Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 161 (2019) 398–405

Procedia Computer Science

www.elsevier.com/locate/procedia

The Fifth Information Systems International Conference 2019

Square Matrix Multiplication Using CUDA on GP-GU

Ali Olow Jimale[a,b], Fakhitah Ridzuan[a], Wan Mohd Nazmee Wan Zainon[a,*]

[a]*School of Computer Sciences, Universiti Sains Malaysia, Penang, 11800, Malaysia*
[b]*Faculty of Computing, SIMAD University, Mogadishu, Somalia*

Abstract

This paper focuses on matrix multiplication algorithm, particularly square parallel matrix multiplication using Computer Unified Device Architecture (CUDA) programming model with C programming language. Matrix multiplication is under the list of time-consuming problems that require s huge computational resources to improve its speedup. As many studies have shown, it is not easy to achieve high performance speedup in sequential matrix multiplication algorithm using larger input. The emphasis of this study is to propose a parallel algorithm to calculate the product of two square matrices with improved speedup performance compared to the sequential and OpenMP algorithms. In this research, biruni (super machine workstation) in the School of Computer Sciences, USM, Malaysia with General Purpose Graphics Processing Unit (GP-GU) was used to parallelize the matrix product algorithm. A comparison between parallel OpenMp versions and sequential algorithm with the proposed CUDA based algorithm of this research was carried out to evaluate the speedup performance of the proposed parallel CUDA based algorithm. The overall results show that CUDA based parallel matrix multiplication is approximately 400 times faster than sequential matrix multiplication and 4 times faster than OpenMp matrix multiplication algorithms, respectively. Therefore, the proposed parallel algorithm can help the researchers working with matrix multiplication application problems. It can also help mathematicians to easily calculate the product of any two matrices and obtain the result in a shorter time.

© 2019 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/)
Peer-review under responsibility of the scientific committee of The Fifth Information Systems International Conference 2019.

Keywords: CUDA; GPGU; parallelization; C language; Matrix multiplication

* Corresponding author. Tel.: +60-465-346-38.
E-mail address: nazmee@usm.my

1877-0509 © 2019 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/)
Peer-review under responsibility of the scientific committee of The Fifth Information Systems International Conference 2019.
10.1016/j.procs.2019.11.138

1. Introduction

The applications of matrix multiplication in real life problems are increasing daily. They are used to analyze weather patterns [1], perform linear algebra operations [2], and recognize human faces for security purpose [3]. It was also applied in autonomous vehicles and robotics [4], process control and graph analysis [5], scientific, and engineering [6]. On other hand, matrix multiplication operations are time-consuming problems [4]. Sequential processing methods, which require more processing time and storage, were traditionally used to perform matrix multiplication, but a Matrix sequential algorithm is not efficient with larger input data [5]. Several Parallel programming algorithms have been proposed to deal with big data challenge in matrix multiplication problem [7]. This paper presents CUDA based parallel solution for square matrix multiplication problem using GP-GU shared memory architecture.

The term matrix (multiplication/ product), denoted C, is the multiplication operation that produces a new matrix from two matrices denoted A and B; it can be mathematically defined as in Eq. 1, where i, j, and k are the elements of the matrices [8].

.

$$(\boldsymbol{C})\boldsymbol{ij} = \sum_{\boldsymbol{k=1}}^{\boldsymbol{m}} \boldsymbol{A_{ik}B_{kj}} \tag{1}$$

In order to multiply two matrices, it must have an equal size of columns, as shown in Eq. 2, where the m's denote the same columns. Eq. 2 also shows the size of the product matrix; it can be observed that the product matrix's size always equals to the size of first matrix's row times the size of second matrix's column. In matrix multiplication, the order of multiplying two matrices does not matte; it is associative operation which produces the same results if AB or BA multiplication is performed [9].

$$(nxm)(mxp) = (nxp) \tag{2}$$

To clarify the idea, let us assume that A= $\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix}$, B = $\begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix}$, then

