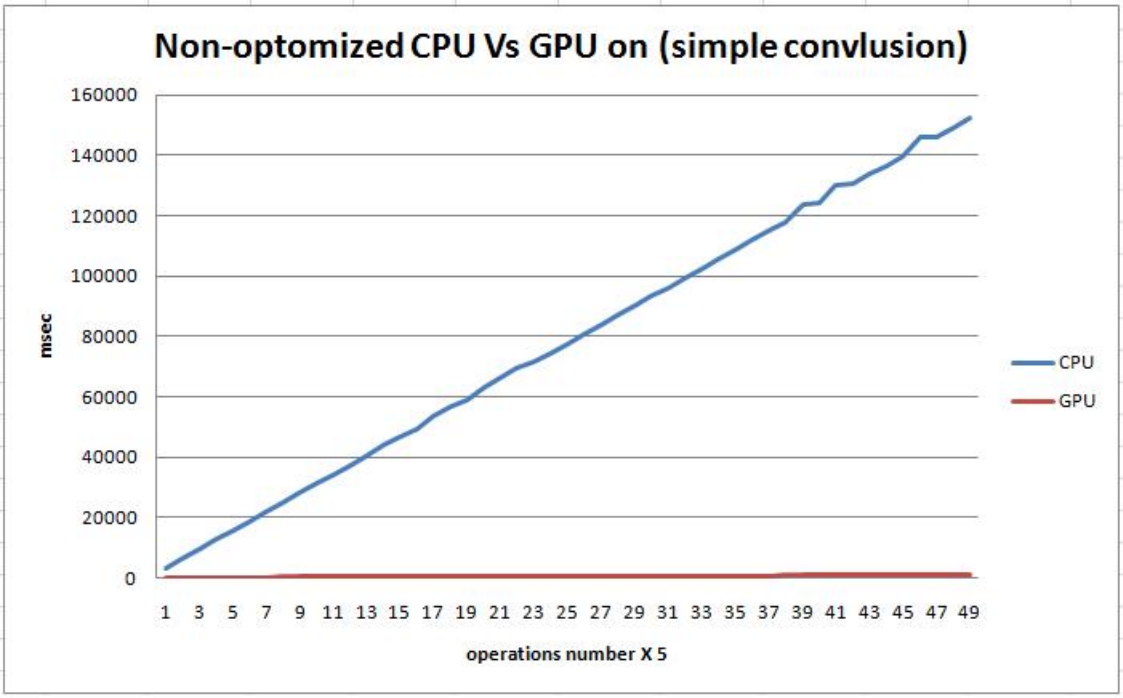


Figure 7.17: Defualt C compiler setting of CPU Vs GPU on simple convolution

CHAPTER 8

CONCLUSION AND FUTURE WORK

8.1 Overview

In this research work, we presented the GPU as a new hardware environment to detect edges using 3D Sobel edge detection filter and compared its performance to the Sobel edge detection performance running on a cluster of CPUs.

8.2 Contributions

In this thesis, there were two main contributions. The first contribution is modifying the Sobel edge detection to work on GPU structure using CUDA language. that was achieved by modifying the memory allocations, threads execution on the Sobel and finally data partitioning. The second contribution is enhancing the GPU speed

of execution Sobel edge detection by performing three level of optimization on the CUDA code, and then compare the best performance with the CPU performance on Sobel edge detection.

8.3 Conclusion

Running Sobel edge detection on GPU is in general performing very well. However, when it comes to the comparison with the CPU, the Read and the Write process seems to be the bottle neck which gives the advantage to the CPU when running using 8 cores machine.

With small number of instructions, CPU is performing faster than GPU. In GPU, 84% of the time is consumed by I/O processes. However, with large number of instructions, the GPU performs better because of the huge number of cores to handle the large number of instructions during one read and write processes.

As batch processes, GPU would not give the performance that would reflect the time effort of rewiring the code. With 4 cores machines, which most of the current machines are setup with, the GPU and the CPU performance breakeven. However when it comes to interactive application the GPU would play a big roll of taking some of the laod from CPU to perform an intensive mathematical calculations.

8.4 Future Work

According to Figure 7.9, the read and write process consumes 84% of the total time. This opens the door for more future enhancement to focus on the I/O process and any increase in the that I/O performance could have a big impact on the total time. In one of the experiments , we tried to minimize the I/O process by allowing the Sobel filter to access the data directly from RAM without copying it to GPU global memory, as a result, the I/O process took almost zero seconds however the Sobel edge detection process took about 1.29 seconds and that is almost 80% more than the best performance we got.

CUDA language supports multi-GPU implementation, which also opens the possibilities for more enhancement on the Sobel edge detection to work on two or more GPUs. This technique would the total performance specially when the data size exceeds the single GPU memory size.

Bibliography

[1] K. alGarni, S. alDossary, , and G. Ananos. Parallelizing 3d sobel edge detection using mpi and mpi/io. 2008.

[2] D. Argialas and O. Mavrantza. "comparison edge detection and hough transform techniques for the extraction of geologic features. In *Proceedings of The international Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, pages 790–795, December 2004.

[3] M. Bahorich and S. Farmer. 3-d seismic discontinuity for faults and stratigraphic features: The coherence cube. *the leading dege*, pages 1053–1058, october 1995.

[4] A. Balevic, L. Rockstroh., and W. Li. Acceleration of a finite-difference method with general purpose gpus - lesson learned. In *Proceedings of Computer and Information Technology, 8th IEEE International Conference*, 2008.

[5] M. Basu. Gaussian-based edge-detection methods-a survey. *IEEE Trans. Syst.*, 32(3):252–260, Aug. 2002.

[6] E. Bourennane, P. Gouton, M. Paindavoine, and F. Truchetet. Generalization of canny-deriche filter for detection of noisy exponential edge. 82(10):1317 1328, 2002.

[7] J. Ching and K. Wei. Interactive 3d seismic attribute volume generation with parallel graphics hardware. In *Proceedings of SEG/San Antonio*, 2007.

[8] S. Chopra and K. Marfurt. Seismic attributes - a promising aid for geologic prediction. 2006.

[9] H. Dai. Parallel processing of prestack kirchhoff time migration on a beowulf cluster.

[10] B. Deschizeaux and J.-Y. Blanc. Imaging earth's subsurface using cuda. *GPU Gems3 NVIDIA*, 2007.

[11] J. Fung and S. Mann. Using graphics devices in reverse: Gpu-based image processing and computer vision. In *Proceedings of IEEE International Conference on Multimedia and Expo*, pages 9–12, 2008.

[12] R. C. Gonzales, , and R. E. Woods. *Digital Image Processing*. Prentice Hall, 2002.

[13] W. Jeong and R. Whitaker. A fast iterative method for eikonal equations. *SIAM Journal of Scientific Computing*, March 2007.

[14] B. J. Kadlec and G. A. Dorn. Leveraging graphics processing units (gpus) for real-time seismic interpretation. page 60 66, January 2010.

[15] D. Kirk and Wen-mei. Cuda programming model. In *NVIDIA*, 2008.

[16] J. Lin and K. Wei. Interactive 3d seismic-attribute volume generation with parallel graphics hardware. 2007.

[17] C. H. Messom and A. L. C. Barczak. High precision gpu based integral images for moment invariant image processing systems, institute of information and mathematical sciences, massey university, auckland, new zealand.

