

Scheduled System Maintenance: On Tuesday, April 6, IEEE Xplore will undergo scheduled maintenance from 1:00-3:00 PM ET. During this time, there may be intermittent impact on performance. We apologize for any inconvenience.

Contents

IEEE.orgIEEE XploreIEEE-SAIEEE SpectrumMore Sites

DownloadPDF

CartCreate AccountPersonal Sign In

Access provided by:  
NANJING UNIVERSITY OF  
AERONAUTICS AND  
ASTRONAUTICS

Sign Out

BrowseMy SettingsHelp

Access provided by:  
NANJING UNIVERSITY OF  
AERONAUTICS AND  
ASTRONAUTICS

Sign Out

All

Q

ADVANCED SEARCH

Conferences > 2016 IEEE Students' Conferenc... ?

Back to Results

Parallel edge detection by SOBEL algorithm using CUDA C

Publisher: IEEE

Cite ThisCite This

PDF

<< Results

Adhir Jain ; Anand Namdev ; Meenu Chawla All Authors

1 Paper Citation

605 Full Text Views

Collabratec

Alerts

ManageContent AlertsAdd to CitationAlerts

More Like This

Parallel Algorithm on Graphics Processing Unit for Harmonic Minimization in Multilevel Inverters  
IEEE Transactions on Industrial Informatics  
Published: 2015

Parallel Programming with GPUs: Parallel Programming Using Graphics Processing Units with Numerical Examples for Microwave Engineering  
IEEE Microwave Magazine  
Published: 2013

Show More

Abstract	<b>Abstract:</b> 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 p... <b>View more</b>
Authors	
Figures	<b>Metadata</b>
References	<b>Abstract:</b> 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 parallelly. 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
Citations	
Keywords	
Metrics	
More Like This	

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.

**Published in:** 2016 IEEE Students' Conference on Electrical, Electronics and Computer Science (SCEECS)

**Date of Conference:** 5-6 March 2016      **INSPEC Accession Number:** 16143127

**Date Added to IEEE Xplore:** 14 July 2016      **DOI:** 10.1109/SCEECS.2016.7509360

**ISBN Information:**

**Publisher:** IEEE

**Conference Location:** Bhopal, India

## SECTION 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.

 **Contents**

## SECTION 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.

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

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

[View Source](#) 

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

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

[View Source](#) 

Sobel operator is used to detect the edge of image M, then horizontal template  $T_x$  and vertical template  $T_y$  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].

## SECTION 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:

 Contents

Download

PDF

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:

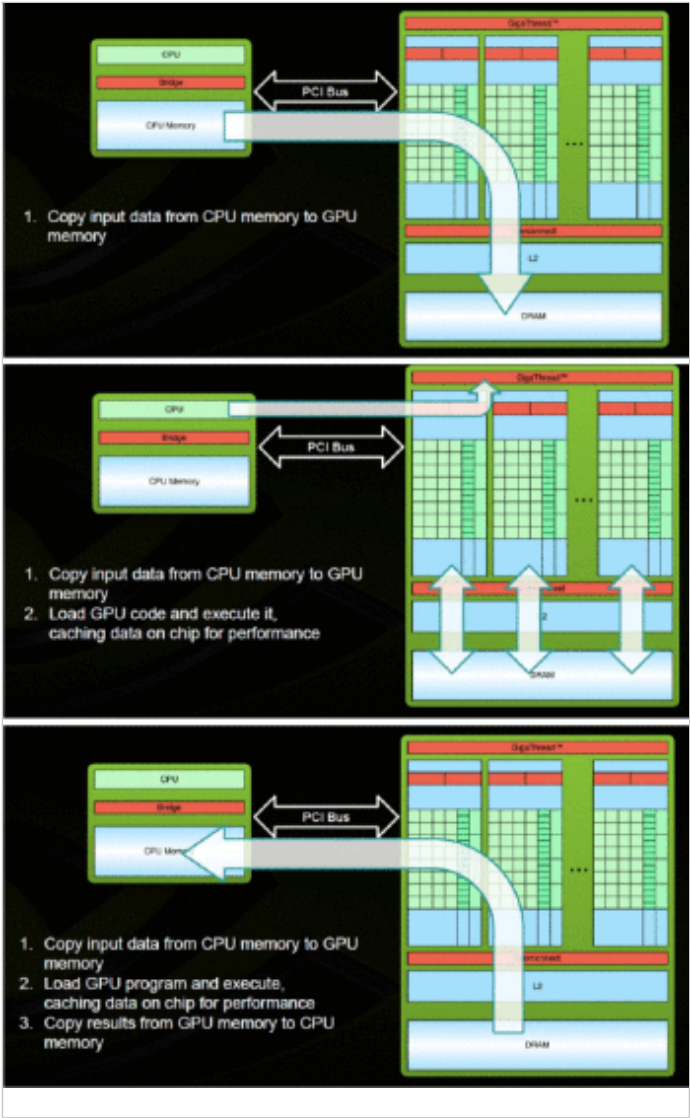


Figure 1.  
CUDA processing flow [12]

SECTION 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.