AB= $\begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} & a_{11}b_{13} + a_{12}b_{23} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} & a_{21}b_{13} + a_{22}b_{23} \\ a_{31}b_{11} + a_{32}b_{21} & a_{32}b_{12} + a_{32}b_{22} & a_{31}b_{13} + a_{32}b_{23} \end{pmatrix}$ and

BA= $\begin{pmatrix} b_{11}a_{11} + b_{12}a_{21} + b_{13}a_{31} & b_{11}a_{12} + b_{12}a_{22} + b_{13}a_{32} \\ b_{21}a_{11} + b_{22}a_{21} + b_{23}a_{31} & b_{21}a_{12} + b_{22}a_{22} + b_{23}a_{32} \end{pmatrix}$ were generating identical results.

Matrix operations, such as matrix multiplication, have numerous applications in science and technology [10]. They are basic to algebra operations [2], graph and number theory, and digital control and signal processing [11]. All of these applications require high ranked computational throughputs. Parallel processing is viable option for today's real-life applications [11]. Many researchers have proposed various CUDA-based matrix multiplication solutions for two main reasons: to teach how CUDA is working or to parallelize matrix multiplication operation. For example,[12] have used hypergraph partitioning technique with CUDA and GPU to parallelize matrix multiplication. The researchers of this study partitioned the matrices into rows and columns to implement parallelization, and found that the hypergraph partitioning technique can be applied to parallelize matrix-multiplication in shared memory architecture; however, this technique does not consider efficiency programming. Moreover, [13] have proposed memory saving matrix multiplication algorithm for NVIDIA Pascal GPUs with high performance. They have managed to reduce memory usage using grouping techniques and utilizing shared memory efficiently. These scientists have achieved 4.3 times speedup in single precision and 4.4 speedup in double precision. In addition to that, [14] proposed an efficient matrix

multiplication algorithm on CUDA GPU. They focused on memory bandwidth efficiency and storage format to achieve 2.5 times speedup. Different authors such as [15] have proposed CUDA based matrix multiplication algorithms to introduce the CUDA programming model. Based on the readings, and reviews made of related studies, a parallel matrix multiplication algorithm was proposed in this research.

2. Design of the parallel program

In general, the proposed solution in this research contains four functions: the main function, cpu_matrix_mult, omp_matrix_mult, and gpu_square_matrix_mult kernel function. Furthermore, input, output and CUDA header files are included at the top of the four functions, and 16 block sizes are defined. The main function is responsible for running the program. Inside the main function, memory allocation and deallocation of both host and device variables occurs. As well as copying data from host to device variables and vice versa; asking the user to input matrices sizes; transferring results to the host; automatically generating input array elements randomly; invoking CPU, OpenMp, and kernel function; and calculating the time taken by both sequential and parallel algorithms.

Due to the length of the main function, the researchers of this work decided to explain the main function's code. At the beginning, the code inside the main functions declares input variables, prompts the user to type matrices size, and holds it in the input variables. After that, it allocates memory for the host variables in the RAM, name h_a for matrix 1, h_b for matrix 2, h_c for the GPU result, and h_cc for the CPU result. Then, it initializes both input matrices (h_a, h_b) using two nested for-loops per each randomly, declares timing measuring variables with their starting and ending events, and starts to count the execution time of GPU version. Next, it allocates memory space for the device variables on the device, name d_a for matrix 1, d_b for matrix 2, d_c for the result. Also, inside the main function, there is code that copies the data of matrices h_a and h_b from host to device memory matrices: d_a and d_b, code that calculates GPU Grid, and Block dimensions, and Calls the kernel function. Also, the main function contains code that transfers results from device to host: d_c to h_c, synchronizes the threads, stops calculating time on GPU and prints its elapsed time. At the bottom of the main function, the code starts the CPU version runtime calculation, invokes CPU function, Stops calculating runtime on the CPU, and prints its elapsed time. Finally, the main function code starts the OpenMp version runtime calculation, invokes OpenMp function, stops calculating runtime on CPU, prints its elapsed time, validates the results computed by GPU, and frees the allocated memory spaces.

