Parallel Implementation of Scheduling Algorithms on GPU using CUDA

Nipun Agarwal
Department of CSE
College of Engineering
Roorkee, Roorkee-247667,
Uttarakhand, India

Aman Goyal
Department of CSE
College of Engineering
Roorkee, Roorkee-247667,
Uttarakhand, India

Gaurav Maheshwari,
Alok Dugtal
Department of CSE
College of Engineering
Roorkee, Roorkee-247667,
Uttarakhand, India

ABSTRACT
The future of computation is the GPU, i.e. the Graphical Processing Unit. The graphics cards have shown the tremendous power in the field of image processing and accelerated generating of 3D scenes, and the computational capability of GPUs have promised its developing into great parallel computing units. It is quite simple to program a graphical processor to perform many parallel tasks. But after understanding the various aspects of the graphical processor, it can be used to perform other useful tasks as well. This paper shows how CUDA can fully utilize the tremendous power of these GPUs. CUDA is NVIDIA's parallel computing architecture which enables terrible increase in computing performance, by gearing the power of the GPU. In the first phase, several operating system algorithms in single threaded CPU environment are implemented using C language, then the same algorithms are implemented on CUDA and CUDA enabled GPU in a parallel environment and finally comparison of their performance and results to their implementation in GPU and CPU are shown.

General Terms
GPU, GPGPU, Parallelization, Multicore

Keywords
CUDA, Scheduling Algorithms, FCFS, SJF, RR, PBS

1. INTRODUCTION
GPU computation has provided a huge bound over the CPU with respect to the computational speed. Hence it is one of the most interesting areas of modern computational research and development. GPU is a graphical processing unit which primarily enables us to run high definition graphics on our PC, which are the essential demand of modern computing world. The main job of the GPU is to compute 3D functions. As these types of calculations are very difficult to perform on the CPU, the GPU can help them to run more efficiently. Though, GPU is introduced for graphical purposes, it has now evolved into computing, accuracy and performance. Using the GPU for processing non graphical objects is known as the General Purpose GPU or GPGPU, whch is used for performing very complex mathematical operations in parallel to achieve low time complexity. The arithmetic power of the GPGPU is a result of its highly specialized computing architecture. [21][22]

This paper proposes the use of CUDA technology on parallel platform to enhance the performance of the operating system scheduling algorithms: First Come First Serve (FCFS) algorithm, Shortest Job First (SJF) algorithm, Round-Robin (RR) algorithm and Priority based scheduling (PBS) algorithm. Scheduling is the process by which threads, processes and data flows are given access to system resources (e.g. processor time and communications bandwidth). This is usually done to maintain balance and share the system resources effectively and achieve a target quality of service. The need for these scheduling algorithm arises from the requirement for most modern systems to perform multitasking (executing more than one process at a time) and multiplexing (send multiple data streams simultaneously across a single physical channel) . The operating system algorithms form the core for resource allocation and their maximum utilization. Scheduling is a complex job which may require an extensive processing and thus is better performed on a parallel processor. The paper demonstrates the performance of scheduling algorithms as to check how fast the algorithm could perform. [7-11] It compares and contrast the working time and corresponding efficiency of normal execution of the serial code implemented in C language on single threaded CPU as compared to GPU implementation. The CUDA implementation of the scheduling algorithms uses the CUDA-C language and the recent NVIDIA CUDA Software Development Kit (SDK) 6.0.

2. GPU COMPUTING
CUDA (Compute Unified Device Architecture) is NVIDIA's manufacturing GPU architecture featured in the graphic cards which has positioned itself as a new means for general purpose computing. CUDA C or C++ is an extension of the C or C++ programming languages for general purpose computing. For running multi threaded applications, there is no need of streaming computing in GPU because the cores can communicate and exchange information with each other. CUDA is only well fitted and useful for highly parallel algorithms. There is a need to have many threads in order to increase the performance of the algorithms while running on GPU.[12] Normally more the number of threads, better is the performance. CUDA can take full advantage of when writing in language C. The primary idea of CUDA is to have thousands of threads which are executing in parallel. All these threads usually execute the very same function or code, known as a kernel. All these threads are executed using the same instructions and different data. Each thread knows its own ID, and based on its own ID, it determines which pieces of data to work on.[3]

