NUS National University of Singapore

Faculty of Engineering

Matrix Multiplication Comparison between single processor and multiple processors

EE5902: Multiprocessor System

for project deadline: November 18, 2019 at 23:59:59

Name: Fredrik Norrstig

Matriculation Number: A0209932B

Name: Han Jie

Matriculation Number: A0116448A

Abstract

The objective of this report is to find out the time complexity of different matrix multiplication algorithm with the help of multiprocessors. The use of multiprocessor in matrix multiplication problem can easily enhance the speed. However, because of the complex procedures to perform the computation in a modern computer, the time spent might be influenced by memory capacity, memory loading speed, data structure used and etc. During the research, three algorithms are implemented using Python. They are vanilla matrix multiplication method, parallel multiplication method and Strassen algorithm. Different matrix sizes and thread numbers are used for testing. The research aspects for this project are matrix size and thread numbers.

There are several findings after analysis of the results. First, the parallel matrix computation time is not decrease as the number of threads increases. Secondly, Strassen algorithm does not outperform normal single thread matrix multiplication. Thirdly, the memory capacity of the computer used for matrix computation makes a huge difference especially when the number of threads increases. Fourthly, different data structure can influence the computation performance even in such simple algorithms like matrix multiplication. As the increasing demand in the area of machine learning, the demand for faster and more efficient matrix multiplication also becomes more and more important. Most of the machine learning framework are designed and implemented on Python platform, which is easier to learn and use comparing to C++ for most non-programming background researchers. Even the underlying algorithms are based on C++, which guarantees the efficiency. We should still consider the efficient and memory use when we implement the algorithms which use matrix multiplication intensively.

Introduction

Background and Motivation

Matrix multiplication has always been a classic linear algebra problem in mathematics and computer science filed. The amount of computational power needed for large matrix is always the constraint for machine learning development. In 1950s, Alan Turing came up with famous “Turing Test”, which is the founding theorem of artificial intelligence. From then on, many mathematicians, computer scientists and even biologists and psychologists have been working in this field [1]. However, due to the hardware constraint, the computational power was not enough to handle the amount of arithmetic operations. Until early 2000s, the hardware development in CPU and more importantly GPU speeds up the machine learning algorithms significantly. That’s the same time when AI became a popular topic. Training machine learning frameworks such as deep neural network involves many steps like forward propagation, backward propagation and gradient descent. However, the very basic arithmetic operation is always matrix multiplication. Due to the data-drive approach used in machine learning, the matrixes are normally huge. Thanks to the development of hardware, multicore and distributed computer architecture provides a simple solution to speed up the process of parallel matrix multiplication.

Because of the intrinsic properties of matrix multiplication, it can be easily computed in parallel and concatenate the results to form the result. By doing so, it can significantly accelerate machine learning training process. However, the speed of matrix multiplication does not only consist of arithmetic operations. It also involves data loading, communication and other operations. The more complex the algorithm is, the more time it costs on data loading, communication and other operations. Hence, there should be a compromise of time with respect to algorithm complexity and other factors.

Problem Statement:

Different matrix multiplication algorithms and architectures have different computational speed and memory consumption. The algorithm complexity and the matrix size do effect computation efficiency significantly. The problem targeted in this project is how the computation speed changes with different algorithm, different matrix size and thread number (for multiprocessing algorithms only).

Related Works

The first algorithm we have is a simple 'ijk'-method, hence forth called vanilla, which is the straightforward implementation with 3 for-loops and has a time complexity of O(n^3). For example, when multiplying two 2x2 matrices together the vanilla method needs 8 multiplication and 4 addition to complete, this will become relevant later when comparing with the other algorithm. Below in figure 1 there is the flowchart of the vanilla implementation.

