PERFORMANCE APPRAISAL OF AN INTEL CORE i7 CPU AND AN NVIDIA QUADRO K420 GPU

[1]G. O. ASORONYE*, [2]L. N. OSUJI & [3]O. J. BOSEDE

*Email: asorgay@gmail.com

[1, 2, 3]Department of Computer Engineering Technology

Akanu Ibiam Federal Polytechnic, Unwana

Afikpo, Ebonyi State

ABSTRACT

Nowadays, demand for High Performance Computing (HPC) is on the rise even though certain problem sets remain within the domains of Super High Performance Computing (SHPC) with the advent of applications such as Big Data analytics, weather forecasting, cognitive computing, quantum physics and climate research, etc. Hence, it behoves that sequential processing is undoubtedly no longer sufficient. NVIDIA, within the commercial realm of computation, has proposed a Compute Unified Device Architecture (CUDA) framework to harness the power of Graphics Processing Units (GPUs) for parallel computing which before now were only been utilized for Graphics Application. Recently, they are used for certain types of high performance computation. In this paper, a comparative analysis of sequential implementation on an intel core i7 Central Processing Unit (CPU) and parallel implementation on an NVIDIA Quadro K420 GPU for dense matrix-matrix multiplication was carried out. A look at the performances on different n square matrices using various grids of variable number of threads per block showed the GPU with its multiple cores provided significant reductions compared to the CPU with speedups of 114, 115, 270 and 941 for 128, 256, 512 and 1024 square matrices respectively. The authors recommended that the use of GPUs for computations with data reuse void of data dependencies should be exploited by scientists and engineers in the building of applications and systems.

Keywords: GPU, CPU, Dense Matrix, Parallel computing, CUDA

INTRODUCTION

Recent GPU architectures support various applications with classifications that are divergent. The scientific and industrial use of GPUs has evolved overtime into the computing mainstream (Mei & Chu, 2017). However, according to Asoronye et al., (2021) there is a general structure to all the programs that are applicative ported to the GPUs. For large computations, the GPGPU delivers on demand amount of compute capability to the application. Matrix multiplication applications are finding more use in real life problems. They are used in weather patterns analysis, linear algebra operations, facial recognition for security purposes, etc. (Ridzuan et al., 2019). Although matrix multiplication operations are time-consuming they have also found use in autonomous vehicles and robotics, process control and graph analysis (Zheng et al., 2013). The product from the multiplication operation from two matrices A and B will produce a new matrix denoted as C. This can be defined mathematically as shown in equation 1, where i, j and k are the elements of the matrices '(Ray et al., 2016).

$$c_{IJ}=\sum_{k=1}^{m} A_{ik}B_{kj} \quad \text{.....(1)}$$

For two matrices to be multiplied, both must have columns of equal size as seen in equation 2. Where the same columns are represented by the k's, the product matrix size is also shown. The size of the first matrix row multiplied by the size of the second matrix's column is equal to the product matrix size as clearly depicted. The order for the multiplication of two matrices is of no effect as it is associative operations that produce the same results if AB or BA multiplication is performed (Somasekhar & Karthikeyan, 2017).

$$(i \times k)(k \times j) = (i \times j) \quad \text{.....(2)}$$

All matrix applications require high ranked computational throughputs. Parallel processing is a viable option for today's real life applications (Ridzuan et al., 2019). Many researchers have proposed various matrix multiplication solutions using CUDA basically for two major reasons, which are to demonstrate matrix operation on CUDA and matrix parallelization operation '(Cui et al., 2009; Kutzkov et al., 2012; Ray et al., 2016). An NVIDIA Pascal GPU memory saving matrix multiplication algorithm was proposed by "Nagasaka et al., (2017). They were able to use both grouping technique and shared memory efficiently to reduce memory usage, thereby achieving a 4.3 x speedup. In this work, the researchers used varied grid of blocks and threads to achieve an optimized dense matrix-matrix multiplication.

