Matrix Multiplication Analysis on Sequential and Parallel Computation using CUDA

Robertus Hudi
Computer Science Department
Universitas Pelita Harapan
Tangerang, Indonesia
hudirobert21@gmail.com

Alessandro Luiz Kartika
Computer Science Department
Universitas Pelita Harapan
Tangerang, Indonesia
sandro.kartika@gmail.com

Dave Joshua Marcell
Computer Science Department
Universitas Pelita Harapan
Tangerang, Indonesia
djoshua449@gmail.com

Winston Renatan
Computer Science Department
Universitas Pelita Harapan
Tangerang, Indonesia
winstonrenatan@gmail.com

2022 1st International Conference on Technology Innovation and Its Applications (ICTIIA) | 978-1-6654-8826-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICTIIA54654.2022.9935904

***Abstract*—This paper aimed to implement both sequential and parallel implementations using CUDA on matrix multiplication to see the differences and effects of it, followed by an analysis of the result. We used the algorithm as that will be elaborated more on the paper, here we would like to generally compare its memory consumption and run time. It is found out that parallel implementation runs faster on an average of 31.23 compared to sequential implementation running the same task. This faster result on parallel programming comes with a trade-off that it consumes more memory rather than sequential implementation. Optimum threads to be used in parallel programming is also needed to be found, here we try to find it with trial-and-error testing. Further research would involve more complex parallel programming implementation and a more controlled testing environment.**

Keywords*—Matrix Multiplication, Sequential, Parallel Computing, CUDA, NVIDIA

I. Introduction

Parallel processing is the division of process into different parts, which performed concurrently by different processors in a computer [1]. Parallel programming is a concept of using two or more processors to complete a task. Parallel programming comes with various benefits from solving larger problems, doing things faster in a more reasonable time, and more cases can be finished [2]. Set side by side with CPU, GPU is taking into consideration to work with parallel computations as it suits more. GPU nowadays have considerably developed from recent years not just for gaming as its early purpose, with the extend of General-purpose computations on GPUs (GPGPU), as one of them is NVIDIA CUDA [3].

In memory hierarchy, on parallel program execution the threads will access data from some memory spaces. Each thread will have both access to local and global memory. This will make memory consumption on parallel computing consumes more rather than serial, because there are a lot of bandwidth usage that happens upon memory transfers [4].

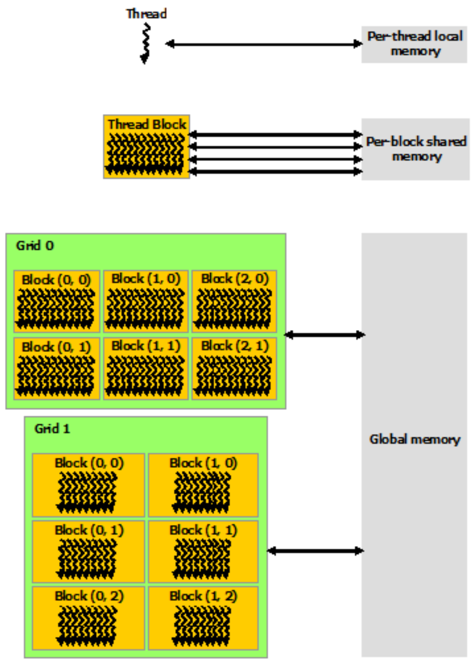


Fig 1.1. Memory Hierarchy [4]

Compute Unified Device Architecture or better known as CUDA is a platform for developer to perform parallel computing developed by NVIDIA on graphical processing units (GPUs). CUDA enables computing to be speed up in a dramatic manner leveraging the power of GPUs a machine has. Developing in CUDA can be done in popular languages such as C, C++, Python, etc [5]. Analyzing tool is also provided using the CUDA Profiling Tools to investigate the details of our program [6].