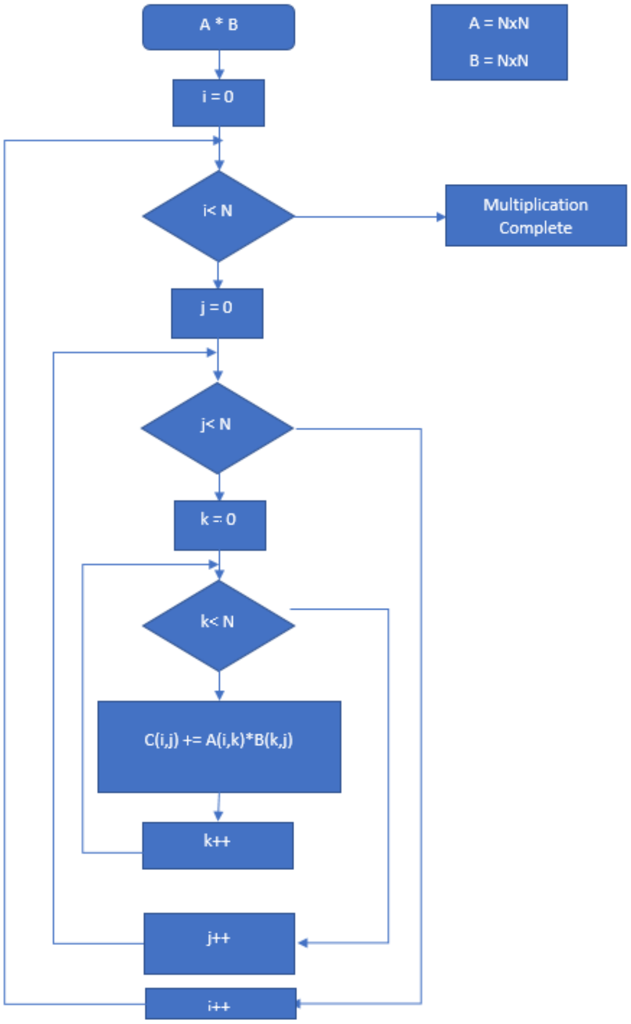


Figure 1: flowchart of the vanilla (ijk-algorithm)

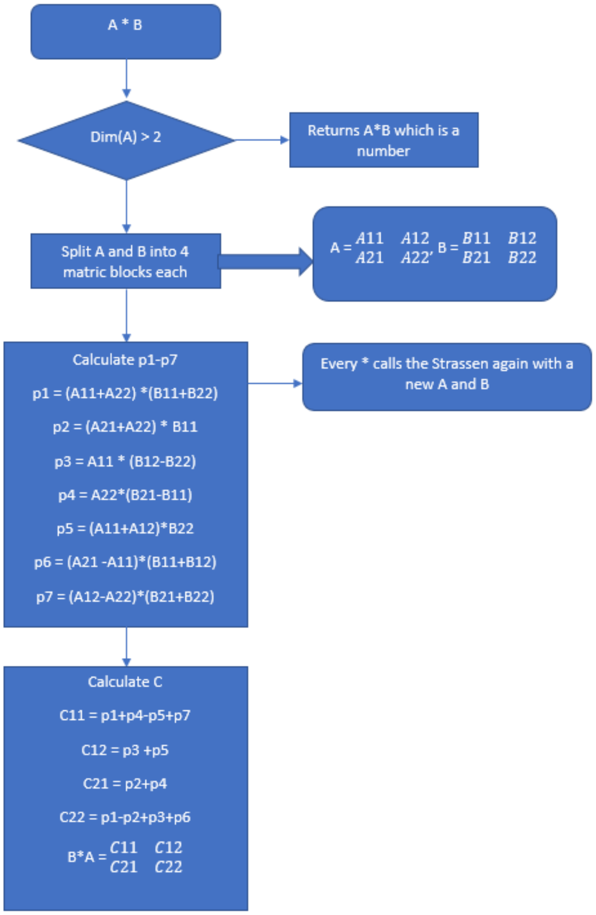


Figure 2: Flowchart of the Strassen-algorithm

The second algorithm is the Strassen-algorithm which reduce the number of multiplications which are needed to be done when multiplying 2 matrices. The Strassen method is a recursive function and has a time complexity of O($n^{2.807}$) [2]. The Strassen multiplication only works on matrices that have the same size and dimensions. When applying the Strassen both matrices are split into its four equal sized block matrices [3]. After this a series of additions and subtractions are done, 10 to be exact, on these block matrices. Then 7 multiplication with the Strassen-algorithm is done when the given matrix is larger. Lastly 8 more addition/subtractions are done and then we have the 4 block matrices of the result. The flowchart of the Strassen algorithm can be seen above.