Second, cpu_matrix_mult function is a naïve function that performs matrix multiplication on the host (CPU) seriously by computing the matrix multiplication one after another. It loops through the indices from i=1 through m, j=1 to k, and h=1 to n using three nested loops. Fig. 1. shows the pseudocode of serial naive matrix multiplication. As the figure shows, three nested for-loops have been used to perform the matrix multiplication. However, the runtime speed of this algorithm is slow compared to the CUDA-based parallel algorithm.

Algorithm: Naïve Matrix multiplication on CPU

Input: Square Matrix size n = {10,20,50,200,500, 1000,…}, this will be used to decide the square matrices dimensions

Output: time taken to multiply two matrices of the given size on CPU, and Speed up

```
void cpu_matrix_mult (int *h_a, int *h_b, int *h_result, int m, int n, int k) {
For i=0 to n-1 do
  {
    For j=0 to n-1 do
    {
      int tmp = 0.0;
      For h=0 to n-1 do
      {
        tmp += h_a [I * n + h] * h_b [h * k + j];
      }
      h_result [I * k + j] = tmp;
    }
  }
}
```

Fig. 1. cpu_matrix_mult function.

gpu_square_matrix_mult is a kernel function that computes the matrix multiplication on the GPU device to improve the speedup. It employs multiple threads to multiply the input matrices elements simultaneously. In addition to that, this parallel algorithm, divides the matrices to be multiplied into blocks, and each block multiplication will be performed by different threads simultaneously. The algorithm uses a CUDA-based technique called tile to increase the computation to the memory ration. Moreover, the algorithm applies __syncthreads () function as a block level barrier. Since all threads are performing the multiplications in different blocks simultaneous, and there is dependency in the results, it is necessary to synchronize the threads, therefore, __syncthreads function stops the thread (s) until all other threads finish their computation and reach the barrier. Fig. 2 demonstrates CUDA-Based parallel matrix multiplication.

The parallel matrix multiplication takes the matrix sizes as an input argument. After dividing the matrices dimension into 16 blocks, the algorithm will automatically employ x times y threads to perform the multiplication simultaneously. After finishing the computation, the algorithm will show the execution time of the algorithm measured in milliseconds. Fig. 3 shows a flow chart for the proposed CUDA-Based parallel program. In Fig. 3, it can be observed that the follow chart contains the design of parallel program that can parallelizes the matrix multiplication.

Algorithm: CUDA-BASED Parallel Matrix multiplication on GPU

Input: Square Matrix size n ={10,20,50,200,500, 1000,…}, this will be used to decide the square matrices dimensions

Output: time taken to multiply two matrices of the given size on both GPU and CPU, and Speed up

```
1. // Create the kernel function to employ the matrix multiplication on GPU as followings: -
2.__global__ void gpu_square_matrix_mult (int *d_a, int *d_b, int *d_result, int n) {
  3. __shared__ int tile_a[BLOCK_SIZE] [BLOCK_SIZE]; // Increase Computation-to-Memory Ratio
  4. __shared__ int tile_b[BLOCK_SIZE] [BLOCK_SIZE]; // Increase Computation-to-Memory Ratio
  5.int row = blockIdx.y * BLOCK_SIZE + threadIdx.y;   // Calculate the row index of tail_a and tail_b
  6. int col = blockIdx.x * BLOCK_SIZE + threadIdx.x;    // Calculate the column index of tail_a and
tail_b
  7. int tmp, idx; // Declare Temporary variable and ID index of threads
  9. for (sub = 0 to gridDim.x) {// Initialize the ID index
  10.    idx = row * n + sub * BLOCK_SIZE + threadIdx.x;
  11.    if (n is not divisible by BLOCK_SIZE)
  12.       tile_a [threadIdx.y] [threadIdx.x] = 0;// Set tile_a equal to zero;
  13.    else
  14.       tile_a [threadIdx.y] [threadIdx.x] = d_a[idx];
  15.    idx = (sub * BLOCK_SIZE + threadIdx.y) * n + col;
  16.    if (n is not divisible by BLOCK_SIZE)
  17.          tile_b [threadIdx.y][threadIdx.x] = 0;// Set tile_b equal to zero
  18.       else
  19.          tile_b [threadIdx.y] [threadIdx.x] = d_b[idx];
  20.    __syncthreads () ;//use __syncthreads () function as a barrier. If 1 thread reaches this, it must wait
until all the threads reach
  21. for (k = 0  to  BLOCK_SIZE; ++k)
  22.       tmp += tile_a[threadIdx.y] [k] * tile_b[k][threadIdx.x];//each thread multiples one block
  23.  __syncthreads (); // block level barrier}
  24. if (row and column do not exceed the array sizes)
  25.       d_result [row * n + col] = tmp;
26. }// End of the algorithm
```

