PARALLEL EDGE DETECTION BY SOBEL ALGORITHM USING CUDA C

Adhir Jain
Dept. of Computer Science & Engg.
MANIT
Bhopal, M.P., India 462003
adhir.jain111@gmail.com

Anand Namdev
Dept. of Computer Science & Engg.
MANIT
Bhopal,M.P., India 462003
anand.nitb19@gmail.com

Dr. Meenu Chawla
Dept. of Computer Science & Engg.
MANIT
Bhopal,M.P., India 462003
chawlam@manit.ac.in

Abstract—**Edge detection is one of the most important paradigm of Image processing. Images contain millions of pixel and each pixel information is independent of its neighbouring pixel. Hence this paper puts to test the capability of Graphics Processing Unit (GPU) to compute in parallel against the millions of pixel calculations involved in image processing. Each pixel operation is independent from other thus GPU can be effectively used along with the help of high level programmable interfaces. More specifically, this paper focuses on Compute Unified Device Architecture (CUDA) as its parallel programming platform and examines the possible gain in time which can be attained for edge detection in images. A well-known algorithm SOBEL for edge detection is used in the experiment. A dataset of images was tested for edge detection both serially and parallely. The results of parallel algorithm were further divided according to the machine on which the algorithm was tested and were further classified according to the number of kernel function used in each machine. Results showed that parallel implementation is about 262 times and 943 times faster when 2 kernel functions were implemented in parallel on GeForce and Tesla machines respectively, and about 120 times and 455 times faster when 3 kernel functions were implemented in parallel on GeForce and Tesla machines respectively as compared to serial implementation for larger images. Statistics also showed a decline in speedup of about 52% when 3 kernels were used than when 2 kernels were used due to increase in communication time. Hence, an analysis came out that the decision regarding which sections of the algorithm to be parallelised, should be taken wisely. If not, would lead to an additional overhead i.e. communication time (time taken in transferring the data from CPU to GPU & back from GPU to CPU) thereby reducing the overall speedup.**

Keywords — *Parallel Edge Detection, Parallel Computing, CUDA, Kernel, Sobel Operator, Speedup, Image Processing, MATLAB, Parallelism v/s Speedup*

I. Introduction

Image processing is one of the most important field widely used today. The human vision has the capability to easily acquire, understand and interpret the information stored in the form of images. On the other hand it is a challenging task to make the machine understand, acquire these information so as to automate the task without human intervention. It is thus important to learn and implement various techniques of the image processing. One of the most important paradigm of the image processing is the Image Edge detection. Image edges are the most basic features of an image. Edges are the set of connected pixel that forms the boundary between two disjoint regions. Edges allows the user to observe feature of an image where more or less abrupt change in intensity occurs. Edge detection has several important applications in digital image processing like pattern recognition, medical field,etc. A number of edge detection algorithms have been described in previous papers. Sobel operator [1] is one of the classic operator used in edge detection as it is a simple operator, insensitive to noise and requires less computations in comparison to other operators. How to quickly and accurately extract the edge information of the images is always a hot research topic. An image consists of millions of pixels and each pixel operation is independent to its neighbouring pixel. Thus parallel programming model can be used which focuses on performing many operations concurrently but slowly rather than performing individual operations rapidly [10].

CUDA [5] is a parallel computing architecture developed by NVidia for massively parallel high performance computing. It is the compute engine in the GPU and is accessible by developers through standard programming languages. The source code for edge detection program consists of both the serial(CPU) and parallel(GPU) code.

This paper compares the performance differences between program code that are run on a sequential processor (CPU) and a parallel processor (GPU). Also it compares the results on different types of GPU machine (GeForce [3] and Tesla [4]) and on the basis of number of kernel function [2] [11] involved in each machine for computation.

