A Fast Way to Compute Matrix Multiplication

Ananya Kharbanda[1], Chua Zhong Ding[2], Shu Jian Jun[3]

[1] Raffles Institution, 1 Raffles Institution Ln, Singapore 575954
[2] River Valley High School, 6 Boon Lay Ave, Singapore 649961
[3] School of Mechanical & Aerospace Engineering, Nanyang Technological University, 50 Nanyang Avenue Singapore 639798

24YANAN481Z@student.ri.edu.sg
chua_zhong_ding@students.edu.sg
mjjshu@ntu.edu.sg

Abstract. Matrix multiplication plays a pivotal role in various domains, including image processing and artificial intelligence, making its optimization a crucial endeavor. This research paper delves into enhancing the efficiency of matrix multiplication by specifically focusing on reducing the number of additions required in the process. Traditional methods typically necessitate between 15 to 18 additions; this study aims to lower this count, thereby streamlining computational requirements.

The methodology encompasses two primary stages:

1. Initially, we assess the practicality of this optimization approach by comparing it with existing matrix multiplication techniques. This comparative analysis not only benchmarks our approach but also highlights potential areas for improvement.
2. Subsequently, we employ a brute force algorithm to derive a novel formula that minimizes the number of additions. This stage is critical in evolving a theoretical concept into a practical application.

Through this multi-faceted approach, the paper aims to contribute to the field, potentially leading to more efficient algorithms in image processing and AI applications.

Keywords: Matrix, Computing, Math

1 INTRODUCTION

1.1 Reasons for undertaking this study

Matrix multiplication is a critical concept in computer science, essential for a wide range of applications. Its role is especially significant in deep learning, where it is the key to calculating outputs in neural networks. In image processing, matrix multiplication is extensively used for manipulating and transforming images. Despite its essential role, matrix multiplication is computationally demanding.

$$Where\ A\ is\ a\ m \times n\ matrix\ and\ B\ is\ a\ m \times p\ matrix$$

$$(AB_{ij}) = \sum_{k=1}^{n} A_{ik} B_{kj}$$

Figure 1. Naive Matrix Multiplication

The standard process of multiplying two matrices of size N typically requires $2(N/2)^3$ operations, involving eight multiplications. This process can be quite time-consuming, especially in complex calculations. As such, there is a need to improve the efficiency of matrix multiplication, such that research in these fields can be sped up.

1.2 Strassen Algorithm

To overcome this challenge, various methods have been developed to speed up matrix multiplication, significantly improving computational efficiency in these important areas.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \quad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix},$$

$$With\, A \times B = C$$

$$M_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$$

$$M_2 = (A_{21} + A_{22}) \times B_{11}$$

$$M_3 = A_{11} \times (B_{12} - B_{22})$$

$$M_4 = A_{22} \times (B_{21} - B_{11})$$

$$M_5 = (A_{11} + A_{12}) \times B_{22}$$

$$M_6 = (A_{21} - A_{22}) \times (B_{11} + B_{12})$$

$$M_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$$

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{bmatrix}.$$

Figure 2. Strassen Algorithm

Strassen Algorithm [1] is a key development in optimizing matrix multiplication. It effectively reduces the number of necessary multiplications from 8 to 7, markedly decreasing the computational complexity. This is particularly advantageous for complex and large-scale calculations. However, this efficiency does not come without a trade-off. While Strassen Algorithm cuts down on multiplications, it increases the number of additions from 4 to 18. Although additions are less computationally intensive, requiring only $(N/2)^2$ operations each, this increase still contributes to a higher overall computation time.

1.3 Winograd Form

$$t = A_{11} \times B_{11}$$

$$u = (A_{21} - A_{11}) \times (B_{21} - B_{22})$$

$$v = (A_{21} + A_{22}) \times (B_{21} - B_{11})$$

$$w = t + (A_{21} + A_{22}) \times (B_{21} - B_{11})$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} A & C \\ B & D \end{bmatrix} = \begin{bmatrix} t + b \times B & w + v + (a + b - c - d) \times D \\ w + u + d \times (B + C - A - D) & w + u + v \end{bmatrix}$$

Figure 3. Winograd Form

The Winograd Form [2] of Strassen's Algorithm reduces the number of additions in matrix multiplication from 18 to just 15. The number of multiplications remains at 7.

