CHOOSING A BETTER ALGORITHM FOR MATRIX MULTIPLICATION

By

XING ZHANG

Master of Project Management

Western Carolina University

Cullowhee, North Carolina

1991

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirement for
the Degree of
MASTER OF SCIENCE
July, 2000

CHOOSING A BETTER ALGORITHM FOR MATRIX MULTIPLICATION

Thesis Approved:

J Chandler

Thesis Adviser

D. E. Hed

Alfred Carlozzi

Dean of the Graduate College

PREFACE

Matrix multiplication is a basic operation of linear algebra, and has numerous applications to the theory and practice of computation. Many applications can be solved fast if the algorithm of matrix multiplication is fast because it is a substantial part of these applications.

This thesis conducts the study of three algorithms; the straightforward algorithm, Winograd's algorithm, Strassen's algorithm, their time complexities, and compares the three algorithms using graphs. The thesis also briefly describes two asymptotic improvements: Pan's of 1983 and Strassen's of 1986.

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to my adviser, Dr. J. P. Chandler, for his constructive and frequent guidance, academic supervision, inspiration and friendly help. I also owe my appreciation to Dr. G. E. Hedrick and Dr. H. K. Dai. I not only learned knowledge for my committee members, but much other common sense as well. A professor has too many students to remember, but each student certainly remembers the professor who taught him.

I always give thanks to my parents, Mr. ZaiKe Zhang and Ms. Peilei Qian, whenever I have something to celebrate, and I always remember their smiling faces while listening quietly to my complaints.

I would also like to take this opportunity to thank the OSU university librarians for their timely assistance. Their assistance is so impressive that I like to have these nice people around me all the time.

My thanks also extend to the Department of Computer Science for providing excellent computing facilities and learning environment.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

LIST OF FIGURES

Figure Page

CHAPTER I

INTRODUCTION

1.1 Matrix Multiplication

Many types of problems involve arithmetic operations, whether it is multiplication, division or more complicated arithmetic operations. The two major problem areas which are the concern of arithmetic complexity of computations are:

1. What is the minimum number of arithmetic operations which are needed to perform the computation?
2. How can we obtain a better algorithm when improvement is possible [6]?

The above two questions pertain to any computation that is arithmetic in nature. From the point of view of arithmetic complexity of computation analysis, when very large numbers or large sequences of numbers are involved, some algorithms that are good for small input become inefficient when the size of the input grows [1].

Matrix multiplication is an important problem in linear algebra, and is closely related to many applications. The computational time required for matrix multiplication often is the dominant part of the total computational time required for all of those applications. That is, all such problems can be reduced to matrix multiplication and can be solved fast if matrix multiplication is solved fast [19]. The simple and straightforward algorithm is to multiply corresponding row and column entries of the two matrices, and add the results of these products to form the new entry in the new matrix. This algorithm works well when the sizes of the two input matrices are small. When the size of the two input matrices become larger, say large than 100, the computation can becomes tediously time consuming.

With this underlying question in mind, many scientists have tried to find a fast algorithm to compute the product of two matrices. A fast algorithm for matrix multiplication implies faster algorithms for many other related problems [2]. This study analyzes different approaches used to improve computational efficiency of matrix multiplication when the size of the input matrices is a major factor.

1.2 Properties of a Matrix and its Operations

This section briefly describes matrix representation and some of the operations relating to matrix multiplication.

If m and n are positive integers, then an $m \times n$ matrix (read "m by n") is a rectangular array

$$\left.\begin{bmatrix} a_{11} & a_{12} & a_{13} & . & . & . & a_{1n} \\ a_{21} & a_{22} & a_{23} & . & . & . & a_{2n} \\ a_{31} & a_{32} & a_{33} & . & . & . & a_{3n} \\ \vdots & \vdots & \vdots & & & & \vdots \\ \underbrace{a_{m1} & a_{m2} & a_{m3} & . & . & . & a_{mn}}_{n \text{ columns}} \end{bmatrix}\right\} m \text{ rows}$$

in which each entry, a_{ij} of the matrix is a number. An $m \times n$ matrix has m rows and n columns. The entry a_{ij} is located in the ith row and the jth column; i is the row subscript because it gives the position among the horizontal lines, and j the column subscript because it gives the position among the vertical lines [3].

It is standard mathematical convention to represent matrices in any one of the following three ways.

1. A matrix can be denoted by an uppercase letter such as

   $A, B, C, \ldots.$

2. A matrix can be denoted by a representative element enclosed in brackets, such as

   $[a_{ij}], [b_{ij}], [c_{ij}], \ldots.$

3. A matrix can be denoted by a rectangular array of numbers

$$A = [a_{ij}] = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{bmatrix}$$

If A and B are matrices of the same size, then the sum $A + B$ is the matrix obtained by adding the corresponding entries of A and B; that is, if A and B are both $m{\times}n$ then $C = A + B$ is the $m \times n$ matrix whose entries satisfy [5]:

$$c_{ij} = a_{ij} + b_{ij}; \qquad i = 1,2, \ldots, m, \qquad j = 1,2, \ldots, n.$$

Addition of matrices of different size is not defined, and matrix addition has the associative property, that is

$$(A + B) + C = A + (B + C).$$

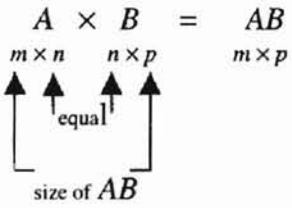
When multiplying a matrix A by a scalar c, the scalar c multiplies each entry in A, that is if $A = [a_{ij}]$ is an $m{\times}n$ matrix and c is a scalar, then the scalar multiple of A by c is the $m{\times}n$ matrix given by $cA = [c\,a_{ij}]$ [5].

If $A = [a_{ij}]$ is an $m \times n$ matrix and $B = [b_{ij}]$ is an $n \times p$ matrix, then the product AB is an $m \times p$ matrix

$$AB = [c_{ij}]$$

Where $c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj} = a_{i1}b_{1j} + a_{i2}b_{2j} + a_{i3}b_{3j} + \cdots + a_{in}b_{nj}$ [3].

This definition means that the entry in the *i*th row and the *j*th column of the product *AB* is obtained by multiplying the entries in the *i*th row of *A* by the corresponding entries in the *j*th column of *B* and then adding the results. In matrix multiplication, the number of columns of the first matrix must equal the number of rows of the second matrix. That is,



However matrix multiplication is not, in general, commutative, that is, it is usually not true that the product *AB* is equal to the product *BA* [3].

The associative property holds for matrix multiplication. If *A* is an $m \times n$ matrix, *B* is an $n \times p$ matrix, and *C* is a $p \times q$ matrix, then $(AB)C = A(BC)$ [4]. The associative law allows matrix multiplication to be computed in different orders and still have the same results.

1.3 Problem Statement

Reducing the number of computations in matrix multiplication is a challenge task.

The traditional way for computing the product of two $n \times n$ matrices uses n^3 multiplications and $n^3 - n^2$ additions. However, trying to find minimum number of arithmetic operations needed to perform the computation and obtain a faster algorithm is always challenging. A computational algorithm that requires less computation time than the regular algorithm is considered better.

By rearranging the order of the computations, Winograd's algorithm can improve the matrix multiplication by a factor of 2 over the straightforward algorithm, and Strassen's algorithm can improve the multiplication from $O(n^3)$ down to $O(n^{2.81})$ [2] by using a different approach.

After Strassen's discovery, many improvements have been made by using new techniques. Pan's method of 1983 and Strassen's method of 1986 use bilinear and trilinear algorithms to further reduced the exponent to 2.67 and 2.48 respectively. Such an improvement over the "very natural" $O(N^3)$ algorithms is theoretically important, even though any substantial impact of the asymptotically fast matrix multiplication on the practice of computing matrix products remains questionable, due to the overhead of those algorithms [19].

This thesis presents the three algorithms, compare Winograd's and Strassen's algorithms and analyzes them, and describes Pan's improvement of 1983 and Strassen's improvement of 1986. For the purpose of comparison, entries of all matrices are taken to be integers.

1.4 Organization of the Thesis

This thesis is composed of seven chapters. Chapter I gives an overview of the matrix multiplication problem as related to computational complexity. The chapter also briefly describes properties of matrices and their operations.

Chapter II presents the straightforward algorithm of matrix multiplication and its computational analysis.

Chapter III raises the importance of the relationship of computation time between input and algorithm, presents Winograd's algorithm, the methodology the algorithm uses and the computational analysis.

Chapter IV presents Strassen's algorithm, one possible deriving method and the computational analysis.

Chapter V describes the asymptotic improvement of Pan's method of 1983 and Strassen's method of 1986.

Chapter VI compares the straightforward algorithm, Winograd's algorithm, and Strassen's algorithms, their requirements and implication.

Chapter VII is the conclusion of the thesis study and a summary of these algorithms.

CHAPTER II

STRAIGHTFORWARD ALGORITHM

2.1 Algorithm Based on the Definition

The straightforward algorithm is based on the definition of matrix multiplication. The product of two matrices $A_{m\times n} = [\,a_{i\,j}\,]$ and $B_{n\times p} = [b_{i\,j}]$ is $AB = c_{i\,j} = \sum_{k=1}^{n} a_{i\,k}b_{k\,j}$.

By definition, an entry in $c_{i\,j}$ is the sum of the products of the *i*th row of *A* and the *j*th column of *B*. Then the algorithm is

```
for $i = 1$ to $m$
    for $j = 1$ to $p$
        $c_{i\,j} = 0$
        for $k = 1$ ton
            $c_{i\,j} = c_{i\,j} + a_{i\,k}b_{k\,j}$.
        end
    end
end
```

Figure 2-1: Straightforward algorithm [5]

The usual order of matrix multiplication is "row times column." If the matrices are A and B, row1 of A multiplies column 1 of B to give the (1,1) entry in the corner of AB. This product of vectors is an inner product, combining n ordinary multiplications of a_{1j} times b_{j1}. If A and B are n by n, there will be n^2 entries in AB and they normally involve n^3 ordinary multiplications [13].

The outer loop controls the row number in the matrix *A*; when the outer loop finishes its iterations, the multiplication is complete. The middle loop controls the column number in the matrix *B*; each iteration of the middle loop goes from column 1 to column p. The inner loop controls column number on matrix *A* and row number on matrix *B*; the inner loop iteration calculates the entry in the matrix *C*.

2.2 Computational Analysis

From the algorithm in Figure 2-1, the number of total additions of the matrix multiplication can be calculated easily. For each entry in c_{ij}, there are $(n-1)$ additions, where n is number of columns in *A* or the number of rows in *B*, so in one row of *C* there are p entries and a total of $(n-1) \times p$ additions, since there are m rows in *C*, so there are a total of $(n-1) \times m \times p$ additions resulting from the straightforward algorithm.

For each value i $(1 \le i \le m)$ in the outer loop, there are p iterations of middle loop, and for each value j $(1 \le j \le p)$ in the middle loop, there are n iterations of the inner loop, so there are $m \times p \times n$ multiplications.

Assuming $m \le n \le p$, then $m \times n \times p \le p \times p \times p$, so the straightforward algorithm results in $O(m^3) \le O(n^3) \le O(p^3)$ computation time. In the case of $m = n = p$, the computation time is $\Theta(n^3)$, and the total number of additions is $(n-1)n^2$.

The straightforward algorithm could be the best practical algorithm unless the matrices are very large. Cormen et al. suggests $n > 45$ [7], and J. Cohen and M. Roth suggest $n > 100$ [8]. Although some authors have different suggestions based on their testing results, for small matrices the straightforward algorithm is preferable [7], and quite efficient.

CHAPTER III

WINOGRAD'S ALGORITHM

3.1 Input vs. Computation Time

Many signal processing problems place a very heavy computational load on even the largest computers which are available, thus a modest reduction in their execution time may have practical significance [6]. In the arithmetic operations, the number of multiplications and additions required for computation has been studied systematically. Speaking of the arithmetic complexity means consideration of both the number of additions and the number of multiplications [6]. In some cases, the number of multiplications is a major factor in determining the asymptotic complexity while in some of the other applications, the attention is on the number of additions, and not just the number of multiplications [6].

In matrix multiplication, addition is preferred to multiplication, since additions can be performed more quickly than multiplications, and they contribute little to the total computational cost relative to multiplication does [19]. Therefore additions are less important than multiplications in matrix multiplication, and multiplication deserves more attention. So a more efficient algorithm is the one with less multiplication and its importance will have a profound impact on the way many computations are being performed [6].

If a program is given large amount of inputs, the algorithm needs to reconsider more the input size to improve the algorithm efficiency than otherwise for small input size algorithms do. If there is a way to rearrange the input to the matrix multiplication, so the input to the matrix multiplication requires fewer multiplications, and thus reduces computation time even with more additions than the original algorithm's, then the algorithm is more efficient and faster.

In 1968, Shmuel Winograd presented his new algorithm which cuts down the number of multiplications required and computes the product of two $n \times n$ matrices using roughly $n^3/2$ multiplications instead of n^3 multiplications which the regular method necessitates [10].

3.2 Vectors and Matrices

Matrix computations are built upon a hierarchy of linear algebraic operations. Dot products involve the scalar operations of addition and multiplication. Matrix-vector multiplication is made up of dot products. Matrix-matrix multiplication amounts to a collection of matrix-vector products [11]. Winograd's algorithm of matrix multiplication uses the concept of vector and vector multiplication (inner product) to rearrange the input to the matrix multiplication.

A vector is an entity characterized by its length and direction. A vector $x = (x_1, x_2, x_3, \cdots, x_n)$ in an $n-$space is an ordered $n-$tuple of real numbers. The set of all ordered $n-$tuples is called $n-$space and is denoted by R^n [3]. The members of R^n are column vectors; on the other hand, the elements of $R^{1\times n}$ are row vectors [11].

$$x \in R^{1\times n} \Leftrightarrow x = (x_1, x_2, x_3, \cdots, x_n)$$

The set of all $m \times n$ matrices with real-valued entries is denoted by $R^{m\times n}$ [7].

$$A \in R^{m\times n} \Leftrightarrow A = [a_{ij}] = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix}$$

and matrix multiplication is $R^{m\times n} \times R^{n\times p} \to R^{m\times p}$ [12]. If $x, y \in R^{n\times 1}$, their dot (inner) product c is

$$c = \sum_{i=1}^{n} x_i y_i = x_1 y_1 + x_2 y_2 + x_3 y_3, + \cdots, x_n y_n .$$

A matrix $A_{m\times n}$ can be viewed as a collection of column (row) vectors, that is,

$$A \in R^{m\times n} \Leftrightarrow A = [a_1, a_2, a_3, \cdots, a_n], \quad a_k \in R^m .$$

Suppose $A \in R^{m\times n}, B \in R^{n\times p}$, and $C \in R^{m\times p}$. The usual matrix multiplication procedure regards AB as an array of dot products to be computed one at a time in left-to right, top-to-bottom order [11]. Using the concept that a matrix is a collection of column (row) vectors, the product C can be computed as $m \cdot p$ inner products. Row and column matrices provide an alternative way of thinking about matrix multiplication [14]. Although equivalent mathematically, it turns out that these versions of matrix multiplication can have very different levels of performance [11].