Fig. 2. CUDA-Based parallel matrix multiplication.

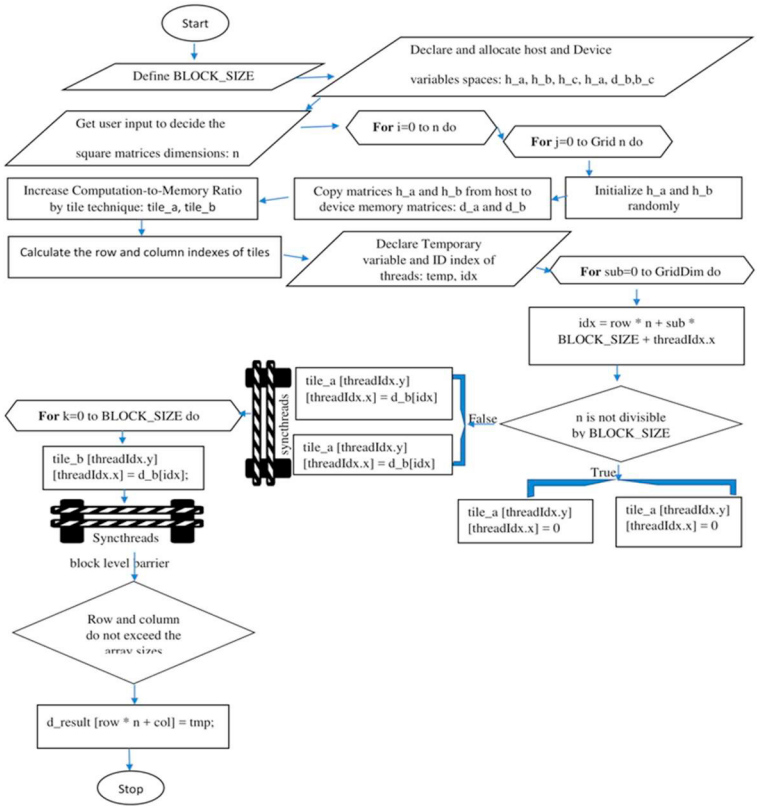

Fig. 3. Flow chart of program design.

3. Experimental design

To know how the proposed algorithm can faster the computation of matrix multiplication, two different 2D square matrices with 10 x 10, 20 x 20, 50 x 50, 200 x 200, 500 x 500, and 1000 x 1000 sizes per two matrices to be multiplied have been tested by using 100, 400, 2500, 40000, 250000, and 1000000 threads, respectively. The calculation of actual threads is carried out by squaring the input size (n); e.g. if n=10, the actual number of threads equals to 100. Each The matrices are divided into block (s), and each block have 256 threads; e.g. if the input size is 20, we divide 20 by 16 blocks, which is almost equals 2, then, by squaring 2, we got 4 blocks; then multiply 4 blocks by 256 threads to get the maximum threads of that multiplication which equals 1024 threads. In this case, only 400 threads will perform the calculation and the rest will be idle. Table 1 shows the matrices sizes multiplied in this research, number of blocks per input, actual threads, and maximum threads.