When multiplying two 2x2 matrices the Strassen algorithm take 7 multiplication and 18 addition/subtraction which is one less multiplication than the vanilla – algorithm.

The third algorithm is the vanilla method but on multiple processors. The basic algorithm is the same as the vanilla-algorithm on a single processor except the i loop is divided between the multiple processor/threads. We have p processors/threads and the matrices are of dimension NxN then each processor gets an interval of N/P to go through in i.

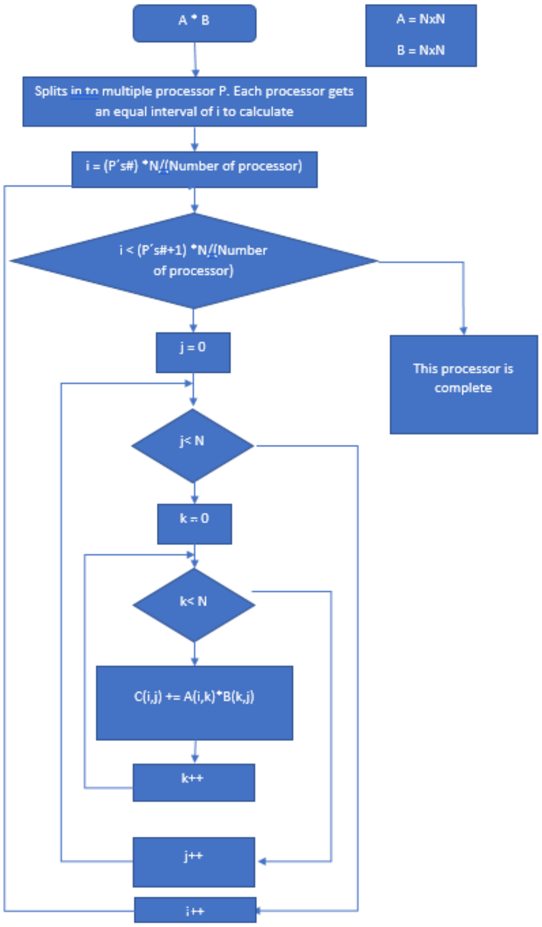
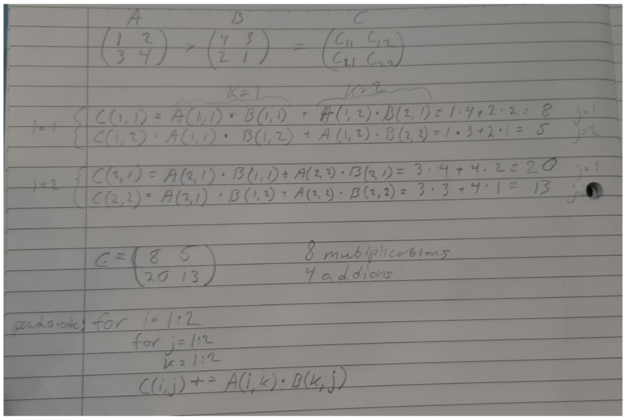

A * B
A = NxN
B = NxN
Splits in to multiple processor P. Each processor gets an equal interval of i to calculate
i = (P´s#) *N/(Number of processor)
i < (P´s#+1) *N/(Number of processor)
This processor is complete
j = 0
j< N
k = 0
k< N
C(i,j) += A(i,k)*B(k,j)
k++
j++
i++

Figure 3: Flow chart of the vanilla-algorithm on multiple processors

The implementation of the Strassen algorithm on multiple processor is a daunting task due to the recursive nature of the algorithm. In the report [2] they have divided the Strassen-algorithm in to 1,7 or 49 threads for calculating the different p values. We were not able to implement this into a working script.