3.3 Winograd's Algorithm

Since matrix multiplication can be represented by dot products, then for $A \in R^{m\times n}, B \in R^{n\times p}$ and $C \in R^{m\times p}$, the matrix AB can be partitioned into

$$A = \begin{bmatrix} a_1{}^T \\ \vdots \\ a_m{}^T \end{bmatrix} \qquad a_k \in R^n$$

and

$$B = [b_1, \cdots, b_p] \qquad b_k \in R^n$$

Then the algorithm to compute the AB product is

```
for $i = 1$ to $m$
    for $j = 1$ to $p$
        $c_{ij} = a_i^T b_j + c_{ij}$
    end
end
```

Figure 3-1: Matrix multiplication: Dot product version [11].

In this way, many of the properties of the dot product are direct consequences of corresponding properties of matrix multiplication.

Let $x = (x, \cdots, x_n)$ and $y = (y_1, \cdots, y_n)$ be two vectors, and if n is even, for each vector, compute the number [1]

$$\xi = \sum_{j=1}^{n/2} x_{2j-1} \cdot x_{2j}$$

and
$$\eta = \sum_{j=1}^{n/2} y_{2j-1} \cdot y_{2j} .$$

The inner product (x, y) is then given by [1]

$$(x, y) = \sum_{j=1}^{n/2} (x_{2j-1} + y_{2j})(x_{2j} + y_{2j-1}) - \xi - \eta = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n .$$

When n is odd, the formula misses the last product of components in the vectors x and y, but this can be remedied easily by adding the product.

Let A be an $m \times n$ matrix and B be an $n \times p$ matrix. Winograd's algorithm to compute the product of AB is:

$$A_i = \sum_{k=1}^{n/2} a_{i.2k-1} \cdot a_{i.2k},$$

$$B_j = \sum_{k=1}^{n/2} b_{2k-1.j} \cdot b_{2k.j}.$$

$$c_{i.j} = \begin{cases} \sum_{k=1}^{n/2} (a_{i.2k-1} + b_{2k.j})(a_{i.2k} + b_{2k-1.j}) - A_i - B_j & \text{if } n \text{ is even} \\ \sum_{k=1}^{n/2} (a_{i.2k-1} + b_{2k.j})(a_{i.2k} + b_{2k-1.j}) - A_i - B_j + a_n b_n & \text{if } n \text{ is odd} \end{cases}$$

Figure 3.2: Winograd's algorithm [10]

Winograd's algorithm shows that rearranging the order of the computations can make a difference, even for expressions such as matrix multiplication. In the algorithm, the A_i s and B_j s need to be computed only once for each row or column. To compute all the A_i s and B_j s requires only n^2 multiplications. The total number of multiplications has thus been reduced to $\frac{1}{2}n^3 + n^2$. The number of additions has increased by about $\frac{1}{2}n^3$. This algorithm is thus better than the straightforward algorithm in cases where additions can be performed more quickly than multiplications [1].

3.4 Computational Complexity

Let N denote total n-dimensional vectors, T denote inner products involving these vectors, and $\lfloor t \rfloor$ denote the integer part of t. For each A_i or B_j, there are $\lfloor n/2 \rfloor$

multiplications and $\lfloor n/2 \rfloor - 1$ additions. There are N vectors, so there are $N\lfloor n/2 \rfloor$ multiplications and $N(\lfloor n/2 \rfloor - 1)$ additions resulting from calculating NA_i or NB_j alone. For each inner product, when $n+1$ is odd, the number of multiplications will be one more than n because the inner product needs to add the product of $x_{n+1}y_{n+1}$. In either case, there are $\lfloor (n+1)/2 \rfloor$ multiplications. For example, n=2, there is one product, $\lfloor (2+1)/2) \rfloor$ =1, n=3, there are one product plus product of $x_3 y_3$, total 2 products, $\lfloor (3+1)/2 \rfloor$ = 2, similarly there are $n+\lfloor n/2 \rfloor$ +1 additions. Since there are T inner products, there are $T(\lfloor (n+1)/2 \rfloor)$ multiplications and $T(n+\lfloor n/2 \rfloor + 1)$ additions resulting from calculating the inner products. The total number of multiplications required is then $N\lfloor n/2 \rfloor$ + $T\lfloor (n+1)/2 \rfloor$. The total number of additions required is $N(\lfloor n/2 \rfloor - 1) + T(n+\lfloor n/2 \rfloor + 1)$ [10]. In the regular way, for N n-dimensional vectors, to perform T inner products, there are $Tn = Nn + (T-N)n$ multiplications as compared with $N\lfloor n/2 \rfloor + T\lfloor (n+1)/2 \rfloor = Nn + (T-N)\lfloor (n+1)/2 \rfloor$, $n > \lfloor (n+1)/2 \rfloor$ for $n > 1$, so if $T > N$, Winograd's algorithm requires fewer multiplications than the regular method. The total number of additions required is $N(\lfloor n/2 \rfloor - 1) + T(n+\lfloor n/2 \rfloor + 1)$, while the regular method requires only $T(n-1)$ additions [10].

Let A be an $m \times n$ matrix , and B be an $n \times p$ matrix. Computing the product AB is equivalent to giving $N = m + p$ vectors and performing mp inner products [10].

The total number of multiplications is

$$(m+p)n + (mp - m - p)\lfloor (n+1)/2 \rfloor$$

$$\approx (m+p)n + (mp - m - p)(n/2)$$

$$= mn + np + mnp/2 - mn/2 - np/2$$

$$= mn/2 + np/2 + mnp/2$$

$$= n^3/2 + n^2 \qquad \text{when } m = n = p$$

If m, n, p are large, Winograd's algorithm requires about $\frac{1}{2}mnp$ multiplications and $\frac{3}{2}mnp$ additions, while the regular method requires mnp multiplications and mnp additions [10].

CHAPTER IV

STRASSEN'S ALGORITHM

4.1 Block Matrix

A matrix can be subdivided or partitioned into smaller matrices by inserting horizontal and vertical lines between selected rows and columns [14]. Column and row partitionings are special cases of matrix blocking. Having a facility with block matrix notation is crucial in matrix computations because it simplifies the derivation of many central algorithms. Moreover, "block algorithms" are increasingly important in high performance computing. Block algorithm essentially means an algorithm that is rich in matrix-matrix multiplication [11].

In general, an m-by-n matrix A can be partitioned by both the rows and columns to obtain

$$A = \begin{array}{cc} \begin{bmatrix} A_{11} & \cdots & A_{1r} \\ \vdots & & \vdots \\ A_{q1} & \cdots & A_{qr} \end{bmatrix} & \begin{array}{c} m_1 \\ \\ m_q \end{array} \\ \begin{array}{ccc} n_1 & \quad\quad & n_r \end{array} & \end{array}$$

where $m_1 + \cdots + m_q = m,\quad n_1 + \cdots + n_r = n$, and $A_{\alpha\beta}$ designates the (α, β) block or submatrix. Block $A_{\alpha\beta}$ has dimension m_α-by-n_β and $A = (A_{\alpha\beta})$ is a q-by-r block matrix [11].

The concepts of matrix operations in Chapter I also carry over to block matrices provided the size of the submatrices are such that the matrix multiplications and additions are defined [14].

Block matrices combine just like matrices with scalar entries as long as certain dimension requirements are met. If

$$B=\begin{array}{cc} \left[\begin{array}{ccc} B_{11} & \cdots & B_{1r} \\ \vdots & & \vdots \\ B_{q1} & \cdots & B_{qr} \end{array}\right] & \begin{array}{c} m_1 \\ \\ m_q \end{array} \\ \begin{array}{ccc} n_1 & \qquad & n_r \end{array} & \end{array},$$

then B is partitioned conformably with the matrix A. The sum $C = A + B$ can also be regarded as a q-by-r block matrix [11]:

$$C=\left[\begin{array}{ccc} C_{11} & \cdots & C_{1r} \\ \vdots & & \vdots \\ C_{q1} & \cdots & C_{qr} \end{array}\right]=\left[\begin{array}{ccc} A_{11}+B_{11} & \cdots & A_{1r}+B_{1r} \\ \vdots & & \vdots \\ A_{q1}+B_{qr} & \cdots & A_{qr}+B_{qr} \end{array}\right].$$

Lemma 4.1.1 [11] If $A \in R^{m\times p}, B \in R^{p\times n}$,

$$A=\begin{array}{cc} \left[\begin{array}{c} A_1 \\ \vdots \\ A_q \end{array}\right] & \begin{array}{c} m_1 \\ \\ m_q \end{array} \end{array} \qquad\qquad B=\begin{array}{c} [B_1,\cdots,B_r] \\ \begin{array}{ccc} n_1 & \quad & n_r \end{array} \end{array},$$

then

$$AB=C=\begin{array}{cc} \left[\begin{array}{ccc} C_{11} & \cdots & C_{1r} \\ \vdots & & \vdots \\ C_{q1} & \cdots & C_{qr} \end{array}\right] & \begin{array}{c} m_1 \\ \\ m_q \end{array} \\ \begin{array}{ccc} n_1 & \qquad & n_r \end{array} & \end{array}$$

where $C_{\alpha\beta} = A_\alpha B_\beta$ for $\alpha = 1,\cdots,q$ and $\beta = 1,\cdots,r$.

Proof [11]. Relate scalar entries in block $C_{\alpha\beta}$ to scalar entries in C. For $1 \le \alpha \le q$, $1 \le \beta \le r$, $1 \le i \le m_a$, and $1 \le j \le n_\beta$, one has

$$[C_{\alpha\beta}]_{ij} = c_{\lambda+i,\mu+j} \qquad \text{where } \lambda = m_1+\cdots+m_{\alpha-1},\ \mu = n_1+\cdots+n_{\beta-1}.$$

But $$c_{\lambda+i,\mu+j} = \sum_{k=1}^{p} a_{\lambda+i,k} b_{k,\mu+j} = \sum_{k=1}^{p} [A_\alpha]_{ik} [B_\beta]_{kj} = [A_\alpha B_\beta]_{ij}.$$

Thus, $C_{\alpha\beta} = A_\alpha B_\beta$.

Lemma 4.1.2 [11] If $A \in R^{m\times p}, B \in R^{p\times n}$,

$$A = \begin{array}{c} [A_1, \cdots, A_s] \\ \;p_1 \quad\;\; p_s \end{array} \quad , \text{ and } \quad B = \begin{array}{c|c} \begin{bmatrix} B_1 \\ \vdots \\ B_s \end{bmatrix} & \begin{array}{c} p_1 \\ \\ p_s \end{array} \end{array}$$

then

$$AB = C = \sum_{\gamma=1}^{s} A_\gamma B_\gamma .$$

Proof [15]. For $1 \le i \le m$ and $1 \le j \le n$,

$$\begin{aligned} c_{ij} &= \sum_{k=1}^{p} a_{ik} b_{kj} \\ &= \sum_{k=1}^{p_1} a_{ik} b_{kj} + \sum_{k=p_1+1}^{p_1+p_2} a_{ik} b_{kj} + \cdots + \sum_{k=p_1+\cdots+p_{s-1}+1}^{p} a_{ik} b_{kj} \\ &= [A_1B_1]_{ij} + [A_2B_2]_{ij} + \cdots + [A_sB_s]_{ij} \\ &= [A_1B_1 + A_2B_2 + \cdots + A_sB_s]_{ij} . \end{aligned}$$

Thus

$$AB = C = \sum_{\gamma=1}^{s} A_\gamma B_\gamma .$$

Theorem 4.1.3 [11] if

$$A = \begin{array}{cc} \begin{bmatrix} A_{11} & \cdots & A_{1s} \\ \vdots & & \vdots \\ A_{q1} & \cdots & A_{qs} \end{bmatrix} & \begin{array}{c} m_1 \\ \\ m_q \end{array} \\ \begin{array}{ccc} p_1 & \quad & p_s \end{array} & \end{array} \quad , \qquad B = \begin{array}{cc} \begin{bmatrix} B_{11} & \cdots & B_{1r} \\ \vdots & & \vdots \\ B_{s1} & \cdots & B_{sr} \end{bmatrix} & \begin{array}{c} p_1 \\ \\ p_s \end{array} \\ \begin{array}{ccc} n_1 & \quad & n_r \end{array} & \end{array} \quad ,$$

the partition of the product $C = AB$ is as follows,

$$C = \begin{array}{cc} \begin{bmatrix} C_{11} & \cdots & C_{1r} \\ \vdots & & \vdots \\ C_{q1} & \cdots & C_{qr} \end{bmatrix} & \begin{array}{c} m_1 \\ \\ m_q \end{array} \\ \begin{array}{ccc} n_1 & \quad & n_r \end{array} & \end{array} \quad ,$$

then

$$C_{\alpha\beta} = \sum_{\gamma=1}^{s} A_{\alpha\gamma} B_{\gamma\beta} \qquad \alpha = 1, \cdots, q\,, \quad \beta = 1, \cdots, r\,.$$

Proof. Set

$$A_{\gamma} = \begin{bmatrix} A_{1\gamma} \\ \vdots \\ A_{q\gamma} \end{bmatrix} \qquad \text{and} \qquad B_{\gamma} = [B_{\gamma 1}, \cdots, B_{\gamma r}]$$

from Lemma 4.1.2 $\quad C = \sum_{\gamma=1}^{s} A_{\gamma} B_{\gamma} = [A_1 B_1] + [A_2 B_2] + \cdots + [A_s B_s]\,.$

from Lemma 4.1.1 $\quad C_{\alpha\beta} = A_{\alpha} B_{\beta}$

then

$$[C_{\alpha\beta}]_{11} = [A_{\alpha} B_{\beta}]_{11} = A_{\alpha 1} B_{1\beta}$$

$$[C_{\alpha\beta}]_{22} = [A_{\alpha} B_{\beta}]_{22} = A_{\alpha 2} B_{2\beta}$$

$$\vdots$$

$$[C_{\alpha\beta}]_{ss} = [A_{\alpha} B_{\beta}]_{ss} = A_{\alpha s} B_{s\beta}$$

$$C_{\alpha\beta} = [\sum_{\gamma=1}^{s} A_{\gamma} B_{\gamma}]_{\alpha\beta} = [A_{\alpha 1} B_{1\beta}] + [A_{\alpha 2} B_{2\beta}] + \cdots + [A_{\alpha s} B_{s\beta}]$$

$$= \sum_{\gamma=1}^{s} A_{\alpha\gamma} B_{\gamma\alpha}$$

By proving that matrix product AB exists and A, B are partitioned into blocks, the product in terms of blocks produced by the partitions is obtained formally in the same way as that in terms of the scalar elements [16]. When matrix A and B are conformably partitioned, the partition of the columns of A must be the same as the partition of the rows of B, but there is no restriction on the partition of the rows of A or the columns of B [17].

