A Study on parallization of genetic algorithms on GPUs Using CUDA

Sagar U S

Assistant Professor Department of Computer Science and Engineering, Srinivas Institute of Technology, Merlapadavu, Valachil, Mangalore -574 143, Karnataka, India

---***---

Abstract - The unit of Graphical processing is in every system consisting of Graphic card and is considered as integral part of computer. Compute unified device architecture is a language built by one of the leading graphic card company NVIDIA in order to carry on the parallel work on the GPU. CUDA has lot of applications in the field of Genetic algorithms, Machine learning algorithms, Numeric analysis etc. It is basically deals with improve in its efficiencies, standardize the operations, increase performance, improve the computation time taken by GPU. It provides a communication medium between the device and the host. And also helps in developing a platform where all the data are analyzed When the computation of the speed off factor is carried out, our aim is to show that the parallel code running on GPU has achieved more gain than the CPU code added more weight age by analyzing the computational values by plotting them in a graphical format. Thus it helps the end user to visually compare the values of parallel GPU and CPU performance and helps to achieve the better results.

Key Words: **GPU, CUDA, genetic algorithms, Floyd and warshall, NVIDIA, parallization**.

1. INTRODUCTION

In the modern era of computers everything has to be more graphical, so the processing unit of Graphics (GPU) mainly to improve the calculation including the graphical and non- graphical units in computers. Currently in our project we make used of NVIDIA GeForce graphic card.

This GPU basically has more than 100 cores, they acts like the co-processors to the CPU that helps while doing graphical computations. Generally CPU takes more computational time when compared it with GPU. So a question might rise in everyone's mind like how GPU is doing faster computations than CPU?

And the answer is simple, the CPU has maximum of 4 to 8 cores and GPU has more than 100 cores. This generally increases the speed of the computation [1]. In this paper, there is implementation of few genetic and numeric analysis algorithms and runs it parallel across the GPU. Then to calculate the Speed of factor of serialize and paralyze results. Genetic algorithms are used to find the solution for the problems related to optimization.

For Example, take a string with no of elements in it, and after the process of testing or by simulation you will get a "n" best case result. In the next iteration what you will do is you breed those "n" values of best case, so that we get the optimized results. This function is called the fitness function. Then finally close the algorithm when satisfactory level of optimization is done. All these operations are running on Nvidia GPU and is written in Compute unified device architecture (CUDA). We study and analyze all these algorithms in order to gain best performance.

GPU generally tasks up to successful execution of the problems related to graphical processing. Mainly graphics are based on how the person visualize, one of the need for the acceptability of graphics is because of its appearance.
There is a concept called "graphics pipeline".Vertex operations are the first stage in graphical pipeline. Triangles can be formed for the objects in all surfaces. Second stage basically converts the data which is the input into triangle (vertices). But there is a chance that 100's of 1000's of vertices, now task every vertex has to be independently calculated, and a formidably high amount of number of units of processing is required so that it can be accelerate the processing of third stage.

The second stage is rasterization, used to determine basically which pixels of screen are included by every number of available triangles. Further stage gives the operations that carry out the fragment option; the content is basically obtained in the context of textures or can say as images and deployed accurately on to a surface through fragment or can say as pixel processing. But we have a very huge quantity of pixels, and also in this stage there is a necessity of maximum amount of units for processing in order to increase the processing units of the fragments. In the last stage of the operations where each and every fragments are combined and stored into a composition way of buffering into the frame. There are certain things called function pipelines which constitutes of hardware which is configurable, but it doesn't support programmable format. It's operated with the Finite State Machines from the API (Application Programming Interface). Basically the API is only one of the standardized layer always allows applications especially the gaming applications to use software or hardware services and also it uses its functionality.

API has become more popular from past two decades and each part of hardware and in the present trend of API which basically includes the improvements which are incremental and are mainly used for the images to display.

