Impact Factor (SJIF): 3.632

International Journal of Advance Research in Engineering, Science & Technology

e-ISSN: 2393-9877, p-ISSN: 2394-2444

Volume 3, Issue 11, November-2016

A Survey on Matrix Multiplication for GPU

Amit kumar singh[1], Rajesh Tiwari[2]

[1]*M.Tech. Scholar, CSE Department, Shri Shsnkaracharya Technical Campus Bhilai, Chhattisgarh*
[2]*Associate Professor, Shri Shsnkaracharya Technical Campus Bhilai, Chhattisgarh*

Abstract –Now a day sequential processing is certainly not sufficient for a large data computation in the area of computer science and technology. The need for high-performance computation is ever growing, even though certain problem sets remain within the area of high-performance computing with applications such as Weather Forecasting, Quantum Physics, and Climate Research etc. Within the commercial area of computation, NVIDIA has an architectural framework (NVIDIA CUDA)to harness the power of GPU's which was previously only been utilized for graphics application like 3D games, but now it has been used for certain types of high-performance computation. In this paper, we will take a critical look at different techniques of Matrix multiplication operation. This paper perform the Matrix multiplication problem with different implementation techniques, and the results has compared on the basis of execution time and find which technique is the most efficient approach for our problem set (matrix operation of n size matrices).

Keyword:-GPU, NVIDIA CUDA, Shared Memory, Tiling, Matrix Multiplication, SIMD.

I. INTRODUCTION

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. Computational mathematics, physics, economics are only some of the areas of their application. Matrix multiplication is a fundamental building block for scientific computing and is one of the most important approaches to understanding parallel programming in GPU [1][2].

The simultaneous use of more than one processor to execute a program is an example of SIMD (single instruction multiple data) process [3]. Ideally, the parallel processing makes a program run faster because there are more engines (CPUs) running [4]. In practice, it is often difficult to divide a program in such a way that separate CPUs can execute different portions without interfering with each other.

CUDA is a general purpose parallel architecture that utilizes a parallel compute engine in GPUs to thus carries out the complex computational problem in less time span than it would have if that same problem would have been executing on a CPU[5][6]. GPUs have been utilized within the graphic field for some time now and have now stepped into other fields that need high-performance computation. Fields such as Medical Imagining, fluid dynamics, and environmental science are some fields are seeking to utilize the potential power of GPUs to solve existing and current problems.

CUDA is a library provided by NVIDIA, it provides extended functionalities in C language by adding CUDA specific functions. Within this paper, we would take a look at the different optimization techniques Naïve matrix multiplication [7] on CPU, matrix multiplication on GPU using Shared and Non-Shared memory and increase floating portion by outer product in trying to optimize a N*N size matrix.

GPU and CUDA

CUDA (Compute Unified Device Architecture) is a library provided by NVIDIA to execute processes in parallel [8].This is an application programming interface (API) to help communicate 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 data after the calculation result is sent back to CPU [9].

GPU (Graphics Processing Unit) was primarily developed to fulfill the need of algorithms used in computer graphics. It has hundreds of cores which were 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 like device memory, shared memory, constant

cache, texture cache, and registers [10][12]. 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 several different memory spaces; we will discuss in brief to all types of memory organization and figure 1 shows the memory organization and basic units of CUDA model.

Global, local, texture, constant, shared and register memory .The only 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.

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 we have used for this implementation has 49152 Bytes per block Shared Memory.

In the GPU, *register* is fastest accessible memory present. 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 we have used for this implementation has 32768 per block registers.

Local memory has the same scope rules as register memory, but performs slower than register.

Data stored 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.

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

Important Units of CUDA Architecture:

CUDA organizes a parallel computation using the abstractions of threads, blocks, and grids which we will describe in brief.