Block multiplication of matrices is of considerable theoretical use, but block multiplication of matrices also has practical use in large-scale computation [15]. Since for two conformably partitioned matrices multiplication is treated just like ordinary scalar-level matrix multiplication, block matrix multiplication can also be partitioned in several possible ways so as their corresponding version of algorithms [14].

4.2 Strassen's Algorithm

In 1968, Volker Strassen discovered a new algorithm to compute the product of two matrices, and reduced the computation time to $O(n^{2.81})$ which is quite efficient for very large matrix multiplications. Although Strassen's algorithm does not use the full theorem of block matrix multiplication, the algorithm does apply the partition of matrix to divide two matrices into half of their size in a series steps to reduce the number of multiplications.

Strassen's algorithm uses a divide-and-conquer method and it is iterative in nature, that is, it reduces the problem of multiplying $n \times n$ matrices to several instances of smaller problems [6].

Assume matrix A and B are two $n \times n$ matrices, and n is power of 2. Let $n = 2m$, then $m = n/2$. Use m to partition both matrices of A and B into four $m \times m$ matrices,

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

The product $AB = C$

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

where the A_{ij}'s, B_{ij}'s and C_{ij}'s are $n/2 \times n/2$ matrices. Using block matrix multiplication, each submatrix is treated as are elements of the $n \times n$ matrices. Then the elements in C is [6]

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21}, \quad C_{12} = A_{11} \times B_{12} + A_{12} \times B_{12},$$

$$C_{21} = A_{21} \times B_{11} + A_{22} \times B_{21}, \quad C_{22} = A_{21} \times B_{12} + A_{22} \times B_{22}.$$

If computing the C_{ij}'s in the straightforward way, there are 8 multiplications. Replacing each multiplication by a recursive call, the recurrence relation is $T(n) = 8T(n/2) + O(n^2)$, which implies that $T(n) = O(n^{\log_2 8}) = O(n^3)$. If there is an algorithm that computes the product of two 2×2 matrices with fewer than 8 multiplications, then the algorithm is asymptotically faster than cubic [1].

The most important part of the recursion is how many multiplications are required to compute the product of two 2×2 matrices. In the design of an algorithm for matrix multiplication, the number of additions does not carry the same weight as the number of multiplications since they always contribute $O(n^2)$ to the recurrence relation, which is not a factor in determining the asymptotic complexity although it does affect the constant factor [1]. Strassen discovered a unique recursive approach in which 7 multiplications and $\Theta(n^2)$ scalar additions and subtractions are enough to compute the product of two 2×2 matrices, and it has a recurrence relation $T(n) = 7T(n/2) + \Theta(n^2) = O(n^{\log_2 7})$ = $O(n^{2.81})$ [7].

Strassen did not give the details of how and what method he used to discovered the submatrix products that make his algorithm work [1][7]. However, his method has four steps:

1. Partition the input matrices A and B into $n/2 \times n/2$ submatrices.

2. Using $\Theta(n^2)$ scalar additions and subtractions, compute 14 $n/2 \times n/2$ matrices.

3. Recursively compute the 7 matrix products P_i.

4. Using additions and subtractions of the P_i to compute the desired submatrices C_{ij} [7].

Udi Manber [1] thinks that the following is possibly a method that could have been used by Strassen to find the algorithm.

Computing the product

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & g \\ f & h \end{bmatrix} = \begin{bmatrix} p & s \\ r & t \end{bmatrix}$$

is equivalent to computing the product

$$\begin{bmatrix} a & b & 0 & 0 \\ c & d & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & c & d \end{bmatrix} \begin{bmatrix} e \\ f \\ g \\ h \end{bmatrix} = \begin{bmatrix} p \\ r \\ s \\ t \end{bmatrix}$$

Write the above matrix multiplication as $A \cdot X = Y$, and look for ways to minimize the number of multiplications required to evaluate Y. The following are four types of special products that are easy to compute.

Type	Product	No. Multiplications
$\alpha)$	$\begin{bmatrix} a & a \\ a & a \end{bmatrix} \begin{bmatrix} e \\ f \end{bmatrix} = \begin{bmatrix} a(e+f) \\ a(e+f) \end{bmatrix}$	1
$\beta)$	$\begin{bmatrix} a & a \\ -a & -a \end{bmatrix} \begin{bmatrix} e \\ f \end{bmatrix} = \begin{bmatrix} a(e+f) \\ -a(e+f) \end{bmatrix}$	1

$$\gamma)\quad \begin{bmatrix} a & 0 \\ a-b & b \end{bmatrix} \begin{bmatrix} e \\ f \end{bmatrix} = \begin{bmatrix} ae \\ ae+b(f-e) \end{bmatrix} \qquad 2$$

$$\delta)\quad \begin{bmatrix} a & b-a \\ 0 & b \end{bmatrix} \begin{bmatrix} e \\ f \end{bmatrix} = \begin{bmatrix} a(e-f)+bf \\ bf \end{bmatrix} \qquad 2$$

If $A \cdot X = Y$ can be expressed in the form of a product of several steps of the types listed above, then it may require fewer multiplications since these types of products use fewer than that the actual number of multiplications. Let

$$B = \begin{bmatrix} b & b & 0 & 0 \\ b & b & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \qquad C = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & c & c \\ 0 & 0 & c & c \end{bmatrix}$$

$$D = \begin{bmatrix} 0 & 0 & 0 & 0 \\ c-b & 0 & 0 & c-b \\ b-c & 0 & 0 & b-c \\ 0 & 0 & 0 & 0 \end{bmatrix}, \qquad E = \begin{bmatrix} a-b & 0 & 0 & 0 \\ 0 & d-b & 0 & b-c \\ c-b & 0 & a-c & 0 \\ 0 & 0 & 0 & d-c \end{bmatrix}.$$

Then, $A = (B + C + D + E)$ and therefore $AX = BX + CX + DX + EX$. BX, CX and DX can be computed with one multiplication using types α or β, but not EX. However E can be expressed as the sum of two matrices $E = F + G$, such that F is of type γ and G is of type δ:

$$F = \begin{bmatrix} a-b & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ c-b & 0 & a-c & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad G = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & d-b & 0 & b-c \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & d-c \end{bmatrix}.$$

So $AX = (BX + CX + DX + EX) \quad = (BX + CX + DX + FX + GX)$. Overall it takes two products of type α, and one product each of type β, γ, and δ.

Let $z_1 = b(e+f)$, $\quad z_2 = c(g+h)$,

$z_3 = (c-b)(e+h)$, $\quad z_4 = (a-b)e$,

$z_5 = (a-c)(g-e)$, $\quad z_6 = (d-c)h$,

$z_7 = (d-b)(f-h)$.

Each of the matrices B, C, D, F and G contributes to the product AX:

B contributes $\begin{bmatrix} z_1 \\ z_1 \\ 0 \\ 0 \end{bmatrix}$, $\quad C$ contributes $\begin{bmatrix} 0 \\ 0 \\ z_2 \\ z_2 \end{bmatrix}$, $\quad D$ contributes $\begin{bmatrix} 0 \\ z_3 \\ -z_3 \\ 0 \end{bmatrix}$,

F contributes $\begin{bmatrix} z_4 \\ 0 \\ z_4 + z_5 \\ 0 \end{bmatrix}$, $\quad G$ contributes $\begin{bmatrix} 0 \\ z_6 + z_7 \\ 0 \\ z_6 \end{bmatrix}$.

So overall, $p = z_1 + z_4$, $\quad r = z_1 + z_3 + z_6 + z_7$, $\quad s = z_2 + z_3 + z_4 + z_5$, and $t = z_2 + z_6$ [1].

Apparently, choosing different matrices of α, β, γ and δ will lead to different z_i's, and thus will have different p, r, s, and t representations. Strassen could have used a similar method to discover the algorithm. The algorithm first proceeds by computing the following seven matrices [6]:

$P_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$, $\quad P_2 = (A_{21} + A_{22}) \times B_{11}$,

$P_3 = A_{11} \times (B_{12} - B_{22})$, $\quad P_4 = A_{22} \times (B_{21} - B_{11})$,

$P_5 = (A_{11} + A_{12}) \times B_{22}$, $\quad P_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12})$,

$P_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$.

Then the algorithm uses these seven matrices to computes the C_{ij}'s:

$$C_{11} = P_1 + P_4 - P_5 + P_7, \qquad C_{12} = P_3 + P_5,$$

$$C_{21} = P_2 + P_4, \qquad C_{22} = P_1 + P_3 - P_2 + P_6.$$

Altogether Strassen's algorithm calls for computing 18 additions of $m \times m$ matrices and 7 multiplications of $m \times m$ matrices [6]. The conventional matrix multiplication involves $(2m)^3$ multiplications and $(2m)^3 - (2m)^2$ additions. If Strassen's algorithm is applied with conventional multiplication at block level, then $7m^3$ multiplications and $7m^3 + 11m^2$ additions are required. If $m >> 1$, then at the each step, the Strassen method involves about $7/8ths$ the arithmetic of the fully conventional algorithm [11].

Starting from the 2-by-2 block matrix multiplication, Strassen's algorithm uses a completely different approach to the matrix-matrix multiplication problem. Strassen's algorithm can be applied to each of the half-sized block multiplication associated with the P_i. Thus, if the original A and B are n-by-n and $n = 2^q$, then Strassen's multiplication algorithm can be applied recursively. As the partitioning of matrices continues, there is no need to recur down to the $n = 1$ level. When the block size gets sufficiently small, that is $n \leq n_{\min}$, it may be sensible to use the conventional matrix multiplication algorithm [11].

The following is the overall procedure of the Strassen's algorithm. Suppose $n = 2^q$ and that $A \in R^{n \times n}$ and $B \in R^{n \times n}$. If $n_{\min} = 2^d$ with $d \leq q$, then the algorithm computes $C = AB$ by applying Strassen's procedure recursively $q - d$ times [11].

```
$C = strass(A, B, n, n_{\min})$
    $if\ n \le n_{\min}$
        $C = AB$
    $else$
        $m = n/2$
        $u = 1\ to\ m$
        $v = m+1\ to\ n$
        $P_1 = strass(A(u,u) + A(v,v),\ B(u,u) + B(v,v), m, n_{\min})$
        $P_2 = strass(A(v,u) + A(v,v),\ B(u,u), m, n_{\min})$
        $P_3 = strass(A(u,u), B(u,v) - B(v,v), m, n_{\min})$
        $P_4 = strass(A(v,v), B(v,u) - B(u,u), m, n_{\min})$
        $P_5 = strass(A(u,u) + A(u,v), B(v,v), m, n_{\min})$
        $P_6 = strass(A(v,u) - A(u,u),\ B(u,u) + B(u,v), m, n_{\min})$
        $P_7 = strass(A(u,v) - A(v,v),\ B(v,u) + B(v,v), m, n_{\min})$
        $C(u,u) = P_1 + P_4 - P_5 + P_7$
        $C(u,v) = P_3 + P_5$
        $C(v,u) = P_2 + P_4$
        $C(v,v) = P_1 + P_3 - P_2 + P_6$
    $end$
```

Figure 4-1: Strassen's algorithm [11].

Strassen's algorithm is one of the most striking examples of a nonintuitive algorithm for a seemingly simple problem [1]. The algorithm is recursive; it calls itself until $n = n_{\min}$. When this condition is reached, the process of partitioning of the matrices stops the recursion. The blocks will then have the size $n_{\min}$.

Strassen's algorithm assumes that the multiplying matrices have the size $n \times n$, and n is a power of 2. In the case when n is not a power of 2, the two matrices can be modified by adding two matrices by $0's$ so as to make their dimension n', a power of 2 [6].

4.3 Computational Complexity

Compared with the straightforward algorithm which requires approximately $2n^3$ (n^3 multiplications and $n^3 - n^2$ additions) arithmetic operations, Strassen's algorithm computes the coefficients of the product of two square matrices A and B of order n from the coefficients of A and B with fewer than $4.7n^{\log_2 7}$ arithmetic operations [18].

Define algorithms $\alpha_{m,k}$ which multiply matrices of order $m2^k$ with $m^3 7^k$ multiplications and $(5+m)m^2 7^k - 6(m2^k)^2$ additions and subtractions of numbers by induction on k:

$$k = 0, \quad \alpha_{m,0} \qquad m^3 7^0 = m^3$$

$$(5+m)m^2 7^0 - 6(m2^0)^2 = 5m^2 + m^3 - 6m^2 = m^3 - m^2$$

$\alpha_{m,0}$ is the usual algorithm for matrix multiplication requiring m^3 multiplications and $m^2(m-1)$ additions.

Assume $\alpha_{m,k}$ is true for the algorithm with $m^3 7^k$ multiplications and $(5+m)m^2 7^k - 6(m2^k)^2$ additions and subtractions of numbers.

Define $\alpha_{m,k+1}$ as follows:

If A, B are matrices of order $m2^{k+1}$ to be multiplied, write

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

Where the A_{ik}, B_{ik}, C_{ik} are matrices of order $m2^k$. Then compute P_1, P_2, P_3, P_4, P_5, P_6, P_7, C_{11}, C_{12}, C_{21} and C_{22} defined in section 4.2 using $\alpha_{m,k}$ for multiplication and the usual algorithm for addition and subtraction of matrices of order $m2^k$ [18].

