Indian J.Sci.Res. 20(2): 227-232, 2018

ISSN: 0976-2876 (Print)
ISSN: 2250-0138(Online)

IMPROVE THE EXECUTION TIME BY USING GPU FOR COMPLEX APPLICATION WITH SIMD

RAJESH TIWARI[a1], MANISHA SHARMA[b] AND KAMAL K. MEHTA[c]

[a]Shri Shsnkaracharya Technical Campus Bhilai, Chhattisgarh, India
[b]Bhilai Institute of Technology, Durg (C.G.), India
[c]O. P. Jindal University Raigarh, (C.G.), India

ABSTRACT

Now a days sequential processing is not sufficient for a large data computation in the area of computer science and technology. To solve the computation problem for large data, the necessity for high-performance calculation is growing day by day. Few common application where high-performance computing is used are Weather Forecasting, Quantum Physics, Climate Research, Heat Distribution Problem etc. An architectural framework has been proposed by NVIDIA to join the power of GPUs with CPUs to improve the execution time. GPUs were previously used only for Graphics Application like Computer games, Multimedia and graphics but now GPU has been used for high-performance computation work. This paper focuses on different techniques for Matrix multiplication operation. This paper performs the Matrix multiplication on resources like CPU, GPU (Shared and Non-shared). Finally the results of execution time with CPU, Shared memory and Non- shared memory are compared and find that the Non – shared memory gives the better result for bulk data.

KEYWORDS: CPU, GPU, Shared Memory, SIMD

Parallel processing is the method of processing program instructions by dividing them into multiple small segments and executes that segments on multiple processors this results the minimum execution time. In the older version (Sequential) of computers, only one program can executed at a time. To solve the complex problem the sequential technique is not used, so the new technique to solve such problem is parallel technique. There are two types of program, one is computation intensive and the other is I/O intensive program. A computation intensive program consider only computation time and I/O intensive program consider only the time spend during the input and output. The interleaved execution of both (computation intensive and I/O intensive) programs together allowed in parallel processing. When the computer system starts an I/O operation, the system is in waiting state till the operation complete. During this time, the compute intensive program starts execution and the utilized the waiting time of the system. This cause in reduction of execution time.

Matrices and matrix operations are widely used in mathematical modeling of various processes, phenomena, and systems. Matrix calculations are the basis of many scientific and engineering calculations few of them are Computational Mathematics, Physics, Economics etc. One of the fundamental building block for scientific computing is the Matrix multiplication and it is one of the most important approaches to understanding parallel programming in GPU [Djinevski et. al., 2013] [Sooknanan and Joshi, 2016].

The concurrent use of more than one processor to execute a program is an example of SIMD (single instruction stream and multiple data stream) process [Sartori and Kumar, 2013]. Generally, the parallel processing makes a program to execute quicker because of more CPUs are running [Shah and Patel, 2014] parallel. In practice, it is a lot difficult to divide a program in such a way that separate CPUs can execute different portions of the program without interfering with each other.

A parallel computation engine is used in GPUs to carries out the complex computational problem in less time than it would have if same problem would have been executing on a single CPU [Cui et. al., 2009] [Shah, 2015]. GPUs have been previously utilized mainly for playing games or the application where large graphics resolutions are required. Now GPU stepped into the fields that need high-performance computation. Fields such as Medical Image, Weather forecasting, and System of linear equations are some fields, where the systems require the high-performance computation to use the possible power of GPUs by which system solve the existing and current problems.

CUDA is a library provided by NVIDIA, it provides extended functionalities in C language by adding CUDA specific functions. This paper shows the different optimization techniques of matrix multiplication [Ohshima et. al., 2006] using CPU, matrix multiplication on GPU using Shared and Non-Shared memory which increases

[1]**Corresponding Author**

floating portion for optimizing a N*N size matrix. N*N matrix means having N rows and N columns.

GPU and CUDA

Compute Unified Device Architecture is a library provided by NVIDIA to execute processes in parallel manner [Ha and Han, 2013]. This is an application programming interface (API) to help communication between device and user. There are CUDA specific functions or methods defined which meant to run on CUDA library only. These are used along with C and C++ programming language. To convert a single processor specific program into CUDA capable programs the programmer needs to modify it accordingly. The CUDA program is generally divided into two parts: the main program executes in the CPU, whereas the parallel portion of the program is executed in GPU. This GPU part is called by the main program and data is sent to GPU for execution where the instructions are executed on the given data, after the calculation result is sent back to CPU [https://developer.nvidia.com].

