An Adaptive Algorithm for Matrix Multiplication

Xudong Han[1], Peiyu Luo[1], Jinqi Wei[1], Xingze Li[1], Chunfeng Hu[2], and Adam Ghandar[3,*]

Abstract—**Matrix multiplication is one of the basic algorithms in computer science. Many kinds of algorithms for matrix multiplication have been developed to pursue faster speed and better quality. In this project, we will compare the brute-force algorithm, Strassen's algorithm and the optimized Strassen's algorithm, and conclude an algorithm that can choose the appropriate method according to the matrix situation.**

Index Terms—**matrix multiplication, Strassen's algorithm, adaptive method, running time**

I. Introduction

Matrix multiplication (MM) is a basic method of data calculation. People developed natural (or brute-force) matrix multiplication algorithm by definition. The time complexity is $O(n^3)$ and when matrices become large, running time is so long that hard to use. People then tried to find faster and more effective algorithm. A major direction is how to effectively reduce the number of use of the arithmetic multiplication in MM, instead of considering the degree of density.

The earliest optimized algorithm was developed by Volker Strassen in 1969 and was named as Strassen algorithm. The basic idea is to reduce the number of times multiplication by replacing multiplication with addition and subtraction. Because multiplication is spend much more running time than addition and subtraction, this idea is effective. The details of Strassen's algorithm will be discussed in Section IV.

Following scholars continue to study and develop faster algorithms. Pan's algorithm was proposed in 1981 which time complexity is decreased to $O(n^{2.494})$. Andrew Stothers (2010) proposed a new algorithm whose time complexity is $O(n^{2.374})$. Up to now, the matrix multiplication algorithm with the lowest time complexity is $O(n^{2.3728639})$, an extension of simplified Stanford's algorithm by François Le Gall (2014).

In this paper, we focus on the Strassen's algorithm and how to optimize it. Three kinds of optimized algorithms, data-processing-optimized Strassen's algorithm, multithreaded Strassen's algorithm and adaptive algorithm, was developed, and we applied them to from small to large matrices to compare their performance. Crossover point was also found to guide the algorithm optimization.

II. Background

The central role of the conventional matrix multiplication algorithm as a building block in solving problems in algebra and scientific computations has generated a significant amount of research into techniques for improving the performance of this algorithm. Since 1960, many matrix multiplication algorithms have been discovered to multiply matrices fast with the cost estimate yielding very complex algorithms that are impractical for many matrices of reasonable sizes. The order of conventional algorithm has been established to be $O\left(n^3\right)$ and a way forward is to reduce this order to $O\left(n^\alpha\right)$ where $\alpha < 3$. Strassen's divide and conquer matrix multiplication algorithm was developed and found to be of order $O\left(n^{2.81}\right)$ with a significant gain of 0.19 (or 19%) in time and cost (see for example the work of Harvard). As a consequence, for sufficiently large values of (in thousands), Strassen's algorithm will run faster on the one hand, as well as offers a significant improvement in cost efficiency on the other hand than the conventional algorithm for matrix multiplication. Adaptive Strassen's matrix multiplication combines the novel implementation of Strassen's ideas with the auto-tuning linear algebra software (ATLAS) or GotoBLAS's matrix multiplication. The author studied the interaction between the effective performance of Strassen and the memory hierarchy, and demonstrated the correct combination of Strassen and ATLAS/GotoBLAS to take full advantage of Strassen's full potential between different architectures, and the paper also proved their approach can reach 30%/22% effect compared with using ATLAS/GotoBLAS. The implementation logic of the adaptive algorithm is as follows: at the top, the cache forgetting algorithm is used to reduce the problem to almost a square matrix; at the middle layer, the Strassen algorithm is deployed to reduce the amount of calculation; at a lower level, ATLA or GotoBLAS are deployed to release the architecture and structure.

There are two basic experiments based on rectangular matrix and square matrix. The first comparison is through HP zv6000, Athlon-64 2GHz and conventional cblasdgemm balanced matrix multiplication and HASA, which shows that Strassen's algorithm can be successfully applied to rectangular matrices and can be significantly improved in performance. However, this improvement is usually due to higher local utilization of data but not just reduced operations. The second experiment is

*This work was supported by Southern University of Science and Technology.