[18] N. W. Moussa. Seismic imaging using gpgpu accelerated reverse time migration. *CS 315A Parallel Computer Architecture and Programming*, 2009.

[19] H. S. Neoh and A. Hazanchuk. Adaptive edge detection for real-time video processing using fpgas. In *GSPx 2004 Conference*, 2004.

[20] J. Nickolls and I. Buck. Scalable parallel programming with cuda. In *Queue*, pages 40–53, Feb. 2008.

[21] NVIDIA. Cuda programming guide. In *NVIDIA*, 2008.

[22] NVIDIA. Nvidia cuda architecture. 2009.

[23] Y. Roodt, W. Visser, , and W. Clarke. Image processing on the gpu: Implementing the canny edge detection algorithm. In *Proceedings of 18th International Symposium of the Pattern Recognition Association of South Africa (PRASA)*, Nov, 2007.

[24] M. Roushdy. Comparative study of edge detection algorithms applying on the grayscale noisy image using morphological filter. *GVIP Journal*, 6(4), December 2006.

[25] S. P. S. Yerneni, D. Bhardwaj, S. Chakraborty, , and R. Rastogi. Imaging subsurface geology with seismic migration on a computing cluster. 88(3), FEBRUARY 2005.

[26] A. L. Tertois and T. Frank. data filtering by 3d convolution. In *Proceedings of 24th GOCAD-MEETING*, 2004.

[27] O. R. Vincent and O. Folorunso. A descriptive algorithm for sobel image edge detection. In *Proceedings of Informing Science, IT Education Conference (InSITE)*, 2009.

[28] H. w. Cooper and R. Cook. Seismic data gathering. In *Proceedings of THE IEEE*, volume 72, October 1984.

[29] X. Wang. Laplacian operator-based edge detectors. 29(5):886 890, May 2002.