Here what we are going to analyze is the usage of parallel programming using CUDA compared to the sequential programming in solving matrix multiplication. Matrix multiplication will be done in a square matrix (NxN matrices), which then the run time and memory consumption will be recorded and compared. Both sequential and parallel solutions will be developed in C++ programming language. The machine used for running all the solutions is HP Pavilion Gaming Laptop 15-dk0xxx, with the details as such: Intel® Core™ i7-9750H Processor @ 2.60GHz (12CPUs), 8192MB RAM, and NVIDIA® GeForce® GTX 1650 Graphics (4 GB GDDR5 dedicated).

II. System Specification

We have done several methods on calculating and solving the matrix multiplication to obtain algorithm with think suits the best and fastest running time compared to the other. The goal is to compute the product matrix C, where C = AB of NxN matrices.

A. *Sequential Algorithm*

As we want to have the matrices as large as possible, it is a problem when storing the numbers in an array. As an array have certain limitation, it is not possible to use fixed multi-dimensional array. So in order to tackle this problem we proposed to use a dynamic multi-dimensional array as shown below. The following function have been tested and able to store up to N=8192.

```
int** create_matrix(int rows, int cols) {
        int** mat = new int* [rows];
        for (int i = 0; i < rows; ++i) {
                mat[i] = new int[cols]();
        }

        return mat;
}
```

After matrix is created, it will then be filled up on another function with random number between 1 and 10. The range was determined that way, so it is easy to verify the result. After all, the point is to check on speed of execution and its memory consumption. The following is the algorithm build for the sequential programming to do the matrix multiplication, in this case we are going to do it on square matrix, but not limited to square matrices if there are several added codes before running the following process.

```
for(int i = 0; i < rowsA; ++i) {
        for(int j = 0; j < colsB; ++j) {
                for(int k = 0; k < colsA; ++k) {
                        matC[i][j] += matA[i][k] * matB[k][j];
                }
        }
}
```

The algorithm above uses iterative method and calculated sequentially the rows of matrix C, the resultant matrix. For every *i* iteration a row of matrix C is created. We can see the illustration on Figure 2.1. To make sure the result is correct, we have verified it to an online matrix calculator up to 64x64 matrix multiplication.

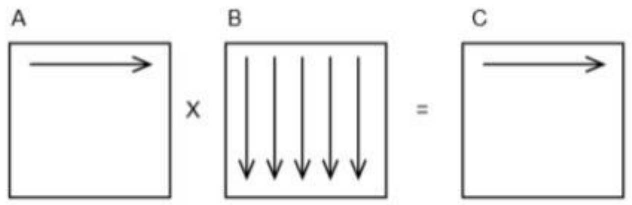


Fig 2.1. Algorithm Illustration [7]

B. *Parallel Algorithm*

Identical to the Sequential Algorithm purpose and goal, which is to compute matrix multiplication as large as possible. The code is modified from Nick's GitHub repository [8]. First of all, the thing that we need to do is to create a kernel that functions to calculate the matrix multiplication leveraging block and thread as below.

```
__global__ void matMul(int* a, int* b, int* c, int ordo) {
        int row = blockIdx.y * blockDim.y + threadIdx.y;
        int col = blockIdx.x * blockDim.x + threadIdx.x;
        c[row * ordo + col] = 0;
        for (int i = 0; i < ordo; i++) {
                c[row*ordo+col] += a[row*ordo+i] * b[i*ordo+col];
        }
}
```

Afterward, following the finishing of creating matrix multiplication kernel or grid, the next step is to give N matrix size and the thread size consecutively. The thread size cannot pass our maximum *Warp Size* which can be discovered in the *deviceQuery* sample program from CUDA samples. In the machine here, the maximum thread to be used is 32. Subsequently, we can calculate the value for block using the formula of *(ordo+thread-1)/thread*. We then proceed with declaring the matrix and fill it with random numbers as given below.

```
generate(matrixA.begin(), matrixA.end(), []() {return rand() %
10+1;});
generate(matrixB.begin(), matrixB.end(), []() {return rand() %
10+1;});
```

The next step to work with is to allocate the device memory together with moving the data or value inside the matrix from the CPU to the GPU. Go on with calling the matrix multiplication kernel that was made before. The last thing to work with is to transfer the matrix multiplication result from GPU back to the CPU and free the memory which have been used.