Table 1. Matrix sizes and threads.

Matrix size	# of blocks	Actual number of the threads	Maximum number of blocks
10x10	1	100	256
20x20	4	400	1024
50x50	16	2500	4096
200x200	169	40000	43264
500x500	1024	250000	262144
1000x1000	3960	1000000	1016064

4. Performance analysis

This section explains the analysis carried out to evaluate the performance of the proposed CUDA based parallel matrix multiplication algorithm. In this research, sequential functionally was also designed to calculate the speedup of the parallel version. In order to find an accurate runtime average results, each experiment was run at least 10 times, and the average results was recorded. To validate and confirm the accuracy of the proposed CUDA based matrix multiplication result, it was compared and confirmed that the CPU matrix multiplication result is identical with GPU matrix multiplication result. First, the time taken by the sequential algorithm was calculated and recorded in a Microsoft Excel spreadsheet table. Fig.4. (a) shows the sequential runtime results measured in milliseconds. The sequential results have confirmed that naïve matrix multiplication algorithm is not good choice as the input size is getting bigger and bigger (Big Data). To overcome the drawbacks of sequential algorithm, CUDA based parallel matrix multiplication algorithm was proposed and its runtime was calculated to check how fast the parallel algorithm could be carried out compared with the sequential. Fig.4. (b) display the parallel runtime in milliseconds.

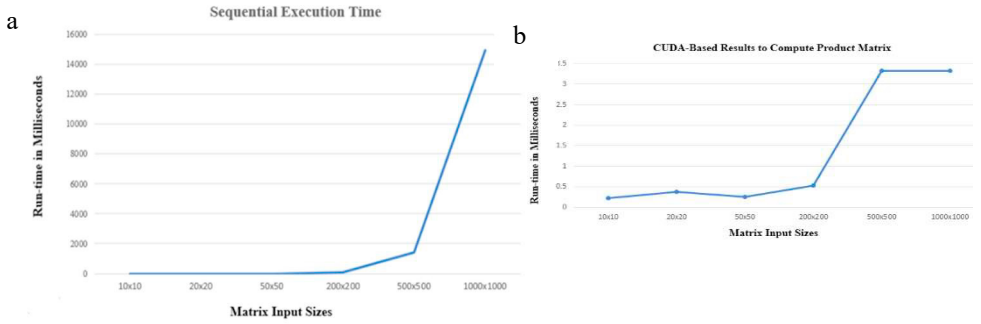

Fig. 4. (a) Sequential runtime; (b) Parallel runtime.

From the results of CUDA-based runtime, it can be observed that the parallel algorithm is only good practice with a big data. After calculating both sequential and parallel algorithms runtimes, a comparison between the two and an OpenMP version is conducted. The aim of this comparison was to determine which parallel programming model (CUDA and OpenMP) achieved better performance in terms of runtime speed in matrix multiplication. To simplify the analysis, a main table (Table 2) containing the three algorithm (sequential, OpenMP and CUDA) runtime results was created.

Table 2. Runtime to compute product matrix.

Matrix size/ programing model	10x10	20x20	50x50	200x200	500x500	1000x1000
Sequential runtime in milliseconds	0.013136	0.0965696	1.384032	88.4987755	1391.242531	14934.1547
OpenMP runtime in milliseconds	0.1517626	0.2649897	1.359649	34.1778403	387.5622438	4013.944263
CUDA runtime in milliseconds	0.2295712	0.3680182	0.2459904	0.5291488	3.3252672	3.3252672

According to Table 2, the sequential algorithm achieved better performance in small input sizes (10x10 and 20x20) compared with both parallel algorithms. In this case, both of the parallel algorithms are slower than the serial algorithm due to not lack of computation to be done by the several threads (100 threads in 10x10 and 400 threads in 20 x20) created and the overhead associated with creating and controlling the threads. As Fig. 5 shows, the time taken by both CUDA and OpenMp algorithms is more than the sequential runtime in small input size.