[1]Xudong Han, Peiyu Luo, Jinqi Wei, and Xingze Li are with the Department of Mechanical and Energy Engineering, Southern University of Science and Technology, Shenzhen, Guangdong 518055, China. {11812519, 11811314, 11811516, 11813223}@mail.sustech.edu.cn

[2]Chunfeng Hu is with the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, Guangdong 518055, China. hucf@sustech.edu.cn

[3]Adam Ghandar is the corresponding author with the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, Guangdong 518055, China. aghandar@sustech.edu.cn

about the relative performance results of Strassen and Faster MM relative to GotoBLAS_DGEMM (Balanced and HASA respectively) and the two architectures Athlon 64 2.4GHz and Pentium 4 3.2GHz. The results shows that: HASA algorithm has fewer instructions than Balanced instructions because it applies Strasssen's division to larger matrices; however, compared with GoerBLAS matrix multiplication, HASA's relative improvement in GotoBLAS matrix multiplication is smaller (the average Balanced reaches 7.2% Acceleration, while HASA reached 5.8%). This shows that algorithms with better data locality provide better performance than algorithms with fewer operations (for Athlon 64 2.4GHz architecture). The Balanced algorithm and the HASA algorithm have similar performance. In other words, on average, HASA can achieve 7.7% acceleration, while Balanced can achieve 7.5% acceleration. Also, for this architecture (Pentium 4 3.2GHz), the performance is very predictable, and the performance graph clearly shows the level of recursion. The previous experiment based on the Square matrix only used two methods to measure the performance of HASA and the performance of 17 different architectures: the relative execution time of HASA on cblas_dgemm and the relative MFLOPS of cblas_dgemm on the ideal machine peak performance.

The results of this article quantify the interaction between the kernels of an application and the underlying architecture can have big significant in helping design complex-but-portable codes. Such metrics can improve algorithm design and can be used as the basis for fully automated methods.

III. Theoretical Analysis

Strassen's matrix multiplication discovered that the original recursive algorithm of complexity $O(n^3)$ can be reorganized in such a way that one computationally expensive recursive matrix multiplication step can be replaced with 18 cheaper matrix additions. Strassen's algorithm embodies two different locality properties because its two basic computations exploit different data locality: matrix multiply has spatial and temporal locality, and matrix addition has only spatial locality. Now we suppose the matrix A of size i × k, the matrix B of size k × j and the matrix C of size i × j which equals to the product of AB. All A, B and C in Strassen's matrix multiplication are composed of four sub-matrices each.

Considering the operand matrix A with $\sigma(A) = i \cdot k$, it is logically composed of four matrices and every part's operand could be expressed as

$$\sigma(A_{11}) = \left\lceil \frac{i}{2} \right\rceil \cdot \left\lceil \frac{k}{2} \right\rceil$$

$$\sigma(A_{12}) = \left\lceil \frac{i}{2} \right\rceil \cdot \left\lfloor \frac{k}{2} \right\rfloor$$

$$\sigma(A_{21}) = \left\lfloor \frac{i}{2} \right\rfloor \cdot \left\lceil \frac{k}{2} \right\rceil$$

$$\sigma(A_{22}) = \left\lfloor \frac{i}{2} \right\rfloor \cdot \left\lfloor \frac{k}{2} \right\rfloor$$

Similarly, B is also composed of four matrices and every part's operand could be expressed as

$$\sigma(B_{11}) = \left\lceil \frac{k}{2} \right\rceil \cdot \left\lceil \frac{j}{2} \right\rceil$$

$$\sigma(B_{12}) = \left\lceil \frac{k}{2} \right\rceil \cdot \left\lfloor \frac{j}{2} \right\rfloor$$

$$\sigma(B_{21}) = \left\lfloor \frac{k}{2} \right\rfloor \cdot \left\lceil \frac{j}{2} \right\rceil$$

$$\sigma(B_{22}) = \left\lfloor \frac{k}{2} \right\rfloor \cdot \left\lfloor \frac{j}{2} \right\rfloor$$

During the Strassen's matrix multiplication, the time of the process for seven multiplications is

$$T = 2\pi(\left\lceil \frac{i}{2} \right\rceil \cdot \left\lceil \frac{k}{2} \right\rceil \cdot j + i \cdot k \cdot \left\lceil \frac{j}{2} \right\rceil + \left\lceil \frac{i}{2} \right\rceil \cdot \left\lfloor \frac{k}{2} \right\rfloor \cdot \left\lceil \frac{j}{2} \right\rceil) \quad (1)$$

where π is the efficiency of matrix multiplication, and $1/\pi$ is simply the floating point operation per second (FLOPS) of the computation.