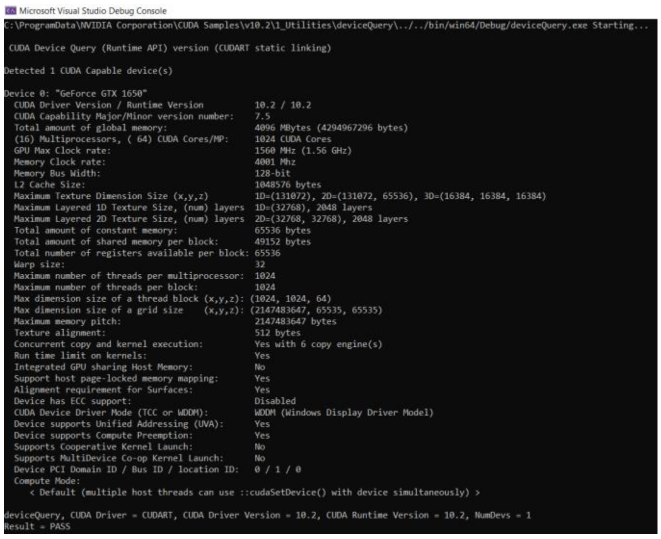
```
cudaMemcpy(matrixAA, matrixA.data(), bytes,
cudaMemcpyHostToDevice);
cudaMemcpy(matrixBB, matrixB.data(), bytes,
cudaMemcpyHostToDevice);
matMul <<<block, thread >>> (matrixAA, matrixBB,
HasilMatrixAB, ordo);

cudaMemcpy(HasilMatrix.data(), HasilMatrixAB, bytes,
cudaMemcpyDeviceToHost);
cudaFree(matrixAA);
cudaFree(matrixBB);
cudaFree(HasilMatrixAB);
```

III. Hypothesis

A. *Grid and Block*

Definitions of grid and block are interconnected one another. Block is a programming abstraction that represents a group of thread, which can be executed either in a serial or parallel way. Grid have the same concept, but it is formed by group of blocks which can be executed in just one kernel [4].

Besides, the following shows the results for the machine *deviceQuery* which explain lot more information about our GPU and its CUDA processing capability. Some of the information shown are *Total amount of global memory*, *Multiprocessors and CUDA Cores/Multiprocessors*, *Memory and GPU max clock rate*, *Maximum number of threads per multiprocessor and block*, *Max dimension size of a thread block and grid size*, etc. At first, the authors thought that the maximum capable thread to be used in the parallel algorithm will be equal to 1024 according to *Maximum number of threads per block*. Eventually, on reality the capable thread being used is as on stated in the *Warp Size* as the authors tried to put in the value accordingly to the given code.



```
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v10.2\1_Utilities\deviceQuery\../../bin/win64/Debug/deviceQuery.exe Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 1650"
  CUDA Driver Version / Runtime Version          10.2 / 10.2
  CUDA Capability Major/Minor version number:    7.5
  Total amount of global memory:                 4096 MBytes (4294967296 bytes)
  (16) Multiprocessors, ( 64) CUDA Cores/MP:     1024 CUDA Cores
  GPU Max Clock rate:                            1560 MHz (1.56 GHz)
  Memory Clock rate:                             4001 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 1048576 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1024
  Maximum number 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): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 6 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  CUDA Device Driver Mode (TCC or WDDM):         WDDM (Windows Display Driver Model)
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 10.2, CUDA Runtime Version = 10.2, NumDevs = 1
Result = PASS
```