GPU (Graphics Processing Unit) was primarily developed to fulfill the need of algorithms used in computer graphics. It has hundreds of cores which are able to execute multiple threads simultaneously. Later it was proposed that this technology can be useful for non-graphic process also if one can divide a single process into multiple threads and distribute them to multiple processors, the overall computation time can be reduced drastically. There are several types of memory present in the GPU [Liu and Vinter, 2014] [Zha and Sahni, 2013] [Barberis et. al., 2013] like device memory, shared memory, constant cache, texture cache, and registers [Lo et. al., 2013] [Salim et. al., 2015] [Anh et. al., 2015] [Eberhardt and Hoemmen, 2016]. To manipulate data in this memory and to use the multiple cores to their programmers must write the CUDA programs very carefully.

Types of CUDA Memory

CUDA devices have different memory spaces, Figure 1 shows the memory organization and basic units of CUDA model. Global, local, texture, constant, shared and register memory. Two types of memory that actually reside on the GPU chip are register and shared memory. Local, Global, Constant, and Texture memory all reside off-chip. Local, Constant, and Texture are all cached.

Shared Memory

Data stored in shared memory is visible to all threads within that block and lasts for the duration of the block. This is invaluable because this type of memory allows for threads to communicate and share data between one another. Each block has a Shared memory which is shared by all its threads for communication within the block. it is around 50 to 100 MB. The hardware which used for this implementation has 49152 Bytes per block Shared Memory.

Register

This is fastest accessible memory present in the GPU. Data stored in register memory is visible only to the thread that wrote it and lasts only for the lifetime of that thread. The hardware which used for this implementation has 32768 per block registers.

Local Memory

Local memory performs slower. It has the same scope rules as register memory.

Global Memory

Stored data in the global memory is visible to all threads within the application (including the host), and lasts for the duration of the host allocation.

Constant memory is used for data that will not change over the course of a kernel execution and is read only. Using constant rather than global memory can reduce the required memory bandwidth, however, this performance gain can only be realized when a warp of threads read the same location.

Texture Memory

Texture memory is another variety of read-only memory on the device. When all reads in a warp are physically adjacent, using texture memory can reduce memory traffic and increase performance compared to global memory.

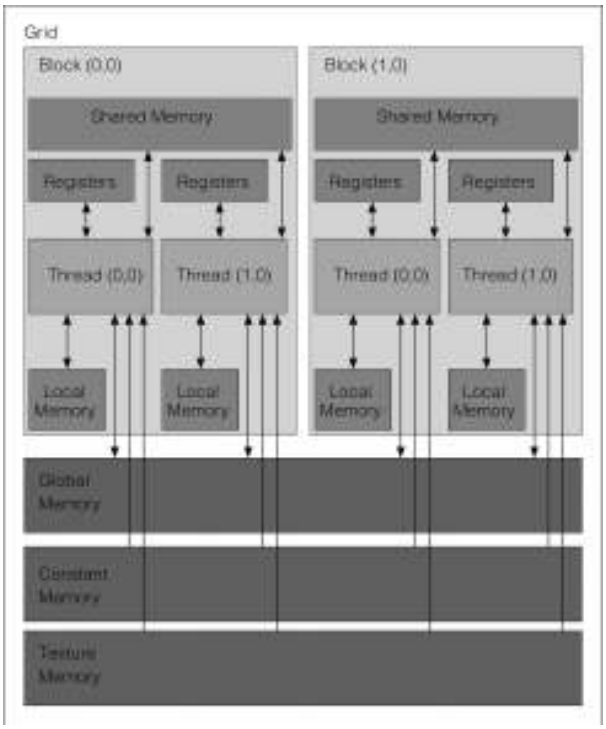

Figure 1: Memory model of the NVIDIA device [Source : NVIDIA web site]

Important Units of CUDA Architecture

A parallel computation used the abstraction of threads, blocks, and grids organized by CUDA which are -

Thread

The basic unit of CUDA architecture is a thread. Each thread runs on separate cores of multiprocessors and each thread can have a pair of Register memory for fast access. Threads are identified by threadIdx, which can be 1D, 2D or 3D. Indexes are used by every thread to access elements in an array such that the collection of all thread cooperatively processes the entire set of data.