A CUDA program consists of some phases that are executed on either the host (CPU) or a device such as a GPU. In the host code, no data parallelism phase is carried out. In some cases, little data parallelism is carried out in host code. In device code, phases which has high amount of data

parallelism are carried out. A CUDA program is a unified source code comprising both, host and device code. The host code is straight forward code in C language, compiled with the help of standard C compiler only. That is what can be said as an ordinary CPU process. The device code is written using CUDA keywords for data-parallel functions, called kernels, and their associated data structures.[13-15] In some cases, kernels can be executed on CPU if there is no GPU device available but this facility is provided with the help of emulations features. The CUDA software development kit provides those features.

CUDA Architecture comprises of three essential parts, which help the programmer to utilize effectively the full computational capability of the graphics card on the respective system. CUDA Architecture splits the GPU device into grids, blocks and threads in a hierarchical structure as shown in fig. 1. Since there are a number of threads in a block and a number of blocks in a grid and a number of grids in a single GPU, the parallelism that is achieved using such a hierarchical architecture is very huge.[23][24]

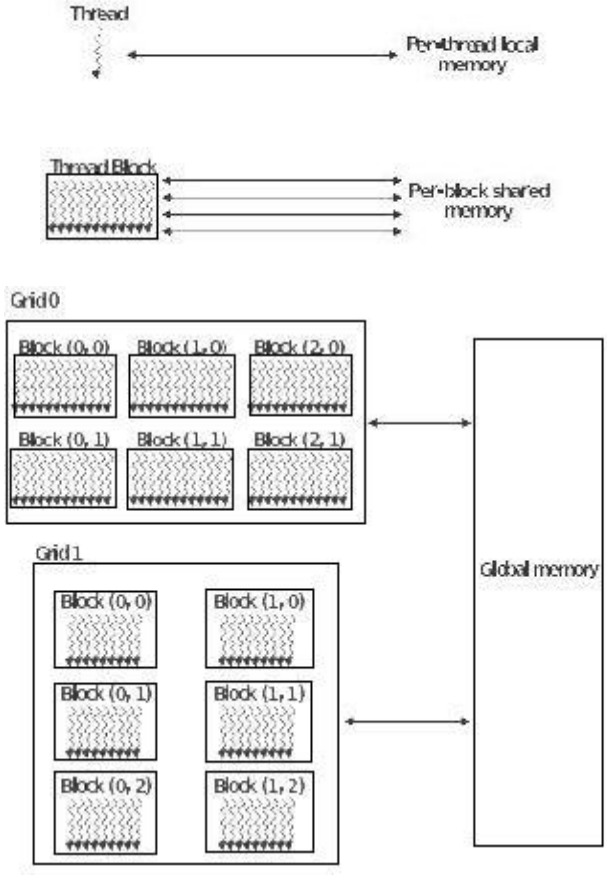


Fig 1: **Cuda Program Structure and memory hierarchy**

A grid is a group of many threads which are running the same kernel. These threads are not synchronized. Every call to CUDA from the CPU is made through one grid. On multi-GPU systems, grids cannot be shared between different GPUs because they use many grids for maximum efficiency. Grids are composed of many blocks. Each block is a logical unit containing a number of co-coordinating threads and a certain amount of shared memory. Blocks too are not shared between multiprocessors. Every block in a grid use the same program. A built in variable "blockIdx" can be used to identify the current block. Blocks are themselves composed of many threads that run on the individual cores of the multiprocessors, but unlike grids and blocks, they are not restricted to a single core. Like blocks, each thread has an own ID called "threadIdx". Thread IDs can be 1D, 2D or 3D based on the block dimensions. The thread ID is relative to the block that contains the thread. [4-6]

3. SCHEDULING ALGORITHMS

Process scheduling is handled by process manager which oversees the removal of the running process from the CPU and also selects some other process based on any particular strategy. Process scheduling is one of the most essential things in multiprogramming operating systems. Using these systems, one can load more than one process at the same time into the executable memory and the processes which are loaded shares the CPU among them.