1.4 Limitations

Despite our best efforts in further reducing the number of additions in matrix multiplication, the computational complexity of finding such an algorithm was too difficult for our computers to handle in the brute force approach.

1.5 Preliminaries

This section will define all significant terms used in this report.
MATRIX MULTIPLICATION is a binary operation that takes a pair of matrices, and produces another matrix. Each element of the resulting matrix is computed as the dot product of the corresponding row of the first matrix and the column of the second matrix.

TIME COMPLEXITY refers to a computational analysis that estimates how the runtime of an algorithm grows as the size of the input increases.

SQUARE MATRICES are matrices that have the same number of rows and columns.

RECURSION is a method where the solution to a problem depends on solutions to smaller instances of the same problem.

NxN MATRIX is a matrix consisting of n rows and n columns.

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}$$

Figure 4. An N x N Matrix where N = 3

2 AIMS / OBJECTIVES

This paper sets out to explore the potential for further optimization in matrix multiplication, specifically targeting a reduction in the number of additions without adding extra computational burdens. Our research is grounded in the hypothesis that an optimal algorithm exists which can achieve this balance. Our study involves, firstly, examining the practicality of minimizing additions in the matrix multiplication process, and secondly, developing an algorithm that can effectively identify and apply a more efficient formula for matrix multiplication. This exploration is driven by the premise that such an algorithm can be realized without necessitating additional computations, thereby enhancing the overall efficiency of the process, a critical factor in numerous applications within the realms of computer science.

3 METHODOLOGY

3.1 Comparing Effectiveness

In order to determine how effective reducing the number of additions in matrix multiplication will be, we will be using a programme to analyse the time taken for matrix multiplication for different sizes of matrices, for square matrices sizes N x N. We will compare the time taken for 3 algorithms, naive multiplication, Strassen algorithm, and the winograd form of Strassen algorithm. We will also analyse how much time an algorithm with only 14 and 13 additions will take. The code for this is shown below.

3.2 Naive Brute Force Approach

To identify an algorithm capable of performing matrix multiplications with 14 or fewer additions, we developed a naive algorithm with two key components.

The Combination Function is the first component of this algorithm. It is a function which takes in 2 arrays of size N, designed to determine a combination of k additions and m multiplications that yields a specified value. This function memorizes the results of different combinations it has previously calculated. By storing these results, the function avoids redundant computations, thus conserving the counts of additions and multiplications used. By iterating through each combination it has already come up with, this algorithm takes $O((k+m)!\ N^2)$ time.
The Matrix Subarray Testing Function, as the second key component of our algorithm, methodically evaluates every viable combination of additions and multiplications within the four subarrays of the matrix. Specifically, it explores combinations that result in exactly K additions and M multiplications. This function efficiently utilizes the results previously computed and stored by the combination function, effectively reusing them when possible to avoid redundant calculations. This strategic reuse of computations not only enhances efficiency but also streamlines the exploration process. As there are K! possible combinations for additions and M! possible combinations for multiplication, the total time complexity for this algorithm would be $O(K! + M!)$

When our Matrix Subarray Testing Function targets a combination of 14 additions and 7 multiplications, it initially faces 364 * 20 = 7,280 potential operations. To optimize this, we've introduced minimum and maximum constraints for the additions and multiplications in each subarray. Specifically, the range for additions is set between 1 and 5, while for multiplications, it's limited to 1 to 3 per subarray. This adjustment significantly narrows down the possibilities to 224 combinations for additions and 16 for multiplications, cumulatively reducing the loop count in our function to 3,584.

Moreover, this refinement not only lessens the loop iterations but also decreases the frequency of running our combination function. Considering the worst-case scenario where the number of additions and multiplications are at their maximum (5 and 3, respectively), the combination function would need to run at most (5+3)! * 22 * 2 = 322,560 times. Consequently, in the most demanding scenario, our algorithm would need to execute 322,560 * 3,584 = 1,156,055,040 times in total.

This strategic approach significantly streamlines the algorithm, ensuring more efficient computation even in the worst-case scenario.

4 RESULTS

4.1 Effectiveness

Table 1 demonstrates the average time taken to execute the Naive, Strassen, and Winograd Algorithm. These algorithms are already well established algorithms.