Fig 3.1. Device Query

B. *Sample Program*

The sample program we tried on is addition on grid and block [9], the first thing that we create is a function that has this ability.

```
_global_ void arradd(int* md, int* nd, int* pd) {
        int myid = blockIdx.x * blockDim.x + threadIdx.x;
        pd[myid] = md[myid] + nd[myid];
        printf("Block Number: %d Thread number : %d.\n",
          blockIdx.x, threadIdx.x);
}
```

Afterwards, we can start to declare variables that we needed to work with in the main function. Continue the process with *cudaMalloc* that functions to allocate memory to GPU. Together with using *cudaMemcpy*, we could copy the data in array from CPU to the memory in GPU.

```
int size = 2000 * sizeof(int);
int m[2000], n[2000], p[2000], * md, * nd, * pd;
int i = 0;

for (i = 0; i < 2000; i++) {
        m[i] = 5;
        n[i] = 3;
        p[i] = 0;
}

cudaMalloc(&md, size);
cudaMemcpy(md, m, size, cudaMemcpyHostToDevice);
cudaMalloc(&nd, size);
cudaMemcpy(nd, n, size, cudaMemcpyHostToDevice);
cudaMalloc(&pd, size);
```

Next thing to work with is declaring grid and block dimension, whereas in the code uses *dim3* which is an integer vector and uses *Block* and *Thread*. After creating grid and block dimension we use *arradd* function, a kernel that has we created before in CUDA. This function will add *m* and *n* then put the results in array *pd* according to the size of grid and block dimension declared. After finishing the process, we can free the memory as we have done in parallel algorithm.

```
dim3 Block(4);
dim3 Thread(5);
arradd << < Block, Thread >> > (md, nd, pd);
cudaMemcpy(p, pd, size, cudaMemcpyDeviceToHost);
cudaFree(md);
cudaFree(nd);
cudaFree(pd);

for (int i = 0; i < 20; i++) {
        printf("\t%d", p[i]);
}
```

Here on the sample program, we learned that *Block* and *Thread* affect the data being performed which is only equals to *Block x Thread*, we think that if the resource is not allocated then the process cannot be done. Besides, the running *Block* is not may always be in order. For example, *Block 3* can run first than *Block 0*.

C. *Execution Time and Memory Consumption Hypothesis*

As on another case we have tried to implement sequential matrix multiplication on C++, we think that it might take a long time to execute even on small matrices. We predict that the memory consumed is more or less the product of NxN which one of the number will represent 4 bytes [10], with the total memory consumption is N x N x 4bytes. We see that from the sequential algorithm the largest cost is on the computation to fill up the resulting matrix, which is a quadratic time of $O(n^3)$.

On parallel implementation, we think that it will always be faster than the sequential algorithm as it leverages GPU not just the CPU. Thread will also come to play a role in the execution time of parallel programming, as we think that the larger the thread is the faster it will execute. On parallel implementation memory, we guess that it will just be more or less two times the memory needed for sequential one. The guess came from that one is used in CPU while the other is consumed on GPU.

IV. Result and Analysis

After the project was done completely, finally we can evaluate the results of each algorithm on its run time and memory consumption. The testing method is described in each sub chapter correspondingly, as it uses different methodology to run on sequential and parallel implementation.

Please be informed that all the calculations and results provided may not be fully accurate as there may be technical errors and many other things to consider going on in the machine (e.g. running other programs together, plugged in to electricity, etc.)

A. *Sequential Implementation*

To obtain the running time of sequential implementation and in order to get the best and most accurate result, for each NxN matrices it is being executed for 30 times. To do it automatically, we put all the algorithm of matrix multiplication described before on chapter two to a function that accepts two parameters, which is its row and column. The function also contains the code below to record time.

```
omp_get_wtime();
```

Meanwhile, for N=8192 matrices it is only executed 7 times due to the time limitation. The trial did not proceed to N=16384, as on that phase there is an unhandled exception error given with the detail below.

```
std::bad_alloc at memory location 0x008FF4B0.
```

From our analysis it is caused by the boundaries given from C++ programming language. As of the memory being used from, the machine is still capable of storing much larger data. The CPU is also being checked and is not used until even half of its power yet.

