International Journal of Emerging Research in Management &Technology
ISSN: 2278-9359 (Volume-4, Issue-6)

Research Article

June 2015

A Study of GPU and CUDA for SIMD Concept

Himanshu Nayak[1], Rajesh Tiwari[2], Dr. Manisha Sharma[3], Dr. Kamal K. Mehta[4]
[1, 2] CSE, SSTC, CSVTU, Bhilai, Chhattisgarh, India
[3] Prof. Bit, Durg, CSVTU, Chhattisgarh, India
[4] Prof. Nirma Unive. Ahmedabad, Gujarat, India

***Abstract*—**

In this era time is very crucial, and when it comes to the digital world time plays a vital role. Today every field in computer science has a huge amount of data, and we need to process them to get valuable information out of it. To make the processing fast and to utilize the processor's capability to its maximum, parallel processing was introduced. With the evolution in processor architectures the parallel processing has also evolved. Now we have multi core systems known as GPUs and using which the degree of parallelism has gone to its highest. This paper Presents Parallel Processing for performing fast execution of a program using NVIDIA GPU Device on CUDA Platform. A comprehensive study of multiple techniques for comparison of GPU system and non-GPU system performances is given.

***Keywords*— Compute Unified Device Architecture, Graphical Processing Unit, Parallel Processing Languages.**

I. INTRODUCTION

Parallel processing divides a bigger task into multiple smaller tasks and executes these tasks on multiple processors concurrently. The main aim of parallel processing is to improve the performance by reducing the execution time, and making better utilization of resources to achieve high degree of parallelism. There are two types of memory available first is shared memory which is available for all processing elements and other one is distributed memory available as a part of each processing element.

The shared memory system supports non-objective part of the sharing which is more costly as compared to distributed memory system. There are several systems to reach to parallel processing using Graphics Processors (GPU), which has multiple cores that can help to achieve high performance parallel computing. The languages used in GPU to develop programs are CUDA, OpenCL, OpenACC, OpenMP.

CUDA

Compute Unified Device Architecture is a Single Instruction Multiple Data stream (SIMD) architecture. One instruction executed by all processors simultaneously and maintain a work load on all processors. It provides a parallel processing platform and GPU operating atmosphere. CUDA is very useful for highly parallel algorithms. If one wants to decrease the execution time of the algorithm when executed on GPU then he/she is required to have multiple threads.

In general, more the number of threads, better the performance, for the most of the serial algorithms. To achieve high performance serial algorithm can be converted into parallel ones, but this is not always feasible. As discussed earlier to induce smart optimization we need to divide the single process into a number of threads, then performance of algorithmic program will increase.

GPU

(Graphical Processing Unit) is an electronic device made for using multiple parallel processes that provide fast execution and fast results, using CUDA platform and GPU achieved time and memory reliability, it has multiple core in a device and each core have a multiple thread those threads execute multiple task parallel. GPU is more powerful as compared to CPU and now days in most of the areas GPU is preferred for processing purposes.

GPU Internal Architecture :

Grid**:** is an internal part of GPU device. There are multiple grids in a Graphical Processing Unit that splits the tasks and perform operations, it internally calls blocks, and threads.

Block**:** is an internal part of grid, each grid has multiple blocks. Each block has shared memory, registers, and local memory.

Thread**:** is a small part of process, multiple thread belongs to every block, each thread executes parallel task and every threads have shared memory, native memory, and registers. Threads in each block shares data through Global Memory, native memory, registers, constant memory, shared memory and texture memory.

Global Memory**:** is sharable among all threads, and perform write and read operation. It is slow in processing and not a cache for serial processing. 16 byte memory is faster than other memory.

Constant Memory**:** stores constant method and values, but it is slow and it has cached.

Texture Memory**:** is a read-only memory that can improve performance and reducing memory traffic when read operation have certain accessing patterns.

Registers**:** is a fast available memory provided for each thread using access and storage purpose.

Shared Memory**:** is a same for each thread and separate for each block, it is smaller than global memory and it uses read and write operation. Threads can execute simultaneously then provide the information of availability of space in a shared memory in block.