Workings and Implementation of the Algorithm

Simple handwritten example of the vanilla-algorithm on a 2x2 sized matrix.



$$\overset{A}{\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}} \times \overset{B}{\begin{pmatrix} 4 & 3 \\ 2 & 1 \end{pmatrix}} = \overset{C}{\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}}$$

$$i=1 \begin{cases} C(1,1) = \underbrace{A(1,1)\cdot B(1,1)}_{k=1} + \underbrace{A(1,2)\cdot B(2,1)}_{k=2} = 1\cdot 4 + 2\cdot 2 = 8 & j=1 \\ C(1,2) = A(1,1)\cdot B(1,2) + A(1,2)\cdot B(2,2) = 1\cdot 3 + 2\cdot 1 = 5 & j=2 \end{cases}$$

$$i=2 \begin{cases} C(2,1) = A(2,1)\cdot B(1,1) + A(2,2)\cdot B(2,1) = 3\cdot 4 + 4\cdot 2 = 20 & j=1 \\ C(2,2) = A(2,1)\cdot B(1,2) + A(2,2)\cdot B(2,2) = 3\cdot 3 + 4\cdot 1 = 13 & j=2 \end{cases}$$

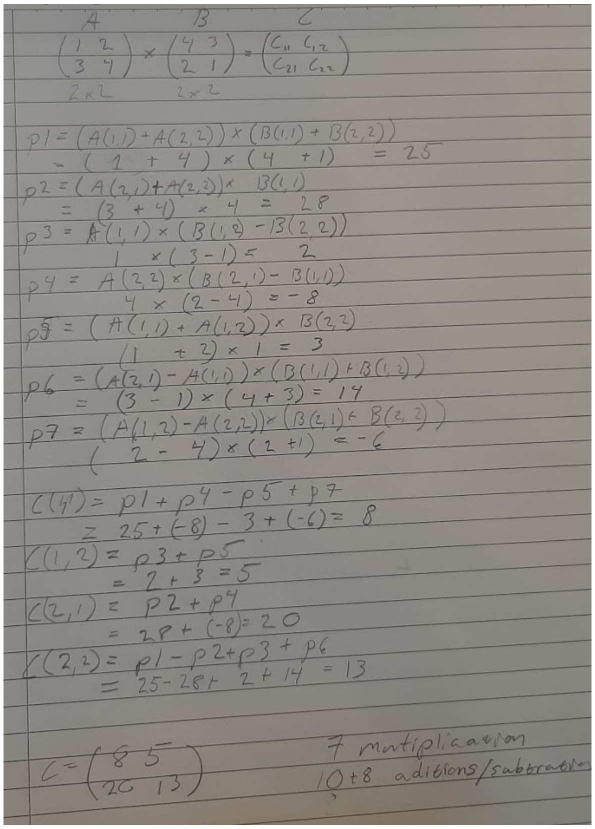
$$C = \begin{pmatrix} 8 & 5 \\ 20 & 13 \end{pmatrix}$$

8 mutiplications
4 addions

```
psudo-code: for i = 1:2
                for j = 1:2
                    k = 1:2
                    C(i,j) += A(i,k)·B(k,j)
```

Figure 4: Hand-written example of the vanilla-algorithm

Below is a handwritten example of the Strassen algorithm on a 2x2 sized matrix. It is more complex for this type of small calculation compared to the vanilla-algorithm. This is because the addition and multiplication are almost the same, computing wise, for small numbers but as the number gets larger the cost of multiplying becomes greater than the cost for additions.



$$\overset{A}{\underset{2\times2}{\begin{pmatrix}1&2\\3&4\end{pmatrix}}}\times\overset{B}{\underset{2\times2}{\begin{pmatrix}4&3\\2&1\end{pmatrix}}}=\overset{C}{\begin{pmatrix}C_{11}&C_{1,2}\\C_{21}&C_{22}\end{pmatrix}}$$