Some terms associated with scheduling algorithms are:
Turnaround time is the elapsed time between the time the job enters and the time that it terminates. This may include the delay of waiting for the scheduler to start the job because some other process is still running and others may be queued ahead.

Turnaround Time=Waiting Time + Service Time

Start time is the time when the task which is scheduled to run and actually gets to start running on the CPU. CPU bursts starts with this.

Response time is the delay that occurs between submitting a process and it being scheduled to run. Thus, it is the delay between a task being ready to run and when it actually starts running.

Throughput refers to the number of processes that are completed within some amount of time. One can compare the throughput of several schedulers and get to know as to which scheduler processes more task as compared to other schedulers. [16-18]

In the following section, we will explore a few scheduling algorithms.

First-come First-served Scheduling (FCFS) follow first in first out strategy. As the process becomes ready, it joins the ready queue. When the current running process ceases to execute, the oldest process which is present in the Ready queue is selected for running after it. That is first entered process among the available processes in the ready queue. The average waiting time for FCFS is often long as compared to other algorithms. It is non-preemptive scheduling. [19]

Shortest Job First Scheduling (SJF) associates with each process the length of the next CPU burst. The process which is expected to take the least processing time is selected for execution, among the available processes in the ready queue. If the next CPU bursts of two processes are the same then FCFS scheduling is used to break the tie. SJF can be preemptive or non-preemptive. [19][20]

In Priority Based Scheduling (PBS), priority (an integer) is associated with each and every process. The CPU is allocated to the process with the highest priority amongst all. Generally process with the smallest integer is given the highest priority. Equal priority processes are scheduled in First Come First

Serve fashion. It can be preemptive or Non-preemptive. [19][20]

Round-Robin Scheduling (RR) algorithm is basically designed for time sharing systems. It is somewhat similar to FCFS with preemption added. Round-Robin Scheduling is also called as time-slicing scheduling and it is a preemptive version with a clock interrupt generated at periodic intervals. If and when the interrupt occurs, the process running is placed in the ready queue and the next ready job is selected on a First-come First-serve basis. This operation is known as time-slicing, because each process is given a slice of time before being preempted. [19][20]

4. IMPLEMENTATION

As stated earlier the main objective of the present work is to analyze the various CPU scheduling algorithms. The foremost criterion for evaluating CPU scheduling is the waiting time and burst time of the processes that are under same set of conditions. The paper has implemented the four major scheduling algorithms namely First Come First Serve (FCFS) Scheduling, Shortest-Job-First (SJF) Scheduling, Priority Based Scheduling (PBS) and Round Robin (RR) Scheduling firstly on single threaded CPU environment and calculated the execution time of each algorithm, then, the same algorithms are implemented with NVIDIA's GPU programming environment, CUDA v6.0. It then compares the execution time of the algorithms on both platform and calculates the speed up achieved in execution of the algorithms on GPU over CPU. The basics of CUDA code implementation are:

a. Allocate the memory on the CPU.

b. Allocate the same amount of memory on GPU using library function "CudaMalloc".

c. Input the data in memory allocated in CPU.

d. Copy data from CPU to GPU memory using another library function CudaMemCpy having parameter (CudaMemcpyHostToDevice) to give the direction of the copy.

e. Processing in now performed in GPU memory using kernal calls. Kernel calls transfer the control from CPU to GPU and also specify the number of grids, blocks and threads required for your program. In other words it defines the parallelism

f. Copy back the final data from GPU to CPU memory using library function CudaMemCpy having parameter (CudaMemcpyHostToDevice)

g. Release the GPU memory or other threads using the library function CudaFree.[9]

Setting up the environment and writing programs in CUDA is a fairly simple task. But, it requires that the one must have a thorough knowledge of the architecture and knowledge of writing parallel codes. The most important part of programming in CUDA is the kernel calls wherein the programmer must determine the parallelism that is required by the program. The division of given data into appropriate number of threads is the major area which defines a successful code.[24]

5. RESULTS AND DISCUSSIONS

In order to analyze the performance of the implemented algorithms the speedup achieved on the execution with respect to time was evaluated for all the test results. All the tests on the algorithms were performed with the similar number of processing nodes or processors and therefore, the speedup in execution is not evaluated based on the number of processors used but by analyzing the speedup in execution time because of the change in parallelizing approach taken up in the program.[1-2]