The organization of this paper is as follows. In section 2, the Sobel edge detection operator is discussed, CUDA processing flow is described in section 3. In the section 4, CUDA thread hierarchy and CUDA kernel is introduced, detailed algorithm to be implemented is explained in section 5. In section 6, GPU programming in MATLAB is presented. Outcomes and results are shown in section 7. In section 8, graphs are analysed along with some observations. Finally, the conclusions are stated in section 9.

II. SOBEL EDGE DETECTION OPERATOR

Edge detection is a common image processing technique used in feature detection and extraction. Applying edge detection on an image can significantly reduce the amount of

data needed to be processed at a later phase while maintaining the important structure of the image. The idea is to remove everything from the image except the pixels that are part of an edge. These edges have special properties, such as corners, lines, curves, etc. A collection of these properties or features can be used to accomplish a bigger picture, such as image recognition. An edge can be identified by significant local changes of intensity in an image. An edge usually divides two different regions of an image. Most edge detection algorithms work best on an image that has the noise removal procedure already applied. A simple edge detection algorithm is to apply the Sobel edge detection algorithm. It involves convolving the image using an integer value filter, which is both simple and computationally inexpensive.
The Sobel operator is widely used in image processing, particularly within edge detection algorithms. The Sobel's operator find the approximate derivative in horizontal and vertical direction.

$$Gx = \{f(x+1,y-1)+2f(x+1,y)+f(x+1,y+1)\} - \{f(x-1,y-1)+2f(x-1,y)+f(x-1,y+1)\}$$

$$Gy = \{f(x-1,y+1)+2f(x,y+1)+f(x+1,y+1)\} - \{f(x-1,y-1)+2f(x,y-1)+f(x+1,y-1)\}$$

And the net gradient is : $g(x,y) = \sqrt{G_x^2 + G_y^2}$
Its convolution template operator as follows:

$$T_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad T_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Sobel operator is used to detect the edge of image M, then horizontal template Tx and vertical template Ty are used to convolute with the image without taking into account the border conditions. Then the total gradient value G may get by adding the two gradient matrices. This G is called as the gradient image. Finally, edge can be detected by applying threshold to gradient image [6].

III. CUDA PROCESSING FLOW

To make the GPU work for general purpose calculations, a certain processing flow is to be maintained which is as follows:

1) Copy input image arrays from CPU memory to GPU memory to load required data on device for computation.
2) Load GPU program and execute, caching data on chip for performance. The time required for evaluation of results on GPU is known as execution time.
3) Copy result image arrays back from GPU memory to CPU memory to further manipulate and display the results.

The above processing flow is depicted in figure 1 for more understanding:

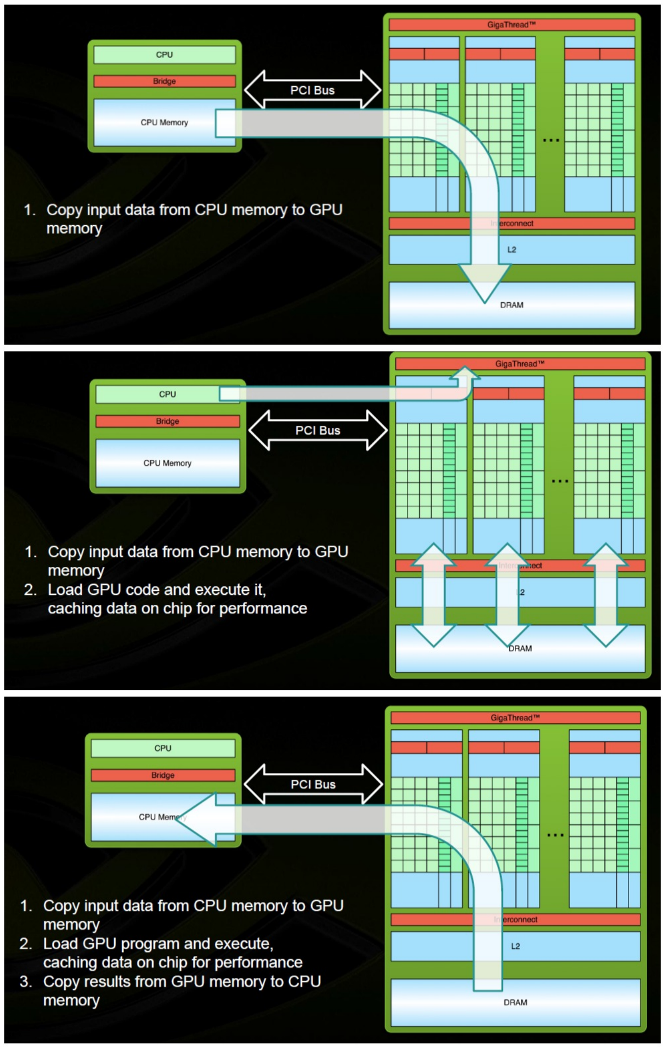

CPU
Bridge
CPU Memory
PCI Bus
GigaThread™
Interconnect
L2
DRAM
1. Copy input data from CPU memory to GPU memory
2. Load GPU code and execute it, caching data on chip for performance
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Figure 1. **CUDA processing flow [12]**

IV. CUDA THREAD HIERARCHY AND CUDA KERNEL

Threads on the device are automatically invoked when a kernel is being executed. The programmer determines the number of threads that best suits the given problem. The thread count along with the thread configurations are passed into the kernel. Figure 2 shows the entire collection of threads responsible for an execution of the kernel, called a grid. A grid is further partitioned and can consist of one or more thread blocks. A block is an array of concurrent threads that execute the same thread program. A thread block can be partitioned into one, two or three dimensions. All threads within a block can cooperate with each other. They can share data by reading and writing to shared memory, and they can synchronize their execution by using syncthreads().

The threading configuration is then passed to the kernel. Within the kernel, this information is stored as built-in variables. BlockDim holds the dimension information of the current block , BlockIdx and threadIdx provides the current block and thread index information. One limitation on blocks is that each block can hold up to 512 threads. Once a kernel is launched, the corresponding grid and block structure is created.

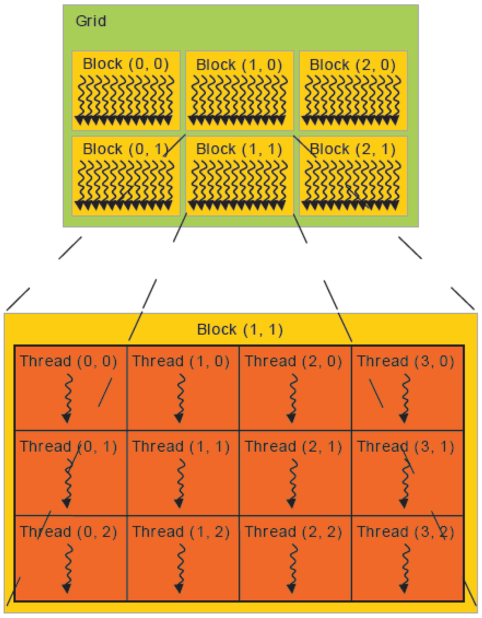


Figure 2. **Grid of thread blocks [7]**

V. DETAILED ALGORITHM

The algorithm is as follows:

1) Fetch the red(R(i,j)), green(G(i,j)) and blue(B(i,j)) components of the input colour image.
2) Go to step 3 directly if the image is already in grayscale format. Otherwise, convert the input image into the grayscale [8] image using the formula:

$$gray(i,j) = 0.2989*R(i,j)+0.5870*G(i,j)+0.1140*B(i,j)$$

3) Apply Sobel‘s operator in both the horizontal and vertical direction to calculate horizontal and vertical gradient.
4) Calculate net gradient and call that image as gradient image.
5) Finally take input from user through a slider depicting the different threshold values and apply on the gradient image.