Doing the execution on main and other function does not give a significant run time difference on the matrix multiplication. Thus, after running the function 30 times using a for loop. The result of average run time is shown below in a table. N.B. n/a means that it is not available as the answer cannot be provided (more or less equal to 0).

TABLE I.
SEQUENTIAL RUN TIME

N	*Time in second*
2, 4, 8, 16	n/a
32	1×10^{-4}
64	9×10^{-4}
128	6.999×10^{-3}
256	5.723×10^{-2}
512	5.400×10^{-1}
1024	5.814×10^{0}
2048	7.108×10^{1}
4096	6.27×10^{2}
8192	4.520×10^{3}

On the other hand, the recording of memory consumption used the default profiler given on Visual Studio 2019. The number displayed on the table is the peak of the memory consumption. While on the process it is actually have several steps especially when filling up matrix a, b, and c which have significant increases on memory consumption. The result is recorded as below.

TABLE II.
SEQUENTIAL MEMORY CONSUMPTION

N	*Memory in MB*
2, 4, 8, 16, 32, 64	n/a
128	1.4×10^{0}
256	2.2×10^{0}
512	4.8×10^{0}
1024	1.49×10^{1}
2048	5.59×10^{1}
4096	1.94×10^{2}
8192	7.725×10^{2}

On the table it is shown that from N=2 until N=64, the memory being consumed is less than 1MB. We conclude that way because it is that even when a 2x2 matrices, it only needs roughly 16bytes (assume each int needs 4bytes). The result cannot be provided as it requires a program to run for one second or more to see diagnostic details on Visual Studio 2019.

B. Parallel Implementation

Corresponding to the sequential implementation, here we will also discuss the result of run time and memory consumption of parallel algorithm implementation on matrix multiplication. Unlike sequential programming, we now need to determine the number of threads to use for parallel programming. The following are the tables of result of calculating NxN matrices using different number of threads.

TABLE III.
THREADS EFFECT ON RUN TIME (IN SECOND)

N/Thread	*1024*	*2048*	*4096*	*8192*
2	2.48×10^{0}	9.59×10^{0}	6.37×10^{1}	4.92×10^{2}
4	1.78×10^{0}	4.02×10^{0}	1.93×10^{1}	1.34×10^{2}
8	1.64×10^{0}	2.64×10^{0}	7.99×10^{0}	4.30×10^{1}
16	1.55×10^{0}	2.45×10^{0}	7.95×10^{0}	4.25×10^{1}
32	1.49×10^{0}	2.56×10^{0}	7.83×10^{0}	4.25×10^{1}

So, what we predict upfront on hypothesis is true that threads are affecting run time of a parallel implementation. It is important to use optimal threads as it is significant and can reduce time up to an average of 73.36% (comparing the slowest and the fastest). We also found out that for some reason the optimal thread on the machine and algorithm is 16 threads on some cases. There is also a stagnant level around threads 8, 16, and 32 compared to using 2 or 4 threads with significant differences.

There are two columns to describe the run time in parallel implementation. The "Time in second" column is the product from Visual Studio 2019 profiler. While the "NVProf" column comes from using CUDA Event API. Where on the first few line we will create initializer for the counter and start to record the time as well, as the code below [11].

```
cudaEvent_t start, stop;
float milliseconds=0;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);
// Parallel Algorithm Here
cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&milliseconds, start, stop);
```

The results will be printed on the variable milliseconds, as it is also recorded on milliseconds instead of second. The results will be shown on the following table, where the milliseconds have been converted to seconds. Where all the time taken is using 32 threads on the execution.