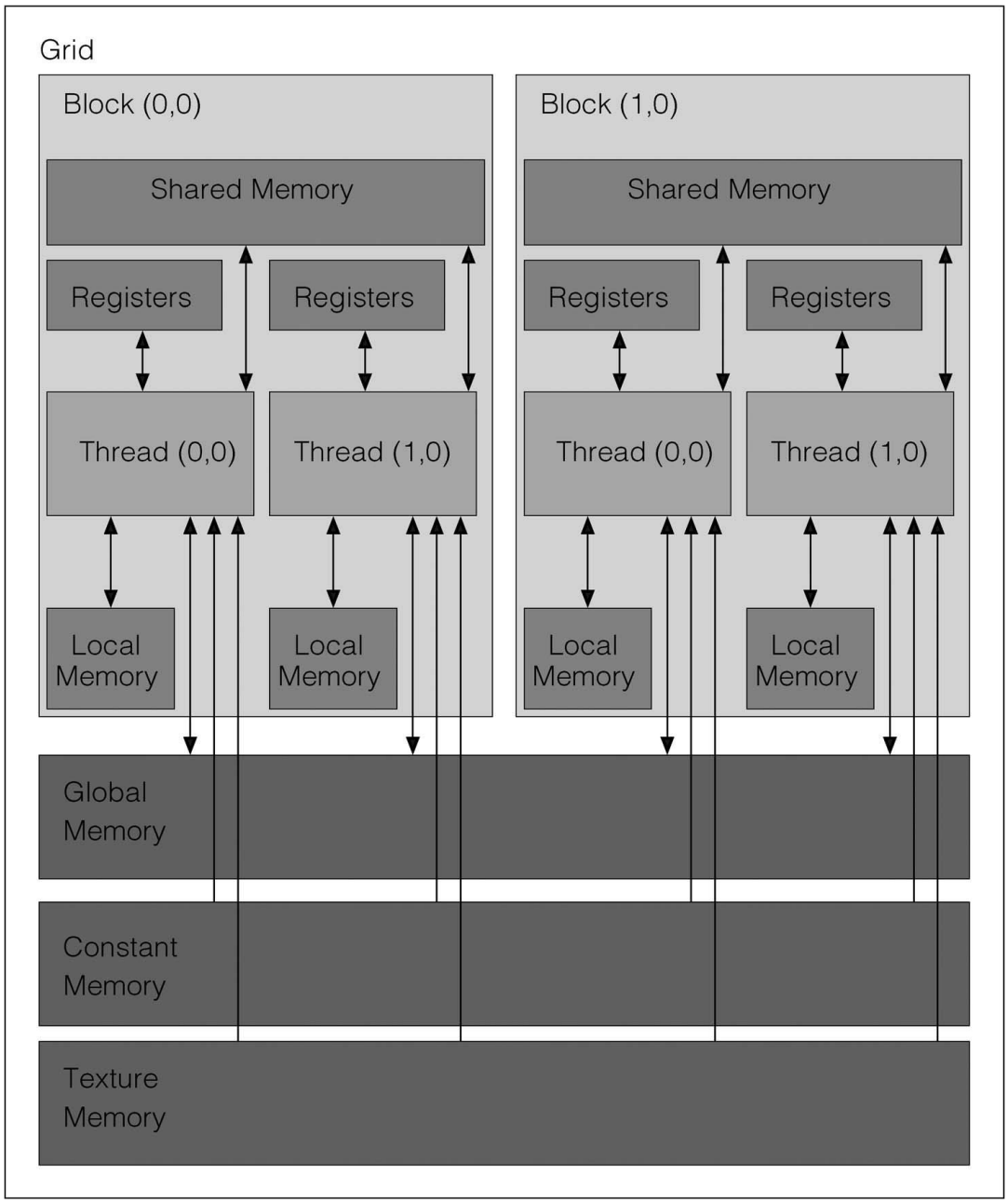

Figure 1: Memory model of the NVIDIA device

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. Every thread uses its index to access

elements in an array such that the collection of all thread cooperatively processes the entire data set.

A *block* is a logical unit which contains multidimensional thread. Block is the group of threads and is identified by blockIdx. The GPU is organized as a collection of multiprocessors (MPs), with each multiprocessor responsible for handling one or more blocks in a grid. A block is never divided across multiple MPs.

A *grid* is a group of blocks. An entire Grid is handled by a single GPU. There is no synchronization at all 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.

II. LITERATURE SURVEY

"Improving performance of Matrix Multiplication and FFT on GPU" shows enhancing the performance of single precision matrix-matrix multiplication subprogram (SGEMM of BLAS) and single-precision FFT using CUDA, which is matrix multiplication operation on the basis of computational-intensive and memory bandwidth.

Conclusion of the paper is that Xiang Cui et al. uses CUBLAS 1.0 and CUBLAS 2.0 and compares the Gigaflops performance of source code with largest problem size and got 5% improved performance of CUBLAS 2.0 than previous version [5].

"An efficient sparse Matrix Multiplication for skewed matrix on GPU" presents an algorithm for sparse matrix multiplication named "ALIGNED_COO" which is an extension to COO (coordinate) format to enhance the performance of large sparse matrix having skewed distribution of non-zero elements. ALIGNED_COO format helps to gain better performance without any extra memory overhead.