Table 1: Time taken to execute each pre existing algorithm

N	**Algorithms Time Taken/s**		
	Naive	***Strassen***	***Winograd***
2^4	$8.958e^{-02}$	$5.244e^{-01}$	$4.495e^{-01}$
2^5	$2.709e^{-02}$	$1.257e^{-01}$	$1.237e^{-01}$
2^6	$2.087e^{-01}$	$9.198e^{-01}$	$8.914e^{-01}$

N	Algorithms Time Taken/s		
	Naive	***Strassen***	***Winograd***
2^7	1.454	5.286	5.196
2^8	1.149e^{01}	3.769e^{01}	4.020e^{01}

Table 2 demonstrates the average time taken to execute an algorithm with 14 and 13 additions respectively. These algorithms are ones we are aiming to search for.

Table 2: Time taken to execute each hypothetical algorithm

N	Algorithms Time Taken/s	
	14 Additions	***13 Additions***
2^4	4.405e^{-01}	4.309e^{-01}
2^5	1.193e^{-01}	1.182e^{-01}
2^6	9.252e^{-01}	8.651e^{-01}
2^7	5.173	5.168
2^8	3.679e^{01}	3.666e^{01}

Figure 5 presents a graphical comparison of each algorithm's performance.

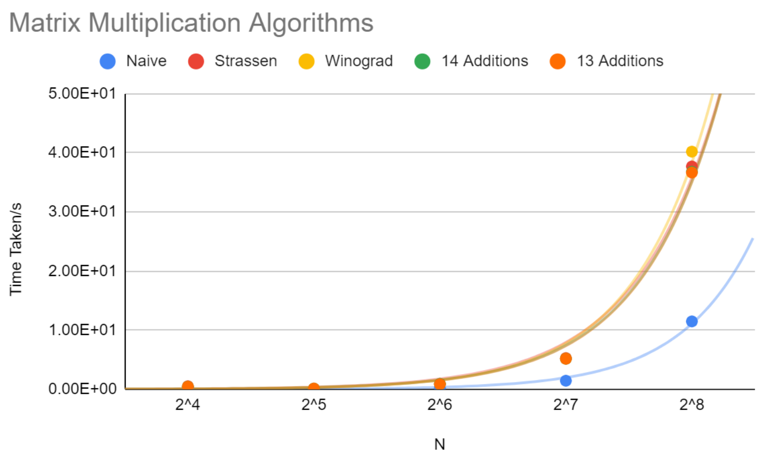

Figure 5. Graph of each algorithm and it's time taken

The collected data reveals that, in general, Winograd's algorithm outperforms Strassen's algorithm in matrix multiplication. Notably, configurations using 14 additions tend to be more efficient than Winograd's version with 15 additions, and similarly, those with 13 additions usually surpass the 14-addition setups in performance. Interestingly, the naive algorithm demonstrates the best performance across these comparisons. This superior efficiency is likely attributable to the overhead associated with the recursive nature of the Strassen algorithm. Furthermore, as the size of the matrices increases, the performance gap between the 13-addition method and the naive algorithm narrows, suggesting that the impact of recursion diminishes with larger data sets.

5 CONCLUSION

In conclusion, achieving a reduction in the number of additions required for matrix multiplication stands as a significant milestone in the ongoing quest to optimize this crucial computational process. While this study has not yet pinpointed the specific algorithm capable of minimizing additions, our proposed brute force approach lays the groundwork for future discovery. This method holds the promise of eventually uncovering the optimal algorithm, though it is contingent upon the availability of additional time and resources. The pursuit of such an algorithm is not just an academic endeavour; it has the potential to substantially impact various fields reliant on efficient computational processes.

ACKNOWLEDGEMENTS
We would like to thank our project supervisor Shu Jian Jun for guiding us in this project. We would also like to thank our NRP teachers in charge from River Valley High School and Raffles Institution.

References

1. Strassen, V. (1969). Gaussian elimination is not optimal. Numerische Mathematik, 13(4), 354–356. https://doi.org/10.1007/bf02165411

2. Coppersmith, D., & Winograd, S. (1990). Matrix multiplication via arithmetic progressions. Journal of Symbolic Computation, 9(3), 251–280. https://doi.org/10.1016/s0747-7171(08)80013-2