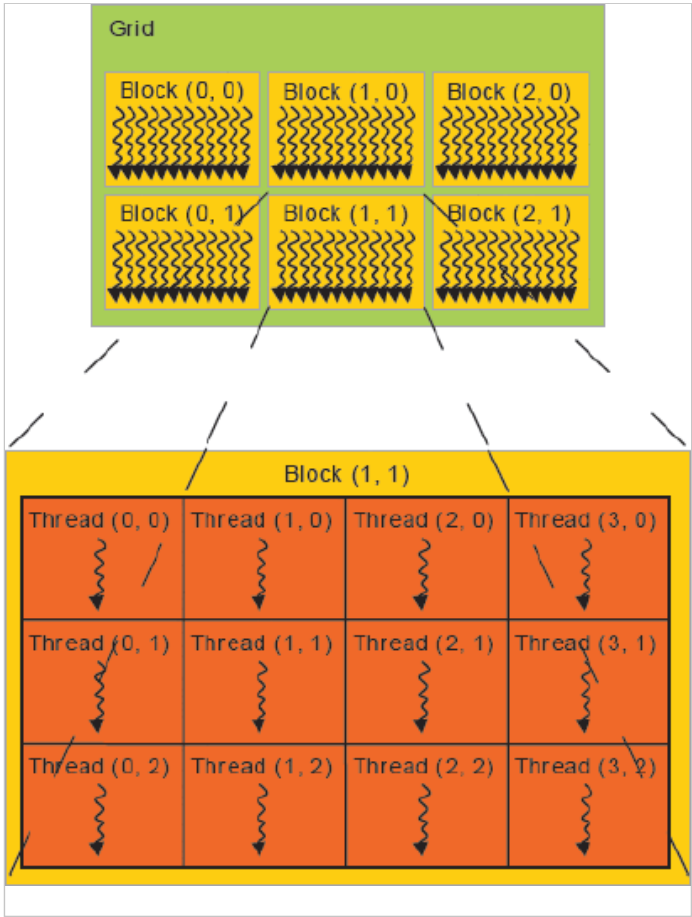


Figure 2. Grid of thread blocks [7]

SECTION 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.

Download  
PDF Contents

- 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:

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

[View Source](#) 

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

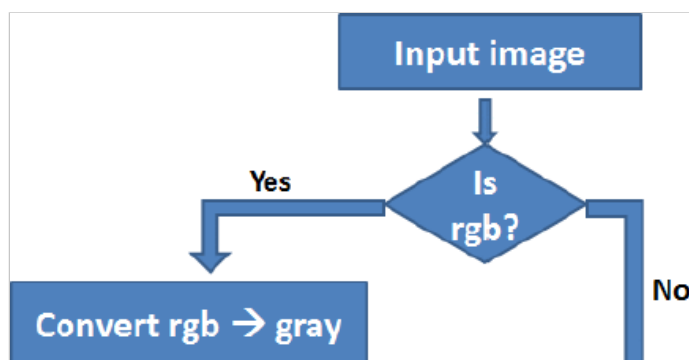
NOTE: According to the need of the user he can set the threshold or treat the gradient image with a predefined optimum threshold [6] value.