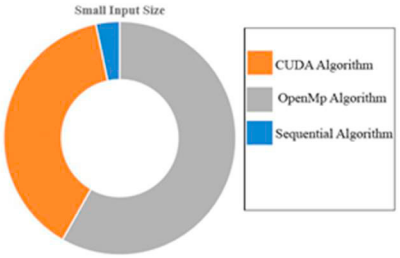

Fig. 5. Small Input Size Result Analysis.

Similarly, the OpenMP algorithm achieved better performance than CUDA algorithm in 10x10 and 20x20 input sizes. This is due to the time taken by the CUDA kernel function to create idle threads (only 100 and 400 threads have been used out of 256 and 1024 threads). As the input size increases (50x50 onwards), the CUDA- based parallel algorithm beats the other algorithms. It achieves higher speedup compared with sequential and OpenMp algorithms. This means that CUDA based matrix multiplication algorithm is applicable in big sizes. Similarly, OpenMp algorithm is better than sequential algorithm in large input size matrices. Fig. 6 displays a computational comparison graph between the three algorithms for big input size.

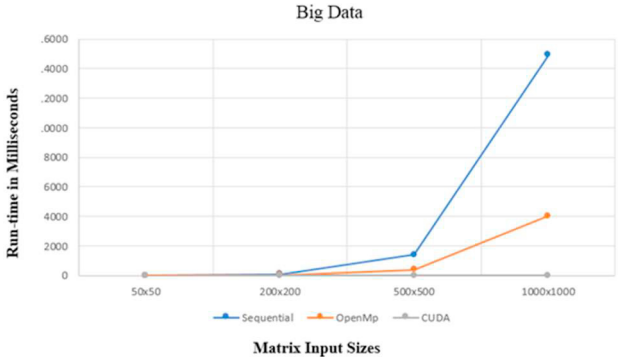

Fig. 6. Big Data Computational Graph.

5. Results and discussion

This research offers the chance to investigate a method to compare the performance of popular parallel programming models for shared memory architecture with sequential algorithm using matrix multiplication. Six different sized matrixes have been multiplied to see the runtime of the sequential algorithm. To compare and calculate the speedup of the parallel algorithms (CUDA-Based algorithm and OpenMp algorithm), the same number of threads has been employed to multiply between same number of matrices input sizes. To get the speedup, the sequential time was divided by the parallel time. Table 3 show the speedup of the proposed parallel program in 10, 20, 50, 200, 500 and 1000 input sizes with 100, 400, 2500, 40000, 250000, and 1000000 threads.

Table 3. Speedup.

Input size	10x10	20x20	50x50	200x200	500x500	1000x1000
Threads	100	400	2500	4000	2500	1000000
OpenMp	0.08655624	0.364427749	1.017933305	2.589361256	3.589726691	3.720568528
CUDA	0.5722063	0.37500182	5.63624272	167.258468	418.3873555	832.84464

From the speedup results, it can be seen that the runtime speed of the first two inputs are less than 1, which indicates that the parallel speedup is less than the sequential speedup due to small input sizes. There are two main reasons reduced the speed of the mentioned input sizes. First, the data associated with the threads are not enough big as providing a lot of processing elements, therefore the available resources cannot be utilized. Second, there is an overhead generated while adding more threads in the computation, which slows down the speed.

6. Conclusion

This paper has presented Square Matrix Multiplication using CUDA on GPGU. There are other two algorithms (Sequential algorithm and OpenMp algorithm) used in this research to calculate the speedup of the proposed algorithm and compare it with the CUAD algorithm. A performance analysis between the three matrix multiplication algorithms are performed. Comparative analysis has shown that the CUDA-based parallel matrix multiplication algorithm runtime speed is better than the sequential and OpenMP matrix multiplication algorithms speed. The results acknowledged the capability of CUDA to reduce the processing time needed in such problems that require big computational resources. This work can be further improved by using hybrid technique (CUDA with OpenMP) to increase the speedup of matrix multiplication. This work was not able to achieve 4.4 speedup due to limited time. The proposed solution in this research can be used as a general guide for information systems practitioners to speed up data processing and data distribution. They can select several suitable processors for their data processing and distributing phase to reduce the overall processing time.