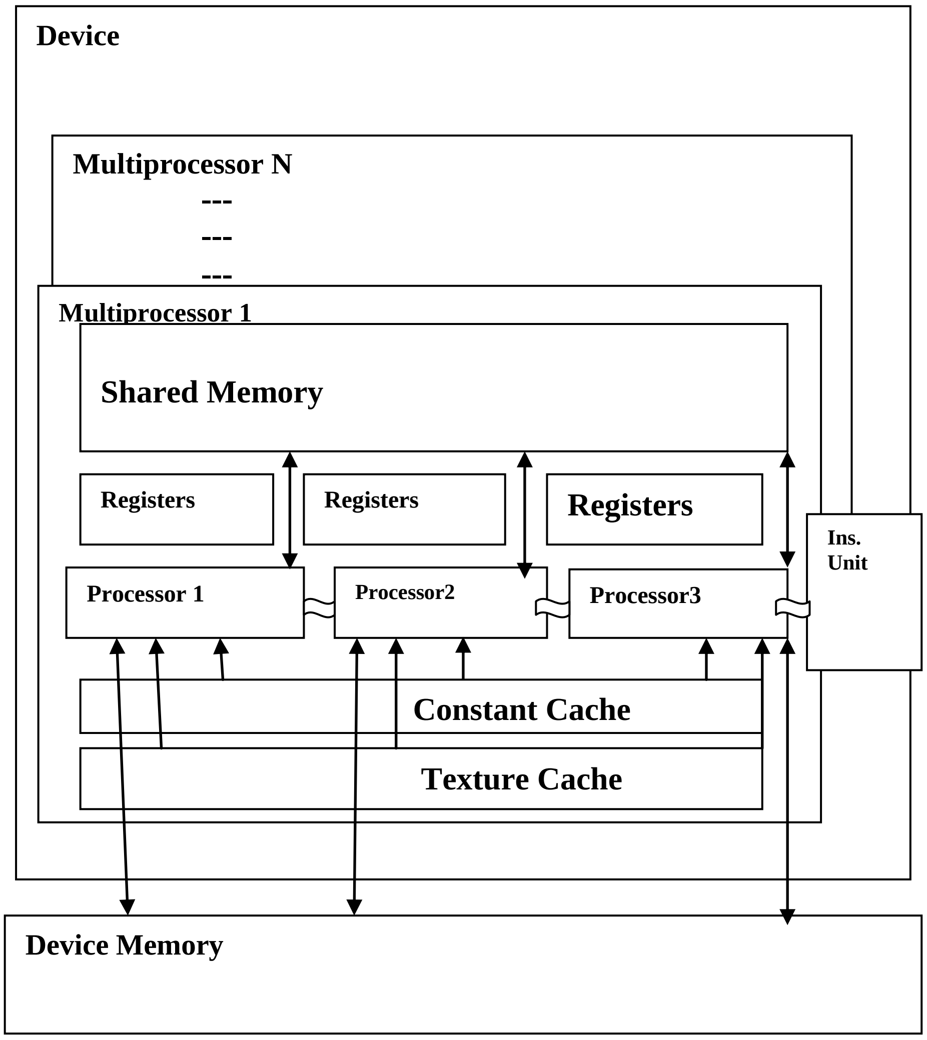

Fig. 1: Compute Unified Device Architecture

Here Host/CPU and Device/GPU communicate through a host function that is shown in fig. 4 kernels as a function instruct to the device (GPU) for any task to execute parallel in a multiple threads in a single block.

II. LITERATURE REVIEW

Barrier Synchronization is a group of processing elements that stops the process at fixed points known as synchronization points. At that point all the processing elements wait until all processes reach that place. When the processing load is unbalanced, barrier synchronization can reduce processor utilization. Many performance optimization strategies have been proposed to optimize the number of synchronization points to enhance the load-balancing among processing units for performance [1][2][3][4][5].

In this paper studied, however, handle counterintuitive phenomena that for GPUs adding unnecessary barriers can enhance application performance. The Barrier synchronization can be used in applications by using _Syncthreads() command in CUDA[6]. This is effective for all threads in a thread block. Using barrier synchronization the authors tested how much memory the process takes and system speedup on various GPU devices. Here in one case using a memory conflict model, which shows the artificial barrier synchronization can relieve memory contention and maintain data access locality by synchronizing the memory demand of a thread block.