Proof. $m^3 7^{k+1}$ and $(5+m)m^2 7^{k+1} - 6(m2^{k+1})^2$. From the algorithm:

	No. multiplications	No. additions and subtractions
P_1	$m^3 7^k$	$(m2^k)^2 + (m2^k)^2 + (5+m)m^2 7^k - 6(m2^k)^2$
P_2	$m^3 7^k$	$(m2^k)^2 + (5+m)m^2 7^k - 6(m2^k)^2$
P_3	same as P_2	same as P_2
P_4	same as P_2	same as P_2
P_5	same as P_2	same as P_2
P_6	same as P_1	same as P_1
P_7	same as P_1	same as P_1
C_{11}		$3(m2^k)^2$
C_{12}		$(m2^k)^2$
C_{12}		$(m2^k)^2$
C_{22}		$3(m2^k)^2$

The total number of multiplications is $7m^3 7^k = m^3 7^{k+1}$ and since the sum of two $m2^k \times m2^k$ matrices uses $(m2^k)^2$ additions, the total number of additions and subtractions is $7((5+m)m^2 7^k) - 6(m2^k)^2) + 18(m2^k)^2$.

$$(5+m)m^2 7^{k+1} - 6(m2^{k+1})^2$$

$$= 7(5+m)m^2 7^k - 6(m2^k 2)^2$$

$$= 7(5+m)m^2 7^k - 24(m2^k)^2$$

$$= 7(5+m)m^2 7^k - 42(m2^k)^2 + 18(m2^k)^2$$

$$= 7((5+m)m^2 7^k - 6(m2^k)^2) + 18(m2^k)^2$$

Fact 1. $\alpha_{m,k}$ computes the product of two matrices of order $m2^k$ with $m^3 7^k$ multiplications and $(5+m)m^2 7^k - 6(m2^k)^2$ additions and subtractions of numbers. Thus, the algorithm multiplies two matrices of order 2^k with 7^k multiplications and fewer than $6 \cdot 7^k$ additions and subtractions [18].

Fact 2. The product of two matrices of order n may be computed with $< 4.7 \cdot n^{\log_2 7}$ arithmetic operations.

Proof [18]. Put

$$k = [\log_2 n - 4], \qquad (1)$$

$$m = [n2^{-k}] + 1. \qquad (2)$$

from (2) $\qquad m - 1 = n / 2^k$

$$2^k (m-1) = n$$

then $\qquad n \leq m2^k$.

Embedding matrices of order n into matrices of order $m2^k$ reduces the task to that of estimating the number of operations of $\alpha_{m,k}$. By Fact 1 this number is

$$m^3 7^K + (5+m)m^2 7^k - 6(m2^k)^2$$

$$= (5 + 2m)m^2 7^k - 6(m2^k)^2$$

$$< (5 + 2(n2^{-k} + 1))(n2^{-k} + 1)^2 7^k$$

$$< 2n^3 (7/8)^k + 12.03n^2 (7/4)^k$$

from (1) $k = [\log_2 n - \log_2 16]$

$$\log_2 n/16 = k$$

$$2^k = n/16$$

use $16 \cdot 2^k \le n$

$$= (2(8/7)^{\log_2 n-k} + 12.03(4/7)^{\log_2 n-k})n^{\log_2 7}$$

let $t = \log_2 n - k$

$$\le \max_{4 \le t \le 5}(2(8/7)^t + 12.03(4/7)^t n^{\log_2 7}$$

$$\le 4.7 \cdot n^{\log_2 7}$$

by a convexity argument [18].

In the general case when n is not a power of 2, embed each matrix in a matrix whose dimension is the next higher power of 2. A more careful "padding", as well as using the regular way for multiplying two matrices where n is small shows that the total number of operations for multiplying two $n \times n$ matrices need not exceed $4 \times n^{\log_2 7}$ [6].

Since $\log_2 7 \approx 2.807$, asymptotically the number of multiplications in Straseen's procedure is $O(n^{2.807})$. However, the number of additions (relative to the number of multiplications) becomes significant as $n_{\min}$ gets small [11].

CHAPTER V

ASYMPTOTIC IMPROVEMENT FOR MATRIX MULTIPLICATION

Strassen's publication of his discovery became one of the exciting news and the most frequently cited results in the study of the time-complexity of algorithms [19]. Strassen's result also encourages us to think that it is possible to further accelerate the matrix multiplication speed by designing new algorithms using some different ideas and techniques.

Strassen's algorithm remained the fastest for about ten years [19]. Many attempts have been made to improve Strassen's famous result with little progress, but much more significant is the use of a new approach of approximating algorithms to find a bound for the multiplication [25]. In 1978, V. Ya. Pan published his asymptotic improvement; the result of was $O(n^{2.795})$ [20]. Pan introduced a new technique of trilinear operations of aggregating, uniting and canceling, and applied the technique to construct a fast linear non-commutative algorithm for matrix multiplication [20]. Pan's method enables construction of a fast algorithm and is asymptotically faster that Strassen's algorithm.

Pan's method also resulted in many attempts to find an improvement not only for matrix multiplication, but also for an asymptotically faster algorithm for questions where matrix multiplication is a substantial part of algorithms for other computational problems of linear algebra and combinatorics [19]. The current best result for matrix multiplication – in terms of asymptotic running time – is $O(n^{2.376})$ [21] by Coppersmith and Winograd.

Because these algorithms have relaxed the ground rules of matrix multiplication, and use more complex techniques and some algorithms do not compute a matrix product at all [21], this chapter only describes Pan's method of 1983 and Strassen's method of 1986.

5.1 Some Notation and Definitions

Any systematic study of algorithms must begin by making precise the notion of the algorithm under investigation [6]. Therefore before further describing Pan's and Strassen's methods, it is necessary to have some notation and definitions.

Notation of LA ([20][22]): If a fast linear algorithm of a certain type (LA) for multiplying two matrices of a specific size was given, then fast algorithms for matrix multiplication, matrix inversion, evaluation of a determinant and solving linear systems of equations, and for some other important problem of any large size could be immediately constructed.

Bilinear forms ([6]). A general system of bilinear forms can be written as

$$f_k = \sum_{j=1}^{s} \sum_{i=1}^{r} a_{ijk} x_i y_j, \qquad k = 1, 2, \cdots, t,$$

where $\{x_1, \cdots, x_r\}$ and $\{y_{1,} \cdots, y_s\}$ are two sets of indeterminates, and the a_{ijk}'s are elements of the field of constants.

Definition ([23]): Bilinear algorithms for matrix multiplication. Given a pair of $m \times n$ and $n \times p$ matrices $X = [x_{ij}]$, $Y = [y_{jk}]$, compute XY in the following order: First evaluate the linear forms in the x-variables and in the y-variables,

$$L_q = \sum f(i, j, q) x_{ij}, \qquad L'_q = \sum_{j,k} f'(j, k, q) y_{jk}, \qquad (5.1)$$

then the products $P_q = L_q L'_q$ for $q = 0,1,\cdots,M-1$, and finally the entries $\sum_j x_{ij} y_{jk}$ of XY, as the linear combinations

$$\sum_j x_{ij} y_{jk} = \sum_{q=0}^{M-1} f''(k,i,q) L_q L'_q, \qquad (5.2)$$

where f_{ijq}, f'_{jkq}, and f''_{kiq} are constants such that (5.1) and (5.2) are the identities in the indeterminates x_{ij}, y_{jk}, for $i = 0,1,\cdots,m-1$; $j = 0,1,\cdots,n-1$; $k = 0,1,\cdots,p-1$. M, the total number of all multiplications of L_q by L'_q is called the rank of the algorithm, and the multiplications of L_q by L'_q are called the bilinear steps of the algorithm, or the bilinear multiplications of the algorithm.

Assume $n = s^h$ for a fixed but sufficiently large s and for $h \to \infty$. The bilinear algorithm (5.1), (5.2) can be applied where all the x_{ij}, y_{jk} and consequently, all the L_q and L'_q denote $s \times s$ matrices for a natural s. Then recursively apply this bilinear algorithm with successive substitutions of $s^g \times s^g$ matrices for x_{ij} and y_{jk} (for all i, j and k), where multiplying a pair of $n \times n$ matrices with $n = s^{g+1}$, $g = h-1,\ h-2,\cdots,1$. This defines a recursive bilinear algorithm for computing XY [23].

Both the straightforward algorithm and Strassen's algorithm are bilinear. The straightforward algorithm for $m \times n$ by $n \times p$ matrix multiplication has rank mnp, and Strassen's algorithm for 2×2 matrix multiplication has rank 7. In both algorithms, the constants f, f', f'' take only the values 0,1, and (for Strassen's algorithm) –1 [24].

If for some pair n, M there exists an algorithm for $n \times n$ matrix multiplication that involves only M multiplications and gives rise to a Strassen-like recursive construction, then

$$\omega \leq \log_2 M / \log_2 n$$

where ω is exponent of matrix multiplication [24].

Theorem ([24]). Given a bilinear algorithm (5.1), (5.2) for $m \times n \times p$ matrix multiplication, such that $mnp > 1$, then

$$\omega \leq 3 \log_2 M / \log_2 (mnp)$$

The theorem reduces estimating the asymptotic complexity of matrix multiplication (represented by ω) to estimating the rank of a specific problem (m, n, p), so it "only" remains to choose appropriate m, n, p and to design a bilinear algorithm (5.1), (5.2) of a smaller rank M. The exponent depends only on the number of bilinear steps of the algorithm (5.1), (5.2) or equivalently on the number of bilinear products $L_q L'_q$ in (5.2) and that the number of linear operations involved in the basic bilinear algorithm does not influence the exponent [19].

Definition of tensor [27][28]: A tensor is an entity having a unique set of components in a given coordinate system. A tensor of rank r associated with a point P of an n-dimensional metric manifold V_n is an r-linear form in the base-vectors associated with the point whose coefficients are in general functions of the co-ordinates of the point and which is invariant to choice of co-ordinate system.

A tensor component may have a set of numbers called indices consisting of either superscripts or subscripts or a combination to indicate its component. The exact number of indices is termed the tensor rank [26].

The number of components in a given tensor depends on both its rank and on "tensor dimensionality". The lowest rank that a tensor may have is 0, and the number of components possessed by a tensor of rank 0 is always 1 regardless of the dimensionality of the tensor. A common name for such a tensor is "scalar" [26].

Scalar tensors may have any rank, A rank-zero scalar tensor is a "scalar scalar ", a rank-one scalar tensor is a "scalar vector", and a rank-two scalar tensor is a "scalar matrix". The number of components in a tensor of rank R in a space of dimensionality D is given by the formula $Z = D^R$ [26].

Schönhage's $\tau -$ theorem [21][25]: Assume we are given a file F and coefficients α_{ijhl}, β_{jkhl}, γ_{kihl} in F(λ) (the field of rational functions in a single indeterminate λ), such that

$$\sum_{l=1}^{L}\left(\sum_{ijh}\alpha_{ijhl}x_{ij}^{(h)}\right)\left(\sum_{jkh}\beta_{jkhl}y_{jk}^{(h)}\right)\left(\sum_{kih}\gamma_{kihl}z_{ki}^{(h)}\right)$$

$$=\sum_{h}\left(\sum_{i=1}^{m_h}\sum_{j=1}^{n_h}\sum_{k=1}^{p_h}x_{ij}^{(h)}y_{k}^{(h)}z_{ki}^{(h)}\right)+\sum_{g>0}\lambda^{g}f_{g}(x_{ij}^{(h)},y_{jk}^{(h)},z_{ki}^{(h)})$$

is an identity in $x_{ij}^{(h)}$, $y_{jk}^{(h)}$, $z_{ki}^{(h)}$, λ, where f_g are arbitrary trilinear forms. Given $\varepsilon > 0$, an algorithm can be constructed to multiply $N \times N$ square matrices in $O(N^{3\tau+\varepsilon})$ operations, where τ satisfies $L = \sum_{h}(m_h n_h p_h)^{\tau}$. Using $O(\lambda)$ as the error term, the above hypothesis becomes

$$\sum_{h}\left(\sum_{ijk}x_{ij}^{(h)}y_{jk}^{(h)}z_{ki}^{(h)}\right)+O(\lambda).$$

Less formally, the hypothesis is a trilinear algorithm, using L bilinear multiplications to (approximately) compute simultaneously several independent matrix products, of dimension $m_h \times n_h$ times $n_h \times p_h$ (written $< m_h, n_h, p_j >$) [25].

Each of the L bilinear multiplications is a linear combination of x variables, times a linear combination of y variables:

$$M_l = \left(\sum_{ijh} \alpha_{ijhl} x_{ij}^{(h)} \right) \left(\sum_{jkh} \beta_{jkhl} y_{jk}^{(h)} \right)$$

Linear combinations of these products M_l are identically equal (up to errors of order λ) to the desired elements w of the answer matrix:

$$w_{ik}^{(h)} \overset{def}{=} \sum_{j=1}^{n_h} x_{ij}^{(h)} y_{jk}^{(h)} = \sum_{l=1}^{L} y_{kihl} M_l + O(\lambda).$$

Multiplying both sides by $z_{ki}^{(h)}$, which can be viewed as a dual to $w_{ik}^{(h)}$, and summing, to obtain the single trilinear identity, this identity contains all the information of the several bilinear identities. In such a situation, the matrix exponent obtained from the construction is defined as $\varepsilon = 3\tau$ [21].