NOTE: According to the need of the user, he can set the threshold or treat the gradient image with a predefined optimum threshold [6] value.
The above algorithm (flowchart in Figure 3) is implemented both, serially on CPU and parallely on GPU. In this paper, three sections of the algorithm have been identified for executing in parallel which are as follows:
(a) Fetching red, blue and green components from colour image.
(b) Conversion of Coloured (RGB) image to GRAYSCALE image.
(c) Applying Sobel‘s mask and calculation of gradient image.

This paper shows implementation of the algorithm in two ways: First by defining 2 kernels by parallelizing only points (b) and (c). Secondly, by defining 3 kernels by parallelizing all of the above points. Finally the results are compared regarding speedup on two machines GeForce and Tesla.
The decision about which section of the code to be parallelised should be taken wisely. As the parallelism in a code increases the speedup increases till a image size limit only, after that any more parallelism in the code results in the overall reduction of the speedup. The reason being parallelism involves huge transfer of the data from CPU to GPU for parallel execution on device side and then again from GPU to CPU to copy the results on the host side. Now as the image size increases, both execution time and communication time increases due to more pixel calculations but communication time i.e. overhead increases at a higher rate. Thus ratio of execution time and communication time decreases with increase in image resolution resulting in the reduction of the overall speedup. This paper clearly explains the above phenomenon and analyses the effect of parallelism versus image resolution with the help of graph in section 8.

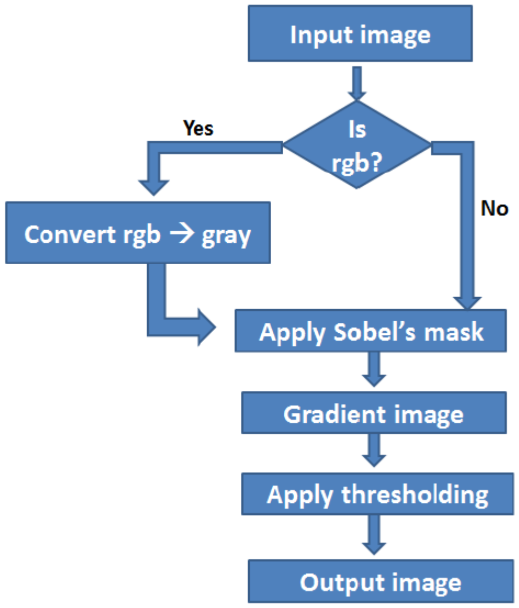


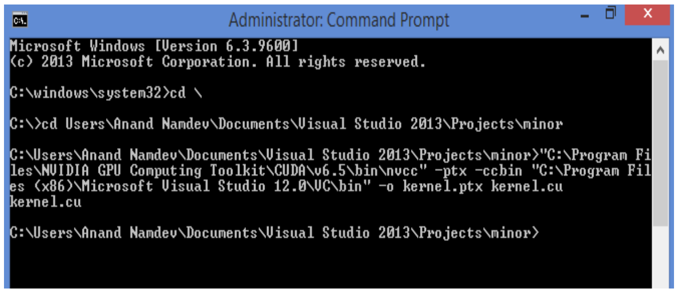
Figure 3. **Algorithmic Flow**

VI. GPU PROGRAMMING IN MATLAB

GPUs are increasingly applied to scientific calculations. Unlike a traditional CPU, which includes no more than a handful of cores, a GPU has a massively parallel array of integer and floating-point processors, as well as dedicated, high-speed memory. A typical GPU comprises hundreds of these smaller processors. The increased throughput made possible by a GPU comes at a cost. Firstly, for computations to be fast enough data must be sent from the CPU to the GPU before calculation and then retrieved from it afterwards. Because a GPU is attached to the host CPU via the Peripheral Component Interconnect (PCI) Express bus, the memory access is slower than with a traditional CPU. This means that our overall computational speedup is limited by the amount of data transfer that occurs in our algorithm. Secondly, programming for GPUs requires a different model and a skill set that can be difficult and time-consuming to acquire. Also, a lot of time is spent on managing and making our code work at these large numbers of threads.
Experienced programmers can write their own CUDA code and can use the CUDA Kernel interface in Parallel Computing Toolbox of MATLAB to integrate the CUDA code with MATLAB thereby creating a MATLAB object that provides access to the existing CUDA kernel which is already converted into PTX code (PTX is a low-level Parallel Threaded eXecution instruction set). They then invoke the feval command to evaluate the kernel on the GPU, using MATLAB arrays as input and output.