Without barriers the threads of various progresses may complete the same cache line which increases cache misses.

Using artificial barrier in any application here recognizes a program model, which is shown in fig. 5. In this pattern, the threads allow the GPU device memory easily and perform simple operation in a program. The access data should have a certain level of locality, under these conditions, adding an artificial barrier at the end of the loop can improve the GPU memory performance.

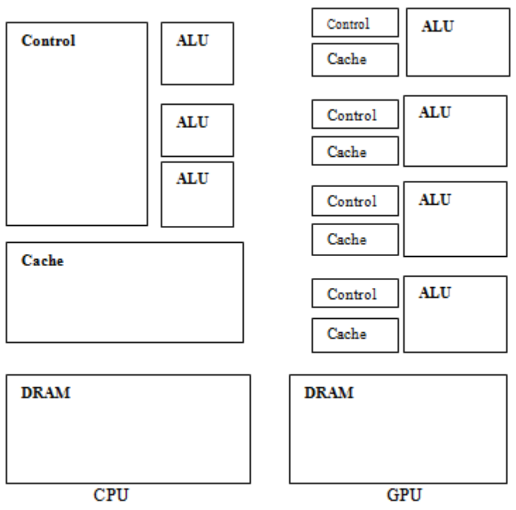

Fig. 2 CPU v/s GPU

There are many multipattern string matching applications available, including network image processing, design recognition, natural language processing. In this paper two methods are implemented Aho-Corasick (AC) [7] multipattern search techniques while Scalpel [8] uses the Boyer-Moor single pattern search algorithm. Since scalpel uses a single pattern searching algorithm, its execution time is a linear in the multiplication of the number of patterns in the pattern directory and the length of the aim string in which the search is being done.
On the other hand, as it uses an efficient multipattern search algorithm, it has a runtime that is free of the number of patterns in the directory and linear in the length of aim pattern. Several investigators tried to improve the performance of multi string matching application through the use of parallelism [9][25][26]. Our attention is to accelerating the AC and Boyer-Moore multipattern string matching algorithms through the use of a GPU. A GPU operates in standard master-slave fashion, in which the GPU is a slave that is connected to a master or host processor under whose control it operates. Algorithm development for the master slave system is affected by the location of the input data and where the results are to be stored. Normally, four cases GPU-to-GPU, Host-to-Host, Host-to-GPU, and GPU-to-Host arise depending on whether the input-output reside initially in the slave (GPU) or the master memory (CPU) [10][11].
In this multipattern string matching, here studied all the phenomena of a given directory of patterns in an aim string.
In this paper, here study addresses the two cases. In our reference there refer to the first case (slave-to-slave) GPU-to-GPU.
Careful observation of the deficiencies of a base implementation of the GPU-to-GPU AC and Boyer-Moore algorithms and a transparent detail of however these insufficiencies are often conquer. Overcome the noted lack of speed the AC implementation by a region between 8 and 9 [12][13].

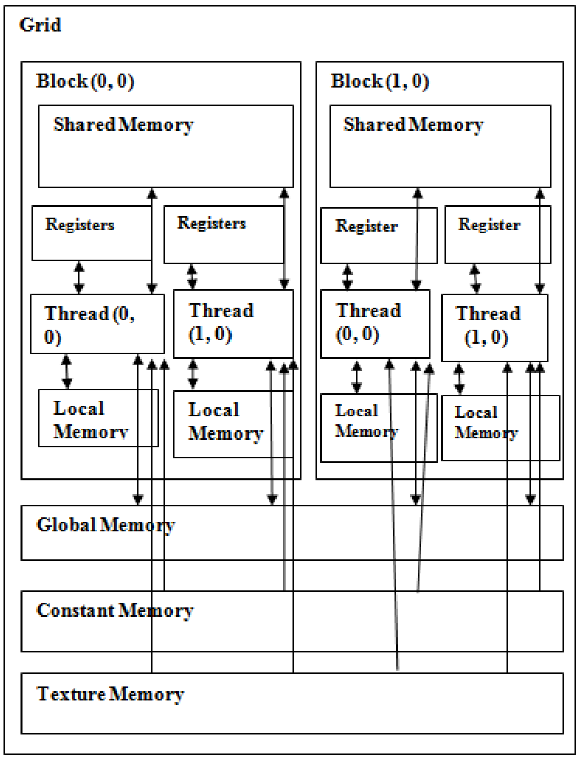

Fig. 3: GPU Accessing Memory

For the Boyer-Moore algorithm, the authors completed only the first step of a 4-step deficiency elimination process and achieved a speedup close to 2. The rest steps weren't completed as a result, it became evident that the remaining steps wouldn't create the GPU-to-GPU Boyer-Moore Formula economical with the improved Aho-Corasick formula.