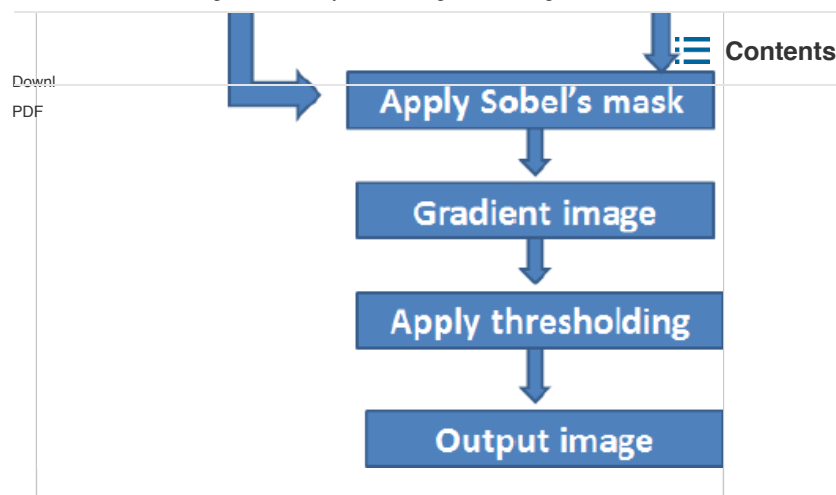
The above algorithm (flowchart in Figure 3) is implemented both, serially on CPU and parallelly on GPU. In this paper, three sections of the algorithm have been identified for executing in parallel which are as follows:

- Fetching red, blue and green components from colour image.
- Conversion of Coloured (RGB) image to GRAYSCALE image.
- 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

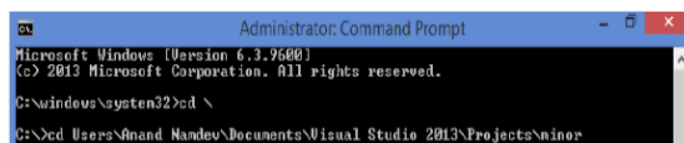
## SECTION VI. GPU Programming in MATLAB

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.

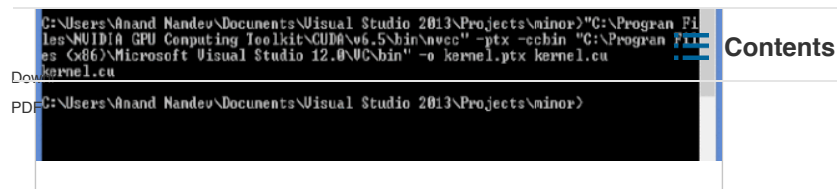
Experienced programmers can write their own CUDA code and can use the CUDA Kernel interface in Parallel Computing Toolbox of MATLAB to integrate the CUDA code with MATLAB thereby creating a MATLAB object that provides access to the existing CUDA kernel which is already converted into PTX code (PTX is 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.







**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');
```

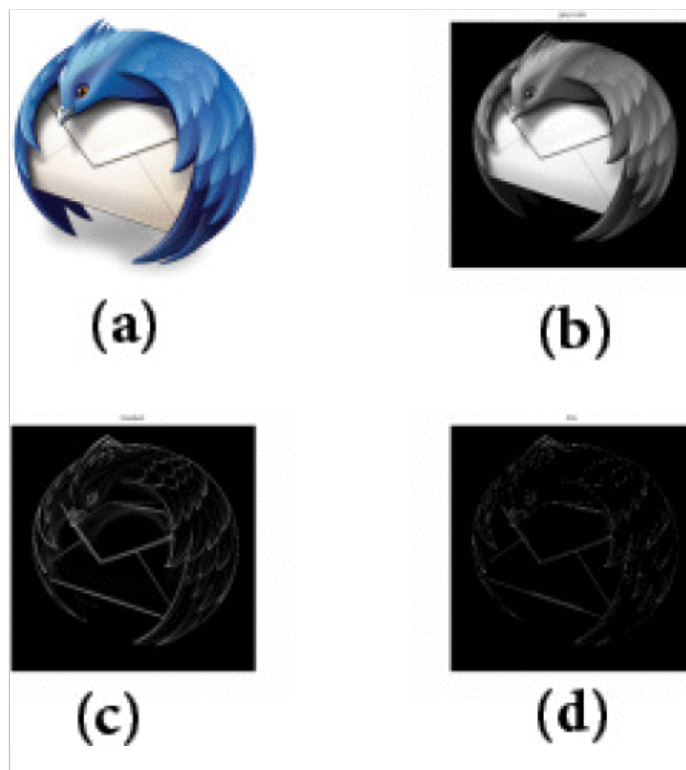
[View Source](#) 

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]
```

## SECTION 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  $1500 \times 1500$  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:

Down!