Block

A block is a logical unit which contains multidimensional thread. Block is the group of threads and is identified by blockIdx. The GPU is a collection of multiprocessors (MPs), [Bernabé et. al., 2013] [Soroushnia et. al., 2014] where each multiprocessor responsible for handles one or more blocks in a grid. A block is never divided across multiple processorss.

Grid

It is a group of blocks. A complete Grid is handled by a single GPU. There is no synchronization between the blocks. A Grid is started in the synchronous form in the CPU, but there can be multiple Grids running at the same time.

PROBLEM IDENTIFICATION

The main resources of a computer system are memory and processor. Memory and processor both plays an important role in high-performance computing, when large amount of data sets used as input. These data sets requires large amount of memory. A single system is not able to fulfill the memory requirements. So multiprocessor or multicomputer systems are used. Multiprocessor system uses the concepts of shared memory and multicomputer uses the concepts of distributed memory (Non – shared).

When large amount of data sets used as input, calculation was not done in proper way, it takes garbage value. The main reason of the problem is cache storage organization and defect caused by mapping of elements of matrix on to single cache set instead of using the entire cache set. Over all degrade the performances of machine which increased execution time instead actual execution time. This paper presents a matrix multiplication problem on the GPU and CPU and comparing the execution time with the use of NVIDIA GeForce GT 525M machine.

SPECIFICATION

The testing platform requirements are as follows:-

Hardware Specification

Intel (R) core (TM) i3-2350M CPU @ 2.30 GHz

System memory:- 4GB(installed memory)

Testing Platform Specifications

Operating System: - windows7 (32-bit operating system)

Software used: - Microsoft visual studio 2010

Language used: - CUDA C

Version of CUDA: - CUDA Toolkit 6.5

GPU Specification

The table shows capabilities of a GPU which we have implemented and performing operation.

Table 1: NVIDIA GPU specification

Device name	NVIDIA GeForce GT 525 M
Compute capability	2.1
Amount of global memory	1024 MB
Number of µp	2 (48 CUDA Core)
Number of streaming prospectors cores	96 CUDA core
Texture fill rate	9.6 billion/second
Memory clock rate	900 MHz
Constant memory amount	65536 bytes
Processor clock tester	1200 MHz
Interface memory	DDR3
Interface width of memory	128 bits
Bandwidth of memory	28.8 GB/second
Shared memory amount per block	49152 bytes
Registers available per block no.	32768
Size of wrap	32
No. of max. threads per µp	1536
No. of threads per block max.	1024
Dimension size of a thread block<x,y,z> max.	1024, 1024, 64
Dimension size of a grid size<x,y,z> max.	65535, 65535, s65535
Texture alignment	512 bytes
Max memory pitch	2147483647 bytes

METHODOLOGY

In this paper the matrix of different size has been stored in file and this file has been used as input. The algorithm [Tiwari et. al., 2015] is as given below: -

Step 1. Input the file of matrix of different size to CPU

Step 2. The time recorder starts (t_s)

Step 3. CPU sends matrix data to GPU

Step 4. GPU receives the data and operation

Step 5. GPU distributes these among threads with scatter function.

Step 6. GPU performs their operations in parallel

Step 7. GPU collects the processed data with gather function.

Step 8. GPU returns the processed data to CPU

Step 9. CPU collects the processed data and produce the Result

Step 10. The time recorder stops (t_e)

The total elapsed time includes the computation time(t_{comp}) as well as total communication time (t_{comm}) which is calculated by :

Elapsed time = $t_e - t_s$ msec.

Here communication time is the time to spend in communication of data and computation time is the time to spend in calculation of data.

RESULTS

All the matrix multiplication operations are performed on predefined parameters as present in Table 1 and got the results which are tabulated in Table 2 and, Figure 2 shows the graphical view for the performance of Matrix Multiplication operation executed for many time and taken an average value for particular sets of all set of N*N size matrices.