APPENDIX A

SOURCE CODE

```
//Sobel_512_512_1.cu (best performance Sobel Edge detection code on the GPU)

#include <stdio.h>
#include <time.h>
#include <fcntl.h>
#include <cutil.h>
#include "util.h"
#include <cutil_inline.h>
# include "cips.h"

/* prototype */
void read_volume_file(char *input_file, float ***the_volume, int nRows, int nCols, int nSlices);
void write_volume_file(float ***the_volume, char *output_file,  int nRows, int nCols, int nSlices);
void read_volume_file_dat(char *input_file, float ***the_volume, int nRows, int nCols, int nSlices);
void write_volume_file_dat(float ***the_volume, char *output_file,  int nRows, int nCols, int nSlice
float ***allocate_volume_array(int nRows, int nCols, int nSlices);
void find_edge_volume_sobel(float ***input_volume, float ***edge_volume,int nRows, int nCols, int nS
float find_max(float *in_array, long length);
float find_min(float *in_array, long length);
void float2gray(float *in_array, unsigned char *out_array, long length);
```

```
void sharp_volume(float ***input_volume, float ***output_volume, int nRows, int nCols, int nSlices);

//these fucntions are usefull for writing and reading  a file.
void find_edge_volume_sobel_GPU(float *input_volume, float *edge_volume, int nRows, int nCols, int n
void write_volume_file_dat_GPU(float *the_volume, char *output_file,  int nRows, int nCols, int nSli
void read_volume_file_GPU(char *input_file, float *the_volume, int nRows, int nCols, int nSlices);
void read_volume_file_dat_GPU(char *input_file, float* the_volume, int nRows, int nCols, int nSlices
void write_volume_file_dat_GPU_unbuffered(float *the_volume, char *output_file,  int nRows, int nCol
void read_volume_file_dat_GPU_unbuffered(char *input_file, float* buffer, int nRows, int nCols, int
//



//For GPU we define the global variable.
#define CACHE_CLEAR_SIZE     (1 << 24)
const   int nblocks= 1; //data partitioning

//This function is defined for working on the GPU
/**
@brief Sobel filter ind 3D, find edges of volumetric data using sobel masks
@param  one dimensional array of the volume data volume
@param  one dimensional array of the output data volume
@param  nRows the numebr of rows
@param  nRows the numebr of columns
@param  N is the dimension of the grid which is defined according to the data volume
see bellow.
*/
__global__ void sobel3d(float* a,float *c, int nRows, int nCols, int nSlices, int N)
{

  int col = blockIdx.x * blockDim.x + threadIdx.x;  //global value of the vertiavl cooridnates
  int row = blockIdx.y * blockDim.y + threadIdx.y; //global value of the horizontal cooridnates
  int index = col + row*N;
```

```
//It is better to define on share memmory because its acces is muche faster than global memmory

  __shared__ int sobel3x[3][3][3]; //constant array wich defines the sobel filter in x-direction
  __shared__ int sobel3y[3][3][3]; //constant array wich defines the sobel filter in y-direction
  __shared__ int sobel3z[3][3][3]; //constant array wich defines the sobel filter in z-direction


 if(threadIdx.y+threadIdx.x==0){ //defines just once for every block
   sobel3x[0][0][0]=1;sobel3x[0][0][1]=1;sobel3x[0][0][2]=1;sobel3x[0][1][0]=0;sobel3x[0][1][1]=0;s
   sobel3x[0][2][0]=-1;sobel3x[0][2][1]=-1;sobel3x[0][2][2]=-1;sobel3x[1][0][0]=1;sobel3x[1][0][1]=
   sobel3x[1][1][0]=0;sobel3x[1][1][1]=0;sobel3x[1][1][2]=0;sobel3x[1][2][0]=-1;sobel3x[1][2][1]=-2
   sobel3x[2][0][0]=1;sobel3x[2][0][1]=1;sobel3x[2][0][2]=1;sobel3x[2][1][0]=0;sobel3x[2][1][1]=0;s
   sobel3x[2][2][0]=-1;sobel3x[2][2][1]=-1;sobel3x[2][2][2]=-1;

   sobel3y[0][0][0]=1;sobel3y[0][0][1]=0;sobel3y[0][0][2]=-1;sobel3y[0][1][0]=1;sobel3y[0][1][1]=0;
   sobel3y[0][2][0]=1;sobel3y[0][2][1]=0;sobel3y[0][2][2]=-1;sobel3y[1][0][0]=1;sobel3y[1][0][1]=0;
   sobel3y[1][1][0]=2;sobel3y[1][1][1]=0;sobel3y[1][1][2]=-2;sobel3y[1][2][0]=1;sobel3y[1][2][1]=0;
   sobel3y[2][0][0]=1;sobel3y[2][0][1]=0;sobel3y[2][0][2]=-1;sobel3y[2][1][0]=1;sobel3y[2][1][1]=0;
   sobel3y[2][2][0]=1;sobel3y[2][2][1]=0;sobel3y[2][2][2]=-1;

   sobel3z[0][0][0]=1;sobel3z[0][0][1]=1;sobel3z[0][0][2]=1;sobel3z[0][1][0]=1;sobel3z[0][1][1]=2;s
   sobel3z[0][2][0]=1;sobel3z[0][2][1]=1;sobel3z[0][2][2]=1;sobel3z[1][0][0]=0;sobel3z[1][0][1]=0;s
   sobel3z[1][1][0]=0;sobel3z[1][1][1]=0;sobel3z[1][1][2]=0;sobel3z[1][2][0]=0;sobel3z[1][2][1]=0;s
   sobel3z[2][0][0]=-1;sobel3z[2][0][1]=-1;sobel3z[2][0][2]=-1;sobel3z[2][1][0]=-1;sobel3z[2][1][1]=
   sobel3z[2][1][2]=-1;sobel3z[2][2][0]=-1;sobel3z[2][2][1]=-1;sobel3z[2][2][2]=-1;

 }

 __syncthreads();

 int  V=nRows*nCols*nSlices;
 if ( (col<N) && (row<N)){
   if (index<V ){
     c[index]=0.0;
```

```
int i,j,k;
int l,m,n;
float sum_x=0, sum_y=0, sum_z=0;
int nRnC=nRows*nCols;

i=index/nRnC;                        //i: is the slice number
k=(index-nRnC*i)/nRows;    //k: is the column number
j=index-nRnC*i-k*nRows;    // j is the row number
int ijk;// if V[i][j][k] is the volume data, them equivalnet one dimensional array notation is
        // V[i*nRows*nCols+k*nRows+j]=V[index]
       // Following this mapping the definition of the Solbel filter is essencailly the same
       // for every value of the index=col + row*N, is associated one thread

if ( (i>0) && i<(nSlices-1)){
  if((j>0) && (j<(nRows-1))){
    if((k>0) && (k<(nCols-1))){



      sum_x = 0;
      sum_y = 0;
      sum_z = 0;
      for(l=-1;l<2;l++){
        for(m=-1;m<2;m++){
          for(n=-1;n<2;n++){
            ijk=(i+l)*nRnC+j+m+(k+n)*nRows;
            sum_x += a[ijk] * sobel3x[l+1][m+1][n+1];
            sum_y += a[ijk] * sobel3y[l+1][m+1][n+1];
            sum_z += a[ijk] * sobel3z[l+1][m+1][n+1];
          }
        }
      }
      sum_x = sum_x * sum_x;
      sum_y = sum_y * sum_y;
      sum_z = sum_z * sum_z;
```

```
            c[index]=sqrtf(sum_x+sum_y+sum_z/3.0f);

          }
        }
      }


    }

  }
}


int main(int argc, char **argv)
{
  double ts_read,tf_read, ts_write, tf_write, tt_read, tt_write,t0, ts_gpu, tf_gpu, tt_gpu;
  t0=r8_cpu_time();



  int NROWS=559;
  int NCOLS=391;
  int NSLICES=2200;
  float *c;

  int nRows = NROWS;
  int nCols = NCOLS;
  int nSlices = NSLICES/nblocks;

  float gs=nRows*nCols*nSlices/(512);
  int gridS=(int)floor(sqrt(gs))+1;
  int NS=gridS*sqrt(512);
  float r_elapsedTimeInMs = 0.0f;
  float w_elapsedTimeInMs = 0.0f;
  float f_elapsedTimeInMs = 0.0f;
```

```
int size=nRows*nCols*nSlices*sizeof(float);

char the_file[512] = "/slc/gad/sharikas/GPU_data/data/input_data.dat";
char out_file_GPU[512] = "/slc/gad/sharikas/GPU_data/data/output_512.512.1_1.dat";
float *the_volume_GPU;
float *the_volume_GPU_Total;
float *edge_volume_GPU_Total;

 cutilSafeCall( cudaHostAlloc( (void**)&the_volume_GPU, nRows*nCols*nSlices*sizeof(float), cudaHos
 cutilSafeCall( cudaHostAlloc( (void**)&the_volume_GPU_Total, nRows*nCols*nblocks*nSlices*sizeof(f
 cutilSafeCall( cudaHostAlloc( (void**)&edge_volume_GPU_Total, nRows*nCols*nblocks*nSlices*sizeof(
 cutilSafeCall( cudaHostAlloc( (void**)&c, nRows*nCols*nSlices*sizeof(float), cudaHostAllocWriteCo

if( cutCheckCmdLineFlag(argc, (const char**)argv, "device") )
cutilDeviceInit(argc, argv);
else
cudaSetDevice( cutGetMaxGflopsDeviceId() );

unsigned char *ad, *cd;