In general, the formula to calculate speedup using execution time for same number of processors is given by:

S = Ts/Tp

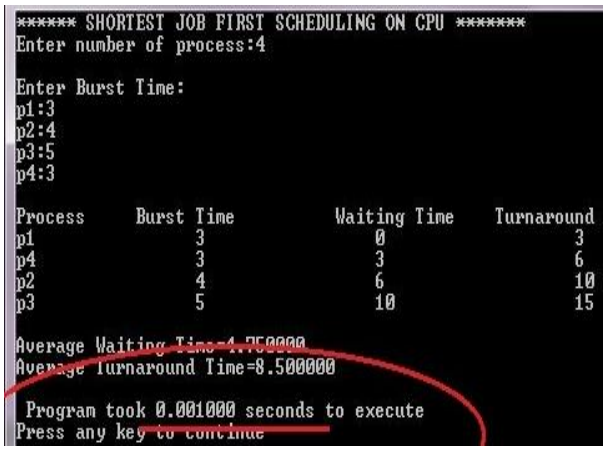
Where, Ts is the time it takes for execution of sequential algorithm and Tp is the it times for execution of parallel algorithm.[1-2]
We have achieved the speedup of around 10-13x over a single-threaded CPU implementation when implemented on GPU. We tested the codes by running them on CPU and GPU respectively. Our test data contain the images shown in fig. 2(a-h)

```
***** FCFS ( FIRST COME FIRST SERVE ) SCHEDULING *****

Enter the number of process 3

Do you need to enter the arrival time of process [y/n]
n
Enter the 1 process burst time : 3
Enter the 2 process burst time : 4
Enter the 3 process burst time : 3

PROCESS BURST TIME      ARRIVAL TIME6
    1         3       0
    2         4       0
    3         3       0

Total Waiting time = 10.000000
Average Waiting time = 10.000000

Total Turn Around Time(TAT) = 20.000000
Average Turn Around Time Avg.(TAT) = 6.666667
 Program took 0.001000 seconds to execute
```

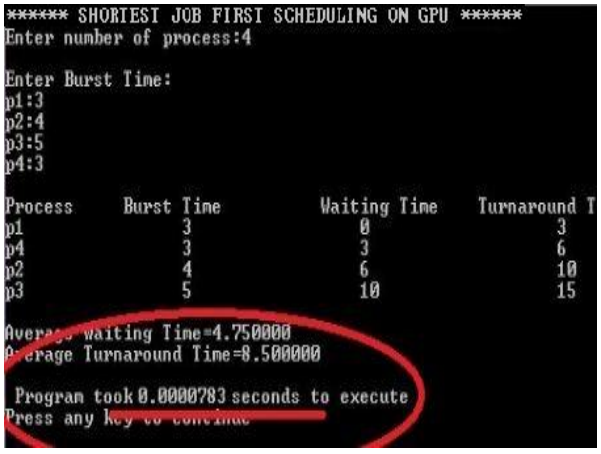
(a) FCFS on CPU

```
***** FCFS ( FIRST COME FIRST SERVE ) SCHEDULING ON  CUD

Enter the number of process 3

Do you need to enter the arrival time of process [y/n] o
3
Enter the 1 process burst time : 4
Enter the 2 process burst time : 3
Enter the 3 process burst time : 3

PROCESS BURST TIME      ARRIVAL TIME6
    1         4       0
    2         3       0
    3         3       0

Total Waiting time = 11.000000
Average Waiting time = 11.000000

Total Turn Around Time(TAT) = 21.000000
Average Turn Around Time Avg.(TAT) = 7.000000
 Program took 0.0000972 seconds to execute
```