And the time of the process for 22 matrix additions (18 matrix additions and 4 matrix copies) is

$$T = \alpha \left\lfloor 5 \cdot \left\lceil \frac{k}{2} \right\rceil \cdot (\left\lceil \frac{i}{2} \right\rceil \cdot \left\lceil \frac{j}{2} \right\rceil) + 3 \cdot i \cdot j \right\rfloor \quad (2)$$

Thus, we find that the problem size [m, n, p] to yield control to the ATLAS/GotoBLAS's algorithm is when Eq. 3 below is satisfied:

$$\left\lfloor \frac{i}{2} \right\rfloor \cdot \left\lceil \frac{k}{2} \right\rceil \cdot \left\lceil \frac{j}{2} \right\rceil \leq \alpha \left\lfloor 5 \cdot \left\lceil \frac{k}{2} \right\rceil \cdot (\left\lceil \frac{i}{2} \right\rceil \cdot \left\lceil \frac{j}{2} \right\rceil) + 3 \cdot i \cdot j \right\rfloor \quad (3)$$

We assume that π and α are functions of the matrix size only, as we explain in the following. So for the square matrix, $i = k = j$, we can find the crossover point:

$$n_1 = 22 \cdot \frac{\alpha}{n} \quad (4)$$

For example, if we assume a ratio $\alpha/\pi = 50$ (this is common for the systems tested in this work), we find that the crossover point corresponds to the problem (matrix) size $n_1 > 1100$. For problems of size $l \cdot n_1 \leq n < (l+1) \cdot n_1$, we may apply Strassen's l times. In fact, the factors π and α are easy to estimate by benchmarking and we can determine the specific recursion point n_1 by a linear search. In Section V, a detailed crossover-point searching by experiment is shown.

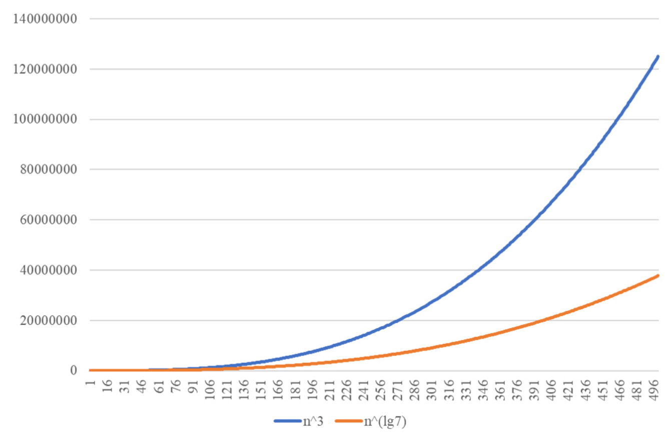

Fig. 1. The theoretical comparison of the complexity of matrix multiplication algorithm (Θ(n^3), green curve) and Strassen's algorithm ($\Theta(n^{lg7})\approx\Theta(n^{2.807})$, red curve).

IV. Methodology

The most important method of matrix multiplication is the general matrix product. It is defined only when the number of columns of the first matrix is the same as that of the second matrix. If A is an i × k matrix and B is an k × j matrix, then their product AB will be an i × j matrix. The elements of the product matrix are obtained as follows:

$$(ab)_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \tag{5}$$

For the natural (or brute-force) matrix multiplication, the operation needs three **for** loops, and the time complexity is $O(n^3)$. The pseudocode of the naive matrix multiplication is as follows:

```
SQUARE–MATRIX–MULTIPLY(A,B)
    n=A.rows
    let C be a new n × n matrix
    for i=1 to n
        for j=1 to n
            C[i,j]=0
            for k=1 to n
                C[i,j]+=A[i,k]∗B[k,j]
    return C
```

In 1969, Volker Strassen proposed the first algorithm, whose time complexity is lower than $\Theta(n^3)$ matrix multiplication algorithm, and the algorithm complexity is $\Theta(n^{lg7})$. As Fig. 1 shown, Strassen's algorithm has great advantages in performance for matrices with large dimension, which can reduce a lot of multiplication calculation.