Experimental setup

The sequential and parallel implementations of the matrix-matrix multiplication were performed on a heterogeneous computer with GPU interconnected with the host (CPU). CUDA Toolkit v9.0 was used, which primarily consisted of the NVIDIA C compiler, debugger and the NVIDIA profiler. Visual studio 2017 community edition was the Integrated Development Environment (IDE) for the CUDA C

integration. CudaEvent_Timer() function was used to record all GPU device timing, while the standard C clock() function was used to measure all CPU timings.

Design of the parallel program

The proposed solution in this work contains three functions: the main, cpu_mm and gpu_mm kernel function. In addition, the input, output and CUDA header files are included at the start of the three functions. The responsibility of the main function is the running of the program. Within the main function block, memory allocation and deallocation of the host and device are carried out. The copying of data from the host to the device and vice-versa also takes place in the main function block. Other computations carried out in the main function includes copying of data from host to device variables and vice versa, taking matrix input sizes from user, result transfer to the host, random generation of input array, CPU and kernel function invocation and computation time calculation for both sequential and parallel algorithms. The researchers due to the main function's length, explained the main function's code. From the start, input variables in the codes are declared, the user is prompted to enter matrix size. Next, memory is allocated for the host variables, h_a and h_b for the square matrices, h_c for the GPU result and h_cc for CPU result. Both matrices are initialized using two nested for-loops randomly, the starting and ending events timer variables are declared, with the execution start time of the GPU activated. Memory allocation for the variables of the device is then declared as d_a and d_b for the square matrices and d_c for the result. Also, the main function accommodates the codes that copies data from the host to the device memory, calculate the GPU grid and block dimensions and calls the kernel function. The codes for transferring results from the device to host d_c, threads synchronizations are in the main function, time calculation on the GPU stops and is printed out. At the ending of the main function, the code for CPU function is invoked, CPU runtime is computed and printed out. Finally, the main function code validates the results computed by the GPU and frees the allocated memory spaces.

The pseudo code for the sequential implementation of the dense matrix-matrix multiplication using CPU is shown below:

1. Declare all parameters
2. Enter the row and column of matrix A
3. Enter the row and column of matrix B
4. Enter the elements of matrix A
5. Enter the elements of matrix B
6. Print out the elements of matrix A in matrix form
7. Print out the elements of matrix B in matrix form
8. Set a loop up to row
9. Set an inner loop up to the column
10. Set another inner loop up to the column
11. Start timer
12. Multiply matrix A and matrix B

store result in matrix C
13. Print final matrix
14. Stop timer

The pseudo code for the parallel implementation of the dense matrix-matrix multiplication using GPU is shown below:

1. Declare all parameters
2. Allocate host memory for matrix A, matrix B and matrix C
3. Initialize input samples in host memory
4. Allocate device memory for matrix A, matrix B and matrix C
5. Copy host memory into device memory
6. Create and start timer
7. Execute the kernel
8. Stop and record time and destroy timer
9. Copy result from devise to host
10. Free device memory
11. Free host memory

Results and discussion

The time measurements of the various *n* squared matrices were first taken from the sequential implementation then the time measurements from the parallel implementation was taken. Comparisons of the execution time from the sequential and parallel implementations were made.

Sequential execution time measurements

The sequential execution times for the various *n* squared dense matrix-matrix multiplications are presented in Table 1. The plot of the execution times for CPU against the various matrix orders is shown in Figure 1 below.

Table 1: Sequential Dense Matrix-Matrix multiplication execution time

Matrix order	Element size	Time taken for Sequential execution (µs)
4,096	64 X 64	19,400
16,384	128 X 128	46,800
65,536	256 X 256	101,400
262,144	512 X 512	472,200
1,048,576	1024 X 1024	3,291,600