Further the present trend of GPUs includes the main part of the hardware with all of the processors of vertex as well as processors of fragments and best thing it is programmable. The processors of vertex generally instructed to carry out the operations related to the vertex and the processors that are related to the fragments were normally instructed in order to carry out the operations that are linked to the fragments. The best part about these processors that they have instruction set for their own use. The coding related entity for this specified hardware was flexible enough and also when related to the "graphics pipelines" where it includes the acceleration part of the unit processing of the graphics was carried away further. When it comes to displaying of different image the graphic part is enabled. Although after doing all these efforts the experts in the field of graphics were not completely agree to the point because there is a fault in the hardware part which is not being able to utilize.

The other part has a very minimum number of the tasks related to programs, and each and every processors of vertex are not at all free when compared with increasing number of operations related to vertex, whereas plenty is left on the parallel side. But in certain tasks related to the programs, certain amounts of the processors of vertex is completely not part of the tasks and are except from task. Basically we are dealing with the small number of the operations, in which each and every part of processors of fragment is not at all free when compared with large amount of the operations related to fragments. The output is the overloaded of the hardware that involves the maximum operations, which seeks the alternate part is set to be completely discharged with minimum operations.

Present generation of evolving GPUs basically has the part of the hardware that includes the USP (shared processor which is united), look alike processors are grouped and were utilized for all the operation related to vertex and also the part of the operations of fragment in order to improve usage of the part of the hardware. The very famous Xbox includes the initial part of the GPU along with USP part that constitute the hardware, after doing all this it still includes the alternate set of instructions for all the operations related to the vertex and the operations includes fragment.

There is difficulty for the general purpose programming. The team has this little amount of freedom related with the programming language (CUDA in our case). From the last 10 year, the GPUs especially NVIDIA were growing as a giant when related to other graphic cart company. And also the best part of this graphic cart is that it has improved coming generations of GPUs, this definitely brought certain amount of improvements with respect to the hardware like the amount of cores, size of the data bus etc.

It is a programming language that is at the top level and is proposed by NVIDIA in the year 2006. It is very much similar to "C"along with certain added extensions along with general added restrictions. It is referred as engine related to the part of computing part of GPU. This language is basically used for programming exclusively on NVIDIA related GPUs. The application is stated and Microsoft visual studio is the complier which is used that supports CUDA extensions. After writing the CUDA program it is compiled by Microsoft visual studio compiler which is running on operating system.

The instruction set of compiler initialize the program with instruction set with PCI and the instruction is passed through RAM on GPUs. The GPUs parallelize the code and returns the instruction set. The Complier then runs the code and displayed the execution time taken by complier to compute the instruction set.

2. RELATED WORK

Parallelizing few Genetic and Numeric analysis Algorithms which is running on the GPUs and analyzing the performance of those algorithms on multiple-core using CUDA programming language. And finally calculating the speed of factor which is calculated by comparing parallel and serial computational time to execute. Then plotting the graph using RSTUDIO to clearly determine the comparison of serial and parallelized CUDA programs. The results will be help to analyze the performance of particular algorithm and also it indicates the need of the parallization for reducing the time and also for the betterment of results.

While explaining the need of multi-core in paper [1], author explains that many of the upcoming brand new applications are based on the concept of multithreading. So basically the trend of computers has recently shifted more towards parallelism. Author mentions three problems that can be solved by introducing parallelism they are:

- Problems regarding heat sync.
- Problems regarding speed of light.
- solves the problem in difficulty of verification and design.

Authors basically explains the how difficult it was to make the single core clock frequency higher. Hence he proposed the parallelism in multi core.

In another paper [2] author petr pospichal explains the concept of parallelizing the genetic algorithms using the programming language CUDA. It's a process of Computing platform for parallization and API created by NVIDIA. The CUDA platform is developed in such a way to work with programming languages such as C++, C and FORTRAN. Currently in this project am using Microsoft visual studio 2013 which supports all these extensions. In this paper he explains how the flow of program works in CUDA.