For the host-to-host computing, they take into account two methodologies to overlap input-output information transfer between host and GPU with GPU calculation. The primary of these, that's intuitive and the given wants further GPU memory than is required by the alternative.

The execution of each strategy is analyzed for master-slave systems that have either one (e.g., GT200 and C1060) or (e.g., C2050) I/O channels between electronic equipment and GPU. The authors have proved the optimally of the first strategy for the systems with 1 and 2 I/O channels and conjointly the sub optimality of the second strategy. For the second strategy, they determined tight performance bounds relative to best performance. These bounds modify the PC user to estimate the trade-off between GPU memory demand and overall application runtime and to in all probability use a hybrid strategy to optimize runtime subject to the constraint of available GPU memory.

They compute first data transfer strategy with their optimized GPU-to-GPU AC implementation to arrive at a host-to-host implementation that achieves a throughput of 10 GBps on a GT200. This throughput is at the high end of the 1.5 to 12 GBps performance achieved by the C1060, which is in the same family as the GT200 implementation [14][15].

In the survey paper an implementation of parallel K-means clustering, referred to as Kps-means, that takes high execution with near-full occupancy calculates kernels while not imposing limits on the quantity of dimensions and statistic points allowable as input, so joining high degree of parallelism and efficiency[16][17]. They have analyzed parallel sorting as processing and change steps, and therefore the finals, it's enforced NVIDIA GPU compare to the processor, Implementations of algorithms on Nvidia’s CUDA-enabled multi-core architectures need to manage two major problems, first restricted size of low-latency high information measure shared memory, and second thread divergence inflicting lost clock cycles through idle threads [18][19][20].

In another survey paper; authors presented a Co-prime Blur scheme that is a video surveillance in public area has increased performance in recent history as a means to stop both terrorism and crime in an urban environment. Whatever, care about the possibility for abuse and the general loss of privacy has also grown along with the number of supervision cameras. In current years the problem of giving visual data in sensitive condition without effect on the privacy of the public has been a major research topic in the machine vision community. The challenge in secure surveillance data lies in how to minimize both the number of personal identity features and people with access to this quality. Studied here newly co-prime blurred pair model in which they have two kernels used for blurring the image are co-prime when mapped to a bivariate polynomial under the z-transform. The blurred data in a single stream are difficult to restore conventional blind deconvention approaches. Here author implements a decryption scheme on the GPU to achieve real time performance. That scheme can effectively protect sensitive identity information in surveillance videos and reconstruct the original non-blurred video stream for people with high-security clearance.