From Figure 2, observe that when matrix size is small, the execution time for matrix multiplication problem on the GPU is more than that of execution time taken by CPU, but when we increase the size of the matrix, execution time taken by CPU is more than the time taken by GPU. The execution time of one or both parallel techniques is less than other, known as a non-shared implementation technique, and the other is known as shared memory tile implementation technique. The reason for time variation between CPU and GPU is data set transfer from CPU to GPU and then the resulting data transfer from GPU to CPU is considerable time as compared to the total execution time. When a small set of data take as input values for multiplication on GPU, the multiprocessor spends more time in transfer data compare to the computation time, meanwhile, the CPU can compute the result in less time for the small data set.

Table 2: CPU and GPU Average Execution Time

S. No.	Size of matrix	CPU Time (ms)	Non-shared memory GPU Time (ms)	Shared memory GPU Time (ms)
1	4*4	0.0026	0.0088	0.0114
2	8*8	0.004	0.0104	0.016356
3	16*16	0.023	0.013702	0.0196492
4	32*32	0.1442	0.0264908	0.0318104
5	64*64	0.424	0.12144	0.155468
6	128*128	2.337	0.75162	1.1218
7	256*256	6.748	2.8682	3.1012

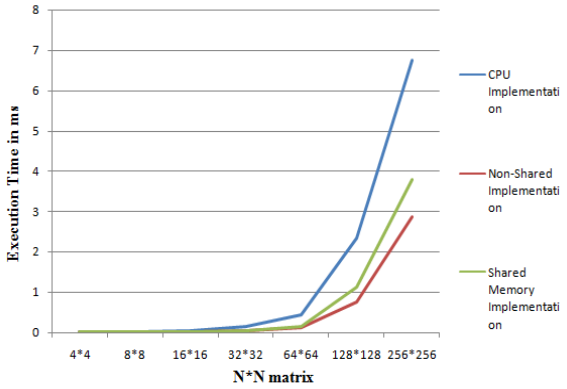

Figure 2: Comparison of Average Execution Time taken by CPU and GPU for N*N Matrix

S. No.	Size of matrix	Non-shared memory GPU Time / CPU Time (Speedup)	(Speedup) = Shared memory GPU Time / CPU Time
1	4*4	3.38	4.38
2	8*8	2.6	4.089
3	16*16	0.5957	0.8543
4	32*32	0.1837	0.2206
5	64*64	0.2864	0.3667
6	128*128	0.3216	0.48
7	256*256	0.425	0.4596

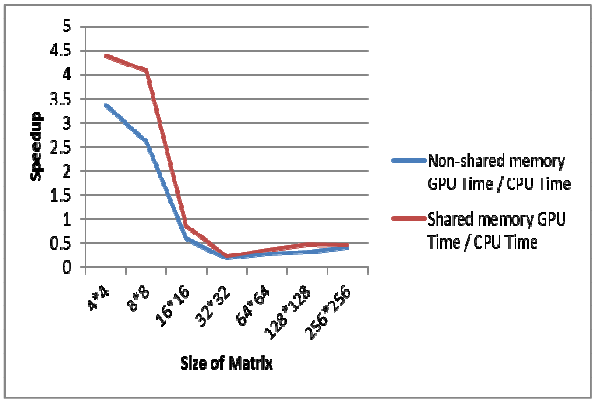

Figure 3: Comparison Chart

CONCLUSION

The performance is comparing between NVIDIA GeForce GT525M and Intel core processor in terms of execution time by using multiple techniques i.e. Simple Matrix Multiplication on Intel core processor (CPU), Non-shared memory Matrix Multiplication and Shared memory Tile Matrix Multiplication on NVIDIA GeForce GPU. Above results show the simulation for Matrix Multiplication problem on both the environment and obtained the results which are discussed and analyzed in the previous section.

In figure 3 the result shows that when the size of the matrix increases then the performance of non-shared parallel matrix multiplication technique is better than the shared memory matrix multiplication on GPU. The computation time for shared memory technique is more than the non-shared matrix multiplication for all data set in this device. In general the shared memory program has less elapsed time than the elapse time in non-shared program. But using shared memory does not necessarily reduce the processing time, and this much depends on the GPU architecture. As when system compare the results of shared and non-shared technique, our GPU device perform better in non-shared memory than the shared memory technique.