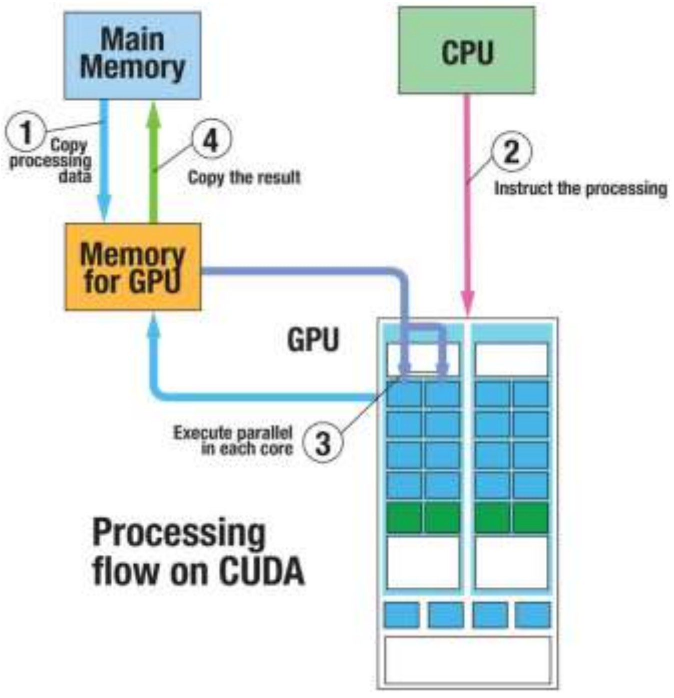


Fig 1: Processing flow on CUDA

In article [3] author Maciej presents a way of implementation of genetic algorithm in CUDA. He basically used a sample algorithm that operates on a huge population and a very complex genotype, so that it overloads the size of the cache memory. Hence it is not completely moved to the graphics card. The presences of modules that execute on the CPU are synchronized through CPU. Considerations were based on weak, but always available graphics cards to test the capability of acceleration algorithms at very low cost.
The literature survey clarifies about the variety of proposed articles and fellow research papers for the application of Compute Unified Device Architecture for parallelization

purpose. It also mentions few of the problems of the proposed models.

Few of the problems of the proposed articles are using multi-core as a part of improving quality of processors and how that can be achieved by using parallelism. The articles do mentioned about how to parallelize the genetic algorithms using CUDA on GPUs [5]. Our proposed platform overcomes all of these limitations and help in achieving better results of parallization when compared to other traditional method of parallelization.

3. SYSTEM MODEL

The design is mainly concerned with developing an important structure of a system. It involves observing the very important components of the system and also it identifies the communications between these components. The architecture that is suited for our proposed system. In the following sub-sections we insight into the various design aspects and the sub-systems involved in this architecture.