We are now going to show the Strassen's matrix multiplication principle. Suppose that both the matrix A and the matrix B are square matrices of n × n(n = 2^n), find $C = AB$, as shown below,

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

And,

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} + \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

The matrix C can be obtained by the following formula,

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

From the above expression, we can conclude that to calculate the multiplication of two n × n matrices requires 8 times multiplication and 4 times addition of two n/2 × n/2. We use $T(n)$ to express the time complexity of n × n matrix multiplication, and according to the above decomposition, we can get

$$T(n) = 8 \cdot T(\frac{n}{2}) + \Theta(n^2) \tag{6}$$

The above steps take time O(1), and create ten n/2 × n/2 matrices, S_1,S_2, ...,S_{10} (time complexity is O(n^2)), which are as shown in following

$$S_1 = B_{12} - B_{22}$$
$$S_2 = A_{11} + A_{12}$$
$$S_3 = A_{21} + A_{22}$$
$$S_4 = B_{21} - B_{11}$$
$$S_5 = A_{11} + A_{22}$$
$$S_6 = B_{11} + B_{22}$$
$$S_7 = A_{12} - A_{22}$$
$$S_8 = B_{21} + B_{22}$$
$$S_9 = A_{11} - A_{21}$$
$$S_{10} = B_{11} + B_{12}$$

Recursively calculate the product P_1, P_2, ..., P_7, of seven matrices, each matrix P_i is n/2 × n/2. And we get

$$P_1 = A_{11} \cdot S_1$$
$$P_2 = S_2 \cdot B_{22}$$
$$P_3 = S_3 \cdot B_{11}$$
$$P_4 = A_{22} \cdot S_4$$
$$P_5 = S_5 \cdot S_6$$
$$P_6 = S_7 \cdot S_8$$
$$P_7 = S_9 \cdot S_{10}$$

Then we can calculate the product C in four parts, C_{11}, C_{12}, C_{21}, C_{22} by P_1, P_2, ..., P_7, which spend time $\Theta(n^2)$.

$$C_{11} = P_5 + P_4 - P_2 + P_6$$
$$C_{12} = P_1 + P_2$$
$$C_{21} = P_3 + P_4$$
$$C_{22} = P_5 + P_1 - P_3 - P_7$$

From the above process, it is not difficult to derive the recursive expression to represent the time complexity of Strassen's algorithm, as Eq. 7.

$$T(n) = \begin{cases} \Theta(1) & , \, if \; n = 1 \\ 7 \cdot T(\frac{n}{2} + \Theta(n^2) & , \, if \; n > 1 \end{cases} \quad (7)$$

We have traded off one matrix multiplication for a constant number of matrix additions. By the master method, Eq. 7 has the solution

$$T(n) = \Theta(n^{lg7}) \quad (8)$$

The pseudocode of Strassen's algorithm is shown below.

```
STRASSEN(A,B)
    n=A.rows
    if n==1
        return A[1,1]*B[1,1]
    let C be a new n × n matrix
    A[1,1]=A[1..n/2][1..n/2]
    A[1,2]=A[1..n/2][n/2+1..n]
    A[2,1]=A[n/2+1..n][1..n/2]
    A[2,2]=A[n/2+1..n][n/2+1..n]
    B[1,1]=B[1..n/2][1..n/2]
    B[1,2]=B[1..n/2][n/2+1..n]
    B[2,1]=B[n/2+1..n][1..n/2]
    B[2,2]=B[n/2+1..n][n/2+1..n]
    S[1]=B[1,2]-B[2,2]
    S[2]=A[1,1]+A[1,2]
    S[3]=A[2,1]+A[2,2]
    S[4]=B[2,1]-B[1,1]
    S[5]=A[1,1]+A[2,2]
    S[6]=B[1,1]+B[2,2]
    S[7]=A[1,2]-A[2,2]
    S[8]=B[2,1]+B[2,2]
    S[9]=A[1,1]-A[2,1]
    S[10]=B[1,1]+B[1,2]
    P[1]=STRASSEN(A[1,1],S[1])
    P[2]=STRASSEN(S[2],B[2,2])
    P[3]=STRASSEN(S[3],B[1,1])
    P[4]=STRASSEN(A[2,2],S[4])
    P[5]=STRASSEN(S[5],S[6])
    P[6]=STRASSEN(S[7],S[8])
    P[7]=STRASSEN(S[9],S[10])
    C[1..n/2][1..n/2]=P[5]+P[4]-P[2]+P[6]
    C[1..n/2][n/2+1..n]=P[1]+P[2]
    C[n/2+1..n][1..n/2]=P[3]+P[4]
    C[n/2+1..n][n/2+1..n]=P[5]+P[1]-P[3]-
        P[7]
    return C
```