REFERENCES

Djinevski L., Arsenovski S., Ristov S. and Gusev M., 2013. "Performance Drawbacks for Matrix Multiplication using Set Associative Cache in GPU devices," in MIPRO 2013, pp. 193-198.

Sooknanan D.J. and Joshi A., 2016. "GPU Computing Using CUDA in the Deployment ofSmart Grids," in SAI Computing Conference 2016, IEEE, pp. 1260-1266.

Sartori J. and Kumar R., 2013. "Branch and Data Herding: Reducing Control and Memory Divergence for Error-Tolerant GPU Applications," IEEE Transactions on Multimedia, **15**(2): 279-290.

Shah M. and Patel V., 2014. "An Efficient Sparse Matrix Multiplication for the skewed matrix on GPU," in 14th International Conference on High-Performance Computing and Communications, IEEE, pp. 1301-1306.

Cui X., Chen Y. and Mei H., 2009. "Improving Performance of Matrix Multiplication and FFT on GPU," in 15th International Conference on Parallel and Distributed Systems 2009, IEEE, pp. 42-48.

Shah M., 2015. "Sparse Matrix Sparse Vector Multiplication -A Novel Approach," in 44th International Conference on Parallel Processing Workshops 2015, IEEE, pp. 67-73.

Ohshima S., Kise K., Katagiri T. and Yuba1 T., "Parallel Processing of Matrix Multiplication in a CPU and

GPU Heterogeneous Environment," In 7th International Meeting on High Performance Computing for Computational Science (VECPAR'06).

Ha S.W. and Han T.D., 2013. "A Scalable Work-Efficient and Depth-Optimal Parallel Scan for the GPGPU Environment," IEEE Transactions on Parallel and Distributed Systems, **24**(12): 2324-2333.

NVIDIA. https://developer.nvidia.com.

Lo S.H., Lee C.R., Kao Q.L., Chung I.H. and Chung Y.C., 2013. "Improving GPU Memory Performance with Artificial Barrier Synchronization," IEEE Transactions on Parallel and Distributed Systems.

Salim M., Akkirman A.O., Hidayetoglu M. and Gurel L., 2015. "Comparative Benchmarking: Matrix Multiplication on a Multicore Coprocessor and a GPU," in IEEE, pp. 38-39.

Anh N.Q., Fan R. and Wen Y., 2015. "Reducing Vector I/O for Faster GPU Sparse Matrix-Vector Multiplication," in 29th International Parallel and Distributed Processing Symposium, 2015, IEEE, pp. 1043-1052.

Eberhardt R. and Hoemmen M., 2016. "Optimization of Block Sparse Matrix-Vector Multiplication on Shared-Memory Parallel Architectures," in International Parallel and Distributed Processing Symposium Workshops 2016, IEEE, pp. 663-672.

Liu W. and Vinter B., 2014. "An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data," in 28th International Parallel & Distributed Processing Symposium 2014, IEEE, pp. 370-381.

Zha X. and Sahni S., 2013. "GPU-to-GPU and Host-to-Host Multi pattern String Matching on a GPU," IEEE Transaction On Computers, **62**(6): 1156-1169.

Barberis A., Danese G., Leporati F., Plaza A. and Torti E., 2013. "Real-Time Implementation of the Vertex Component Analysis Algorithm on GPUs," IEEE Geoscience and Remote Sensing Letters, **10**(2): 251-255.

Bernabé S., Sánchez S., Plaza A., López S., Benediktsson J.A. and Sarmiento R., 2013. "Hyper spectral Unmixing on GPUs and Multi-Core Processors: A Comparison," IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, **6**(3): 1386-1398.

Soroushnia S., Daneshtalab M., Plosila J., Pahikkala T. and Liljeberg P., 2014. "high performance pattern matching on heterogeneous platform", Journal of Integrative Bioinformatics, **11**(3):253.

Tiwari R., Sharma M. and Mehta K.K., 2015. "Dynamic Load Balancing in Parallel Processing using MPI Environment to Improve System Performance " International Journal of Advance Research in Computer Science and Software Engineering, **5**(6): 730 – 734.