References

[1] Yegnanarayanan, V. (2013) “An Application of Matrix Multiplication.” *Resonance* **18**: 368-377.
[2] Krishnan, Krishnan, and Jarek Nieplocha. (2004) “Optimizing Parallel Multiplication Operation for Rectangular and Transposed Matrices”, in *Proceedings, Tenth International Conference on Parallel and Distributed Systems, ICPADS 2004.* pp. 257-266.
[3] Sotiropoulos, Ioannis. and Ioannis Papaefstathiou. (2009) “A Fast Parallel Matrix Multiplication Reconfigurable Unit Utilized in Face Recognitions Systems”, in *International Conference on Field Programmable Logic and Application*. pp. 276-281.
[4] Idris, Mohd. Yamani Idna, Noorzaily Mohamed Noor, Emran Mohd. Tamil, Zaidi Razak, and Hamzah Arof. (2010) “Parallel Matrix Multiplication Design for Monocular SLAM”, in *Fourth Asia International Conference on Mathematical/Analytical Modelling and Computer Simulation (AMS)*. pp. 492-497.
[5] Zheng, Jian-Hua, Liang-Jie Zhang, Rong Zhu, Ke Ning, and Dong Liu. (2013) “Parallel Matrix Multiplication Algorithm Based On Vector Linear Combination Using MapReduce”, in *IEEE Ninth World Congress on Services (SERVICES)*. pp. 193-200.
[6] Kim, Minwoo, Yong J. Jang, and Won W. Ro. (2011) “Parallel Transpose of Matrix Multiplication Based On The Tiling Algorithms”, in *IEEE 54th International Midwest Symposium on Circuits and Systems (MWSCAS)*. pp.1-3.
[7] Li, Hui Li, Geoffrey Fox, and Judy Qiu. (2012) “Performance Model for Parallel Matrix Multiplication With Dryad: Dataflow Graph Runtime”, in *Second International Conference on Cloud and Green Computing 2012 (CGC2012)*. pp. 675-683.
[8] DataChant. (2017) “Excel Matrix Multiplication – Replacing MMULT With Power Query.” Available from: https://datachant.com/2016/06/01/excel-matrix-multiplication-replacing-mmult-with-power-query/. [Accessed 13rd November 2017].
[9] Weisstein, Eric. (2017) “Matrix Multiplication.” Available from: http://mathworld.wolfram.com/MatrixMultiplication.html
[10] Deng, Jixia. (2014) “Application of Matrix in Computer Science and Technology.” *Applied Mechanics & Materials*.
[11] Piedra, Rose Maria. (1994) “*A Parallel Approach For Matrix Multiplication On The TMS320c4x dsp*.” *Texas Instruments SPRA107*. pp. 1-24.
[12] Murni, A. Bustamam, Ernastuti, T. Handhika, and D. Kerami. (2017) “Hypergraph Partitioning Implementation for Parallelizing Matrix-Vector Multiplication Using CUDA GPU-based Parallel Computing”, in *AIP Conference Proceedings*.
[13] Nagasaka, Yusuke, Akira Nukada, and Satoshi Matsuoka. (2017) “High-Performance and Memory-Saving Sparse General Matrix-Matrix Multiplication for NVIDIA Pascal GPU”, in *46th International Conference on Parallel Processing (ICPP2017)*. pp. 101-110.
[14] Bell, Nathan, and Michael Garland. (2008) “*Efficient sparse Matrix-Vector Multiplication on CUDA*.” N*vidia Technical Report NVR-2008-004*, Nvidia Corporation.
[15] Hochberg, Robert. (2012) “Matrix Multiplication with CUDA-A Basic Introduction to the CUDA Programming Model.” Available from: http://www.shodor.org/media/content/petascale/materials/UPModules/matrixMultiplication/moduleDocument.pdf. [Accessed 11st August 2012].