V. Experiment Design

In this section, we are going to introduce each part of our work to implement matrix multiplication. This will be expanded in the order of modules within the code.

A. Class Matrix

The class named Matrix represent a matrix in "row-major" order with parameter "rows" and "columns". Some basic operations of the matrix is also applied in Matrix, as shown below.

- "getValueAt" and "setValueAt": are to get and set the element value at the specified position.
- "getMatrixAt": is to obtain the small matrix of the specified area in the matrix.
- "partitionMatrix": is to divide the matrix into four pieces.
- "mergeMatrix": merging the four small matrices into one large matrix.
- "plus" and "minus": are to add and subtract the matrix.
- "toString" and "show": converting to a string (and printed on the screen).

B. Square Matrix Multiply

This method use the natural (or brute-force) mathematical method to calculate the value of each element of the matrix using triple for loops. And the pseudocode is shown in Section IV. It is quite easy to understand.

C. Strassen matrix multiply

Strassen' algorithm is originally designed for even-order matrix. But adaption to any order matrix is important. To make the algorithm meet the square matrix of any size, when the matrix is divided into blocks, the even-order matrix is naturally divided into four blocks, and the odd-order matrix is divided into "half of the odd-order plus one"-order matrix. In this recursion, the matrix of any order can be divided and operated. The representation of all matrices in the operation uses the Matrix object.

In order to optimize the running time of the block matrix, merge the matrix, and reduce the large amount of space and time caused by the continuous recursive generation and return of new objects, we have changed the function calling method and the specific implementation of matrix multiplication. The original functions all take the result of the operation as the return value of a function and return a new Matrix object. The new function directly calculates the one-dimensional array of the matrix, avoiding repeated disassembly and assembly of data. Then it passes the array storing the result matrix to the function as an input parameter, and modify the value of the array directly inside the arithmetic function. This is equivalent to passing the result's pointer, reducing the time consumption of generating objects. And for some continuous addition and subtraction operations in matrix operations, special functions for continuous addition and subtraction are written, instead of repeatedly calling the addition or subtraction function, which reduces the time of switching between functions and saves the space for storing intermediate calculation results.

Another optimizing idea is to use parallel algorithm. We used multithreading to optimize the code. The process of calculating the intermediate matrix in the Strassen algorithm is executed in parallel, which greatly reduces the time consumption of calculating the intermediate matrix.

Of course, whether it is optimizing the function calling method or using multi-threading to optimize, it is all modified on the basis of the standard matrix multiplication implemented by the Matrix object. Variable controlling is to be used to examine the optimization effects of different optimization methods on operating speed in Section VI.

D. Crossover Point

To find the effective crossover point of the running time of the Strassen's algorithm and brute-force algorithm, we tested and continuously recorded ten sets of running time data. When for the running time of the Strassen's algorithm is larger than that of brute-force algorithm for the first five data and smaller for the last five data, the intermediate point is considered as the intersection point. This is to avoid data jitter caused by fluctuations in the operating environment. If the data fluctuates severely under special circumstances, when the running time of the Strassen's algorithm is smaller than that of the brute-force algorithm in ten consecutive sets of data, the starting point is considered as the intersection point. The upper limit of the crossover-point-finding test is 4098. If no intersection is found before the upper limit is reached, the default intersection 350 is used, which is empirical value based on experiments.