5.2 Pan's Improvement (1983 [19][24])

Let $X = [x_{ij}], Y = [y_{jk}], U = [u_{jk}]$ and $V = [v_{ki}]$ denote $m \times n$, $n \times p$, $n \times p$, $p \times m$ matrices respectively, $i = 0, \cdots, m-1$; $j = 0, \cdots, n-1$; $k = 0, \cdots, p-1$ and assume no relations among the entries of the four given matrices, using the following bilinear algorithm for the simultaneous evaluation of two matrix products XY and UV.

$$\sum_i x_{ij} y_{jk} = \sum_i (x_{ij} + u_{jk})(y_{jk} + v_{ki}) - \sum_j u_{jk} y_{jk} - \sum_j (x_{ij} + u_{jk}) v_{ki} \quad \text{for all } k, i \quad (5.3)$$

$$\sum_k u_{jk} v_{ki} = \sum_k (x_{ij} + u_{jk})(y_{jk} + v_{ki}) - \sum_k u_{jk} y_{jk} - x_{ij} \sum (y_{jk} + v_{ki}) \text{ for all } i, j \quad (5.4)$$

Evaluate the mnp bilinear products $(x_{ij} + u_{jk})(y_{jk} + v_{ki})$ for all i, j, k; the mn products $x_{ij} \sum_k (y_{jk} + v_{ki})$ for all i, j; the np products $u_{jk} y_{jk}$ for all j, k, and the pm products $\sum_j (x_{ij} + u_{jk}) v_{ki}$ for all k, i. This amounts to a total of

$$M_{m,n,p} = mnp + mn + np + pm \text{ [24]} \qquad (5.5)$$

bilinear products of the form $L_q L'_q$, and $M_{m,n,p}$ is the rank of the bilinear algorithm. After calculating all the $M_{m,n,p}$ products $L_q L'_q$, use (5.3), (5.4) to evaluate the matrix product XY and UV as linear combinations of those $L_q L'_q$ with the coefficients f'' equal to 0,1 and -1 and defined by (5.3), (5.4).

Because no relations were assumed among the entries of the four matrices, the products of XY and UV are disjoint matrix product and the entries are indeterminate.

From (5.3), (5.4), the bound ω can be deduced

$$\omega \leq 3 \log_2 (M_{m,n,p} / 2) \log_2 (nmp).$$

Because the algorithm (5.3), (5.4) simultaneously solves two equally hard problems (m, n, p) and (n, p, m), a factor of 2 appears in the bound [24].

Modify (5.3), and (5.4) by introducing an auxiliary nonzero parameter λ to accentuate the power of the algorithm (5.3), (5.4) [19]

$$\sum_j x_{ij} y_{jk} = \sum_j (x_{ij} + u_{jk})(y_{jk} + \lambda v_{ki}) - \sum_j u_{jk} y_{jk} - \lambda \sum_j (x_{ij} + u_{jk}) v_{ki} \qquad (5.5)$$

$$\sum_k u_{jk} v_{ki} = \lambda^{-1} \left\{ \sum_k (x_{ij} + u_{jk})(y + \lambda v_{ki}) - \sum_k u_{jk} y_{jk} - x_{ij} \sum_k (y_{jk} + \lambda v_{ki}) \right\} \quad (5.6)$$

For every value $\lambda \neq 0$, the algorithm (5.5), (5.6) defines a bilinear algorithm of rank $M_{m,n,p}$ that evaluates XY and UV, also consider the case where λ is a real or complex parameter that converges to zero. When λ become very small, that is

$$\lim_{\lambda \to 0} \lambda \sum_{j} (x_{ij} + u_{jk}) v_{ki} = 0$$

the products $-\lambda \sum_{j} (x_{ij} + u_{jk}) v_{ki}$ can be deleted from (5.5), and the bilinear algorithm of products is

$$M_{m,n,p,\lambda} = mnp + mn + np \text{ [19]} \qquad (5.7)$$

that approximately evaluates XY and UV with a precision that can be made arbitrarily high by choosing λ small.

An example [24]:

$$p_1 = (\lambda x_{01} + x_{10})(\lambda y_{01} + y_{10}), \qquad p_2 = (\lambda x_{00} - x_{10})(y_{00} + \lambda y_{01}),$$

$$p_3 = (-\lambda x_{01} + x_{11})(y_{10} + \lambda y_{11}), \qquad p_4 = x_{10}(y_{00} - y_{10}),$$

$$p_5 = (x_{10} + x_{11}) y_{10},$$

$$q_{00} = x_{00} y_{00} + x_{01} y_{10} = \lambda^{-1}(p_1 + p_2 + p_4) - \lambda(x_{00} + x_{01}) y_{01},$$

$$q_{10} = x_{10} y_{00} + x_{11} y_{10} = p_4 + p_5,$$

$$q_{11} = x_{10} y_{01} + x_{11} y_{11} = \lambda^{-1}(p_1 + p_3 - p_5) - \lambda x_{01}(y_{01} - y_{11}).$$

This is a λ-algorithm for partial matrix multiplication. It evaluates three out of four entries of the 2×2 matrix $Q = XY$ with arbitrary precision as λ decreases. Reduce the rank to 5 after ignoring the vanishing products. Consider the problem of the evaluation of the product Q^* of a 2×2 matrix X^* by a 2×3 matrix Y^*. Represent Q^* as the sum $Q^* = Q^0 + Q^1$ where Q^0, Q^1 have the form

$$Q^0 = \begin{bmatrix} q^*_{00} & 0 & 0 \\ q^*_{10} & q^*_{11} & 0 \end{bmatrix}, \qquad Q^1 = \begin{bmatrix} 0 & q^*_{01} & q^*_{02} \\ 0 & 0 & q^*_{12} \end{bmatrix}.$$

Then evaluate Q^0 and the transpose of Q^1 by the above algorithm. This defines a bilinear λ-algorithm of rank 10 for Q^* and consequently defines the exponent ω=2.799. However, one can use a complicated recursive construction that starts directly from the above design of rank 5 for partial matrix multiplication and finally derive the exponent $\omega \le 3\log_6 5 = 2.694$ [25].

By generating a recursive construction starting from the λ-algorithm (5.5), (5.6) that defines the exponent

$$\omega \le 3\log_2(M_{m,n,p,\lambda}/2)/\log_2(nmp)$$

substitute (5.7) into the above inequality bound for $m = p = 7$, $n = 1$ and deduce that

$$\omega \le 3\log_2 31.5/\log_2 49 = 2.669 \text{ [24]}.$$

5.3 Strassen's Improvement (1986 [21] [29])

In 1986, Volker Strassen relaxed the ground rules for computing matrix multiplication by using a basic trilinear form which is not a matrix product [21]. His method for estimating the exponent ω of matrix multiplication has led to the bound $\omega < 2.48$. Coppersmith and Winograd have summarized Strassen's construction of the algorithm in the following [21].

In his construction, Strassen observed that, using the ability to multiply a pair of $N \times N$ matrices, one can "approximately" (in the λ sense) multiply $(3/4)N^2$ pairs of independent scalars, that is, compute

$$\sum_{l=1}^{(3/4)N^2} x_l y_l z_l + O(\lambda)$$

where all the x_l, y_l, z_l are independent. Setting $g = [\tfrac{3}{2}(N+1)]$ one can to obtain

$$\sum_{1 \le i,j,k \le N} (x_{ij}\lambda^{i^2+2ij})(y_{jk}\lambda^{j^2+2j(k-g)})(z_{ki}\lambda^{(k-g)^2+2(k-g)i}) = \sum_{\substack{i+j+k=g \\ 1 \le i,j,k \le N}} x_{ij} y_{jk} z_{ki} + O(\lambda)$$

since the exponent of λ, $i^2 + 2ij + j^2 + 2j(k-g) + (k-g)^2 + 2(k-g)i = (i+j+k-g)^2$ is zero when $i+j+k=g$ and is positive otherwise. Since any two indices i, j uniquely determine the third $k = g-i-j$, each variable x_{ij} is involved in at most one product. There are about $[(3/4)N^2]$ triples (i,j,k), $1 \le i,j,k \le N$, $i+j+k=g$. Call this construction (*) [21].

Strassen uses the following basic trilinear algorithm, which uses $q+1$ multiplications:

$$\sum_{i=1}^{q}(x_0 + \lambda x_i)(y_0 + \lambda y_i)(z_i / \lambda) + (x_0)(y_0)\left(-\sum z_i / \lambda\right)$$

$$= \sum_{i=1}^{q}(x_i y_0 z_i + x_0 y_i z_i) + O(\lambda). \qquad (1)$$

This is viewed as a block inner product: $\sum_{i=1}^{q}(x_i^1 y_0^1 z_i + x_0^2 y_i^2 z_i) + O(\lambda)$. The superscripts denote indices in the block inner product: $X^1Y^1Z + X^2Y^2Z$, or, dually, $W = X^1Y^1 + X^2Y^2$ [21].

This is the block structure of an inner product, or matrix product of size $<1,2,1>$, where the 1×2 block matrix (row vector) X is multiplied by a 2×1 block matrix (column vector) Y to yield a 1×1 block matrix (block scalar) W. Because x_i and x_0 are different variables, they are labeled with different superscripts, that is, put them into different blocks; so are y_i and y_0. However, the z-variables are shared in both blocks. Thus, this algorithm does not in itself represent a matrix product [21].

Examine the fine structure. The first block, $\sum x_i^1 y_0^1 z_i$, represents a matrix product of size $< q,1,1 >$ [29]. A $q \times 1$ matrix (column vector) x is multiplied by a 1×1 matrix (scalar) y_0 to yield a $q \times 1$ matrix (column vector) w, which is dual to z [21]. In the second block, $\sum x_0^2 y_i^2 z_i$ represents a matrix product of size $< 1,1,q >$ [29]. A 1×1 matrix (scalar) x_0 is multiplied by a $1 \times q$ matrix (row vector) y to yield a $1 \times q$ matrix (row vector) w. When adding the two blocks, the indices i of w (or z) cannot be identified as row indices or as column indices [21]. This is the difficulty solved in Strassen's construction.

Take the construction (1) and the two constructions gotten by cyclic permutations of the variables x, y, z, and tensor them together [29], to get an algorithm requiring $(q+1)^3$ multiplications to compute [21]

$$\sum_{i,j,k=1}^{q}(x_{ij0}^{11} y_{0jk}^{11} z_{i0k}^{11} + x_{ijk}^{21} y_{0jk}^{11} z_{i00}^{12} + x_{ij0}^{11} y_{00k}^{12} z_{ijk}^{21} + x_{ijk}^{21} y_{00k}^{12} z_{ij0}^{22} + x_{0j0}^{12} y_{ijk}^{21} z_{i0k}^{11} +$$

$$+ x_{0jk}^{22} y_{ijk}^{21} z_{i00}^{12} + x_{0j0}^{12} y_{i0k}^{22} z_{ijk}^{21} + x_{0jk}^{22} y_{i0k}^{22} z_{ij0}^{22}) + O(\lambda).$$

This is a block 2×2 matrix product (indicated by the superscripts). Within each block is a smaller matrix product; for example the block $I = 1, J = 1, K = 2$ is the matrix product $\sum_{i,j,k=1}^{q}(x_{ij0}^{11} y_{00k}^{12} z_{ijk}^{21})$, which can be interpreted as a matrix product of size $< q^2,1,q >$:

$$\sum_{i,j,k=1}^{q} x_{ij,0}^{11} y_{00,k}^{12} z_{k,ij}^{21} \quad [21].$$

Taking the N^{th} tensor power of this construction, one can get a construction requiring

$(q+1)^{3N}$ multiplications, and producing a block $2^N \times 2^N$ matrix product, each block of which is a matrix product of some size $< m,n,p >$ where $mnp = q^{3N}$ [29]. Applying construction (*) to the block structure, one then obtains $(3/4)(2^N)^2$ independent matrix products, each some size $< m,n,p >$ where $mnp = q^{3N}$ [21]. Applying the $\tau-$theorem, one gets

$$\omega \le 3\tau_N, \qquad (q+1)^3 = \tfrac{3}{4} 2^{2N} (q^{3N})^{\tau_N}.$$

Taking N^{th} roots and letting N grow, the $(\tfrac{3}{4})$ becomes insignificant, then one has $\omega \le 3\tau$, $(q+1)^3 = 2^2 q^{3\tau}$. Letting $q = 5$, Strassen obtains an upper bound for ω [29].

$\omega \le \log_2(6^3 / 2^2) / \log_2 5 = \log_5 54 < 2.4785$.

CHAPTER VI

COMPARISON

6.1 Implementation

The implementation of the three algorithms consists of six classes: a random number generator [30] class provides random numbers to fill elements of the matrix A and the matrix B; an array class [31] creates two arrays objects for Winograd's algorithm; a matrix class implements necessary member functions for matrix objects to perform operations; a straightforward algorithm class; a Winograd's algorithm class; and a Strassen's algorithm class.

The implementations of the algorithms are written in C++ and were compiled on a Sun Microsystems 64-bit Solaris 7 platform. Timing of the speed of algorithms is accomplished by starting a clock before executing the algorithm and stopping the clock after the execution. The difference is then divided by C++ library function <time.h> macro CLOCKS_PER_SEC to get CPU execution time of a particular section of algorithm in seconds. To compare three algorithms, all algorithms use the same operand matrix A and B of the same size.

6.2 Performance Comparisons

To gain better comparisons, all testing matrix sizes are power of 2 to avoid the overhead cost involved with padding which has small constant of proportionality for the number of arithmetic operations. After running on different matrix sizes, the following tables give the results and comparisons.

Matrix size	Standard	Winograd	Strassen (original)
	Number of Operation	Number of Operation	Number of Operation
8 X 8	1,024	1,200	1,576
16 X 16	8,192	8,928	12,184
32 X 32	65,536	68,544	89,896
64 X 64	524,288	536,448	647,704
128 X 128	4,194,304	4,243,200	4,607,656
256 X 256	33,554,432	33,750,528	32,548,504
512 X 512	268,435,456	269,220,864	229,019,176
1024 X 1024	2,147,483,648	2,150,627,328	1,607,852,824

Figure: 6-1 Operation count for 2^d x 2^d matrices

For the purpose of comparison, recursion in Strassen's algorithm is carried out until matrices are partitioned down to reach the 2x2 level, and then the multiplication is switched to the straightforward method.