PDF

Image Resolution in pixel<sup>2</sup>

Serial Time in seconds

Parallel Time when two kernels were used in seconds

Speedup when two kernels were used

Parallel Time when three kernels were used in seconds

Speedup when three kernels were used

$\Delta(2k, 3k)$  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)	$\Delta(2k,3k)$
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)	$\Delta(2k,3k)$
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  $256 \times 256$  to  $10000 \times 10000$ . 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

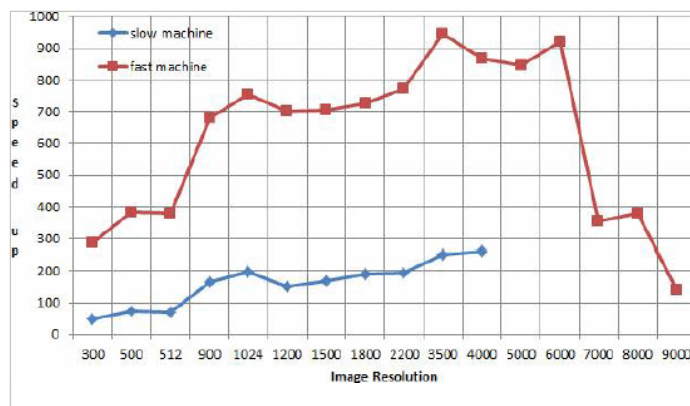
Property	CPU	GeForce	Tesla
No. of cores	8	96	448
Memory capacity (GB)	8	2	8
Memory speed (GHz)	1.90	1.62	1.50

## SECTION VIII.

### Graphs and Observations

Figure 6 shows the graph that has been plotted against image resolution (in pixel<sup>2</sup>) and the speedup on both machines using 2 kernel functions.

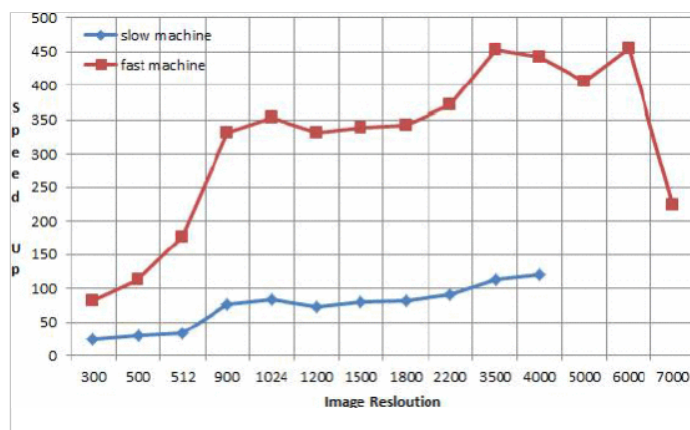
From the graph in figure 6 it can be seen that as the image resolution increases, the speedup increases but the increase in the speedup is observed only up to a resolution value of 6500, that is any further increase in the size of the image results in the decrease of the speedup. This huge decrease in 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)

## SECTION IX.

## Contents

Download  
PDF

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  $10000 \times 10000$  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.

---

Authors




---

Figures




---

References




---

Citations




---

Keywords




---

Metrics

---



IEEE Personal Account

CHANGE USERNAME/PASSWORD

Purchase Details

PAYMENT OPTIONS

VIEW PURCHASED DOCUMENTS

Profile Information

COMMUNICATIONS PREFERENCES

PROFESSION AND EDUCATION

TECHNICAL INTERESTS

Need Help?

US &amp; CANADA: +1 800 678 4333

WORLDWIDE: +1 732 981 0060

CONTACT &amp; SUPPORT

Follow



About IEEE Xplore | Contact Us | Help | Accessibility | Terms of Use | Nondiscrimination Policy | Sitemap | Privacy & Opting Out of Cookies

A not-for-profit organization, IEEE is the world's largest technical professional organization dedicated to advancing technology for the benefit of humanity.

Download PDF

© Copyright 2021 IEEE - All rights reserved. Use of this web site signifies your agreement to the terms and conditions.

IEEE Account	Purchase Details	Profile Information	Need Help?
<div>» Change Username/Password</div> <div>» Update Address</div>	<div>» Payment Options</div> <div>» Order History</div> <div>» View Purchased Documents</div>	<div>» Communications Preferences</div> <div>» Profession and Education</div> <div>» Technical Interests</div>	<div>» <b>US &amp; Canada:</b> +1 800 678 4333</div> <div>» <b>Worldwide:</b> +1 732 981 0060</div> <div>» Contact &amp; Support</div>

About IEEE Xplore | Contact Us | Help | Accessibility | Terms of Use | Nondiscrimination Policy | Sitemap | Privacy & Opting Out of Cookies

A not-for-profit organization, IEEE is the world's largest technical professional organization dedicated to advancing technology for the benefit of humanity.

© Copyright 2021 IEEE - All rights reserved. Use of this web site signifies your agreement to the terms and conditions.