E. Generate Random Matrix

To test and verify the matrix multiplication algorithm, there must be a method to generate random matrices. We were to generate a random number in the interval $[-1, 1]$, and first generated an integer in the interval $[0, 201)$, then divided the result by 100, and randomly added a sign to it. Class "DecimalFormat" is used to keep all two decimal places to eliminate the excessive number of decimal places caused by the positive and negative numbers. The generated random numbers were filled into the matrix circularly to generate a random matrix of any size for testing.

VI. Empirical Analysis

First of all, we compared the running time of the brute-force algorithm and Strassen's algorithm without any optimization. The original Strassen's algorithm always took longer running time than the brute force algorithm, regardless of the size of the matrix, mainly because of the time taken to split the matrix. (Fig. 2) Therefore, the optimization for Strassen's algorithm is necessary. We developed optimized Strassen's algorithm, multithreaded Strassen's algorithm and adaptive algorithm.

We are going to discuss brute-force and other four optimized algorithms and compare their running time. They are described in details below.

- Brute-force algorithm: $\Theta(n^3)$ matrix multiplication algorithm. This is a simple and natural method using definition. The pseudocode is shown before in SQUARE-MATRIX-MULTIPLY(A,B).
- Data-Processing-Optimized Strassen's algorithm: Based on Strassen's algorithm mentioned before, the data processing is improved by changing the function calling method from data copying to pointer passing,to make it easier to store and modify and the speed is higher. The pseudocode is shown before in STRASSEN(A,B).
- Multithreaded Strassen's algorithm: Strassen's algorithm applied parallel computation which greatly improved the processing efficiency. Here, we only use nested parallelism. And the parallelism of multithreaded Strassen's algorithm is $\Theta(n^{lg7}/lg^2 n)$. The pseudocode is shown below as P-STRASSEN(A,B).

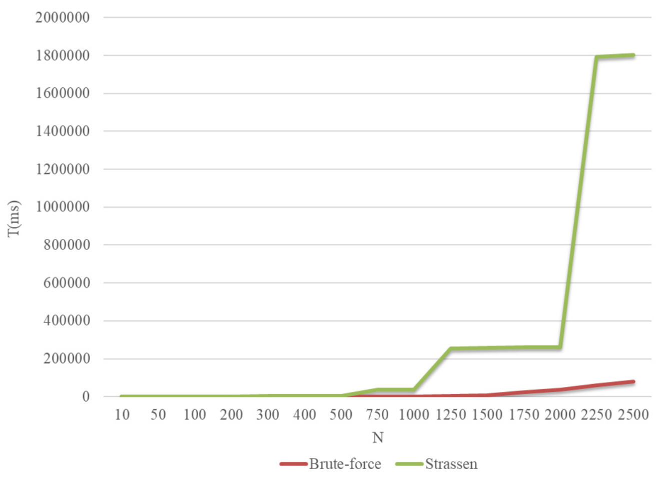

Fig. 2. The comparison between brute-force algorithm and original Strassen's algorithm

```
P-STRASSEN(A,B)
    n=A.rows
    if n==1
        return A[1,1]*B[1,1]
    let C be a new n × n matrix
    parallel
        A[1,1]=A[1..n/2][1..n/2]
        A[1,2]=A[1..n/2][n/2+1..n]
        A[2,1]=A[n/2+1..n][1..n/2]
        A[2,2]=A[n/2+1..n][n/2+1..n]
        B[1,1]=B[1..n/2][1..n/2]
        B[1,2]=B[1..n/2][n/2+1..n]
        B[2,1]=B[n/2+1..n][1..n/2]
        B[2,2]=B[n/2+1..n][n/2+1..n]
    parallel
        S[1]=B[1,2]-B[2,2]
        S[2]=A[1,1]+A[1,2]
        S[3]=A[2,1]+A[2,2]
        S[4]=B[2,1]-B[1,1]
        S[5]=A[1,1]+A[2,2]
        S[6]=B[1,1]+B[2,2]
        S[7]=A[1,2]-A[2,2]
        S[8]=B[2,1]+B[2,2]
        S[9]=A[1,1]-A[2,1]
        S[10]=B[1,1]+B[1,2]
    parallel
        P[1]=STRASSEN(A[1,1],S[1])
        P[2]=STRASSEN(S[2],B[2,2])
        P[3]=STRASSEN(S[3],B[1,1])
        P[4]=STRASSEN(A[2,2],S[4])
        P[5]=STRASSEN(S[5],S[6])
        P[6]=STRASSEN(S[7],S[8])
        P[7]=STRASSEN(S[9],S[10])
    parallel
        C[1..n/2][1..n/2]=P[5]+P[4]-P
            [2]+P[6]
        C[1..n/2][n/2+1..n]=P[1]+P[2]
        C[n/2+1..n][1..n/2]=P[3]+P[4]
        C[n/2+1..n][n/2+1..n]=P[5]+P
            [1]-P[3]-P[7]
    return C
```