(b) FCFS on GPU

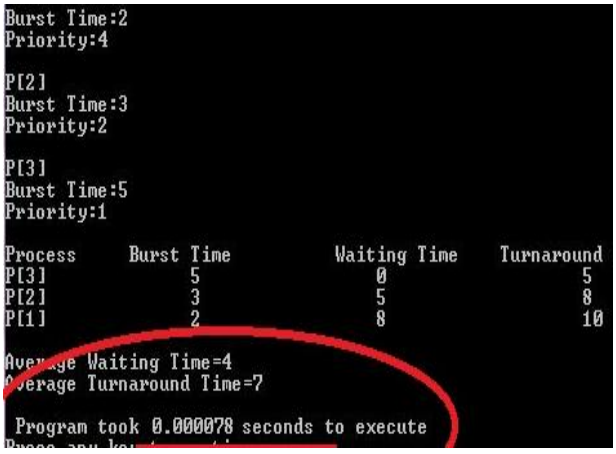


(c) SJF on CPU

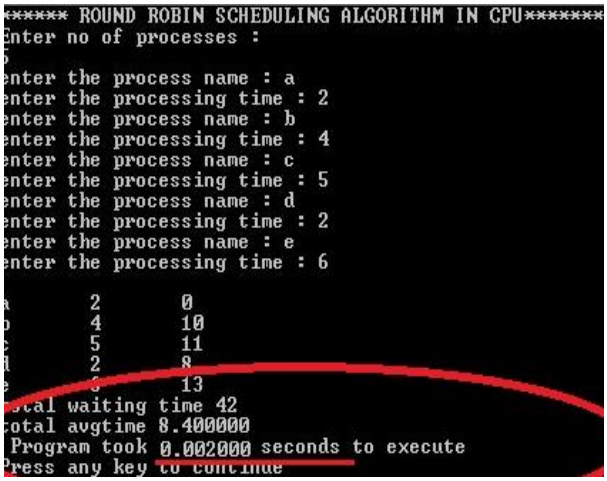


(d) SJF on GPU

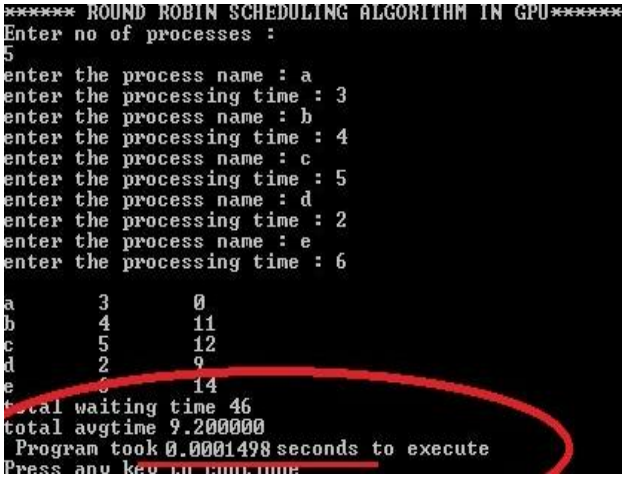
(e) PBS on CPU



(f) PBS on GPU



(g) Round Robin on CPU



(h) Round Robin on GPU

Fig 2- (a-h). Execution time observed for implementation of algorithms on CPU and GPU