cudaMalloc((void**)&ad, size);
cudaMalloc((void**)&cd, size);
cudaEvent_t start, stop, w_start, w_stop, f_start, f_stop;
ts_read=r8_cpu_time();

read_volume_file_dat_GPU(the_file, the_volume_GPU_Total, nRows, nCols, nblocks*nSlices);
tf_read=r8_cpu_time();
tt_read = tf_read - ts_read;

int nsize=nRows*nCols*nSlices;
float togpu=0.0;
float fromgpu=0.0;
float tfilter=0.0;
float sizeTotal=(float)size*nblocks/(1024.0*1024.0);
float bandwidthInMBs;
```

```
double rtime;
// loop on the number of blocks
for(int block=0;block<nblocks;block++){
                the_volume_GPU=the_volume_GPU_Total+block*nsize;
                unsigned int hTimer;
                cutilCheckError( cutCreateTimer(&hTimer) );
                cutilSafeCall  ( cudaEventCreate( &start ) );
                cutilSafeCall  ( cudaEventCreate( &stop ) );
                cutilCheckError( cutStartTimer( hTimer));
                cutilSafeCall( cudaEventRecord( start, 0 ) );
                ts_gpu=r8_cpu_time();
                cutilSafeCall( cudaMemcpyAsync(ad,the_volume_GPU, size,cudaMemcpyHostToDevice, 0) );
                cutilSafeCall( cudaEventRecord( stop, 0 ) );

                // make sure GPU has finished copying
                cutilSafeCall( cudaThreadSynchronize() );
                cudaEventSynchronize(stop);
                //get the the total elapsed time in ms
                cutilCheckError( cutStopTimer( hTimer));
                cutilSafeCall( cudaEventElapsedTime( &r_elapsedTimeInMs, start, stop ) );
                r_elapsedTimeInMs = cutGetTimerValue( hTimer);

                dim3 dimBlock(512, 1);
                dim3 dimGrid(gridS, gridS);

                cutilSafeCall( cudaThreadSynchronize() );
                unsigned int f_Timer;
                cutilCheckError( cutCreateTimer(&f_Timer) );
                cutilSafeCall  ( cudaEventCreate( &f_start ) );
                cutilSafeCall  ( cudaEventCreate( &f_stop ) );
                struct timespec t1,t2;
                clock_gettime(CLOCK_REALTIME,&t1);
                sobel3d<<<dimGrid,dimBlock>>>((float *) ad,(float *)cd, nRows, nCols, nSlices,NS);
                cutilSafeCall( cudaThreadSynchronize() );
```

```
                // make sure GPU has finished copying
                cutilSafeCall( cudaThreadSynchronize() );
                cudaEventSynchronize(stop);
                //get the the total elapsed time in ms
                clock_gettime(CLOCK_REALTIME,&t2);
                rtime = (double) (t2.tv_sec - t1.tv_sec) + 1.0e-9*(t2.tv_nsec - t1.tv_nsec);
                CUDA_SAFE_CALL( cudaThreadSynchronize() );
                unsigned int w_Timer;
                cutilCheckError( cutCreateTimer(&w_Timer) );
                cutilSafeCall  ( cudaEventCreate( &w_start ) );
                cutilSafeCall  ( cudaEventCreate( &w_stop ) );
                cutilCheckError( cutStartTimer( w_Timer));
                cutilSafeCall( cudaEventRecord( w_start, 0 ) );
                cutilSafeCall( cudaMemcpyAsync(c,cd, size,cudaMemcpyDeviceToHost, 0) );
                cutilSafeCall( cudaEventRecord( w_stop, 0 ) );
                cudaEventSynchronize(w_stop);
                cutilCheckError( cutStopTimer(w_Timer) );
                cutilSafeCall( cudaEventElapsedTime( &w_elapsedTimeInMs, w_start, w_stop ) );
                w_elapsedTimeInMs = cutGetTimerValue( w_Timer);
                fromgpu=fromgpu+cutGetTimerValue(hTimer);
                tf_gpu=r8_cpu_time();
                //copy the data back to CPU
                memcpy(&edge_volume_GPU_Total[block*nsize],c,nsize*sizeof(float));
        }
tt_gpu =  tf_gpu - ts_gpu;


CUDA_SAFE_CALL(cudaFree(ad));
CUDA_SAFE_CALL(cudaFree(cd));
//printf("comnco\n");
 ts_write=r8_cpu_time();
 //write the resutls back to a file
 write_volume_file_dat_GPU_unbuffered(edge_volume_GPU_Total, out_file_GPU, nRows,nCols, nblocks*n
 tf_write=r8_cpu_time();
 tt_write = tf_write - ts_write;
```

```
   int fd=open(out_file_GPU,O_WRONLY);

//    printf("\n");
//    printf("*******************************Total*********************************************
//    printf("\n");
   printf("%f␣#",tt_read);
   printf("%f␣#", r_elapsedTimeInMs/1000.0);
   printf("%f␣#", rtime);
   printf("%f␣#",w_elapsedTimeInMs/1000.0);
   printf("%f␣#", tt_gpu);
   printf("%f␣\n",tt_write);

  return 0;
}


/*****************  sub routines  ******************/


/*******************************************************
* Read .dat volume data
* The .dat volume data contains no information about the
* dimension of the volume, so the number of rows, columns
* and slices must be specified when calling this sub routine.
* Each voxel is float type data in .dat file.
*******************************************************/

void read_volume_file_dat(char *input_file, float ***the_volume, int nRows, int nCols, int nSlices)
{
        FILE *fp;
        float *buffer;
        int i,j,k;

        buffer = (float *) malloc(nRows*nCols*nSlices*sizeof(float));
```

```
        if ((fp = fopen(input_file, "rb")) == NULL)
        {
                printf("Error␣open␣input␣file␣%s\n", input_file);
                perror("");
                exit(-1);
        }
        fseek(fp,0,SEEK_SET);
        fread(buffer, 1, nRows * nCols * nSlices * sizeof(float), fp);


        for(i=0; i<nSlices; i++)
        {
                for(k=0; k<nCols; k++)
                {
                        for(j=0; j<nRows; j++){
                                the_volume[i][j][k] = (float) buffer[i*nRows*nCols+k*nRows+j
                        }
                }
        }


        fclose(fp);
        free(buffer);
}

void read_volume_file_dat_GPU(char *input_file, float* buffer, int nRows, int nCols, int nSlices)
{
        FILE *fp;
//      float *buffer;
        int i,j,k;

//      buffer = (float *) malloc(nRows*nCols*nSlices*sizeof(float));
        if ((fp = fopen(input_file, "rb")) == NULL)
        {
```

```
                printf("Error open input file %s\n", input_file);
                perror("");
                exit(-1);
        }
        fseek(fp,0,SEEK_SET);
        fread(buffer, 1, nRows * nCols * nSlices * sizeof(float), fp);




        fclose(fp);
//      free(buffer);
}


void read_volume_file_dat_GPU_unbuffered(char *input_file, float* buffer, int nRows, int nCols, int
{
        int fp;
//      float *buffer;
        //int i,j,k;

//      buffer = (float *) malloc(nRows*nCols*nSlices*sizeof(float));
        if ((fp = open(input_file, O_RDONLY)) == -1)
        {
                printf("Error open input file %s\n", input_file);
                perror("");
                exit(-1);
        }
        lseek(fp,0,SEEK_SET);
        read(fp, buffer, nRows * nCols * nSlices * sizeof(float));
```

```
        close(fp);
//      free(buffer);
}



/*************************************************************
* allocate_volume_array:
* allocate array for volume with size
* nRows-by-nCols-by-nSlices, return an pointer array if succeed
*
*************************************************************/
float ***allocate_volume_array(int nRows, int nCols, int nSlices)
{
        int i,j;
        float ***the_array;
        the_array = (float ***) malloc( nSlices* sizeof(float**));