- Adaptive 16 algorithm: hybrid algorithm using SQUARE-MATRIX-MULTIPLY(A,B) when the size is under 16 and STRASSEN(A,B) when the size is above 16. The pseudocode is shown below as ADAPTIVE-MATRIX-MULTIPLY(A,B).

```
ADAPTIVE-MATRIX-MULTIPLY(A,B)
    n=A.rows
    if n==1
        return A[1,1]*B[1,1]
    let C be a new n × n matrix
    if n<=16
        return SQUARE-MATRIX-MULTIPLY(
            A,B)
    A[1,1]=A[1..n/2][1..n/2]
    A[1,2]=A[1..n/2][n/2+1..n]
    A[2,1]=A[n/2+1..n][1..n/2]
    A[2,2]=A[n/2+1..n][n/2+1..n]
    B[1,1]=B[1..n/2][1..n/2]
    B[1,2]=B[1..n/2][n/2+1..n]
    B[2,1]=B[n/2+1..n][1..n/2]
    B[2,2]=B[n/2+1..n][n/2+1..n]
    S[1]=B[1,2]-B[2,2]
    S[2]=A[1,1]+A[1,2]
    S[3]=A[2,1]+A[2,2]
    S[4]=B[2,1]-B[1,1]
    S[5]=A[1,1]+A[2,2]
    S[6]=B[1,1]+B[2,2]
    S[7]=A[1,2]-A[2,2]
    S[8]=B[2,1]+B[2,2]
    S[9]=A[1,1]-A[2,1]
    S[10]=B[1,1]+B[1,2]
    P[1]=STRASSEN(A[1,1],S[1])
    P[2]=STRASSEN(S[2],B[2,2])
    P[3]=STRASSEN(S[3],B[1,1])
    P[4]=STRASSEN(A[2,2],S[4])
    P[5]=STRASSEN(S[5],S[6])
```

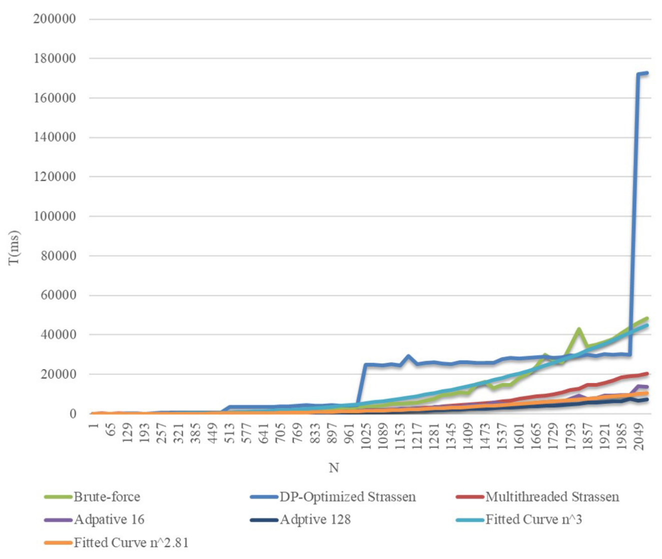

Fig. 3. The comparison between brute-force algorithm (Brute-force), data-processing-optimized Strassen's algorithm (DP-Optimized Strassen), multi-threaded Strassen's algorithm (Multithreaded Strassen), adaptive 16 algorithm (Adaptive 16), adaptive 128 algorithm (Adaptive 128), and two curves that fit brute-force algorithm (Fitted Curve nˆ3) and optimized Strassen's algorithm (Fitted Curve nˆ2.81)

```
    P[6]=STRASSEN(S[7],S[8])
    P[7]=STRASSEN(S[9],S[10])
    C[1..n/2][1..n/2]=P[5]+P[4]-P[2]+P
        [6]
    C[1..n/2][n/2+1..n]=P[1]+P[2]
    C[n/2+1..n][1..n/2]=P[3]+P[4]
    C[n/2+1..n][n/2+1..n]=P[5]+P[1]-P
        [3]-P[7]
    return C
```