$p1=(A(1,1)+A(2,2))\times(B(1,1)+B(2,2))$
$=(1+4)\times(4+1)=25$
$p2=(A(2,1)+A(2,2))\times B(1,1)$
$=(3+4)\times4=28$
$p3=A(1,1)\times(B(1,2)-B(2,2))$
$1\times(3-1)=2$
$p4=A(2,2)\times(B(2,1)-B(1,1))$
$4\times(2-4)=-8$
$p5=(A(1,1)+A(1,2))\times B(2,2)$
$(1+2)\times1=3$
$p6=(A(2,1)-A(1,1))\times(B(1,1)+B(1,2))$
$=(3-1)\times(4+3)=14$
$p7=(A(1,2)-A(2,2))\times(B(2,1)+B(2,2))$
$(2-4)\times(2+1)=-6$

$C(1,1)=p1+p4-p5+p7$
$=25+(-8)-3+(-6)=8$
$C(1,2)=p3+p5$
$=2+3=5$
$C(2,1)=p2+p4$
$=28+(-8)=20$
$C(2,2)=p1-p2+p3+p6$
$=25-28+2+14=13$

$C=\begin{pmatrix}8&5\\20&13\end{pmatrix}$

7 mutiplication
10+8 aditions/subtratio

Figure 5: Hand-written example of the Strassen algorithm

Implementation:

The algorithms are implemented in Python3. The platform used is macOS Catalina. Some other configuration of the testing machine is:

Processor: 2.8 GHz Quad-Core Intel Core i7

Memory : 16 GB 2133 MHz LPDDR3

The result and outcome given in this report are only based on the specific testing machine used. If the same code is used in another different machine, the result can vary, and this is discussed in the next section.

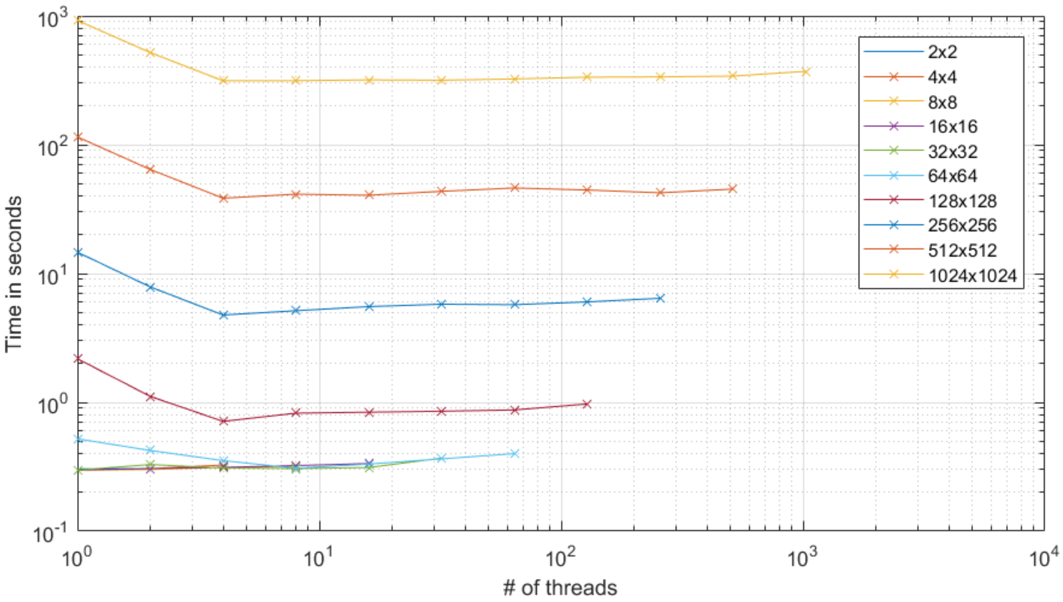


Figure 6: #of threads compared to time for differently sized matrixes

Result:

The data collected from testing can be interpreted from many perspectives. This section shows the result in the forms of tables and charts, and the discussion and hypothesis regarding these results is in next section.

1. Using parallel multithreading algorithm.

As shown in figure 6, when the matrix size is small, <64, the number of threads and the time consumption does not have a significant relationship, neither linear nor exponential. Some datasets even show a small increase in time when number of threads increases, i.e. matrix size =4.