The number of operations for an algorithm for a certain matrix size includes the number of additions, the number of subtractions and the number of multiplications. A portion of the number of additions for any size matrix multiplication comes from the fact that counts the addition of zero after c_{ij} is first initialized to zero in equation $c_{ij} = c_{ij} + a_{ik} \times b_{kj}$. This addition is unnecessary when using hand calculation, but unavoidable in the algorithm. For example, for 8x8 matrices, by definition, there are $n^3 - n^2 = 8^3 - 8^2 = 448$ additions, in the algorithm. Adding 0 right after c_{ij} is initialized to zero is also one operation, so there are n^2 times more operations of addition of zeros causing the total number of additions to be 512. This is true for both the straightforward algorithm and Strassen's algorithm implementation since both algorithms use the same multiplying method, but not Winograd's algorithm. At the 2x2 level, Strassen's algorithm, in theorem, has a total of 12 additions + 6 subtractions + 7 multiplications = 25 operations; in its implementation, there are 12 additions + 7 additions (resulting form initializing c_{ij} = 0 of 7 multiplications) + 6 subtractions + 7 multiplications = 32 operations. By definition in Winograd's algorithm there are a total number of $N\lfloor n/2 \rfloor + T\lfloor (n+1)/2 \rfloor$ multiplications and $N\lfloor n/2 \rfloor + T\ (n + \lfloor n/2 \rfloor\ +1)$ additions and subtractions, where $N = m + p$ and $T = mp$. For matrices of size 32x32, the total number of multiplications is: $16n^2 + 32n = 16384 + 1024 = 17408$, which is $\frac{1}{2}n^3 + n^2$, compared with $n^3 (= 32768)$ for the straightforward algorithm. The total number of additions and subtractions is: $n^3 + 17n^2 + 30n = 32768 + 17408 + 960 = 51136$. With large size matrices, this number will be close to $\frac{3}{2}n^3$, that is, compared with the straightforward algorithm, Winograd's algorithm will increase the number of additions by about $\frac{1}{2}n^3$.

Since the implementation of Strassen's algorithm uses full recursion, a relatively large portion of the total number of operations results from calculating at small size. For instances, at the 2x2 level, the straightforward method involves 8 multiplications and 4 additions, while Strassen's algorithm uses 7 multiplications and 18 additions and subtractions which is not efficient.

Looking at the total number of operations alone, Strassen's construction is advantageous over the straightforward algorithm and Winograd's algorithm at 256x256 level and greater, and Winograd's method seems to have similar performance as the regular one.

Matrix size	Standard # of Multiplication	Winograd # of Multiplication	Strassen (original) # of Multiplication
8 X 8	512	320	392
16 X 16	4,096	2,304	2,744
32 X 32	32,768	17,408	19,208
64 X 64	262,144	135,168	134,456
128 X 128	2,097,152	1,064,960	941,192
256 X 256	16,777,216	8,454,144	6,588,344
512 X 512	134,217,728	67,371,008	46,118,408
1024 X 1024	1,073,741,824	537,919,488	322,828,856

Figure 6-2: Multiplication count for 2^d x 2^d matrices

Even with the full recursion, as the matrices get large, Strassen's construction seems to outperform the other two methods. At levels of 64x64 and 128x128, the total number of multiplications is half of those of the straightforward method, and also thousands of times less than that of Winograd's. At level 256x256 and greater, the total number of multiplications of Strassen's construction is about one third of that of the regular and algorithm millions less than that of Winograd's.

Looking at the number of multiplications alone, Strassen's construction outperforms the other two except at 32x32 level and less, while Winograd's algorithm has about half of the number of multiplications of the straightforward algorithm. The results suggest that among these sizes of matrices, there should be a point at which the recursion of Strassen's algorithm is stopped and the process is switched to the standard method. In general, it seems that the point is machine dependent. By considering the number of multiplications alone, it seems as if choosing 64x64 as the truncation point where recursion is switched to straightforward method is a good idea.

Matrix size	Standard Time (second)	Winograd Time (second)	Strassen (original) Time (second)
8 X 8	0.0	0.0	0.0
16 X 16	0.01	0.01	0.03
32 X 32	0.04	0.04	0.27
64 X 64	0.32	0.23	1.74
128 X 128	2.54	1.85	12.28
256 X 256	21.21	16.19	86.51
512 X 512	175.19	144.82	625.76
1024 X 1024	1,667.83	1,362.58	4,391.0

Figure 6-3: Time count for $2^d \times 2^d$ matrices

The observed time for each different matrix size is repeated several times. For size below 128x128, the result of each run can differ by a single digit after the decimal. For sizes above 128x128, the results could differ in the ten's digit. Overall, the runtime of each algorithm is quite consistent.

From the figure, it can be seen that runtime for matrix size 64x64 and below among three algorithms is not significant. Starting from size 128x128, the runtime for both the straightforward algorithm and Winograd's algorithm increase by a factor of average 8.5 when n is doubled while runtime for Strassen's algorithm increases by a factor of 7.1. It proves that even with a fully recursive implementation, Strassen's algorithm does run faster than the other two.

At matrix size 128x128 and above, the relative large run time for Strassen's algorithm could be due to some factors. When parameters are passed by value the recursive structure requires a lot of spaces in the heap memory. Each time Strassen's algorithm calls itself, parameters passed by value are costly due to the time spent copying the large structure and this also causes the increase of number of memory access. The large value of run time for large matrix size could also be due to the coding.

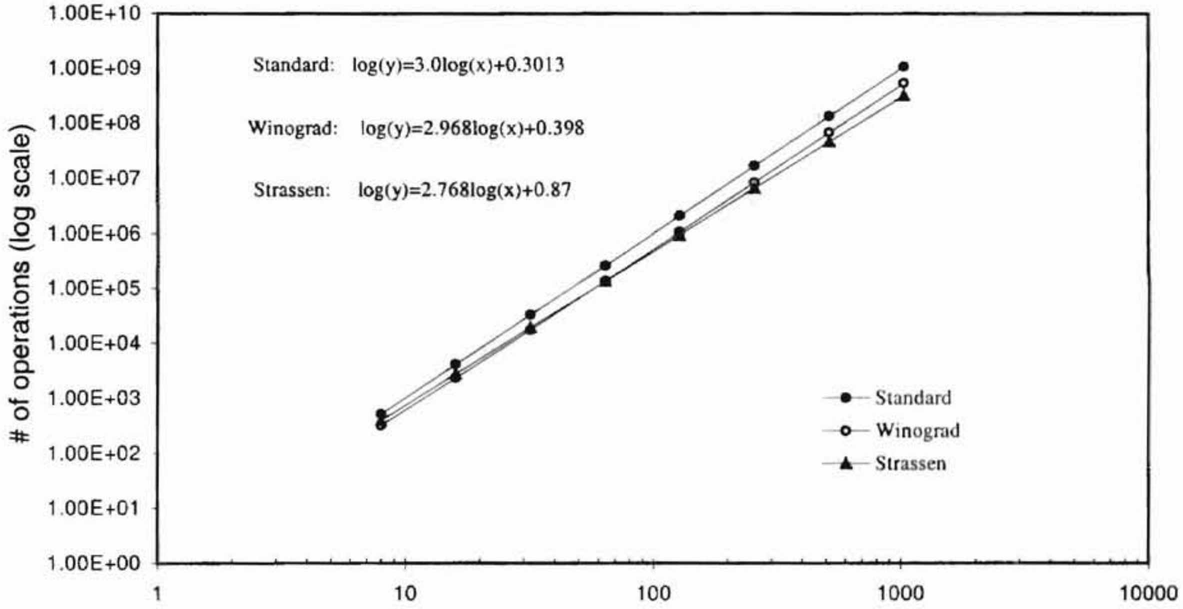


Figure 6-4: # of operations vs. n (log-log scales)

Figure 6-4 shows taking the logarithms of both total multiplications and input size for each algorithm, and plotting them to get straight lines. The slope of a line is the exponent of the number of multiplications for that algorithm. By definition, the straightforward algorithm requires n^3 multiplications, and the slope for its line is 3.

Strassen's algorithm has a theoretical upper bound of 2.80735, and for this empirical line the slope is 2.768. This result is close to the theoretical value.

For the straightforward line, $\log(y) = 3.0\log(x) + 0.3013$. Its slope is 3 and this value is slightly larger than the other two. This implies the line goes upward relatively more quickly than the other two lines do, and the straightforward algorithm will result in more total operations than the other two algorithms do as the input size getting larger. The slope for Winograd's algorithm line $\log(y) = 2.968\log(x) + 0.398$, is very close to that of straightforward line, this implies, as the input size gets larger, both lines will be relatively parallel, and have similar number of total operations.

From Figure 6-4, line, $\log(y) = 2.968\log(x) + 0.398$, for Winograd's algorithm have an interception with line, $\log(y) = 2.768\log(x) + 0.87$, for Strassen's algorithm at approximate input size of 64x64. Below this point, Strassen's algorithm results in more multiplications than Winograd's algorithm does, above the point, Winograd's algorithm results in more multiplications than Strassen's algorithm does.

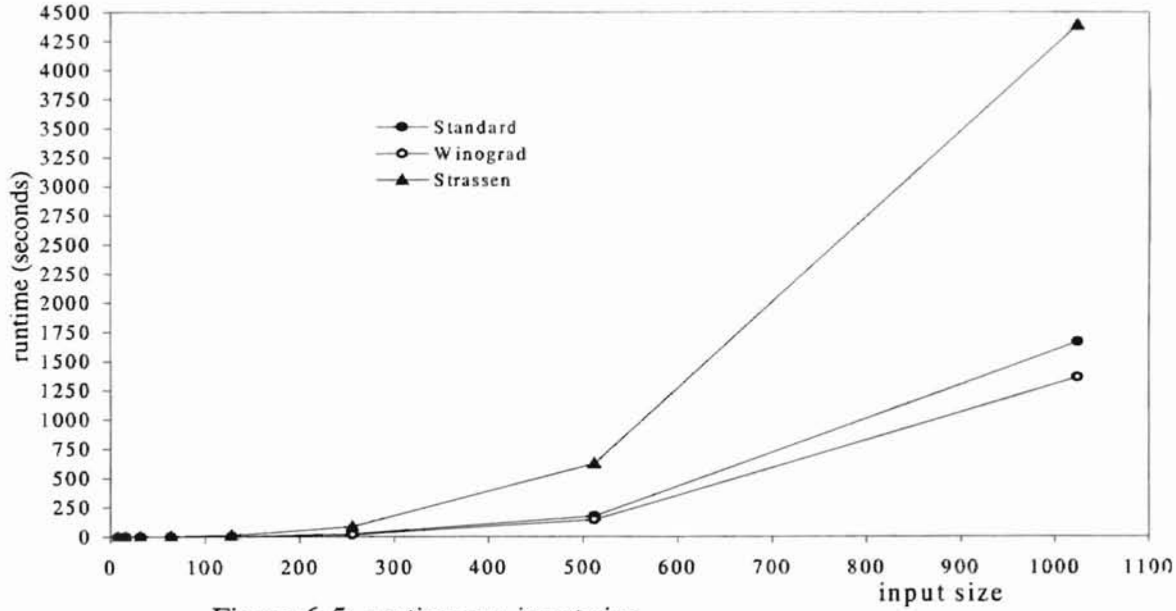


Figure 6-5: runtime vs. input size

Figure 6-5 shows a graph of run time vs. input size in seconds. Because of full recursion, Strassen's algorithm requires more time copying large structures and accessing the memory, and its run time increases rapidly as the input size increases, for fairly small n the differences in run time are negligible.

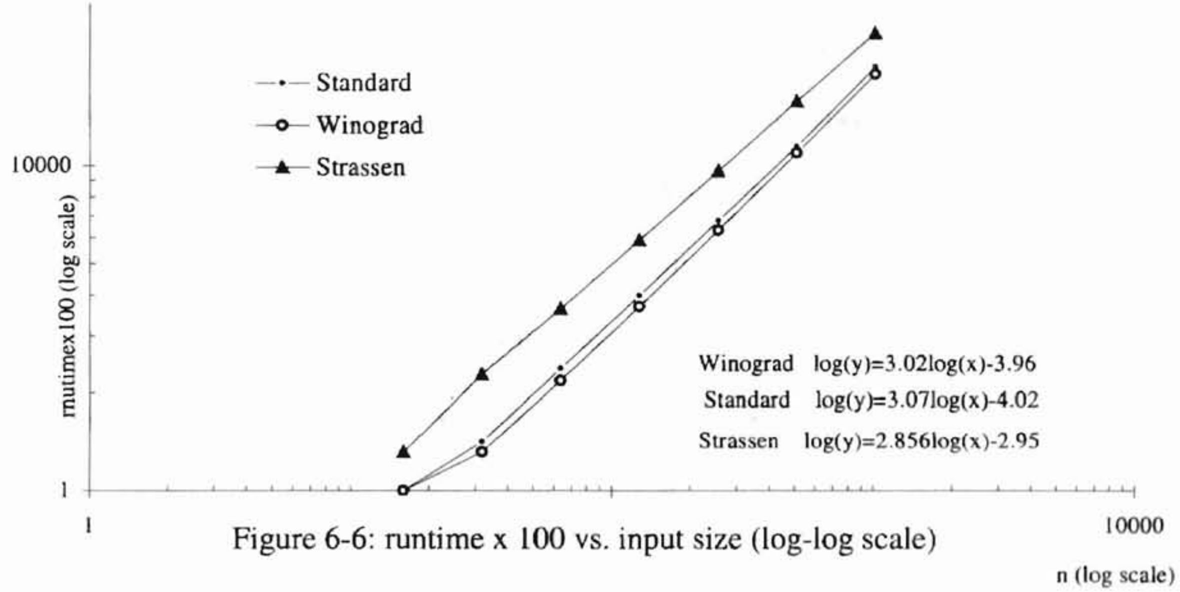


Figure 6-6: runtime x 100 vs. input size (log-log scale)

Figure 6-6 shows the plotting of runtime vs. input size on a log scale. Because runtime at small input size takes less than 1 second, runtime for all input sizes at and above 16x16 are multiplied by 100 before taking logarithms, and runtime for size 8x8 is dropped because it is not measurable.

From the figure, it can be seen that graphs for both the straightforward algorithm and Winograd's algorithm have close value of slopes. Their lines are relatively parallel for the given input sizes, since the difference between the two slope values is fairly small, it can be expected that the two lines could keep the parallel trend for large input size.

At full recursion, Strassen's algorithm has relative large run time at and above size 16x16 for the reasons described before. The value of slope for Stassen's algorithm is relatively smaller than the other two, and this implies that the line for Strassen's algorithm will intercept the other time lines somewhere as input size increases and stay below the other two lines.

6.3 Performance with truncation at 64x64

Values in Figure 6-1 suggest a truncation point at size of 64x64, and Figure 6-7 shows the result of operation count at that truncation level. With the number of operations for the straightforward algorithm and Winograd's algorithm unchanged, Strassen's construction has a big reduction of the total operations at and above the truncation level. On average, below the truncation level, the total number of operations decrease by 31%; at and above the truncation level, the decrease rate is 18.5%.