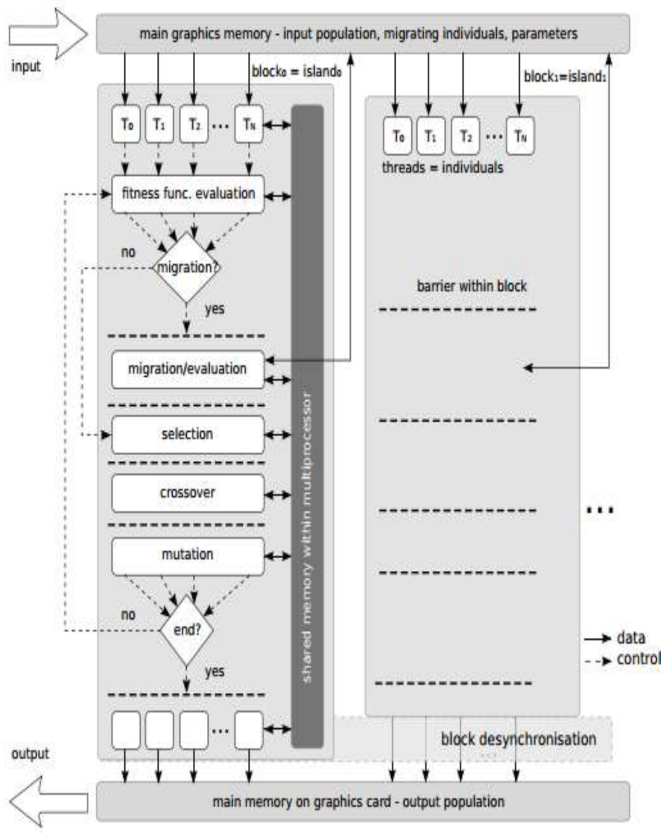


Fig 2: Architecture of CUDA flow [4]

According to the Compute Unified Device Architecture framework is used to execute our Genetic Algorithm on GPU. The CUDA toolkit version 7.5 achieved best speedups on GPU and among the wide community of developers. This toolkit can be run on any of the NVIDIA graphics card mainly from the Geforce [7] generation basically running on Windows and Linux. Parallelism is achieved on GPU which is expressed by directives of compile then it is added to the C programming language, the graphic NVIDIA GPUs[6] mainly contains of multiprocessors that is able to carry out work in parallel. Tasks running in this particular unit are compositely lightweight and can be constantly monitored using barriers so that consistency of data is managed which can be executed with very minimum impression regarding the multiprocessor and its performance, but not comparing the multiprocessors. This problem with multiprocessor will forces us to choose either completely independent or migrations of performance between these two asynchronously. The memory basically linked to graphics cards is constituted into two levels which is the main memory and the next one is chip memory. Main memory generally has a lot of capacity and it carries a complete dataset as well as programs of user related. It also provides an output point when communicating with CPU. But the problem is, high latency over weights the lot of capacity. One more problems are, chip memory is quick, but it is contained to limited size. Other than per-thread registers, the chip memory particularly contain the useful multiprocessor segments which is shared. This array which is of 16KB acts as a L1 cache which is user managed. The size of memory of chip is a main controlling factor for designing genetic algorithms efficiently, but existing applications of CUDA deals with much more benefit.

4. COOPERATIVE REQUIREMENTS

The main aim of the programming phase or the execution phase is to convert the basic design of the problem created during the design process into the code in a programming language[8][9][10] that is CUDA in our case, that is basically compiled and run by a computer and that performs the calculation of the specified CPU and GPUs by the design.
Implementation of any program is always followed by correct decisions in the field of selection of domain used, the programming language used, etc. These important decisions are basically influenced by very crucial factors such as the working environment.

There are 3 important decisions during the implementation. They are:

- Choosing the operating system.
- Choosing the hardware and software required for implementation.
- Choosing the programming language on which the work is carried on.

4.1 Floyd and Warshall

The main purpose of this algorithm is to find the shortest path of the weighed graph.

Step 1: Declaring the CUDA functions for Floyd algorithm

Step 2: Finding Shortest path on Parallel GPU and Serial CPU

The above function calculates the CPU and GPU timings from the start time to end time taken to find the path which is shortest with respect to the weighted graph.

Once the path is found the time is initialized and function copies all the information to the device and basically it runs on the Kernel of CUDA.

The graph is generated based on the random weights which can be seen in the step 3 of Floyd-Warshall implementation on CUDA.

Step3: Copy the memory and free the device

In the above step the error sting is printed and copies back the memory from one unit to another. Once it is successfully copied the CUDA device memory is freed. The memory which is providing the error string is freed.

Basically this step is to generate the random graph error free, if any memory location is having the error then it is detected and freed in this step.

Step 4: Generating random graphs

The graph generated has to be random because it helps the GPU timings better as the computer randomly generates the values and also it decreases the human effort for entering the values.

Step 5: Generate result file

The final step will be generating the result file set. If there is any error the print that the program is unable to find the shortest path for randomly generated distance.