When matrix size is larger than 64, the trend is approximately the same for different sizes. As the number of number of threads increases, the time goes down first then it keeps almost constant.

Given same number of threads, the time for multiplication is exponential to the matrix size when the matrix size is larger than 64.

Similarly, when the matrix size is small, the time consumption and the size itself do not have a significant relationship.

2. For the comparison across algorithms:

The time consumption increases exponentially as the matrix size increases linearly for all the algorithms.

When the size of the matrix is small the vanilla method and the Strassen-algorithm are faster than the 2 threads method of vanilla.

As the size increases the Strassen-algorithm becomes comparably slower than the vanilla-algorithm. This goes against our expectations, after knowing the time complexity of both these algorithms. A plausible reason for this is in the Strassen implementation there are a lot of new variables being used in every recursive loop.

As expected, the 2 threads vanilla takes over the Vanilla method after a while.

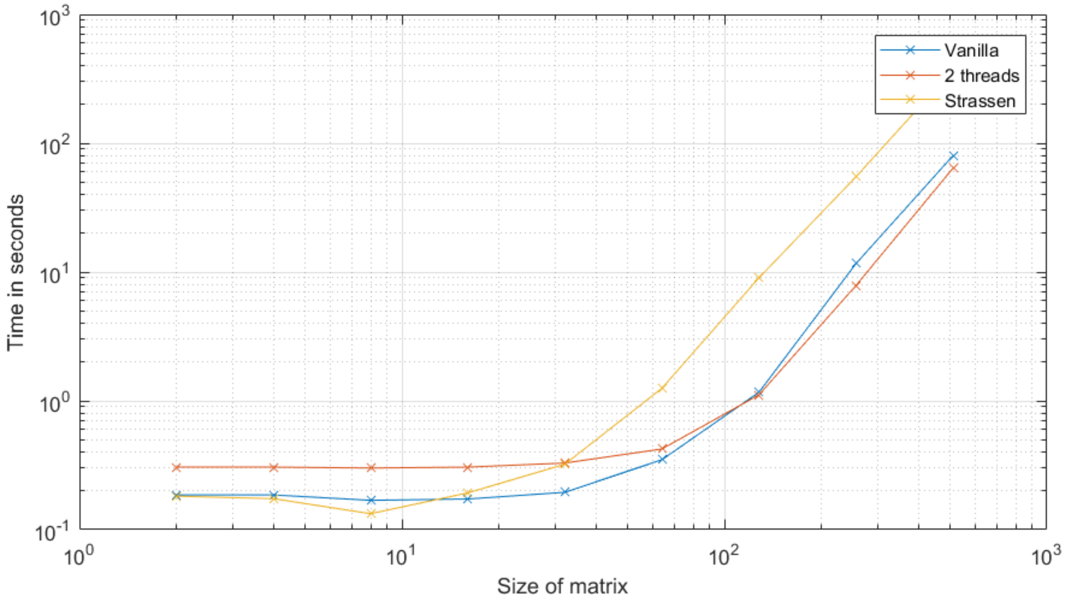


Figure 7: Comparison of time between different algorithms on different sized matrices

Discussions

1. The code is implemented using Python and Python multiprocessing package. The mechanism behind is launching threads to perform specific tasks. Hence, instead of real multi-processing, this project is actually multi-threading. The difference is that when the number of threads is larger than the number of cores, more than one threads are assigned to same core. In this case, it is not parallel computing anymore. In each core, it becomes time multiplexing. This explains the strange trend in the figure when the time decreases and keeps constant after 4 thread. It is because the machine we used only have 4 cores. However, from 1 thread, 2 threads and 4 threads, the decreasing thread can already justify the usage of multiprocessing algorithm significantly improve the performance.
2. When the matrix size is very small, the relationship between matrix size, number of threads are unclear. This is mainly because the time counted in the figure is the overall time, which consist of not only the arithmetic operation time but also the time for data loading from hard disk to memory, from memory to registers. When the matrix size is very small, i.e. less than 64, the arithmetic operation time is so

insignificant that the other time dominates the overall time.