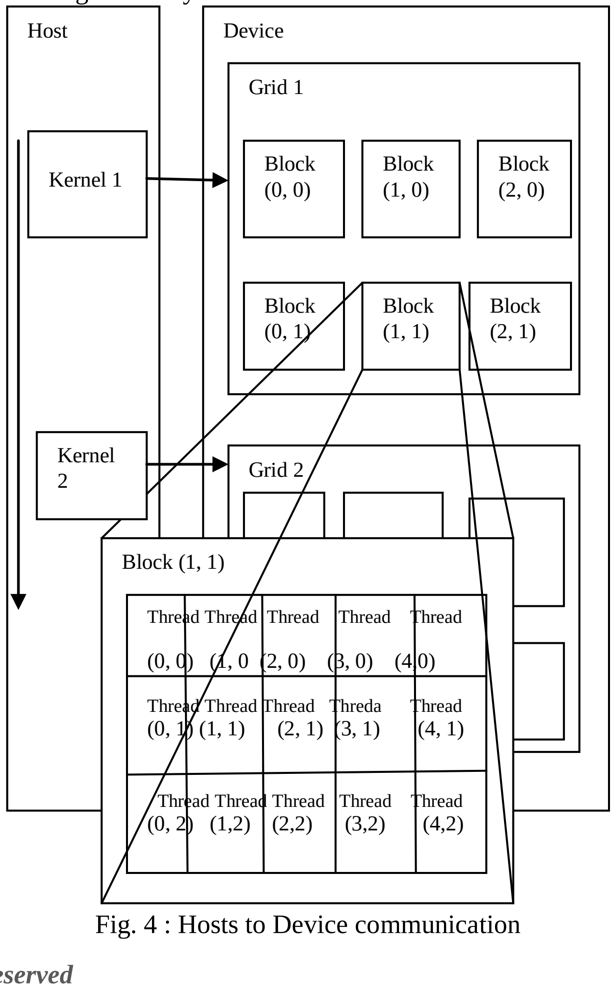

Fig. 4 : Hosts to Device communication

```
Method (….)
{ ……..
Loop(…..)
{
   1. Local memory access that has data and contention for
      the shared-memory system
   2. Logical operation
   3. ***Add an artificial barrier here
      }
      ….. }
```

Fig. 5 Patterns for using Artificial Barrier in the program

III. RELATED WORK

Several researchers have accomplished the power and capacity of GPU. So a lot of work is being done currently every day to implement GPU into current technology and method, to enhance their presentation. Parallel computing has not only helped to increase the speed and resource utilization but together build several complicated problem executions probable, to all its advantages here called high computation process.

1. The latest development in the scope of GPUs has brought some amount of computing power to system which delivers a path to increase many numerical methods.

Danilo De Donno et al [21] give an explanation to GPU characteristics by calculating how the compute time of Finite-Difference Time-Domain (FDTD) method can be decreased. Appropriately, they explain how to improve a CUDA model of the FDTD algorithm that achieves peak performance.

2. To bring for parallel programming on GPUs exploitation C, CUDA may be a straightforward stage. David Luebke[22] make a case for this methodology to perform matrix operation on GPUs utilization C/C++.

The performance increases as the author divided loop variables into threads, grids and blocks. Author focused on various memories present in the hardware and how one can use them in an effective manner. In the conclusion he deduced that the GPU can speed up the entire execution by implementing small threads in GPU, as compared to CPU.

3. Miguel C'ardenas-Montes et al[23] try to resolve large amount of optimized problem using GPUs, they study the small and available common problem solving approaches including time boundary and other factors. They characterize the design of a GPGPU-based parallel Swarm method, to deal with such problems protecting a restricted execution time limit. This implementation gains from an accomplice mapping of the data element (swarm of high dimension particles) to the parallel processing elements of the GPU. In conclusion, a graph between speedup curve and increase in dimensionality is generated.

4. The GPU is being to help register and analysis their data faster with the best insurance policies. The process of computing the rate of assurance change accordingly to many parameters for various rule owners. There are a lot of things in insurance agencies which can be too complex, for example, of the maturity of policy to constantly changing rates and shares. The data always increases and so the complexity, and then to calculate policy and the maturity amount CPU can take more time to analyze the data. In such places we can use a GPU processor that takes a huge number of parameters and calculates very fast and provide the result quickly.[24]

IV. EXPERIMENTAL STEPS

The Software Development Toolkit or SDK is a best way to study about CUDA. Using this we can write and compile the example code and execute it using the CUDA programming toolkit. The SDK is easily accessible on NVIDIA's official website and can be downloaded by anyone. Anyone have basic programming knowledge can start writing a CUDA code from C with some settings that is in a position to run on the GPU.

Start to Develop our Own Application Steps

1. Install Microsoft Visual Studio's latest version as an environment for CUDA programing.
2. Attach CUDA capable GPU device in the system and install its driver for compute capability of the Graphics Card.
3. Write a CUDA code in C/C++ or Java according to choice.
4. Select some internal settings like to include the CUDA library, and working environment.
5. Using CUDA library function, we can achieve the parallel execution and transfer instruction CPU to GPU for execution and after execution the result copy to host memory space.
6. And show output in command prompts according to our inputs.