Else print the success statement of Floyd program along with the computational time it has taken to execute the input. And then determine the speed off factor when comparing it with the serial values.

4.2 Monte Carlo pi value determination

The Monte Carlo algorithm is a one which determines whether the value to be either true or false. Either it gives the result or it doesn't give the expected output.

In this case I have considered the calculation of the value of pi in both CPU and GPUs and also this program will help us to calculate the percentage of increase in computational timings of GPU with respect to CPU.

Step 1: Defining Constants and checking CUDA error string

The above pseudo code in fig 5.6 first initializes the constant threads and blocks required during the execution of program. Next step will be checking for CUDA errors for the threads created. If the error is found then debug those errors and get the string which is the reason for error.

Step 2: Creating and allocating shared memory blocks

The next step will be creating the shared memory blocks for the threads in order to hold all blocks of memory.

There is a function called _syncthreads() which will wait till all the threads are catch up. Once the threads are at same point for each block the summing is done based on the previously created shared memory.

Step 3: Reducing parallel to speed up the computation

In order to increase the speed of execution of the program the thread is executed in parallel. Mainly here we make use of shared memory concept as described in previous step.

The sum of the threads is reduced by running it in parallel and by sharing its memory. For each block of thread the parallization is done based on the dimension of particular block of thread. And thus we can able to increase the speed of execution of each thread running parallel.

Step 4: Calculating the start and end time of execution

In this step the beginning time and finished time of execution of program is calculated. And also check out the errors if any, then the data is copied from GPU to CPU.

Finally print the execution statement and free the CUDA memory which is allocated in step 1 and then reset the device as it is in the beginning using cudaDeviceReset() function.

4.3 Odd-Even Merge Sort

This is one of the sorting methods also called as Batcher even-odd merge sort. It is genetic algorithm mainly used for parallelization purpose.

Splitting the elements based on odd and even digits and finally merging and sorting the individual elements.

Step 1: CUDA function for Odd sort

The above function in fig 5.10 is the CUDA function for sorting the odd values from the given set of individuals. And assign the values of odd one to the created thread and store it in a temporary array variable.

Step 2: Generating the CUDA even function

The function in fig 5.10 is the CUDA function in order to sort the even values from the randomly generated set of individuals. And add the values that are even to the thread id created and store it in array variable.

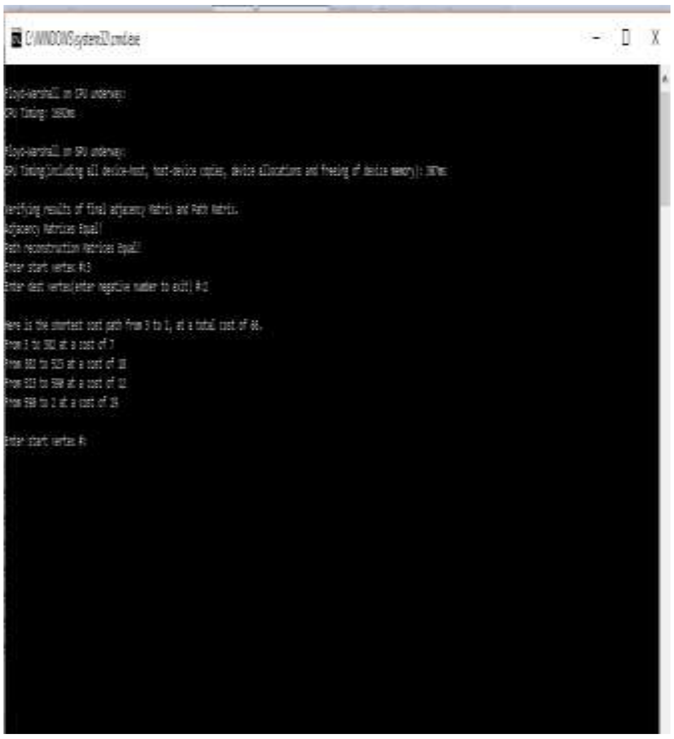
Step 3: Allocate CUDA memory and compute time elapsed

In the final part of this odd-even merge sort the memory is allocated to both odd as well as even function.