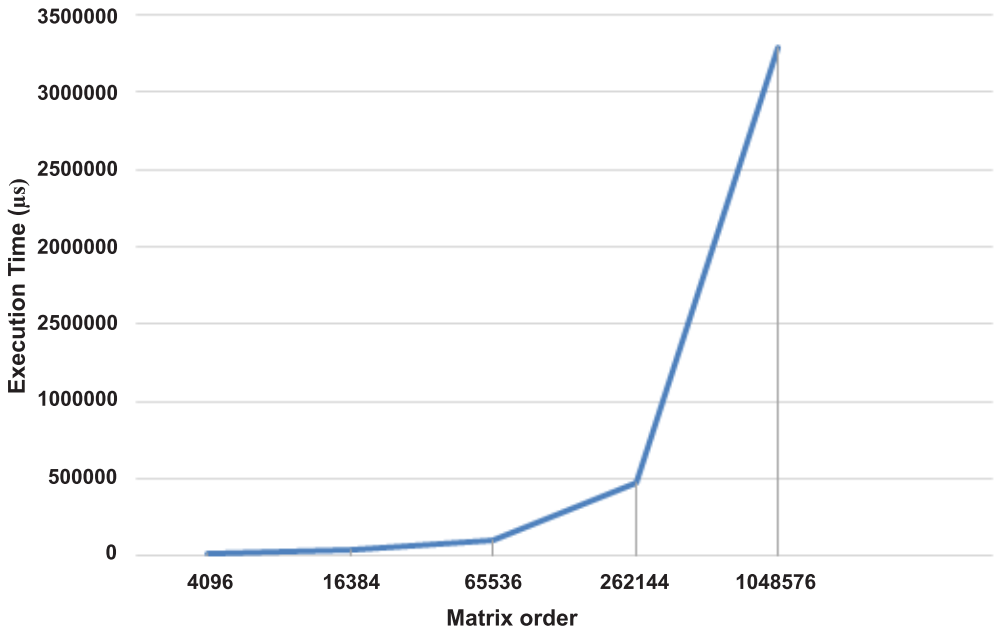

Figure 1: Plot of sequential execution time against matrix order.

Parallel execution time measurements

The parallel execution times for the various *n* squared dense matrix-matrix multiplications using various grids of variable number of threads per block as a technique for optimizing performance are presented in Table 2. The plot of the optimized execution times for GPU against the various matrix orders is shown in Figure 2 below.

Table 2: Sequential vs Parallel Matrix multiplication execution time

Matrix order	Element size	Grid	Time taken for parallel execution (µs)
4096	64 X 64	<<<32,1>>>	206
16384	128 X 128	<<<16,8>>>	412
65536	256 X 256	<<<8,32>>>	879
262144	512 X 512	<<<4,128>>>	1752
1048576	1024 X 1024	<<<2,512>>>	3497

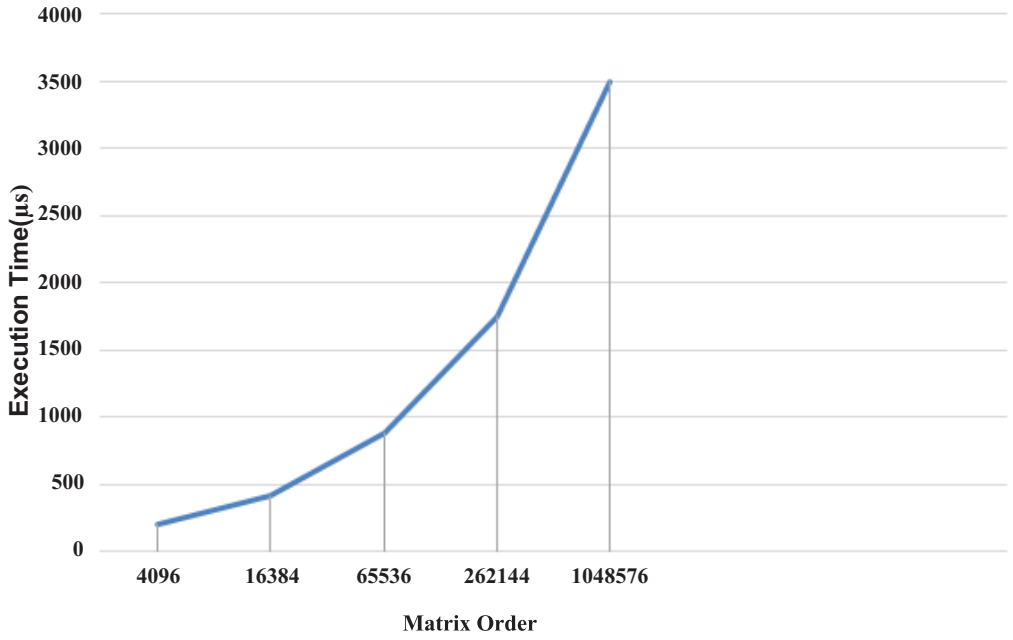

Figure 2: Plot of Parallel execution time against matrix order.