- Adaptive 128 algorithm: hybrid algorithm using SQUARE-MATRIX-MULTIPLY(A,B) when the size is under 128 and STRASSEN(A,B) when the size is above 128. The pseudocode is much similar to adaptive 16 algorithm and would not show again.

We applied all the five algorithms to matrices whose size are from 2 × 2 to 2081 × 2081. The testing platform is Intel Core i7-7500U @2.70GHz with ROM 16G. We recorded all the code running time and plotted the data as a line graph. According to Fig. 3, we can get the following results.

a) Implementation of Optimization: The data-processing-optimized Strassen's algorithm become faster than the brute-force algorithm after $n = 1793$. The multithreaded Strassen's algorithm become faster than the brute-force algorithm after $n = 353$. And depending on the design of adaptive algorithm, it must be faster than the brute-force algorithm. So the optimization of matrix multiplication algorithm is successfully implemented.

b) Efficiency of Adaptive Method: The adaptive method shows excellent performance and its running time is less than one third of that of the violent algorithm. This is mainly due

TABLE I
CROSSOVER POINT EXPERIMENT FOR DATA-PROCESSING-OPTIMIZED STRASSEN'S ALGORITHM AND MULTITHREADED STRASSEN'S ALGORITHM AGAINST BRUTE-FORCE ALGORITHM

Serial Number	DP-Optimized	Multithreaded
1	1780	350
2	1791	332
3	1776	330
4	1785	345
5	1788	343
Average	1784	340

to the fact that Strassen algorithm is no longer used in the calculation of small-size matrix multiplication, but the brute-force algorithm with fewer steps and more direct operation is adopted in this state, which speeds up all small-size matrix multiplication, thus obtaining a faster operation speed.

c) Fitting accuracy: The optimized Strassen's algorithm is basically consistent with the curve n^2.81, and the brute-force algorithm is consistent with the curve n^3. These prove the correctness of the previous part of time complexity calculation.

d) Step Effect: Discontinuity appeared in the running time of Strassen's algorithm. This was because when the order place of the matrix was between 2^n and 2^{n+1}, the number of matrix blocks was 2^{n+1}, and the time consumed by matrix blocks was relatively large, so step effect occurred.

Though the running time of algorithms may be different on different testing platforms, but the above results still valid. The above results verify that the optimized algorithm has a faster speed compared with the violent algorithm. In particular, the adaptive algorithm shows good performance and the running time is reduced to less than one third of that of the brute-force algorithm.

For the crossover point, we conducted an experiment to find the point between the data-processing-optimized Strassen's algorithm and the brute-force algorithm and the multithreaded Strassen's algorithm and the brute-force algorithm respectively. The results (Table I) shows that the crossover point is stable over a certain interval. And of course, based on the theoretical analysis in Section III, the point value still depends on the testing platforms.

VII. CONCLUSION

In this paper, we first implemented the brute-force algorithm of matrix multiplication and Strassen's algorithm. After that, the existing algorithms are optimized from two directions of multithreading and code structure, and more ideal results are obtained. After comparison, when the matrix size is small, the brute-force algorithm is faster. But when the matrix size is large, the Strassen's algorithm has greater advantages. We tried to optimize the Strassen's algorithm in two ways: data-processing and multithreading. Both two optimized algorithm can improve the running speed. Finally, we propose an adaptive method that can select different algorithms according to the matrix itself, so as to improve the operational efficiency of matrix multiplication. Of course, the algorithm provided in this paper may still have optimization space, and we will continue to improve the work.

REFERENCES

[1] Paolo D'Alberto and Alexandru Nicolau. Adaptive Strassen's matrix multiplication. In Proceedings of the 21st Annual International Conference on Supercomputing, pages 284–292, June 2007

[2] TH Cormen, CE Leiserson, RL Rivest, C Stein. Introduction to algorithms. MIT Press, 2009