Then Create the Start and stop event in order to calculate the time taken by the program to sort the values. When the sorting is done calculate the time and then free the allocated CUDA memory.

5 Experimental Results

Floyd-Warshall Result:


C:\WINDOWS\system32\cmd.exe

```
Floyd-Warshall on CPU underway:
CPU Timing: 1692ms

Floyd-Warshall on GPU underway:
GPU Timing(including all device-host, host-device copies, device allocations and freeing of device memory): 307ms

Verifying results of final adjacency Matrix and Path Matrix.
Adjacency Matrices Equal!
Path reconstruction Matrices Equal!
Enter start vertex #:3
Enter dest vertex(enter negative number to exit) #:1

Here is the shortest cost path from 3 to 1, at a total cost of 66.
From 3 to 382 at a cost of 7
From 382 to 515 at a cost of 18
From 515 to 598 at a cost of 12
From 598 to 1 at a cost of 29

Enter start vertex #:
```

Fig 3: Floyd and Warshall results

From the above figure the CPU timings is 1692 ms and the optimized paralyzed GPU timings is 307ms.

So the speed off factor is 1692 / 307 = 5.5
Therefore the parallization of GPU on Floyd-Warshall algorithm has shown more than 5 times increase in the result when compared it with the CPU timings.

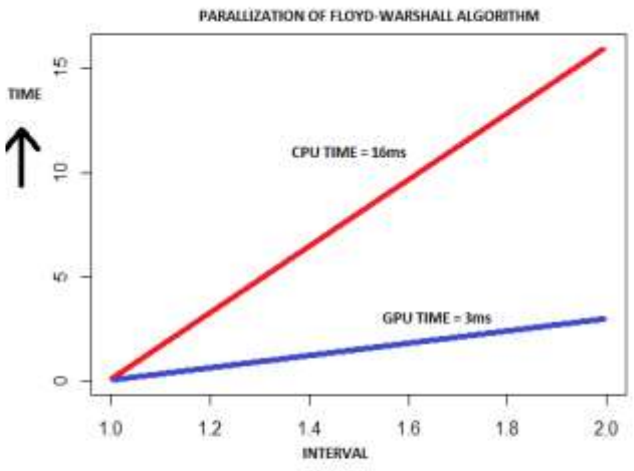
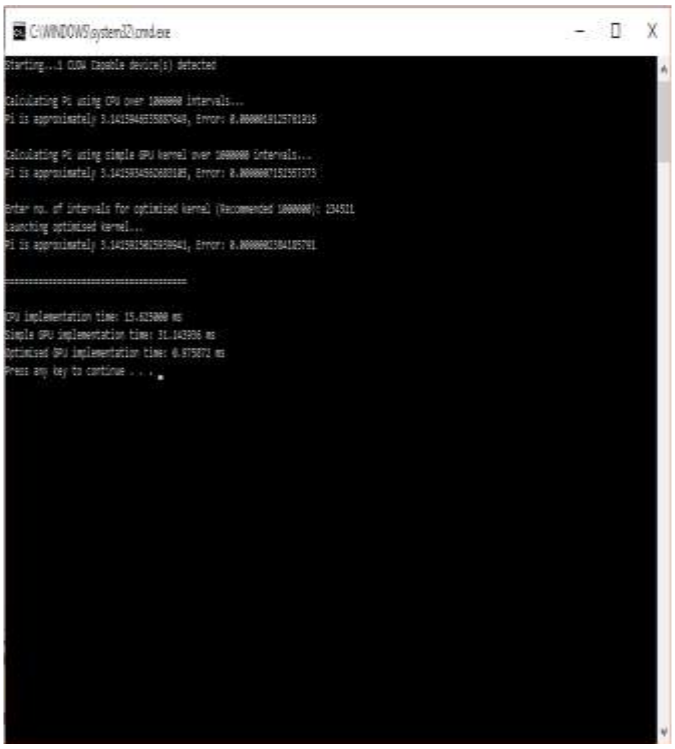