A. 64bit .ptx generation

The CUDA kernel code which is written separately cannot be directly run in MATLAB, hence it is converted to ptx code which is then executed from MATLAB commands by creating an object of the created ptx file.
This can be done by running CMD as administrator and typing in the required command. A screenshot has been given for proper visualization as shown in Figure 4.



```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\windows\system32>cd \

C:\>cd Users\Anand Namdev\Documents\Visual Studio 2013\Projects\minor

C:\Users\Anand Namdev\Documents\Visual Studio 2013\Projects\minor>"C:\Program Fi
les\NVIDIA GPU Computing Toolkit\CUDA\v6.5\bin\nvcc" -ptx -ccbin "C:\Program Fil
es (x86)\Microsoft Visual Studio 12.0\VC\bin" -o kernel.ptx kernel.cu
kernel.cu

C:\Users\Anand Namdev\Documents\Visual Studio 2013\Projects\minor>
```

Figure 4. **Command to generate ptx file [9]**

B. Evaluating the CUDA Kernel in MATLAB

To load the kernel into MATLAB, path is provided to the compiled PTX file and source code:

ptx_object = parallel.gpu.CUDAKernel('Kernel.ptx', 'Kernel.cu');

Once the ptx_object is created a few setup task must be completed before one can launch it, such as initializing return data and setting the sizes of the thread blocks and grid. The kernel can then be used just like any other MATLAB function, except that the kernel is launched using the feval command, with the following syntax:
output = feval(ptx_object, input_Arguments)[11]

VII. OUTCOMES AND RESULTS

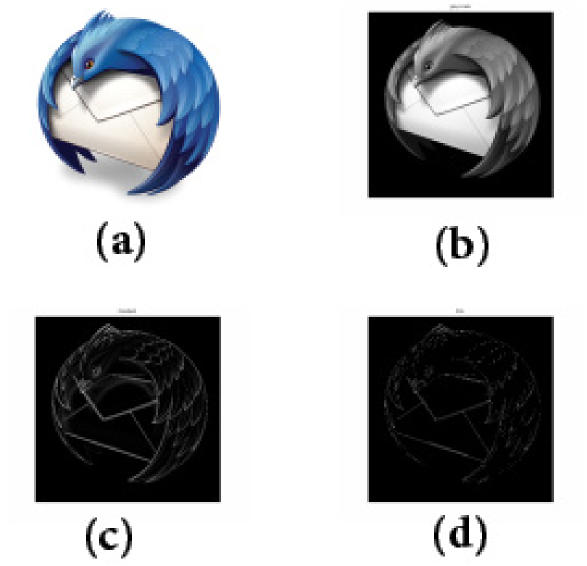


Figure 5. **The above four figures are the outcomes of the experiment done with parallel algorithm for Sobel's edge detection. (a) original image Thunderbird of size 1500x1500 pixels, (b) is greyscale converted image, (c) is gradient image and image (d) is final edge detected image.**

The abbreviations used in the below tables are stated as follows:
IR= Image Resolution in pixel^2
ST = Serial Time in seconds
PT(2k)= Parallel Time when two kernels were used in seconds
S(2k)= Speedup when two kernels were used
PT(3k)= Parallel Time when three kernels were used in seconds
S(3k)= Speedup when three kernels were used
Δ(2k,3k)= Percentage decrease in speedup

Table I
ALGORITHM RUNNING TIME AND SPEEDUP ON GEFORCE MACHINE (SLOW)