Matrix size	**Standard** Number of Operation	**Winograd** Number of Operation	**Strassen (original)** Number of Operation
8 X 8	1,024	1,200	1,024
16 X 16	8,192	8,928	8,192
32 X 32	65,536	68,544	65,536
64 X 64	524,288	536,448	524,288
128 X 128	4,194,304	4,243,200	3,743,744
256 X 256	33,554,432	33,750,528	26,501,120
512 X 512	268,435,456	269,220,864	186,687,488
1024 X 1024	2,147,483,648	2,150,627,328	1,311,531,008

Figure: 6-7 Operation count for 2^d x 2^d matrices with truncation at 64x64

Matrix size	Standard # of Multiplication	Winograd # of Multiplication	Strassen (original) # of Multiplication
8 X 8	512	320	512
16 X 16	4,096	2,304	4,096
32 X 32	32,768	17,408	32,768
64 X 64	262,144	135,168	262,144
128 X 128	2,097,152	1,064,960	1,835,008
256 X 256	16,777,216	8,454,144	12,845,056
512 X 512	134,217,728	67,371,008	89,915,392
1024 X 1024	1,073,741,824	537,919,488	629,407,744

Figure 6-8: Multiplication count for 2^d x 2^d matrices with truncation at 64x64

Figure 6-8 shows the comparison results after using truncation at size 64x64. Because the recursion is switched to the standard algorithm when truncation point is reached, the total number of multiplications for Strassen's algorithm has increased by overall average 61%. Above the truncation point, the increase rate is even higher, 94%, this increase results from multiplying matrices at size 64x64. For example, for size 64x64, the standard algorithm results in 262144 multiplications, with full recursion, the same size results in about half the number (134456). Figure 6-9 shows the graphs of results of Figure 6-8 after taking logarithms.

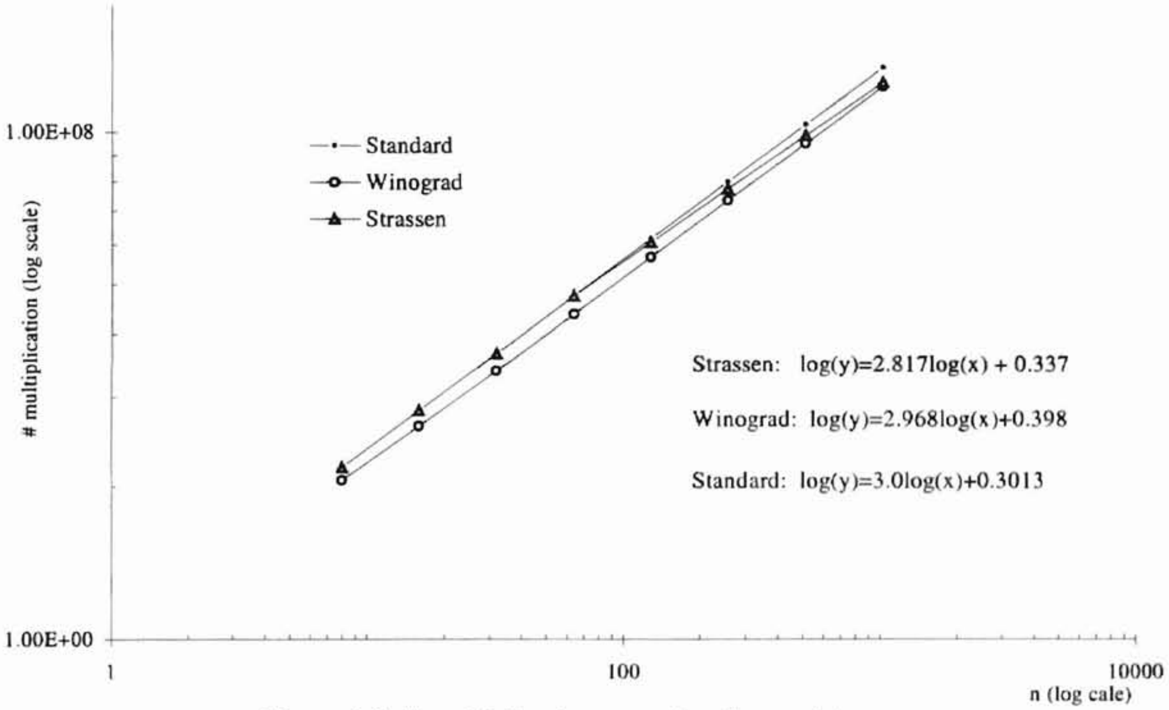


Figure 6-9: # multiplication vs. n (log-log scale)

The line for Strassen's algorithm can be viewed as two parts. The first part is below the size of 64x64, because below this level, the number of multiplications are the same as those of the straightforward algorithm. Above 64x64, the recursion first partitions the large matrices until reaching the size of 64x64, then the straightforward algorithm takes over the calculation. Size 64x64 is the turning point above which

Strassen's algorithm increases the number of multiplications by 94%, so the slope of the Strassen's line is calculated for the part above 64x64 level.

Matrix size	Standard Time (second)	Winograd Time (second)	Strassen (original) Time (second)
8 X 8	0.0	0.0	0.0
16 X 16	0.01	0.01	0.01
32 X 32	0.04	0.04	0.04
64 X 64	0.34	0.25	0.36
128 X 128	2.61	2.02	2.45
256 X 256	20.2	15.07	16.39
512 X 512	173.3	132.41	117.67
1024 X 1024	1684.3	1,389.16	817.39

Figure 6-10: Time count for $2^d \times 2^d$ matrices with truncation at 64x64

Figure 6-10 shows the results of run time after using truncation. From the figure, it can be seen that the run time for Strassen's algorithm has been reduced dramatically. The average speed up is 5.07. At input size 1024x0124, the speed up is 4391.0/817.39=5.37.

Figure 6-11 shows the plotting of logarithms of the results in Figure 6-10. The plotting dropped results at 8x8 level since they were all zeros. The line function for Strassen's construction is constructed between input size 64x64 and 1024x1024 to reflect the trend of using truncation. The other two line functions are constructed between input size 16x16 and 1024x1024.

It can be seen from the graph that both the straightforward algorithm and Winograd's algorithm take a similar amount of time for a given size. Strassen's algorithm has a smaller value of slope for its runtime plotting, and its line will move apart from both straightforward line and Winograd's line as the input size increases which means Strassen's construction uses less time than both the straightforward algorithm and Winograd's algorithm for a given input size as the input size increases as expected from the asymptotic complexities.

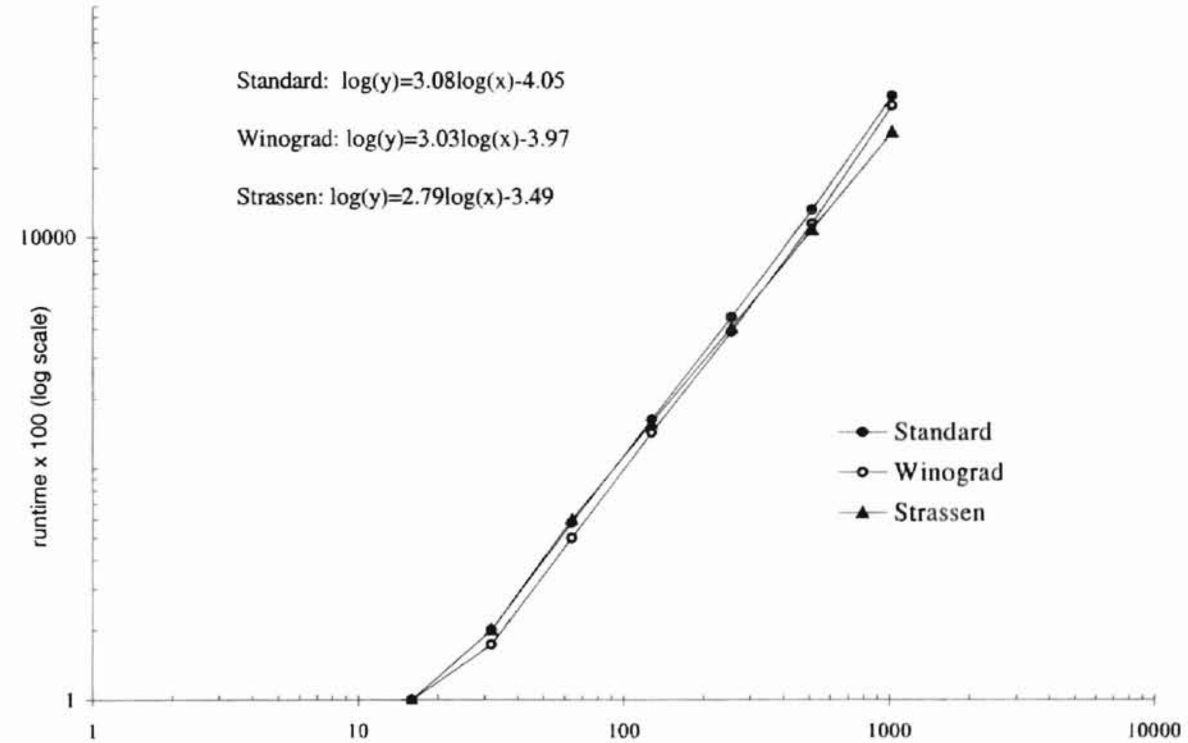


Figure 6-11: runtime x 100 vs. n (log-log scale)

6.4 Strassen's algorithm implementation considerations

Implementation of an algorithm affects time and space efficiency of the algorithm. Strassen's algorithm is supposed to be faster for larger input matrices. Up to this point, this study uses a recursive method of implementation. The parameters in the method are passed by value. Passing parameters by value consumes not only memory space but also execution time. When the input matrix size increases, a proportional execution time is used to copy matrices instead of doing the calculation.

One way to reduce this time is to pass parameters by reference; only the addresses of the data objects are passed. Data are manipulated at their original location in memory. The implementation of Strassen's algorithm using passing by reference shows considerable time saving as the input matrix size getd larger. Table 6-12 shows the results of run time for the two passing methods, their run time difference, and fraction of reduced time relative to the time of passing by value.

Matrix size	Pass by value (second)	Pass by reference (second)	Difference (second)	Fraction (diff / by value)
8 X 8	0.0	0.0	0.0	0
16 X 16	0.01	0.0	0.01	0
32 X 32	0.04	0.04	0.0	0
64 X 64	0.36	0.31	0.05	0.14
128 X 128	2.45	2.19	0.26	0.11
256 X 256	16.39	15.75	0.64	0.04
512 X 512	117.67	111.0	6.67	0.06
1024 X 1024	817.39	769.07	48.32	0.06

Figure 6-12: run time comparison (pass by value vs. pass by reference)

The table shows that the time saved is proportional to the input size. At 1024x1024 level, the time saved is 48 seconds by using passing by reference.

Considering, when passing by value, that multiplying two 256x256 matrices uses only less than 17 seconds, the time saved is quite significant. Memory space saved is also significant, although it cannot be seen, since data manipulations are taking place at their memory location. The speed up because of using passing by reference at level 512x512 is 117.67 / 111.0 = 1.06. At level 1024x1024, the speed up is 817.39 / 769.07 = 1.06.

Using run time resulting from passing by reference to replot the graph of run vs. input size. The run time for the straightforward algorithm and Winograd's algorithm remain the same, so their line functions are the same as in Figure 6-11. The new line for Strassen's algorithm is constructed between input size 64x64 and 1024x1024 to reflect the use of truncation at 64x64. The graph uses the same scale as that of in Figure 6-11 in order to compare the difference.

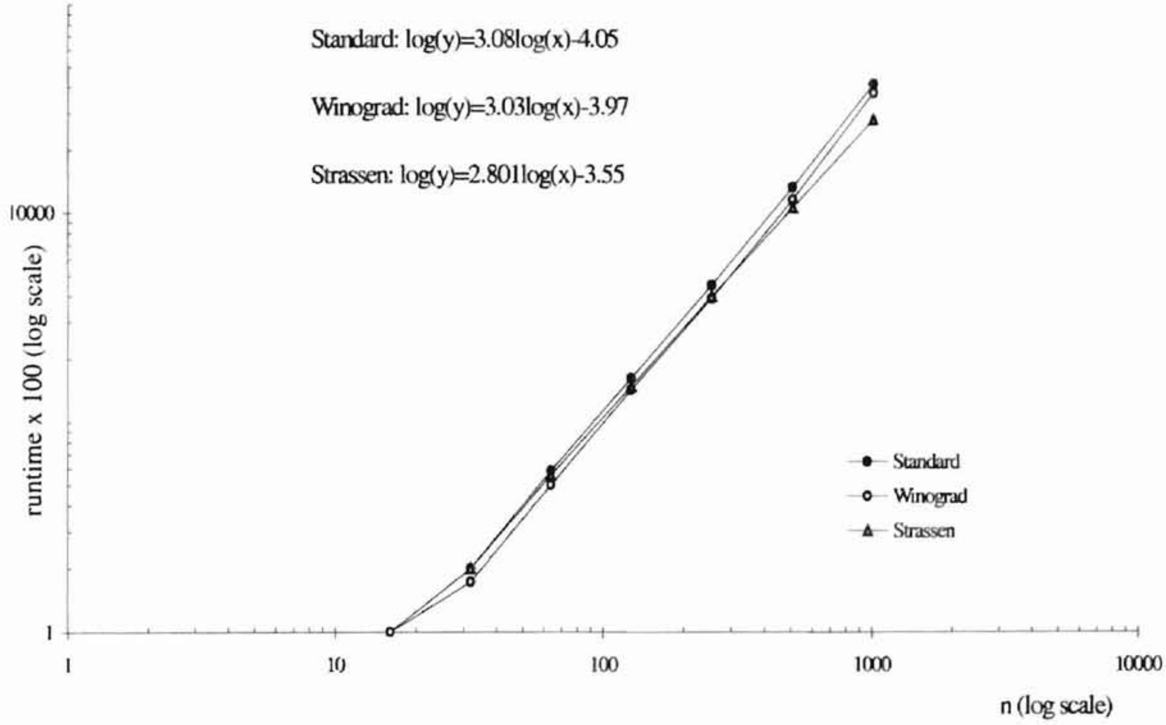


Figure 6-13: runtime x 100 vs. n, (log-log scale) pass by reference

The slope value for the line of Strassen's algorithm is 2.816 when using 4 significant digits during the calculation, 2.801 when using 3 significant digits. Both values are close to the theoretic value 2.807. This result shows that an implementation of an algorithm can make a difference both in time and space efficiency.