Comparison of parallel and sequential *n* square matrix-matrix multiplication execution times

The comparison of the parallel implementation of the n square matrix-matrix multiplication using the varied grids in Table 2 and the sequential implementation of the *n* square matrix-matrix multiplication execution times in Table 1 are presented in Table 3 and the plot of the parallel and sequential execution times against dense matrix order is presented in Figure 3.

Table 3: Sequential vs Parallel Dense Matrix-Matrix multiplication execution times

Matrix order	Element size	Time taken for Sequential execution (µs)	Time taken for parallel execution (µs)
4096	64 X 64	19400	206
16384	128 X 128	46800	412
65536	256 X 256	101400	879
262144	512 X 512	472200	1752
1048576	1024 X 1024	3291600	3497

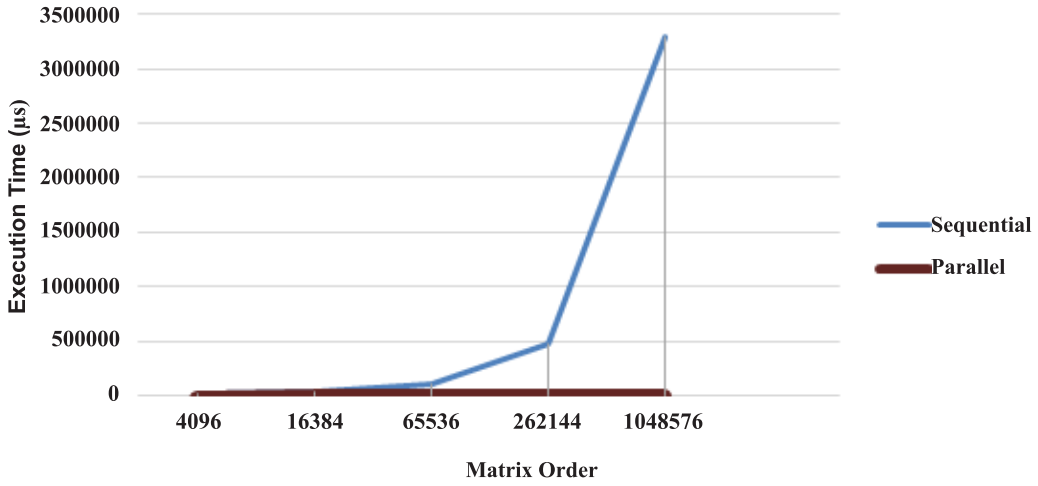

Figure 3: Plot of the parallel vs sequential execution times against dense matrix order

Achieved speedup computation

The achieved speedup presented in Table 4 was computed by dividing the sequential execution time with the parallel execution time for the various n square dense matrix-matrix multiplication. The plot for the various speedups gotten from the ratio of the sequential execution time to the parallel execution time is presented in Figure 4.

Table 4: Speedup obtained

Matrix order	Element size	Speedup (S) = Ts/Tp
4096	64 X 64	94
16384	128 X 128	114
65536	256 X 256	115
262144	512 X 512	270
1048576	1024 X 1024	941