3. Figure 6 shows an expected result when the matrix size is larger than 64 and number of threads smaller or equal to 4. The time decreases proportional as number of threads increases. The time increases exponentially as the size increases.
4. The result we got where not what we expected. For starters the Strassen-algorithm should be faster than the ijk(Vanilla) algorithm due to the previously proven time complexity, Strassen having an O(n^2.807) and ijk have O(n^3). Our thoughts here is that our implementation takes too much space and creates to many new variables inside each recursive loop to slow it down significantly. Moreover, the vanilla algorithm is much easier to implement so the data structure used is also simpler. The Strassen algorithm is much more complex so our implementation might not to optimal. The use of data structure might be another factor that influence the performance.

Conclusion

The result shows that the multiprocessing architecture is able to significantly accelerate the matrix multiplication time. However, the efficiency is largely dependent on the number of cores. If the number of cores is smaller than the number of the threads, the acceleration capability is constrained. Within the hardware capacity, the speed is proportional to the number of cores used in parallel.

For complex algorithm, number of cores may not be the only consideration. Some recursive algorithm may require a large memory for its recursion stack. The implementation of the algorithm also plays an essential role in the performance of the algorithm. It is not easy to implement the complex algorithms in the most optimal way. If it is the case, the algorithm may not execute as expected.

However, most of the software engineers and data scientists do not need to worry about the architectures of the matrix multiplication algorithm since they are already well studied and optimized to the latest computers and operation systems.

Future Work:

Just like the advancement of hardware significantly changes the field of machine learning and artificial intelligence. Computer architecture is also being changed with new technologies such as distributed computing architecture and the use GPU also large accelerate the training process.

As the computational power becomes cheaper and cheaper and communication price goes down, the distributed computation can be used for machine learning and AI[4]. Instead of constrained by the number of processors like the machine used in this project, the researcher can make use of cloud providers such as Amazon Web Services and Azure. With the help of new technologies, large matrix multiplication can be more efficient and cost effective.

References

[1] M. Son and K. Lee, "Distributed matrix multiplication performance estimator for machine learning jobs in cloud computing," in 2018, . DOI: 10.1109/CLOUD.2018.00088.

[2] Chou C-C, Deng YF, Li G, Wang Y. Parallelizing Strassen's method for matrix multiplication on distributed-memory MIMD architectures. Computers and Mathematics with Applications. 1995 Jan 1;30(2):49-69. https://doi.org/10.1016/0898-1221(95)00077-C

[3] Huang, Jianyu & Smith, Tyler & Henry, Greg & van de Geijn, Robert. (2016). Strassen's Algorithm Reloaded. 690-701. 10.1109/SC.2016.58.

[4] Kushida, Kenji & Murray, Jonathan & Zysman, John. (2015). Cloud Computing: From Scarcity to Abundance. Journal of Industry, Competition and Trade. 15. 10.1007/s10842-014-0188-y.

[5] Gupta, Himanshu & Sadayappan, Ponnuswamy. (1994). Communication Efficient Matrix Multiplication on Hypercubes.. Parallel Computing - PC. 12. 320-329. 10.1145/181014.181434.

[6] Van De Geijn, R. A. and Watts, J. (1997), SUMMA: scalable universal matrix multiplication algorithm. Concurrency: Pract. Exper., 9: 255-274. doi:10.1002/(SICI)1096-9128(199704)9:4<255::AID-CPE250>3.0.CO;2-2

[7] D'Alberto, Paolo & Bodrato, Marco & Nicolau, Alexandru. (2011). Exploiting Parallelism in Matrix-Computation Kernels for Symmetric Multiprocessor Systems Matrix-Multiplication and Matrix-Addition Algorithm Optimizations by Software Pipelining and Threads Allocation. ACM Trans. Math. Softw.. 38. 2. 10.1145/2049662.2049664.

Fredrik Norrstig	Han Jie
Strassen implementation	Ijk implementation
Graph in MATLAB	Parallel algorithm implementation
Flow charts of the different implementation	Data collection
Discussion on different algorithms	Discussion on parallel algorithms with different size and thread
Report formatting	Abstract
Data Processing	Introduction
Examples in the report	Conclusion