TABLE IV.
PARALLEL RUN TIME

N	*Time in second*	*NVProf in second*
2, 4, 8, 16, 32, 64	n/a	n/a
128	1.185×10^{0}	5×10^{-4}
256	1.423×10^{0}	1.6×10^{-3}
512	1.425×10^{0}	1.1×10^{-2}
1024	1.487×10^{0}	8.25×10^{-2}
2048	2.56×10^{0}	5.506×10^{-1}
4096	7.831×10^{0}	3.950×10^{0}
8192	4.248×10^{1}	3.134×10^{1}
16384	2.943×10^{2}	2.537×10^{2}

It is quite obvious that the parallel algorithm runs faster for matrix multiplication rather than the sequential one. The run time can be easily noticeable especially on bigger N. When N≤256, the CUDA role is not so significant compared to the total time. On larger N, it is shown that CUDA plays a big role as on N=16384 it consumes 85,99% of the total time. We think that it may be caused that on smaller N, the computing power on CUDA is not really needed. On the other hand, larger N needs more computational power, which parallel programming provide through the usage of GPU. But there are some trade-offs from the fast run time of parallel algorithm that is shown below on the memory consumption.

TABLE V.
PARALLEL MEMORY CONSUMPTION

N	*Memory in MB*
2, 4, 8, 16, 32, 64	n/a
128	7.338×10^1
256	2.776×10^1
512	3.902×10^1
1024	9.133×10^1
2048	2.095×10^2
4096	4.975×10^2
8192	1.6×10^3
16384	6.1×10^3

On the table it is shown that from N=2 until N=64, the memory being consumed is not available using the default profiler on Visual Studio 2019. As can be seen on N=128, it consumes memory more than N=256 and N=512, of course it is a weird phenomenon. There has been some spike data on N=128 until N=512, which later on will be more stable on N=1024 and so. The memory needed for such errors sometimes give less than 10MB on an execution and sometimes even more than 50MB on the other tries. It is remained unknown what cause this to happen, we speculate that it might be that there may be errors when placing or filling the data.

C. *Comparison and Analysis*

This sub chapter will compare the results of both sequential and parallel algorithm implementation by putting the results side by side. First of all, it is obvious that the parallel algorithm cost less time in the execution compared to the sequential algorithm. We will perform calculation of the total speedup using the following equation.

$$total\ speedup = \frac{T_s}{T_p} \quad (1)$$

The equation above is Amdahl's Law [12]. Let T_s be the computation time needed in sequential (without parallel computation) and T_p be the computation time needed in parallel computation. The comparison and speedup calculation are given in the table below.

TABLE VI.
RUN TIME COMPARISON AND SPEED UP

N	*Sequential Run in second*	*Parallel Run in second*	*Total Speedup in multiple*
2, 4, 8, 16	n/a	n/a	n/a
32	1×10^{-4}	n/a	n/a
64	9×10^{-4}	n/a	n/a
128	6.999×10^{-3}	1.185×10^0	0.006
256	5.723×10^{-2}	1.423×10^0	0.040
512	5.400×10^{-1}	1.425×10^0	0.379
1024	5.814×10^0	1.487×10^0	3.911
2048	7.108×10^1	2.56×10^0	27.764
4096	6.27×10^2	7.831×10^0	80.123
8192	4.520×10^3	4.248×10^1	106.403