Fig 4: Floyd and Warshall Graphical Analysis

Monte Carlo Result:


C:\WINDOWS\system32\cmd.exe

```
Starting...1 CUDA Capable device(s) detected

Calculating Pi using CPU over 1000000 intervals...
Pi is approximately 3.1415946535887648, Error: 0.0000019125781316

Calculating Pi using simple GPU kernel over 1000000 intervals...
Pi is approximately 3.1415934562680185, Error: 0.0000007151557373

Enter no. of intervals for optimised kernel (Recommended 1000000): 234511
Launching optimised kernel...
Pi is approximately 3.1415915825959941, Error: 0.0000002038108791

====================================

CPU implementation time: 15.625000 ms
Simple GPU implementation time: 31.143936 ms
Optimised GPU implementation time: 0.975872 ms
Press any key to continue . . .
```

Fig 5: Monte Carlo results

From the above figure the CPU timings is 3100 ms and the optimized paralyzed GPU timings is 1512 ms.

So the speed off factor is 3100 / 1512 = 2.1
Therefore the comparison of GPU on Monte Carlo algorithm to calculate the pi value has shown more than 2 times increases in the result when compared it with the CPU timings.

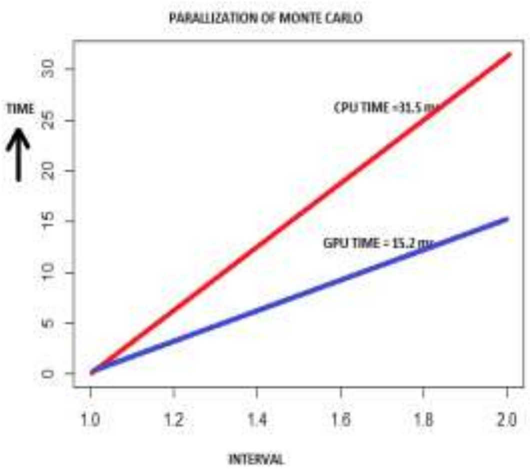
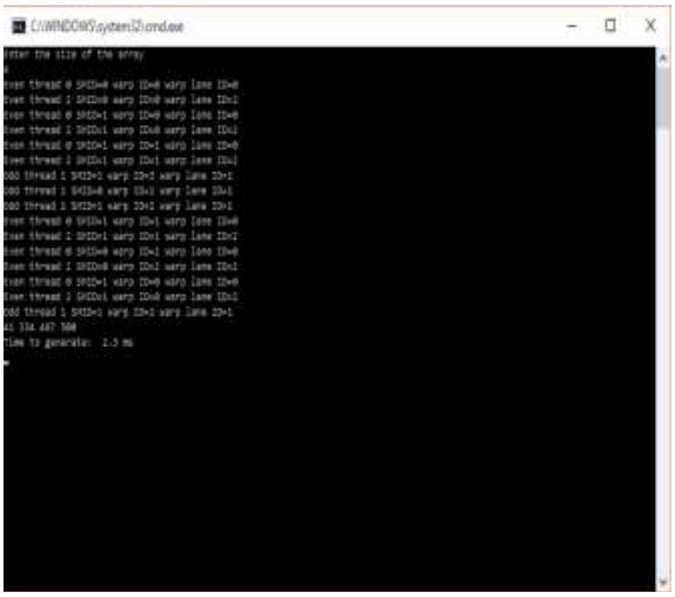