IR	**ST**	**PT (2k)**	**S (2k)**	**PT (3k)**	**S (3k)**	**Δ(2k,3k)**
1500x1500	59.50	0.35	168.36	0.74	79.83	52.58
1800x1800	86.73	0.45	191.79	1.06	81.74	57.38
2200x2200	137.18	0.70	194.43	1.49	91.81	52.77
3500x3500	425.67	1.69	250.74	3.74	113.63	54.68

Table II
ALGORITHM RUNNING TIME AND SPEEDUP ON TESLA MACHINE (FAST)

IR	**ST**	**PT (2k)**	**S (2k)**	**PT (3k)**	**S (3k)**	**Δ(2k,3k)**
1500x1500	59.50	0.08	705.32	0.17	338.52	52.04
1800x1800	86.73	0.11	725.56	0.25	342.14	52.84
2200x2200	137.18	0.17	771.71	0.36	372.25	51.76
3500x3500	425.67	0.45	943.88	0.93	454.16	51.88

Table 1 and Table 2 show the results of image datasets. The same experiment was performed on around 25 image dataset with their resolution ranging from 256x256 to 10000x10000. The experiment was performed on two machines GeForce and Tesla. The experiment results showed that on an average there was about 52% decrease in speedup when 3 kernels were used as compared to when 2 kernels were used. Hence it is not always required to impose parallelism unnecessarily as it comes more with a cost of communication time as explained above in the paper. For simulation environment, the hardware used is described below:

Table III
HARDWARE SPECIFICATIONS

Property	**CPU**	**GeForce**	**Tesla**
No. of cores	8	96	448
Memory capacity (GB)	8	2	8
Memory speed (GHz)	1.90	1.62	1.50

VIII. Graphs and Observations

Figure 6 shows the graph that has been plotted against image resolution (in pixel^2) and the speedup on both machines using 2 kernel functions.
From the graph in figure 6 it can be seen that as the image resolution increases, the speedup increases but the increase in the speedup is observed only up to a resolution value of 6500, that is any further increase in the size of the image results in the decrease of the speedup. This huge decrease in speedup was mainly due to the communication time i.e. it consists of first copying the data on the GPU side and then executing the data over device side. Hence as the size of the image increases the time of transferring the data over the GPU side i.e. communication time increase at a higher rate than the executing time which results in the reduction of the speedup.

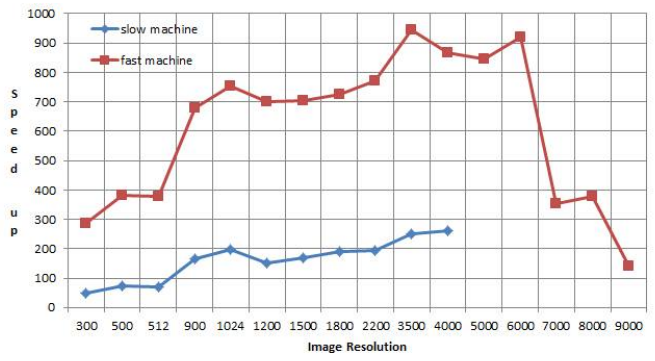


Figure 6. **Comparative study (2 kernels)**

Figure 7 shows the graph which has been drawn against test cases when GPU code consisted of 3 kernels. One extra kernel was added in order to produce maximum parallelism in the code and the results were then analysed.
From the graph it can be clearly observed that the speed up decreased in comparison to the earlier GPU code with 2 kernels, the reason being the increase in communication overhead due to that extra kernel which also involved transferring the whole data on the GPU side and then executing the kernel on the GPU side. This simply adds to the total time. The graph has been plotted upto a resolution of 7000 on the faster machine while upto a resolution of 5000 on the slower machine, as memory was insufficient to store the huge arrays of the image on the GPU side. The graph also shows that the speed up gradually decreases after a resolution of 6500 because of the extra added burden of the communication time of the 3 kernels.