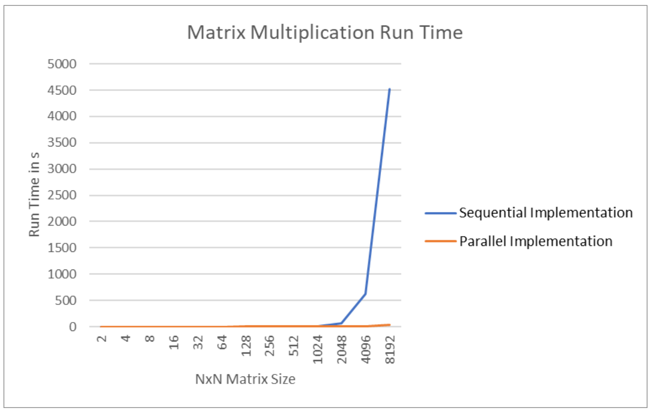


Figure 4.1. Run Time Comparison

From the table and figure presented above, we can see that by making a program parallel is very significant to its run time. Even though, on N≤512 it takes longer time to finish computation with parallel solution as the $total\ speedup < 1$. We indicate that it may be because of overhead in parallelism in the task (thread or process) start up and termination cost and maybe combined with other problems [13][14]. So, our hypothesis on the previous chapter is incorrect as we predict that parallel will functions well and faster no matter the matrices size are. But in average, the speedup done by making a program parallel is 31.232 times faster. Thus, using parallel implementation in solving problems is still recommended.

Besides comparing the run time, the memory consumption is also clearly stated head-to-head in Table VII. In addition to the memory consumption needed, a new column called "Ratio" was added to show how many times more do the parallel memory consumption needs compared to sequential memory consumption.

$$Ratio = \frac{MC_p}{MC_s} \quad (2)$$

The equation above is made to easily identify the ratio of memory consumed on each process. Where MC_p denotes the memory consumption on a parallel computation. On the other hand, MC_s denotes the memory consumption for a sequential computation.

TABLE VII.
MEMORY CONSUMPTION AND MULTIPLE

N	*Sequential Run in MB*	*Parallel Run in MB*	*Ratio*
2, 4, 8, 16, 32, 64	n/a	n/a	n/a
128	1.4×10^0	7.338×10^1	52.411
256	2.2×10^0	2.776×10^1	12.618
512	4.8×10^0	3.902×10^1	8.129

N	*Sequential Run in MB*	*Parallel Run in MB*	*Ratio*
1024	1.49×10^1	9.133×10^1	6.130
2048	5.59×10^1	2.095×10^2	3.478
4096	1.94×10^2	4.975×10^2	2.562
8192	7.725×10^2	1.6×10^3	2.071

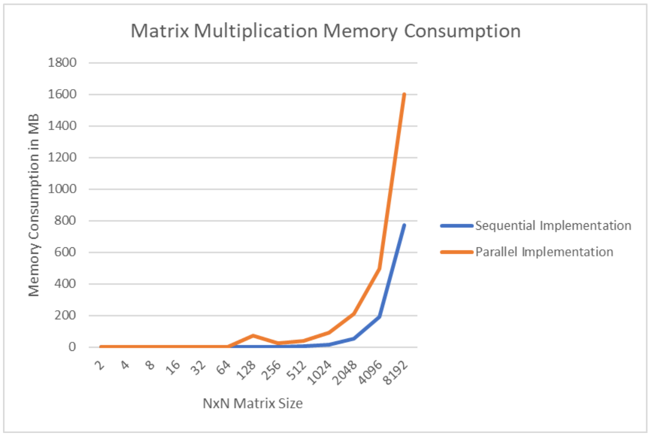


Fig 4.2. Memory Consumption Comparison

On the table and figure above, it is assumed for the memory consumed on N≤64 is 0 as it is not available to do the calculation. It is also shown on both on the table and figure that the difference is quite much whether it is on the run time or on the memory consumption. But vice versa to the run time, memory consumption for parallel computation cost more than sequential. The amount of memory required for parallel computation can be greater than serial, due to the need of replicating data and overheads associated with libraries used and subsystems [14].