Fig 6: Monte Carlo Graphical Analysis

Odd-Even Merge Sort:



```
Enter the size of the array
4
Even thread 0 SMID=0 warp ID=0 warp lane ID=0
Even thread 1 SMID=0 warp ID=0 warp lane ID=1
Even thread 0 SMID=1 warp ID=0 warp lane ID=0
Even thread 1 SMID=1 warp ID=0 warp lane ID=1
Even thread 0 SMID=1 warp ID=1 warp lane ID=0
Even thread 1 SMID=1 warp ID=1 warp lane ID=1
Odd thread 1 SMID=1 warp ID=1 warp lane ID=1
Odd thread 1 SMID=0 warp ID=1 warp lane ID=1
Odd thread 1 SMID=1 warp ID=1 warp lane ID=1
Even thread 0 SMID=1 warp ID=1 warp lane ID=0
Even thread 1 SMID=1 warp ID=1 warp lane ID=1
Even thread 0 SMID=0 warp ID=1 warp lane ID=0
Even thread 1 SMID=0 warp ID=1 warp lane ID=1
Even thread 0 SMID=1 warp ID=0 warp lane ID=0
Even thread 1 SMID=1 warp ID=0 warp lane ID=1
Odd thread 1 SMID=1 warp ID=1 warp lane ID=1
41 334 487 500
Time to generate:  2.5 ms
```

Fig 7 : Odd-Even Merge Sort

From the above figure the CPU timings is 4.2 ms and the optimized paralyzed GPU timings is 2.6 ms.

So the speed off factor is 4.2 / 2.6 = 1.6
Therefore on observing the value of GPU on algorithm to calculate the odd-even sort value has shown more than 1.6 times increases in the result when compared it with the CPU timings.

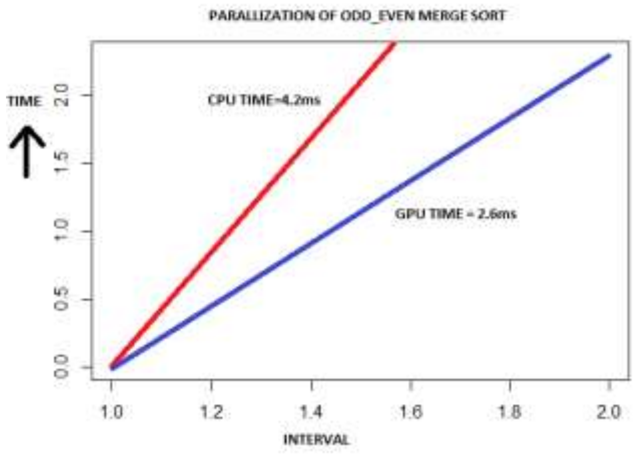


Fig 8: Odd-Even Merge sort graphical Analysis

5. CONCLUSIONS

The very important contribution done is to make an easily accessible algorithm which is running parallel and develop the design for architecture of graphical processing units

When the calculation of the speed off factor is performed, The results shows that the parallel code running on GPU has achieved at least minimum of more than 1.5 times more gain than the CPU code. Thus it helps the end user to visually compare the values of parallel GPU and CPU performance and helps to achieve the better results.

Finally with this work we have proved that the Parallel GPU can achieve higher degree of optimality when compared to CPU code.

REFERENCES

1. Tutorial by D Kirk, Computing the speed of Computation in NVIDIA graphic card.
2. “A Parallel development of algorithm on GPU” by Steven and ovideau, Dept of CSE 2014.
3. “Parallel genetic algorithm on CUDA architecture” by Peter poshpical Springer-2010.
4. Parallel experience of computing with CUDA, Michael and Grand John, IEEE-2008
5. “Implementation of genetic algorithms on CUDA” by Maciej, Springer-2014
6. CUDA application design and Development by NVIDIA 2011 edition.
7. Parallel programming with CUDA by Ian Buck.

8. https://en.wikipedia.org/CUDA

9. https://en.wikipedia.org/Parallel_computing

10. http://docs.nvidia.com/cuda/cuda-c-programming-guide/