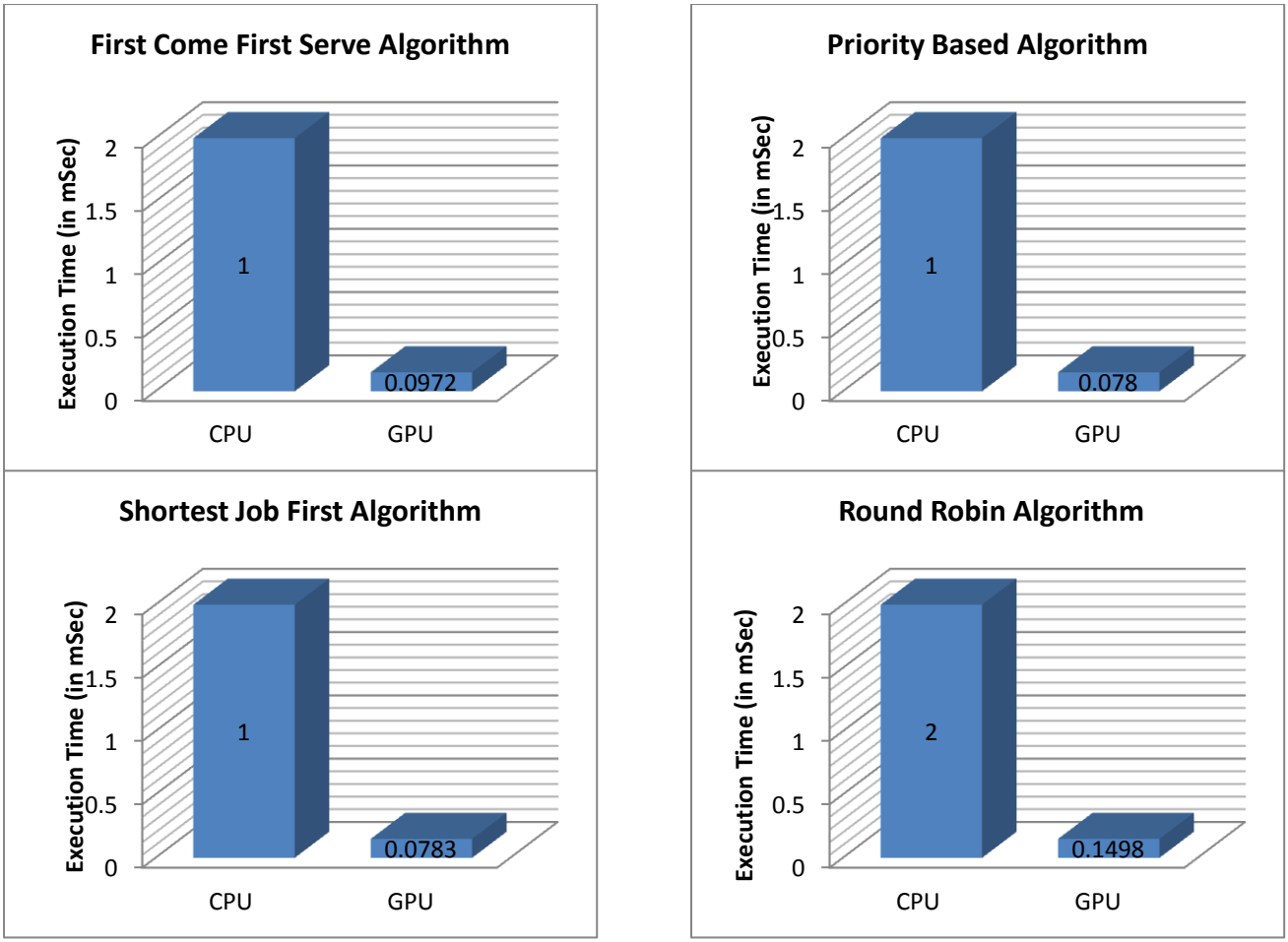


Fig 3: Comparative graphs of execution time observed (in milliseconds)

Table 1 contains the data about the execution time as observed for the scheduling algorithms when executed on CPU and GPU respectively. The time complexity is improved to a good extent on GPU due to parallelized approach.

Table 1: Execution time obtained for various algorithms

Algorithm Used	CPU Time (in ms)	GPU Time (in ms)
First Come First Serve	1.0	0.0972
Shortest Job First	1.0	0.0783
Priority Based	1.0	0.0780
Round Robin	2.0	0.1498

Overall comparison of speed up achieved in different operating system scheduling algorithms as compared with execution time on CPU is represented in the Fig 4.