Conclusion of the paper is that M. Shah at el. try to improve the performance of mainly three factors of sparse matrix which is as load balancing, alignment and synchronization free distribution of work load on GPU. The performance of ALIGNED_COO is found better over other sparse formats for large set of sparse matrix [4].

"Reducing Vector I/O for Faster GPU Sparse Matrix-Vector Multiplication" show the majority of the bandwidth consumed by accesses to the vector I/O for most matrices and the problem is fully utilization of power of GPU. Two techniques used to significantly reduces the I/O for vector accesses, CUSP SpMV algorithm and yaSpmv algorithm.

Conclusion of the paper is that with the use of these two techniques we can decreases the I/O for vector accesses for GPU sparse matrix vector multiplication which forms a dense row from sparse ones, and partial sums, which compute sums using values in shared memory to reduce I/O in the next SpMV iteration and then full utilizing the power of GPU and get consistently better performance [12].

"Performance Drawbacks for Matrix Multiplication using Set Associative Cache in GPU Device" introducing drawbacks of performance of shared memory processors within the execution of simple matrix multiplication algorithm and, why drawbacks appears for specific problem sizes.

Conclusion of the paper is that the main reason is cache storage organization and drawbacks caused by mapping of matrix elements onto one cache set, instead of using the entire cache set. Set associative cache in GPU can seriously degrade the performance of GPU and can increases execution time [1].

"High Performance Pattern Matching on Heterogeneous Platform" introducing a pattern matching algorithm named as PFAC algorithm (parallel failure-lessAho-corasick algorithm) which is an efficient and extended approach of Aho-corasick (AC) algorithm with linear complexity and, compare the sequential and parallel approach.

Conclusion of the paper is that, when sequential version of the algorithm is used with single thread on CPU, single thread build state machine and start to search the input data character by character which is very slower approach but when using PFAC algorithm the memory efficiency of GPU version is very much more than sequential version due to removing the failure transition from state transition machine. The experimental result shows that PFAC algorithm is 15% speedup than Aho-corasick algorithm [18].

"Real-Time Implementation of the Vertex Component Analysis Algorithm on GPUs" presenting a new parallel implementation of the vertex component analysis (VCA) algorithm for spectral unmixing of remotely sensed hyper spectral data on commodity graphics processing units. There has four VCA algorithms use in the experiment, first develop a serial version then another three are parallel version algorithms, one using NVIDIA's CUDA, another is CUDA basic linear algebra subroutines library (CUBLAS), and the last is CUDA linear algebra library (CULA).

Conclusion of the paper is that, this implementation has tested on an NVIDIA Fermi GPU and is faster than the corresponding C version and the original Mat lab codes (both serial and parallel code). It concluded that CUBLAS parallel version VCA algorithm gives better performance than the rest of another parallel and serial version algorithm [16].

"Improving GPU Memory Performance with Artificial Barrier Synchronization" introducing an important mechanism for a block of threads to monitor data consistency on GPU, called Barrier Synchronization. This study provides a different viewpoint for barrier synchronization on GPU such as, adding barrier synchronization can improve the performance of some memory-intensive applications and preserve access data locality.

Conclusion of the paper is that the artificial barrier synchronization model relieves contention for the caches and DRAM system and preserve access data locality on the GPU [10].

III. PROBLEM IDENTIFICATION

After studied all the above papers the conclusion is that the basic problem is coming in memory section i.e. memory organization, when uses large amount of data as input, calculation is 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 and machine taking 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.

IV. SPECIFICATION

The testing platform information is as followed:-

CPU specification:-

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

System memory:- 4GB(installed memory)

Software environment of the testing platform:-

Software environment:- Windows7 (32-bit operating system)

Development tools:- Microsoft visual studio 2010

Development language: - CUDA c

CUDA version: - CUDA Toolkit 6.5

GPU specification:-

The table shows capabilities of a GPU which use for performing experiment.