        for(i=0; i<nSlices; i++)
        {
                the_array[i] = (float **) malloc(nRows* sizeof(float*));
                for(j=0; j<nRows; j++)
                {
                        the_array[i][j] = (float *)malloc(nCols * sizeof(float));
                        if(the_array[i][j] == '\0')
                        {
                                printf("\n\t malloc of the_volume[%d,%d] failed", i,j);
                        }
                }
        }
        return(the_array);
}

/*************************************************************
* write_volume_file:
```

```
* write the volume data to .raw file
*
*
************************************************************/
void write_volume_file(float ***the_volume, char *output_file,  int nRows, int nCols, int nSlices)
{
        FILE *fp;
        float *buffer;
        int i,j,k;
        unsigned char *buffer_gray;

        buffer = (float *) malloc(nRows*nCols*nSlices*sizeof(float));
        if (buffer == NULL){
                printf("\n Insufficient memory!\n");
                exit(0);
        }

        for(i=0; i<nSlices; i++)
        {
                for(j=0; j<nRows; j++)
                {
                        for(k=0; k<nCols; k++){
                                buffer[i*nRows*nCols+j*nCols+k] = the_volume[i][j][k];
                        }
                }
        }


        /* normalize the range to 0-255 */
        buffer_gray = (unsigned char *) malloc(nRows*nCols*nSlices*sizeof(char));
        float2gray(buffer, buffer_gray, nRows*nCols*nSlices);

        if ((fp = fopen(output_file, "wb+")) == NULL)
```

```
        {
                printf("Error␣open␣input␣file␣%s\n", output_file);
                perror("");
                exit(-1);
        }
        fseek(fp,0,SEEK_SET);
        fwrite(buffer_gray, 1, nRows * nCols * nSlices, fp);

        fclose(fp);
        free(buffer);
        free(buffer_gray);
}




/*************************************************************
* write_volume_file_dat:
* write the volume data to .dat file
* each votex in the .dat file is of float data type, see also
* read_volume_file_dat for more information.
*************************************************************/
void write_volume_file_dat(float ***the_volume, char *output_file,  int nRows, int nCols, int nSlice
{
        FILE *fp;
        float *buffer;
        int i,j,k;


        buffer = (float *) malloc(nRows*nCols*nSlices*sizeof(float));
        if (buffer == NULL){
                printf("\n␣Insufficient␣memory!\n");
                exit(0);
```

```
        }


        for(i=0; i<nSlices; i++)
        {
                for(k=0; k<nCols; k++)
                {
                        for(j=0; j<nRows; j++){
                                buffer[i*nRows*nCols+k*nRows+j] = the_volume[i][j][k];
                        }
                }
        }




        if ((fp = fopen(output_file, "wb+")) == NULL)
        {
                printf("Error open input file %s\n", output_file);
                perror("");
                exit(-1);
        }
        fseek(fp,0,SEEK_SET);
        fwrite(buffer, 1, nRows * nCols * nSlices*sizeof(float), fp);


        fclose(fp);
        free(buffer);


}


/*******************************************************
* find_edge_volume_sobel:
*  find edges of volumetric data using sobel masks
*
*******************************************************/
void find_edge_volume_sobel(float ***input_volume, float ***edge_volume,
```

```
                                        int nRows, int nCols, int nSlices)
{
        int i,j,k,l,m,n;
        float sum_x=0, sum_y=0, sum_z=0;


        /* mask for sobel operator */
        int sobel3_x[3][3][3] = {{{1,  1,  1}, {0,  0,  0}, {-1, -1, -1}},
       {{1,  2,  1}, {0,  0,  0}, {-1, -2, -1}},
        {{1,  1,  1}, {0,  0,  0}, {-1, -1, -1}}};


        int sobel3_y[3][3][3] = {{{1, 0, -1}, {1, 0, -1}, {1, 0, -1}},
        {{1, 0, -1}, {2, 0, -2}, {1, 0, -1}},
        {{1, 0, -1}, {1, 0, -1}, {1, 0, -1}}};

        int sobel3_z[3][3][3] = {{{1, 1, 1},{1, 2, 1}, {1, 1, 1}},
        {{0, 0, 0}, {0, 0, 0}, {0, 0, 0}},
        {{-1, -1, -1}, {-1, -2, -1}, {-1, -1, -1}}};

        for(i=0; i<nSlices; i++){
                for(j=0; j<(nRows); j++){
                        for(k=0; k<(nCols); k++){
                                edge_volume[i][j][k] = 0;
                        }
                }
        }

        for(i=1; i<(nSlices-1); i++){
                for(j=1; j<(nRows-1); j++){
                        for(k=1; k<(nCols-1); k++){

                                /* along x direction */
                                sum_x = 0;
```

```
                                        sum_y = 0;
                                        sum_z = 0;
                                        for(l=-1;l<2;l++){
                                                for(m=-1;m<2;m++){
                                                        for(n=-1;n<2;n++){
                                                                sum_x += input_volume[i+l][j+m][k
                                                                sum_y += input_volume[i+l][j+m][k
                                                                sum_z += input_volume[i+l][j+m][k
                                                        }
                                                }
                                        }
                                        sum_x = sum_x * sum_x;
                                        sum_y = sum_y * sum_y;
                                        sum_z = sum_z * sum_z;
                                        edge_volume[i][j][k] = (float) sqrt((sum_x+sum_y+sum_z)/3);
                                }
                        }
                }

}

/*********************************************************
* find_max:
* find max value of an array
*********************************************************/
float find_max(float *in_array, long length)
{
        long i;
        float max_value = in_array[0];

        for(i=0; i<length; i++){
                if (max_value < in_array[i]){
                        max_value = in_array[i];
                }
```

```
        }
        return(max_value);
}

/*******************************************************
* Read raw volume data
* The .raw volume data contains no information about the
* dimension of the volume, so the number of rows, columns
* and slices must be specified when calling this sub routine.
* Each voxel is 8 bit in the .raw file.
*******************************************************/
void read_volume_file(char *input_file, float ***the_volume, int nRows, int nCols, int nSlices)
{
        FILE *fp;
        unsigned char *buffer;
        int i,j,k;

        buffer = (unsigned char *) malloc(nRows*nCols*nSlices*sizeof(char));
        if ((fp = fopen(input_file, "rb")) == NULL)
        {
                printf("Error␣open␣input␣file␣%s\n", input_file);
                perror("");
                exit(-1);
        }
        fseek(fp,0,SEEK_SET);
        fread(buffer, 1, nRows * nCols * nSlices, fp);


        for(i=0; i<nSlices; i++)
        {