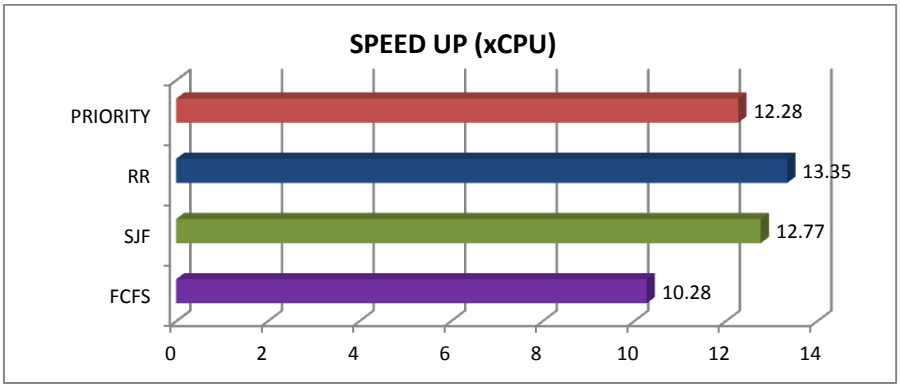


Fig 4: Speedup achieved on various algorithms

6. CONCLUSION

This paper introduces a GPU implementation of the operating system scheduling algorithms. The algorithms were designed for the NVIDIA CUDA platform to work in parallel with many threads in execution. It implements the operations of calculating the waiting time and turnaround time on the GPU and it uses the latest features of the NVIDIA CUDA SDK 6.0 on Geforce 740m processor. With the help of the GPU the execution time of various algorithms was found to be 10.28-13.35x faster than on the CPU using C code. This is a significant enhancement in the performance of the algorithms. GPU has shown tremendous potential and a further performance increase is expected with better optimization and more advanced GPUs.

7. REFERENCES

[1] Shuai C., Michael B., Jiayuan M., David T., Jeremy W. S., Kevin S., Performance Study of General-Purpose Applications on Graphics Processors Using CUDA

[2] Maria Andreina F. Rodriguez, "CUDA: Speeding Up Parallel Computing".

[3] Wikipedia- "http://en.wikipedia.org/wiki/CUDA"

[4] Anthony Lippert – "NVIDIA GPU Architecture for General Purpose Computing"

[5] David Kirk/NVIDIA and Wen-mei Hwu, 2006-2008 – "CUDA Threads"

[6] Yadav K., Mittal A., Ansari M. A., Vishwarup V., "Parallel Implementation of Similarity Measures on GPU Architecture using CUDA"

[7] Direct Compute Programming Guide (http://developer.download.NVIDIA.com/compute/DevZone/docs/html/DirectCompute/doc/DirectCompute_Programming_Guide.pdf)

[8] Singh B.M., Mittal A., Ghosh D., Parallel Implementation of Niblack's Binarization Approach on CUDA.

[9] Peter Zalutaski "CUDA – Supercomputing for masses."

[10] Practical Applications for CUDA (http://supercomputingblog.com/cuda/practical-applicationsfor-cuda/)

[11] Matthew Guidry, Charles McClendon, "Parallel Programming with CUDA".

[12] NVIDIA Corporation. NVIDIA CUDA Compute Unified Device Architecture Programming Guide, June 2008.

[13] Danilo De Donno et al., "Introduction to GPU Computing and CUDA Programming: A Case Study on FDTD," IEEE Antennas and Propagation Magazine, June 2010.

[14] Practical Applications for CUDA http://supercomputingblog.com/cuda/practical-applications-for-cuda

[15] GPU Gems 2, Chapter 35. GPU Program Optimization http://http.developer.NVIDIA.com/GPUGems2/gpugems_chapter35.html

[16] Process Scheduling. Available online: https://www.cs.rutgers.edu/~pxk/416/notes/07-scheduling.html 2003-2015.

[17] CPU Scheduling. Available online: http://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/5_CPU_Scheduling.html

[18] Types of Scheduling. Available online: http://www.go4expert.com/articles/types-of-scheduling-t22307/

[19] Alexandra Fedorova. "Operating System Scheduling for Chip Multithreaded Processors", https://www.cs.sfu.ca/~fedorova/thesis

[20] Daniel Alexander Taranovsky, CPU "Scheduling in Multimedia Operating Systems",1999

[21] David Tarditi, Sidd Puri, Jose Oglesby, "Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses", October 2006

[22] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Kevin Skadron, "A Performance Study of General-Purpose Applications on Graphics Processors Using-CUDA"

[23] Manish Arora, "The Architecture and Evolution of CPU-GPU Systems for General Purpose-Computing".

[24] Jayshree Ghorpade , Jitendra Parande , Madhura Kulkarni , Amit Bawaskar, "GPGPU PROCESSING IN CUDA ARCHITECTURE" Advanced Computing: An International Journal (ACIJ), Vol.3, No.1, January 2012