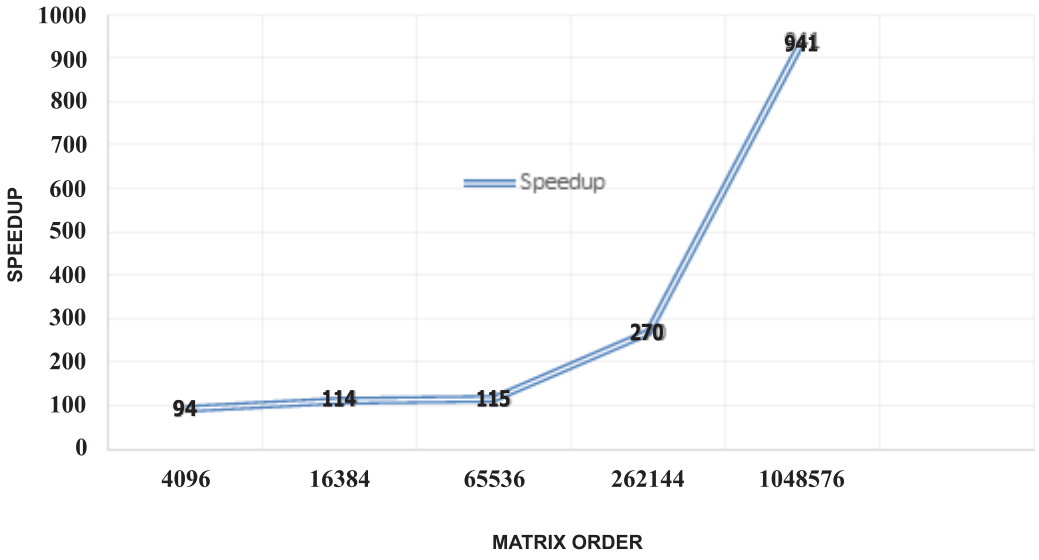

Figure 4: Plot of Speedup against the various matrix orders

Conclusion and recommendations

This research, investigated the comparison of dense matrix-matrix multiplication performance for parallel implementation on a GPU and sequential implementation on a CPU. Comparative analysis from the results obtained showed that the CUDA based parallel *n* squared matrix-matrix multiplication algorithm runtime speed outperformed the sequential *n* squared matrix-matrix multiplication algorithm implemented on the CPU. The results acknowledged the capability of CUDA using varied grid technique to reduce the processing time needed in such problems that require enormous computational resources. With speedup of 1000x attained, it is recommended that problems that provide for data reuse can be efficiently solved using GPUs. Also computations void of data dependencies can be parallelized since better speedups are guaranteed.

REFERENCES

Asoronye, G. O., Onyibe, C. O., Oko, G. E., & Obi, D. (2021). Single input multiple threads architecture optimization using CUDA. Paper presented at International Conference on Resource Extraction for Sustainable Development in a Post COVID-19 Era., held at Alex Ekwueme Federal University, Ndufu-Alike Ikwo, Ebonyi State, Nigeria.

Cui, X., Chen, Y., & Mei, H. (2009). Improving performance of matrix multiplication and FFT on GPU. Proceedings of the International conference on parallel and distributed systems - ICPADS, 4248. https://doi.org/10.1109/ICPADS.2009.8

Kutzkov, K., Campagna, A., & Pagh, R. (2012). On parallelizing matrix multiplication by the column-row method . Society for Industrial and Applied Mathematics, 122132.

Mei, X., & Chu, X. (2017). Dissecting GPU memory hierarchy through microbenchmarking. IEEE transactions on parallel and distributed systems, 28(1), 7286. https://doi.org/10.1109/TPDS.2016.2549523

Nagasaka, Y., Nukada, A., & Matsuoka, S. (2017). High-performance and memory-saving sparse general matrix-matrix multiplication for NVIDIA Pascal GPU. 46th International Conference on Parallel Processing, 101110. https://doi.org/10.1109/ICPP.2017.19

Ray, U., Hazra, T. K., & Ray, U. K. (2016). Matrix multiplication using Strassens Algorithm on CPU & GPU. International Journal of Computer Sciences and Engineering, 4(10), 99105.

Ridzuan, F., Mohd, W., Wan, N., Olow, A., & Sciences, C. (2019). Square matrix multiplication using CUDA on GP-GU. Procedia Computer Science, 161, 398405. https://doi.org/10.1016/j.procs.2019.11.138

Somasekhar, G., & Karthikeyan, K. (2017). Fast matrix multiplication with Big Sparse Data. CYBERNETICS AND INFORMATION TECHNOLOGIES , 17(1), 1630. https://doi.org/10.1515/cait-2017-0002

Zheng, J., Zhang, L., Zhu, R., Ning, K., & Liu, D. (2013). Parallel matrix multiplication algorithm based on Vector Linear Combination using MapReduce b ¦. 2013 IEEE Ninth World Congress on Services. https://doi.org/10.1109/SERVICES.2013.67