                for(j=0; j<nRows; j++)
                {
                        for(k=0; k<nCols; k++){
                                the_volume[i][j][k] = (float) buffer[i*nRows*nCols+j*nCols+k
```

```
                        }
                }
        }


        fclose(fp);
        free(buffer);
}


void read_volume_file_GPU(char *input_file, float *the_volume, int nRows, int nCols, int nSlices)
{
        FILE *fp;
        unsigned char *buffer;
        int i,j,k;

        buffer = (unsigned char *) malloc(nRows*nCols*nSlices*sizeof(char));
        if ((fp = fopen(input_file, "rb")) == NULL)
        {
                printf("Error open input file %s\n", input_file);
                perror("");
                exit(-1);
        }
        fseek(fp,0,SEEK_SET);
        fread(buffer, 1, nRows * nCols * nSlices, fp);


        for(i=0; i<nSlices; i++)
        {
                for(j=0; j<nRows; j++)
                {
                        for(k=0; k<nCols; k++){
                                the_volume[i*nRows*nCols+j*nCols+k] = (float) buffer[i*nRows
                        }
```

```
                }
        }


        fclose(fp);
        free(buffer);
}

void write_volume_file_dat_GPU(float *the_volume, char *output_file,  int nRows, int nCols, int nSli
{
        FILE *fp;
//      float *buffer;
//      int i,j,k;

        if ((fp = fopen(output_file, "wb+")) == NULL)
        {
                printf("Error open input file %s\n", output_file);
                perror("");
                exit(-1);
        }
        fseek(fp,0,SEEK_SET);
        fwrite(the_volume, 1, nRows * nCols * nSlices*sizeof(float), fp);
       // printf("passo\n");
        fclose(fp);


}


void write_volume_file_dat_GPU_unbuffered(float *the_volume, char *output_file,  int nRows, int nCol
{
        int fp;
//      float *buffer;
//      int i,j,k;
```

```
        if ((fp = open(output_file, O_WRONLY)) == -1)
        {
                printf("Error open input file %s\n", output_file);
                perror("");
                exit(-1);
        }
        lseek(fp,0,SEEK_SET);
        write(fp,the_volume, nRows * nCols * nSlices*sizeof(float));
       // printf("passo\n");
        close(fp);


}

void find_edge_volume_sobel_GPU(float *input_volume, float *edge_volume,
                                        int nRows, int nCols, int nSlices)
{
        int i,j,k,l,m,n;
        float sum_x=0, sum_y=0, sum_z=0;


        /* mask for sobel operator */
        int sobel3_x[3][3][3] = {{{1,  1,  1}, {0,  0,  0}, {-1, -1, -1}},
       {{1,  2,  1}, {0,  0,  0}, {-1, -2, -1}},
        {{1,  1,  1}, {0,  0,  0}, {-1, -1, -1}}};


        int sobel3_y[3][3][3] = {{{1, 0, -1}, {1, 0, -1}, {1, 0, -1}},
        {{1, 0, -1}, {2, 0, -2}, {1, 0, -1}},
        {{1, 0, -1}, {1, 0, -1}, {1, 0, -1}}};

        int sobel3_z[3][3][3] = {{{1, 1, 1},{1, 2, 1}, {1, 1, 1}},
        {{0, 0, 0}, {0, 0, 0}, {0, 0, 0}},
```

```
{{-1, -1, -1}, {-1, -2, -1}, {-1, -1, -1}}};


for(i=0; i<nSlices; i++){
        for(j=0; j<(nRows); j++){
                for(k=0; k<(nCols); k++){
                        edge_volume[i*nRows*nCols+k*nRows+j] = 0;
                }
        }
}


for(i=1; i<(nSlices-1); i++){
        for(j=1; j<(nRows-1); j++){
                for(k=1; k<(nCols-1); k++){

                        /* along x direction */
                        sum_x = 0;
                        sum_y = 0;
                        sum_z = 0;
                        for(l=-1;l<2;l++){
                                for(m=-1;m<2;m++){
                                        for(n=-1;n<2;n++){
                                sum_x += input_volume[(i+l)*(nRows*nCols)+j+m+(k+n)*
                                sum_y += input_volume[(i+l)*(nRows*nCols)+j+m+(k+n)*
                                sum_z += input_volume[(i+l)*(nRows*nCols)+j+m+(k+n)*
                                        }
                                }
                        }
                        sum_x = sum_x * sum_x;
                        sum_y = sum_y * sum_y;
                        sum_z = sum_z * sum_z;
                        edge_volume[i*nRows*nCols+k*nRows+j] = (float) sqrt((sum_x+s
                }
        }
}
```

```
}



/*************************************************************
* find_min:
* find min value of an array
*************************************************************/
float find_min(float *in_array, long length)
{
        long i;
        float min_value = in_array[0];

        for(i=0; i<length; i++){
                if (min_value > in_array[i]){
                        min_value = in_array[i];
                }
        }
        return(min_value);
}



/*************************************************************
* float2gray:
* normalize the range of input array into range [0 255]
*************************************************************/
void float2gray(float *in_array, unsigned char *out_array, long length)
{
        long i;
        float max_gray = 255;
        float min_gray = 1;
        float max, min;
        max = find_max(in_array, length);
```

```
        min = find_min(in_array, length);
        for(i=0; i<length; i++){
                out_array[i] = (unsigned char) floor((max_gray-min_gray)*(in_array[i]-min)/(max-min)

        }

}

/***********************************************
* this function uses the sharp mask to sharp the volume
* data
***********************************************/
void sharp_volume(float ***input_volume, float ***output_volume,
                                        int nRows, int nCols, int nSlices)
{

        int i,j,k,l,m,n;
        int mask[3][3][3] = {{{0,-1,0},{-1,-2,-1},{0,-1,0}},
                             {{-1,-2,-1},{-2,25,-2},{-1,-2,-1}},
                             {{0,-1,0},{-1,-2,-1},{0,-1,0}}};


        for(i=0; i<nSlices; i++){
                for(j=0; j<(nRows); j++){
                        for(k=0; k<(nCols); k++){
                                output_volume[i][j][k] = 0;
                        }
                }
        }

        for(i=1; i<(nSlices-1); i++){
                for(j=1; j<(nRows-1); j++){
                        for(k=1; k<(nCols-1); k++){
```

```
                                        for(l=-1;l<2;l++){
                                                for(m=-1;m<2;m++){
                                                        for(n=-1;n<2;n++){
                                                                output_volume[i][j][k] +=
                                                                        input_volume[i+l][j+
                                                        }
                                                }
                                        }


                                }
                        }
                }

} /* end of routine */
```

```
//main.cpp

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <cutil_inline.h>
#include <xmmintrin.h>
#include "convolutionSeparable_common.h"

////////////////////////////////////////////////////////////////////////////////
// Reference CPU convolution
////////////////////////////////////////////////////////////////////////////////