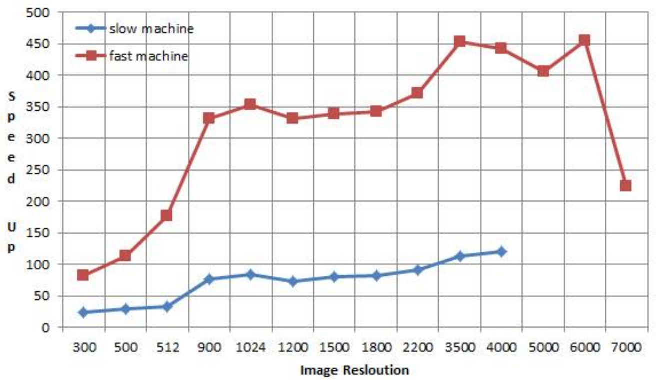


Figure 7. **Comparative study (3 kernels)**

IX. Conclusion

This project work now concludes that if parallelism is used optimally then one can achieve higher speedups, even 1000 times or 10000 times faster. The implementation contains both the sequential version and the parallel version. This

allows the reader to compare and contrast the performance differences between the two executions. The implementation used in this project achieves a maximum speedup of around 950 times, however if one tries to maximize parallelism where even not necessary then it results in reduced speedup due to much increase in communication time which is overhead for performance . This project explains the two separate speedups : one with (2 kernels) and the other with (3 kernels) and the observation comes out to be that speed up reduced by around 52% on both machines : one GeForce with 96 CUDA cores and the other Tesla with 448 CUDA cores. So one thing is clear that proper and efficient use of CUDA programming or parallelism can perform complex computations in much less time and provide you with higher speedups.
Future work will involve the edge detection of images with resolution greater than 10000x10000 with the help of Image Segmentation as GPU memory becomes insufficient to store large arrays for high resolution images. Image segmentation is the process of dividing an image into multiple sub images such that the result is a set of segments that collectively cover the entire image. Due to this segmentation of image into parts, instead of transferring whole array to GPU, calculation can be done in parts and the result can be merged which will help to evaluate results for more higher resolution images.

Acknowledgment

We express our sincere gratitude to our guide Dr. Meenu Chawla and thank her for her guidance and support in completing this paper.

References

[1] SOBEL, I., *An Isotropic 3x3 Gradient Operator,*. Machine Vision for Three – Dimensional Scenes, Freeman, H., Academic Pres, NY, 376-379, 1990.
[2] CUDA C Programming Guide PG-02829-001_v7.0 Page 9-10,2015
[3] http://www.geforce.com/hardware/notebook-gpus/geforce-gt-630m/specifications
[4] Michael Garland, *"Parallel Computing Experiences with CUDA"* in IPDPS 2010, pp. 13-27
[5] Jayshree Ghorpade, Jitendra Parande, Madhura Kulkarni, Amit Bawaskar, *"GPGPU Processing in CUDA Architecture"* in Advanced Computing: An International Journal (ACIJ), Vol.3, No.1, January 2012, pp 105-120
[6] Jin-Yu., Yan. ,Xiang., *"Edge Detection of Images Based on Improved Sobel Operator and Genetic Algorithms"*, IEEE International Conference on Image Analysis and Signal Processing, IASP 2009,April 2009,pp. 31-35
[7] CUDA C Programming Guide PG-02829-001_v7.5 Page 11,2015
[8] R. C. Gonzalez and R.E. Woods, *Digital Image Processing,* Prentice Hall, New Jersey, 2008.
[9] Generating CUDA ptx files from Visual Studio [online] Available: http://stackoverflow.com/questions/13426170/convert-cu-file-to-ptx-file-in-windows?rq=1
[10] Tinku Acharya and Ajay K. Ray,*Image Processing Principles and Applications,* John Wiley & Sons, New Jersey 2005
[11] Cliff Woolley,*CUDA Overview*,NVIDIA Developer Technology Group, pp. 20-30
[12] Cyril Zeller,*CUDA C/C++ Basics,*NVIDIA Corporation, Supercomputing Tutorial,pp. 9-11, 2011