V. Conclusion

We present matrix multiplication both on sequential and parallel model to see the differences. To conclude, it is recommended to use parallel algorithm with optimal thread numbers on bigger cases (breaking point should be found) as it solves things faster with speedup on average of 31.23, instead of the sequential. Some of the important notes to be taken for parallel usage is that it consumes lots of memory compared to sequential. Thus, it is afraid that on parallel computation time would not be a problem, but memory of the machine would be the limitation of solving cases. On further research it should also be explored more about giving constant value on *cudaMemcpy* and parallel implementations for machine learning or deep learning as well as it consumes lots of time and resources which parallel programming can deal with [15].

There are also some errors that remained unsolved on the project as the spike data and the data provided may not be 100% accurate. Some of the recommendations that we might give is to use profiler other than default from Visual Studio 2019 as it may not be that accurate. The machine quality also needs to be maintained (e.g. no running software when executing code, plugged in electricity, heat sink quality, etc.). Implementing matrix multiplication either on sequential and parallel may use a better algorithm rather than the one that provided.

References

[1] "parallel-processing noun - Definition, pictures, pronunciation and usage notes | Oxford Advanced Learner's Dictionary at OxfordLearnersDictionaries.com." [Online]. Available: https://www.oxfordlearnersdictionaries.com/definition/english/parallel-processing. [Accessed: 05-Apr-2020].

[2] L. Woodard, "Introduction to Parallel Programming What is Parallel Programming?," 2013.

[3] S. Ohshima, K. Kise, T. Katagiri, and T. Yuba, "Parallel Processing of Matrix Multiplication in a CPU and GPU Heterogeneous Environment," *High Perform. Comput. Comput. Sci. - VECPAR 2006*, vol. 4395, no. June, pp. 490–503, 2007.

[4] "Programming Guide :: CUDA Toolkit Documentation." [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#scalable-programming-model. [Accessed: 13-Apr-2020].

[5] "CUDA Zone | NVIDIA Developer." [Online]. Available: https://developer.nvidia.com/cuda-zone. [Accessed: 05-Apr-2020].

[6] "Profiler :: CUDA Toolkit Documentation." [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide/index.html. [Accessed: 15-Apr-2020].

[7] V. P. Gergel, "S08 Parallel Methods for Matrix Multiplication."

[8] "cuda_programming/mmul.cu at master · CoffeeBeforeArch/cuda_programming." [Online]. Available: https://github.com/CoffeeBeforeArch/cuda_programming/blob/master/matrixMul/baseline/mmul.cu. [Accessed: 08-Apr-2020].

[9] Aulia Hening Darmasti, "Cuda Programming Tutorial (Bahasa Indonesia) - YouTube," 21-Mar-2017. [Online]. Available: https://www.youtube.com/watch?v=JjQwNlz_NTo. [Accessed: 13-Apr-2020].

[10] "C++ Data Types - GeeksforGeeks." [Online]. Available: https://www.geeksforgeeks.org/c-data-types/. [Accessed: 13-Apr-2020].

[11] "How to Implement Performance Metrics in CUDA C/C++ | NVIDIA Developer Blog." [Online]. Available: https://devblogs.nvidia.com/how-implement-performance-metrics-cuda-cc/. [Accessed: 11-Apr-2020].

[12] "Parallel Speedup — Parallel Computing Concepts." [Online]. Available: http://selkie.macalester.edu/csinparallel/modules/IntermediateIntroduction/build/html/ParallelSpeedup/ParallelSpeedup.html. [Accessed: 09-Apr-2020].

[13] "Overhead of Parallelism - Chanaka Balasooriya - Medium." [Online]. Available: https://medium.com/@chanakadkb/overhead-of-parallelism-d1d3c43abadd. [Accessed: 16-Apr-2020].

[14] "Introduction to Parallel Computing." [Online]. Available: https://computing.llnl.gov/tutorials/parallel_comp/. [Accessed: 16-Apr-2020].

[15] V. Hegde and S. Usmani, "Parallel and Distributed Deep Learning." [Online]. Available: https://web.stanford.edu/~rezab/classes/cme323/S16/projects_reports/hedge_usmani.pdf. [Accessed: 01-Sep-2022]