Implementing Basic CUDA Code

1. CUDA API provides functions kind qualifiers like __global__, __device__, __host__.
2. Using "CudaMalloc" function for GPU memory allocation.
3. Take as input data in the CPU.
4. Copy data from CPU to GPU using library function CudaMemCpy with parameter as (CudaMemcpyHostToDevice)
5. Then, using function calls the control will be transferred to CPU to GPU they have specified the number of grids, blocks and threads i.e. that provide parallelism in our program.

6. Copy the result data in CPU, memory by using library function CudaMemCpy with Parameter (CudaMemcopyHostToDevice).
7. After that's given result, free the GPU memory using library function CudaFree.

As discussed above setting up CUDA is easy and writing code is also not difficult if anyone is familiar with computer programming. The most crucial part of in CUDA programming is deciding how many threads are required for proper parallelism. The kernel calls are important, as they can make the code run very efficiently or can also break the system if not properly managed.
CUDA Pros:

1. The CUDA functions are written in basic C\C++ languages, thus coder writes someone easier.
2. C functions are pointer supported.
3. C++ constructors are also supported.
4. Uses simple API.
5. CUDA increased GPU performance using libraries.
6. CUDA supports large number of languages.
7. CUDA provides documentation to write a code in different platform.

Cons:

1. It is NVIDIA GPU proprietary.
2. If GPU hardware is not available, then CPU does not take care about data backup and a task is aborted.
3. CUDA programs require settings that are difficult to set.

Benefits:

1. CUDA provides high level language C programming model for flexible programming.
2. GPU provides more thread facility compared to the CPU.
3. CUDA provides a shared memory for all threads that can communicate data easily.

Limitations:

1. GPU doesn't support recursive function, with help of the loop, we can achieve this.
2. Texture version not there.
3. Only support the NVIDIA GPU.

V. CONCLUSION AND FUTURE WORK

After comparing GPU and CPU programming and CUDA paradigms of parallel computing, we can say that the Nvidia GPU Architecture future is very good hand and that initiates the change of parallel computing in both software and hardware area. There are the fast executions and more effective purposes GPU is used in our literature survey. we studied and it is better for other and all parallel computing area.
Using CUDA API and libraries enhanced the performance of programming on parallel computing in various algorithms like sorting, multiplication, addition, string matching we will implement them and compare the results with CPU and GPU in terms of execution time and memory utilizations.

REFERENCES

[1] K. Avinash, Monu Pandey, Ajinkya Deshpande, and Prof. M. Rajasekhara Babu, “Improved GPU Co-processor Sorting Algorithm with Barrier Synchronization”, IJAER, ISSN 0973-4562, Vol. 8, No. 19 (2013).
[2] Xinyan Zha and Sahni fellow IEEE Transactions on Computers , “GPU to GPU and Host to Host Multipattern String Matching on a GPU”, Vol 62, No. 6,June 2013.
[3] Kai J. Kohlhoff, Vijay S. Pande, and Russ B. Altman, “K-Means for Parallel Architectures using All-prefix-Sum Sorting and Updating Steps” , IEEE Transactions on Parallel and Distributed Vol. 24, No. 8, August 2013..
[4] Christopher Thorpe, Feng Li, Zijia Li, Zhan Yu, David Saunders, and jingyi Yu, Member, IEEE “A Coprime Blur Scheme for Data Security in Video Surveillance”, IEEE Transaction on Pattern Analysis and Machine Intelligence, Vol. 35, No.12, December 2013.
[5] M. C. Rinard, “Using early phase termination to eliminate load imbalances at barrier synchronization points,” in proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications,2007, pp. 369-368.
[6] R Gupta, “The fuzzy barrier: a mechanism for high speed synchronization of processors”, in proceedings of the third international conference on Architectural support for programming languages and operating systems, 1989, pp. 54-63.
[7] D. Cederman and P. Tsigas, “On dynamic load balancing on graphics processors,” in Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, 2008, pp. 57-67.
[8] S. Sengupta, M. Harris, and M. Garland, “Efficient parallel scan algorithms for GPUs,” NVIDIA, NVIDIA Technical Report NVR-2008-003, 2008.
[9] M. Burtscher and K. Pingal,” An efficient cuda implementation of the tree-based barnes hut n-body algorithm”, in GPU Computing Gems Emerald Edition, W. –m. W. Hwu, Ed. Morgan Kaufmann, 2011, ch. 6, pp. 75-92.