extern "C" void convolutionRowCPU(
    float *h_Result,
    float *h_Data,
    float *h_Kernel,
    int imageW,
    int imageH,
    int kernelR
);

extern "C" void convolutionColumnCPU(
    float *h_Result,
    float *h_Data,
    float *h_Kernel,
    int imageW,
    int imageH,
    int kernelR
);
```

```
////////////////////////////////////////////////////////////////////////////////
// Main program
////////////////////////////////////////////////////////////////////////////////
int main(int argc, char **argv){
    float
        *h_Kernel,
        *h_Input,
        *h_Buffer,
        *h_OutputCPU,
        *h_OutputGPU;

    float
        *d_Input,
        *d_Output,
        *d_Buffer;



//    const int imageW =24576;
//    const int imageH = 24576/16 ;
    const unsigned int iterations = 10;
    const int imageW =10240;
    const int imageH = 10240;

    unsigned int hTimer;

    //Use command-line specified CUDA device, otherwise use device with highest Gflops/s
    if ( cutCheckCmdLineFlag(argc, (const char **)argv, "device") )
        cutilDeviceInit(argc, argv);
    else
//        cudaSetDevice( cutGetMaxGflopsDeviceId() );
//        cudaSetDevice( 1 );
        cudaDeviceProp deviceProp;

  /* Get the device selected by the user or default to 0, and then set it. */
```

```
        int deviceCount = 0;
        int idev =1;
        char *device = "1";

    cudaGetDeviceCount(&deviceCount);
    idev = atoi(device);
    if(idev >= deviceCount || idev < 0)
    {
      fprintf(stderr, "Invalid device number %d, using default device 0.\n",
              idev);
      idev = 0;
    }

  cutilSafeCall(cudaSetDevice(idev));

    cutilCheckError(cutCreateTimer(&hTimer));

    printf("%i x %i\n", imageW, imageH);
    printf("Allocating and intializing host arrays...\n");
        cutilSafeCall( cudaHostAlloc( (void**)&h_Kernel,KERNEL_LENGTH * sizeof(float), cudaHostAlloc
        cutilSafeCall( cudaHostAlloc( (void**)&h_Input,imageW * imageH * sizeof(float), cudaHostAllo
        cutilSafeCall( cudaHostAlloc( (void**)&h_Buffer,imageW * imageH * sizeof(float), cudaHostAll
        cutilSafeCall( cudaHostAlloc( (void**)&h_OutputGPU ,imageW * imageH * sizeof(float), cudaHos

/*      float *CPUh_Kernel = (float *) _mm_malloc( KERNEL_LENGTH * sizeof(float),16);
        float *CPUh_Input = (float *) _mm_malloc( imageW * imageH * sizeof(float),16);
        float *CPUh_Buffer = (float *) _mm_malloc( imageW * imageH * sizeof(float),16);
        h_OutputCPU = (float *) _mm_malloc( imageW * imageH * sizeof(float),16);
*/
        float *CPUh_Kernel = (float *) malloc( KERNEL_LENGTH * sizeof(float));
        float *CPUh_Input = (float *) malloc( imageW * imageH * sizeof(float));
        float *CPUh_Buffer = (float *) malloc( imageW * imageH * sizeof(float));
        h_OutputCPU = (float *) malloc( imageW * imageH * sizeof(float));
```

```
    srand(200);
    for(unsigned int i = 0; i < KERNEL_LENGTH; i++)
        h_Kernel[i] = (float)(rand() % 16);
    for(unsigned i = 0; i < imageW * imageH; i++)
    {
        h_Input[i] = (float)(rand() % 16);
            CPUh_Input[i] = (float)(rand() % 16);
    }


printf("Allocating and initializing CUDA arrays...\n");
    cutilSafeCall( cudaMalloc((void **)&d_Input,   imageW * imageH * sizeof(float)) );
    cutilSafeCall( cudaMalloc((void **)&d_Output,  imageW * imageH * sizeof(float)) );
    cutilSafeCall( cudaMalloc((void **)&d_Buffer , imageW * imageH * sizeof(float)) );

    setConvolutionKernel(h_Kernel);


printf("Running GPU convolution (%u identical iterations)...\n", iterations);
    for(int jj=1;jj<50;jj++)
    {
    cutilSafeCall( cudaThreadSynchronize() );
    cutilCheckError( cutResetTimer(hTimer) );
    cutilCheckError( cutStartTimer(hTimer) );
        for(unsigned int i = 0; i < iterations; i++){
    cutilSafeCall( cudaMemcpyAsync(d_Input, h_Input, imageW * imageH * sizeof(float),cudaMemcpyH
          for(int ii=0;ii<jj;ii++)
            {
            convolutionRowsGPU(
                d_Buffer,
                d_Input,
                imageW,
                imageH
```

```
                );


    /*              convolutionColumnsGPU(
                      d_Output,
                      d_Buffer,
                      imageW,
                      imageH
                );
*/
                }
            }
    //printf(" %d Reading back GPU results...\n",jj);
        cutilSafeCall( cudaMemcpyAsync(h_OutputGPU, d_Output, imageW * imageH * sizeof(float),cudaMe
        cutilSafeCall( cudaThreadSynchronize() );
        cutilCheckError(cutStopTimer(hTimer));
        float gpuTime = cutGetTimerValue(hTimer) / (float)iterations;
 //   printf("Average GPU convolution time : %f msec //%f Mpixels/sec\n", gpuTime, 1e-6 * imageW * i


//    printf("Checking the results...\n");
//        printf("...running convolutionRowCPU()\n");


        cutilCheckError( cutResetTimer(hTimer) );
        cutilCheckError( cutStartTimer(hTimer) );
              for(int ii=0;ii<jj;ii++)
                {
        convolutionRowCPU(
            CPUh_Buffer,
            CPUh_Input,
            CPUh_Kernel,
            imageW,
            imageH,
            KERNEL_RADIUS
        );
```

```
//          printf("...running convolutionColumnCPU()\n");
/*          convolutionColumnCPU(
              h_OutputCPU,
              CPUh_Buffer,
              CPUh_Kernel,
              imageW,
              imageH,
              KERNEL_RADIUS
          );
*/
          }
          cutilCheckError(cutStopTimer(hTimer));
          float cpuTime = cutGetTimerValue(hTimer) / (float)iterations;
          printf("Average %10d     %10.3f   %10.3f     %10.3f\n", jj,cpuTime, gpuTime,gpuTime/cpuTime);

  }
          printf("...comparing the results\n");
          double sum = 0, delta = 0;
          for(unsigned i = KERNEL_RADIUS; i < imageW-KERNEL_RADIUS ; i++){
          for(unsigned j = KERNEL_RADIUS; j <  imageH-KERNEL_RADIUS; j++){
              delta += (h_OutputGPU[j*imageH+i] - h_OutputCPU[j*imageH+i]) * (h_OutputGPU[j*imageH+i]
              sum   += h_OutputCPU[j*imageH+i] * h_OutputCPU[j*imageH+i];
          }
          }
          double L2norm = sqrt(delta / sum);
          printf("Relative L2 norm: %E\n", L2norm);
      printf((L2norm < 1e-6) ? "TEST PASSED\n" : "TEST FAILED\n");

      printf("Shutting down...\n");
          cutilSafeCall( cudaFree(d_Buffer ) );
          cutilSafeCall( cudaFree(d_Output) );
          cutilSafeCall( cudaFree(d_Input) );
          free(h_OutputGPU);
```

```
        free(h_OutputCPU);
        free(h_Buffer);
        free(h_Input);
        free(h_Kernel);

    cutilCheckError(cutDeleteTimer(hTimer));

    cudaThreadExit();

    cutilExit(argc, argv);
}
```

```
//convolutionSeparable_gold.cpp


#include "convolutionSeparable_common.h"


////////////////////////////////////////////////////////////////////////////////
// Reference row convolution filter
////////////////////////////////////////////////////////////////////////////////
extern "C" void convolutionRowCPU(
    float *h_Dst,
    float *h_Src,
    float *h_Kernel,
    int imageW,
    int imageH,
    int kernelR
){
int y;
int x;
#pragma omp parallel for private (y,x)
    for(y = 0; y < imageH; y++)