Table 1: NVIDIA GPU specification

Device name	NVIDIA GeForce GT 525 M
Compute capability	2.1
Total amount of global memory	1024 MB
Number of multiprocessors	2(48 CUDA Core/MP)
Number of streaming prospectors cores	96 CUDA core
Texture fill rate	9.6 billion/second
Memory clock rate	900 MHz
Total amount of constant memory	65536 bytes
Processor clock tester	1200 MHz
Memory interface	DDR3
Memory interface width	128 bit
Memory bandwidth	28.8 GB/second
Total amount of shared memory per block	49152 bytes

Total no. of registers available per block	32768
Warp size	32
Max. no. of threads per multiprocessor	1536
Max. no. of threads per block	1024
Max. dimension size of a thread block<x,y,z>	1024,1024,64
Max. dimension size of a grid size<x,y,z>	65535,65535,65535
Texture alignment	512 bytes
Max memory pitch	2147483647 bytes

V. EXPECTED RESULT

Implementation of matrix multiplication problem is also test on an NVIDIA Fermi GPU, so the expectation is that the parallel version of CUDA code for GPU should performs better then the CPU version. Barrier synchronization function is used in implementation of paper, according to the conclusion of paper is that improve the computational performance of GPU when uses this function in source code. Barrier synchronization function uses in matrix multiplication algorithm for GPU version, so there should be enhanced elapsed time for GPU execution on this particular GPU machine. This paper, perform a matrix multiplication problem in sequential and parallel approach using three different techniques, CPU execution, Non-shared GPU execution and, shared memory GPU execution. Both technique of GPU execution is parallel approach and the CPU execution is sequential technique approach. After simulation process of execution of programs, comparing CPU and GPU performance as execution time of each data set or matrix size and expect that the CPU execution time for small set of matrix size is less than the GPU execution time of both parallel approaches. But when increase the matrix size, the execution time of CPU should be always more than that of GPU execution time. Because time spend for transferring data from/to CPU to GPU and vice versa on the GPU is negligible for large amount of data set or input data where as the calculation time for CPU is higher than GPU. Main reason behind this the GPU has number of hundred processors which works parallel instead of single processor as a CPU.

VI. REFERENCES

[1] L. Djinevski, S.Arsenovski,S. Ristov and M.Gusev, "Performance Drawbacks for Matrix Multiplicationusing Set Associative Cache in GPU devices," in MIPRO 2013, 20-24 May 2013, pp. 193-198.

[2] D. J. Sooknanan, A. Joshi, "GPU Computing Using CUDA in the Deployment ofSmart Grids," in SAI Computing Conference 2016, July 13-15, IEEE 2016, pp. 1260-1266.

[3] J. Sartori and R. Kumar, "Branch and Data Herding: Reducing Control and Memory Divergencefor Error-Tolerant GPU Applications," IEEE Transactions on Multimedia, Vol. 15, No. 2, February 2013, pp. 279-290.

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

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

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

[7] S. Ohshima, K. Kise, T. Katagiri, and T. Yuba1,"Parallel Processing of Matrix Multiplicationin a CPU and GPU Heterogeneous Environment,"

[8] S.W. Ha and T.D. Han, "A Scalable Work-Efficient and Depth-OptimalParallel Scan for the GPGPU Environment," IEEE Transactions on Parallel and Distributed Systems, Vol. 24, No. 12, December 2013, pp. 2324-2333.

[9] NVIDIA. https://developer.nvidia.com.

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

[11] M. Salim, A. O. Akkirman, M. Hidayetoglu, and L. Gurel,"Comparative Benchmarking: Matrix Multiplicationon a Multi-core Coprocessor and a GPU," in IEEE 2015, pp. 38-39.

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

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

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

[15] X. Zha and S.Sahni, "GPU-to-GPU and Host-to-HostMulti-pattern String Matching on a GPU," IEEE Transaction On Computers, Vol. 62, No. 6, June 2013, pp. 1156-1169.

[16] A. Barberis, G. Danese, F. Leporati, A. Plaza, and E. Torti, "Real-Time Implementation of the VertexComponent Analysis Algorithm on GPUs," IEEE Geoscience and Remote Sensing Letters, Vol. 10, No. 2, March 2013, pp. 251-255.

[17] S.Bernabé, S. Sánchez, A.Plaza,S. López, Member, J. A. Benediktsson, and R. Sarmiento, "HyperspectralUnmixing on GPUs and Multi-CoreProcessors: A Comparison," IEEE Journal Of Selected Topics In Applied Earth Observations And Remote Sensing, Vol. 6, No. 3, June 2013, pp. 1386-1398.

[18] S. Soroushnia, M. Daneshtalab, J. Plosila, T. Pahikkala and P. Liljeberg, "high performance pattern matching on heterogeneous platform" journal of integrative Bioinformatics 11(3):253, 2014.