[10] N. Jacob and C. Bradley, “Offloading IDS Computation to the GPU, ” Roc. 22nd Ann. Computer Security Application Conf. ,2006.

[11] D.E. Knuth, J.H. Morrise Jr., and V. Pratt, “Fast Pattern Matching in String,” SIAM J. Computing, vol. 6, 323-350,1977.

[12] L. Marziale, G. Richard III, and V. Roussev, “Massive Threading: Using GPUs to Increase the performance of digit Forensics Tools,” Science Direct, vol. 4, pp. 73-81, 2007.

[13] C. Lin et al., “Accelerating String Matching sing Multi-Threaded Algorithm on GPU,” Proc. IEEE Globecom, 2010.

[14] G. Navarro and K. Frederiksson, “Average Complexcity of Exact and Approximate Multiple Streang Matching,” Theoritical Computer Sceince, Vol. 32, pp. 283-290, 2004.

[15] A. Pal and N. Memon, “The Evolutaion of File Carving,” IEEE Singnal Processing Magazine, Vol. 26, No. 2,pp. 59-72, Mar. 2009.

[16] R. wu, B. Zhang, M. Hsu, and Clustering, “Billions of Data Points Using GPUs”, Proc. ComBinde Workshops Unconventional High Performance Computing Workshop Plus Memory Access Workshop(UCHPC-MAW’09), 2009,doi10.1145/1531666.1531668

[17] M. Zechner and M.Granitizer, “ Accelerating K-Means on the Graphics Processor via CUDA,” Proc. First Int’l Conf. Intensive Applications and Services (INTENSIVE’09), pp. 7-15 ,2009, doi:10.1109/ INTENSIVE.2009.19

[18] H. Bai, L. He, D. Ouyang, Z. Li, and H. Li, “K-Means on Commodity GPUs with CUDA,” Proc. WRI World Congress Computer Sceince and Information Eng., Vol. 3, pp.651-655,2009,doi:10.11.1109/CSIE.2009.491

[19] S.A.A. Shalom, M. Dash, and M. Tue, “Efficient K-Means Clustering Using Accelerated Graphics Processors,” Proc. 10th Int’l Conf. Data Warehousing and knowledge Discovery I. Song, J. Eder, and T. Nguyen, eds., pp. 166-175, 2008, doi: 10.1007/978-3-540-85836-2_16.

[20] T. Zhang, R. Ramakrishana, and M. Livny, “BIRCH: An Efficient Data Clustering Method for Very Large Databases,” Proc. ACM SIGMOD Int’l Conf. Management of Date, pp. 103-114, 1996, doi:10.1145/23/235968.233324.

[21] Danlio De Donno et al., “Introduction to GPU Computing and CUDA Programing: A Case Study on FDTD,” IEEE Antennas and Propagation Magazine, June 2010.

[22] Miguel C’ardenas-Montes,” Accelerating Particle Swaram Algorithm with GPGPU,” 19th International Euromicro Confrence on parallel, Distributed and Network-Base Processing, 2011.

[23] M. Zechner and M. Granitizer, “Accelareting K-Means on the Graphics Processor via CUDA,” Proc. First Int’l Conf. Intensive Applications and Services (INTENSIVE ’09), pp. 7-15, 2009, doi: 10.1109/INTENSIVE.2009.19

[24] Jayshree Ghorpade, Jitendra Parande, Madhura Kulkarni, Amit Bawaskar ”GPGPU Processing in CUDA Architecture” ACIJ, Vol.3, No.1, January 2012.

[25] Rajesh Tiwari et al. , “ The Efficient load balancing in the parallel Computer” , International Journal of Advanced Research in Computer and Communication Engineering , Vol. 2, Issue 4, April 2013 , pp 1667 – 1671 , ISSN : 2319-5940 .

[26] Rajesh Tiwari et al. , “Analysis of Various Decentralized Load Balancing Techniques” International Journals of Recent and Innovative Trends in Computing and Communication (IJRITCC), Vol2 , Issue 11 , November 2014, pp 3360 – 3365 , ISSN: 2321 – 8169.