        for(x = kernelR; x < imageW-kernelR; x++){
            float sum = 0;
#pragma ivdep
#pragma vector always
            for(int k = -kernelR; k <= kernelR; k++){
                int d = x + k;
                    sum += h_Src[y * imageW + d] * h_Kernel[kernelR - k];
            }
            h_Dst[y * imageW + x] = sum;
        }
//      }
}
```

```
////////////////////////////////////////////////////////////////////////////////
// Reference column convolution filter
////////////////////////////////////////////////////////////////////////////////
extern "C" void convolutionColumnCPU(
    float *h_Dst,
    float *h_Src,
    float *h_Kernel,
    int imageW,
    int imageH,
    int kernelR
){

        int y;
        int x;
#pragma omp parallel for private (y,x)
    for(y = kernelR; y < imageH-kernelR; y++)
        {
        for(x = 0; x < imageW; x++){
            float sum = 0;
#pragma ivdep
#pragma vector always
            for(int k = -kernelR; k <= kernelR; k++){
                int d = y + k;
                    sum += h_Src[d * imageW + x] * h_Kernel[kernelR - k];
            }
            h_Dst[y * imageW + x] = sum;
        }
        }
}
```

```
//convolutionSeparable.cu

#include <assert.h>
#include <cutil_inline.h>
#include "convolutionSeparable_common.h"

////////////////////////////////////////////////////////////////////////////////
// Convolution kernel storage
////////////////////////////////////////////////////////////////////////////////
__constant__ float c_Kernel[KERNEL_LENGTH];

extern "C" void setConvolutionKernel(float *h_Kernel){
    cudaMemcpyToSymbol(c_Kernel, h_Kernel, KERNEL_LENGTH * sizeof(float));
}


////////////////////////////////////////////////////////////////////////////////
// Row convolution filter
////////////////////////////////////////////////////////////////////////////////
#define   ROWS_BLOCKDIM_X 16
#define   ROWS_BLOCKDIM_Y 4
#define ROWS_RESULT_STEPS 4
#define   ROWS_HALO_STEPS 1

__global__ void convolutionRowsKernel(
    float *d_Dst,
    float *d_Src,
    int imageW,
    int imageH,
    int pitch
){
    __shared__ float s_Data[ROWS_BLOCKDIM_Y][(ROWS_RESULT_STEPS + 2 * ROWS_HALO_STEPS) * ROWS_BLOCKD
```

```
//Offset to the left halo edge
const int baseX = (blockIdx.x * ROWS_RESULT_STEPS - ROWS_HALO_STEPS) * ROWS_BLOCKDIM_X + threadI
const int baseY = blockIdx.y * ROWS_BLOCKDIM_Y + threadIdx.y;

d_Src += baseY * pitch + baseX;
d_Dst += baseY * pitch + baseX;

//Main data
#pragma unroll
for(int i = ROWS_HALO_STEPS; i < ROWS_HALO_STEPS + ROWS_RESULT_STEPS; i++)
    s_Data[threadIdx.y][threadIdx.x + i * ROWS_BLOCKDIM_X] = d_Src[i * ROWS_BLOCKDIM_X];

//Left halo
for(int i = 0; i < ROWS_HALO_STEPS; i++){
    s_Data[threadIdx.y][threadIdx.x + i * ROWS_BLOCKDIM_X] =
        (baseX >= -i * ROWS_BLOCKDIM_X ) ? d_Src[i * ROWS_BLOCKDIM_X] : 0;
}

//Right halo
for(int i = ROWS_HALO_STEPS + ROWS_RESULT_STEPS; i < ROWS_HALO_STEPS + ROWS_RESULT_STEPS + ROWS_
    s_Data[threadIdx.y][threadIdx.x + i * ROWS_BLOCKDIM_X] =
        (imageW - baseX > i * ROWS_BLOCKDIM_X) ? d_Src[i * ROWS_BLOCKDIM_X] : 0;
}

//Compute and store results
__syncthreads();
#pragma unroll
for(int i = ROWS_HALO_STEPS; i < ROWS_HALO_STEPS + ROWS_RESULT_STEPS; i++){
    float sum = 0;

    #pragma unroll
    for(int j = -KERNEL_RADIUS; j <= KERNEL_RADIUS; j++)
        sum += c_Kernel[KERNEL_RADIUS - j] * s_Data[threadIdx.y][threadIdx.x + i * ROWS_BLOCKDIM
```

```
        d_Dst[i * ROWS_BLOCKDIM_X] = sum;
    }
}

extern "C" void convolutionRowsGPU(
    float *d_Dst,
    float *d_Src,
    int imageW,
    int imageH
){
    assert( ROWS_BLOCKDIM_X * ROWS_HALO_STEPS >= KERNEL_RADIUS );
    assert( imageW % (ROWS_RESULT_STEPS * ROWS_BLOCKDIM_X) == 0 );
    assert( imageH % ROWS_BLOCKDIM_Y == 0 );

    dim3 blocks(imageW / (ROWS_RESULT_STEPS * ROWS_BLOCKDIM_X), imageH / ROWS_BLOCKDIM_Y);
    dim3 threads(ROWS_BLOCKDIM_X, ROWS_BLOCKDIM_Y);

    convolutionRowsKernel<<<blocks, threads>>>(
        d_Dst,
        d_Src,
        imageW,
        imageH,
        imageW
    );
    cutilCheckMsg("convolutionRowsKernel() execution failed\n");
}

////////////////////////////////////////////////////////////////////////////////
// Column convolution filter
////////////////////////////////////////////////////////////////////////////////
#define   COLUMNS_BLOCKDIM_X  16
#define   COLUMNS_BLOCKDIM_Y  8
```

```
#define COLUMNS_RESULT_STEPS 4
#define    COLUMNS_HALO_STEPS 1

__global__ void convolutionColumnsKernel(
    float *d_Dst,
    float *d_Src,
    int imageW,
    int imageH,
    int pitch
){
    __shared__ float s_Data[COLUMNS_BLOCKDIM_X][(COLUMNS_RESULT_STEPS + 2 * COLUMNS_HALO_STEPS) * CO

    //Offset to the upper halo edge
    const int baseX = blockIdx.x * COLUMNS_BLOCKDIM_X + threadIdx.x;
    const int baseY = (blockIdx.y * COLUMNS_RESULT_STEPS - COLUMNS_HALO_STEPS) * COLUMNS_BLOCKDIM_Y
    d_Src += baseY * pitch + baseX;
    d_Dst += baseY * pitch + baseX;

    //Main data
    #pragma unroll
    for(int i = COLUMNS_HALO_STEPS; i < COLUMNS_HALO_STEPS + COLUMNS_RESULT_STEPS; i++)
        s_Data[threadIdx.x][threadIdx.y + i * COLUMNS_BLOCKDIM_Y] = d_Src[i * COLUMNS_BLOCKDIM_Y * p

    //Upper halo
    for(int i = 0; i < COLUMNS_HALO_STEPS; i++)
        s_Data[threadIdx.x][threadIdx.y + i * COLUMNS_BLOCKDIM_Y] =
            (baseY >= -i * COLUMNS_BLOCKDIM_Y) ? d_Src[i * COLUMNS_BLOCKDIM_Y * pitch] : 0;

    //Lower halo
    for(int i = COLUMNS_HALO_STEPS + COLUMNS_RESULT_STEPS; i < COLUMNS_HALO_STEPS + COLUMNS_RESULT_S
        s_Data[threadIdx.x][threadIdx.y + i * COLUMNS_BLOCKDIM_Y] =
            (imageH - baseY > i * COLUMNS_BLOCKDIM_Y) ? d_Src[i * COLUMNS_BLOCKDIM_Y * pitch] : 0;

    //Compute and store results
```

```
    __syncthreads();
    #pragma unroll
    for(int i = COLUMNS_HALO_STEPS; i < COLUMNS_HALO_STEPS + COLUMNS_RESULT_STEPS; i++){
        float sum = 0;
        #pragma unroll
        for(int j = -KERNEL_RADIUS; j <= KERNEL_RADIUS; j++)
            sum += c_Kernel[KERNEL_RADIUS - j] * s_Data[threadIdx.x][threadIdx.y + i * COLUMNS_BLOCK

        d_Dst[i * COLUMNS_BLOCKDIM_Y * pitch] = sum;
    }
}

extern "C" void convolutionColumnsGPU(
    float *d_Dst,
    float *d_Src,
    int imageW,
    int imageH
){
    assert( COLUMNS_BLOCKDIM_Y * COLUMNS_HALO_STEPS >= KERNEL_RADIUS );
    assert( imageW % COLUMNS_BLOCKDIM_X == 0 );
    assert( imageH % (COLUMNS_RESULT_STEPS * COLUMNS_BLOCKDIM_Y) == 0 );

    dim3 blocks(imageW / COLUMNS_BLOCKDIM_X, imageH / (COLUMNS_RESULT_STEPS * COLUMNS_BLOCKDIM_Y));
    dim3 threads(COLUMNS_BLOCKDIM_X, COLUMNS_BLOCKDIM_Y);

    convolutionColumnsKernel<<<blocks, threads>>>(
        d_Dst,
        d_Src,
        imageW,
        imageH,
        imageW
    );
    cutilCheckMsg("convolutionColumnsKernel() execution failed\n");
}
```

VITAE

- Abdulaziz S. Al-Sharikh.
- Nationality : Saudi
- Born in Kuwait city, Kuwait, on November 29, 1976.
- Address : P. O. Box 8171, Dhahran, 31311, Saudi Arabia
- Phone Number: 0504825572
- Earned a Bachelors of Science degree in Computer Science from King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, in June 2000.
- E-mail: sharikas@aramco.com, sharikh.abdulaziz@gmail.com