6.5 Choosing an optimal truncation for 1024x1024 matrices

Strassen's algorithm results in better performance for large size matrix multiplication. With truncation, Strassen's algorithm can result in even better performance.

Strassen's algorithm responds differently in terms of total operations, total number of multiplications and execution time at different truncation levels. So determining the truncation level is a matter of finding where stopping the matrix partition and calling the straightforward algorithm is faster among all possible truncations. Figure 6-14 lists the empirical testing results for multiplying 1024x1024 matrices, and it is believed that the measures of total operation, total number of multiplications and execution time vary from machine to machine even at the same truncation.

Truncation Size	Total Operations	Total Multiplication	Time (second)
8 X 8	1,153,257,088	421,654,016	920.5
16 X 16	1,138,198,016	481,890,304	791.28
32 X 32	1,198,434,304	550,731,776	779.3
64 X 64	1,311,531,008	629,407,744	826.44
128 X 128	1,466,073,075	719,323,136	903.27
256 X 256	1,657,143,296	822,083,584	1,017.54
512 X 512	1,883,766,784	939,524,096	1,195.09
1024 X 1024	2,147,483,648	1,073,741,824	1,667.83

Figure 6-14: Performance at different truncation size for 2^{10} x 2^{10} matrices

From the Figure, setting truncation at higher level results in more operations, more multiplications and higher execution time. This is because at and below that truncation level, the straightforward algorithm takes over; setting truncation at too low level also results in relative high total operations and execution time. From the Figure, it can be believed that if the truncation is lower than 8x8, both the total number of operations and execution time will also increase.

If the truncation criteria are the combination of total operations, total number of multiplications and execution time, among all possible truncations from 8x8 to 1024x1024, truncation at 16x16 or 32x32 seem to be a better choice.

CHAPTER VII

CONCLUSIONS

The straightforward algorithm is always the best choice for small size matrix multiplication. It provides stable results with precision and efficiency. As to what is the appropriate size of matrix fits the straightforward algorithm, there could be many concerns, but its complexity of n^3 and the nature of the algorithm makes it theoretically less attractive for large n.

Winograd's algorithm cuts down the number of multiplication by a factor of 2, and this is consistent regardless of the input size. The use of the concept of inner product to reduce the number of multiplications rests on the idea that additions can be performed more quickly than multiplications. The total number of operations of Winograd's algorithm is about the same as those of the straightforward algorithm [10].

Strassen's algorithm, for the first time, improved on the asymptotic complexity of the straightforward algorithm, and gave the hope of further improvement and acceleration. Strassen's algorithm has some drawbacks in terms of required input size, less stable results than the straightforward algorithm, that is, for similar errors in the input, Strassen's algorithm will probably create large round off errors in the output, and less efficient as truncation size to stop the recursion becomes small.

Although Strassen's construction results in asymptotic improvement. At large input size, the run time can be greatly reduced by choosing a suitable input size at which the recursion is stopped, and the calculation is then switched to the straightforward process. This empirical study has shown that run time for Strassen's algorithm can be reduced dramatically by choosing truncation point at 64x64 for input size range 8x8 to 1024x1024.

After Strassen's discovery, many new approaches to find asymptotical improvements of matrix multiplication exhibit characteristics of approximating algorithm [25], that is, what is the greatest lowest bound ω on the exponents. Put in a different way, $\omega \leq \log_n M$ where M is the number of multiplication, and n is the input size [19]. The most exciting aspect of Strassen's approach of 1986 is that it eliminates a major barrier to proving $\omega = 2$. Schönhage's $\tau -$theorem enables to deal with not a fixed algorithm but with a family of algorithms. In a search for more starting algorithms of the Strassen variety, one of them might yield the elusive $\omega = 2$ [21].

SELECTED BIBLIOGRAPHY

1. Udi Manber, "Introduction to Algorithms." Addison-Wesley, Reading, MA, 1989, 301-304.

2. Steven S. Skiena, "The Algorithm Design Manual." TELOS, Springer-Verlag New York Inc., New York, 1997, 204-206.

3. Ron Larson and Bruce H. Edwards, "Elementary Linear Algebra." D. C. Heath and Company, Lexington, MA, 1996, 13-85.

4. Bill Jacob, "Linear Functions and Matrix Theory." Springer-Verlag New York, Inc., New York, 1995, 119-120.

5. Charles G. Cullen, "Linear Algebra with Applications." Addison-Wesley, Reading, MA, 1997, 47-58.

6. Shmuel Winograd, "Arithmetic Complexity of Computations." Society for Industrial and Applied Mathematics, Regional Conference Series in Applied Mathematics, 1980, 1-23

7. T. Cormen, C. Leiserson, and R. Rivest, "Introduction to Algorithms." MIT Press, Cambridge MA, 1990, 731-744.

8. J. Cohen and M. Roth, "On the Implementation of Strassen's Fast Multiplication Algorithm." Acta Informatica, 6:341-355, 1976.

9. Alfred V. Aho, Hohn E. Hopcroft, and Jeffrey D. Ullman, "Data Structures and Algorithms." Addison-Wesley, 16-19, 1983.

10. Shmuel Winograd, "A New Algorithm for Inner Product." IEEE Transactions on Computers, C-17: (1968), 693-694.

11. Gene H. Golub and Charles F. Van Loan, "Matrix Computations." The Johns Hopkins University Press, Baltimore, 1996, 3-34.

12. Gene H. Golub and Charles F. Van Loan, "Matrix Computations." The Johns Hopkins University Press, Baltimore, 1985, 1-16.

13. Gilbert Strang, "Introduction to Applied Mathematics." Wellesley-Cambridge Press, Massachusetts, 1986, 25-28.

14. Howard Anton, "Elementary Linear Algebra." John Wiley & Sons, Inc., New York, 1994, 25-31.

15. Howard Eves, "Elementary Matrix Theory." Allyn and Bacon, Inc., Boston, 1966, 37-40.

16. Peter Lancaster, "Theory of Matrices." Academic Press, Inc., New York, 1969,16-19.

17. Ross A. Beaumont and Richard W. Ball, "Introduction to Modern Algebra and Matrix Theory." Rinehart & Company, Inc., New York, 1957, 25-29.

18. Volker Strassen, "Gaussian Elimination is Not Optimal." Numerische Mathematik, 13, (1969), 354-356.

19. Victor Pan, "How Can We Speed up Matrix Multiplication?" Society for Industrial And Applied Mathematics Review, V.26, 393-415.

20. V. Ya. Pan, "Strassen's Algorithm is Not Optimal. Trilinear Technique of Aggregating, Uniting and Canceling for Constructing Fast Algorithm for Matrix Operations." Proceedings 19th Annual Symposium on Foundations of Computer Science, pp. 166-176.

21. Don Coppersmith and Shmuel Winograd, "Matrix Multiplication via Arithmetic Progressions." In Proceedings Nineteenth ACM Symposium, Theory of Computing, Page 1-6, 1987.

22. V. Ya. Pan, "New Fast Algorithms for Matrix Operations." Society for Industrial and Applied Mathematics, Journal of Computing, Vol. 9, No. 2, 1980, pp. 321-342.

23. Dario Bini and Victor Y. Pan, "Polynomial and Matrix Computations." Vol. 1, Birkhäuser, Boston, 1994, 314-317.

24. Victor Pan, "How to Multiply Matrices Faster." Lecture Notes in Computer Science, Edited by G. Goos and J. Hartmanis, Springer-Verlag, New York, 1984, 1-93.

25. A. Schönhage, "Partial and Total Matrix Multiplication." Society for Industrial and Applied Mathematics, Journal of Computing, Vol. 10, No. 3, 1981, pp. 434-454.

26. John A. Eisele and Robert M. Mason, "Applied Matrix and Tensor Analysis." John Wiley & Sons, Inc., New York, 1970, 206-264.

27. Banesh Hoffmann, "About Vectors." Prentice-Hall, Inc., New Jersey, 1966, 111-128.

28. A. P. Wills, "Vector and Tensor Analysis." Prentice-Hall, Inc., New York, 1931, 242-259.

29. Volker Strassen, "The Asymptotic Spectrum of Tensors and The Exponent of Matrix Multiplication." Symposium On Foundations of Computer Science, IEEE, 1986, pp. 49-54.

30. Mark R. Headington and David D. Riley, "Data Abstraction And Structures Using C++." Jones and Bartlett Publishers, Boston, 1997, 75-80.

31. George J. Pothering and Thomas L. Naps, "Introduction to Data Structures and Algorithm Analysis with C++." West Publishing Company, New York, 1995, 45-61.

APPENDIX

```
//===========================================================
// matrix.h head file
// This file creates matrix object of arbitrary size and enables some simple matrix
// operations
//===========================================================

#include<iostream.h>
#include<assert.h>
#include<string.h>

#ifndefMATRIX_H
#define MATRIX_H


 long int addition=0, subtraction=0, multiplication=0;


template<class index, class anytype>
class Matrix
{

        public:
                Matrix(index, index);
                Matrix(const Matrix<index, anytype> &);
                ~Matrix();
                Matrix& operator=(const Matrix<index, anytype>&);
                void assign(index, index, const anytype &);
                anytype retrieve(index, index);
                anytype &operator()(index, index);
                void statistic(long int& ,long int &,long int &);
                int rowsize, colsize;

        private:
                anytype *matrixData;
};
```

```
template<class index, class anytype>
Matrix<index, anytype>::Matrix(index row, index col)
{
        matrixData = new anytype[row * col];
        rowsize = row;
        colsize = col;
}


template<class index, class anytype>
Matrix<index, anytype>::Matrix(const Matrix<index, anytype> &initmatrix)
{
        rowsize = initmatrix.rowsize;
        colsize = initmatrix.colsize;
        matrixData = new anytype[rowsize * colsize];
        assert(matrixData !=0);
        memcpy(matrixData, initmatrix.matrixData, rowsize*colsize*sizeof(anytype));
}


template<class index, class anytype>
Matrix<index, anytype>::~Matrix()
{
        delete [] matrixData;
}


template<class index, class anytype>
Matrix<index, anytype> &Matrix<index, anytype>::
                        operator=(const Matrix<index, anytype> &source)
{
        if(rowsize != source.rowsize || colsize != source.colsize)
        {
                delete [] matrixData;
                rowsize = source.rowsize;
                 colsize = source.colsize;
                 matrixData = new anytype[rowsize * colsize];
                 assert(matrixData !=0);
        }
        memcpy(matrixData, source.matrixData, rowsize*colsize*sizeof(anytype));

        return *this;
}
```

```
template<class index, class anytype>
void Matrix<index, anytype>::assign(index row, index col, const anytype &val)
{
        matrixData[(row-1)*colsize + (col-1)] = val;
}


template<class index, class anytype>
anytype Matrix<index, anytype>::retrieve(index row, index col)
{
        return matrixData[(row-1)*colsize + (col-1)];
}

template<class index, class anytype>
anytype &Matrix<index, anytype>::operator()(index row, index col)
{
        return matrixData[(row-1)*colsize + (col-1)];
}

template<class index, class anytype>
Matrix<index,anytype> operator*(Matrix<index,anytype> &A,
                                Matrix<index,anytype> &B)
{
        assert(A.colsize == B.rowsize);
        Matrix<int, int> C(A.rowsize, B.colsize);

        for(int i=1; i<=A.rowsize; i++)
                for(int j=1; j<=B.colsize; j++)
                {
                        C(i,j) = 0;
                        for(int k=1; k<=A.colsize; k++)
                        {
                                C(i,j) = C(i,j) + A(i,k)*B(k,j);
                                addition++;
                                multiplication++;
                        }
                }
                return C;
}


template<class index, class anytype>
Matrix<index, anytype>& operator+(Matrix<index,anytype> &A,
Matrix<index,anytype>&B)
{
        assert(A.rowsize==B.rowsize ||A.colsize==B.colsize);
        for(int i=1; i<=A.rowsize; i++)
                for(int j=1; j<=A.colsize; j++)
                {
                        A(i,j) = A(i,j) + B(i,j);
                        addition++;
                }
        return A;
}


// enable objects subtraction
template<class index, class anytype>
Matrix<index,anytype>& operator-(Matrix<index,anytype> &A,
                                                        Matrix<index,anytype> &B)
{
        assert(A.rowsize==B.rowsize || A.colsize==B.colsize);
        for(int i=1; i<=A.rowsize; i++)
                for(int j=1; j<=A.colsize; j++)
                {
                        A(i,j) = A(i,j) - B(i,j);
                        subtraction++;
                }
                return A;
}


// return counters
template<class index, class anytype>
void Matrix<index, anytype>::statistic(long &plus, long &minus, long &mult)
{
        plus = addition;
        minus = subtraction;
        mult = multiplication;
}

#endif
```

```
//=================================================================
// This header file exports facilities for pseudorandom number
// generation. Machine dependency: long ints must be at least 32 bits
//=================================================================

#ifndef RAND_H
#define RAND_H
#include <assert.h>

static long currentSeed;              //Updated as a globbal variable
const long A = 16807;                 //Multiplier = 7**5
const long M = 2147483647;            //Modulus = 2**31-1
const long Q = 127773;                //Quotient of M /A
const long R = 2836;                  //Remainder of M /A


class Rnumber
{
        public:
                void SetSeed(long);
                float NextRand();
};

void Rnumber::SetSeed(long initSeed)
{
        assert(initSeed >= 1);
        initSeed = initSeed % M;
        currentSeed = (initSeed > 0) ? initSeed : 1;
        assert(currentSeed >=1 && currentSeed < M);
}

float Rnumber::NextRand()
{
        long temp = A *(currentSeed % Q) - R *(currentSeed/Q);
        assert(currentSeed>=1 && currentSeed<M);
        currentSeed = (temp>0) ? temp : temp + M;
        float result = float (currentSeed)/float(M);
        assert(currentSeed>=1 && currentSeed<M);
        assert(result>0 && result<1); //result should be floating num

        return result;
}

#endif
```

```
//============================================================
// header file: array.h
// This file enables array object to check index range, compare two objects; assign one
// object to another; retrieve element of the object, and some of  necessary object
// operations
//============================================================

#include<iostream.h>
#include<assert.h>

typedef enum boolean{FALSE, TRUE} BOOLEAN;

#ifndef ARRAY_H
#define ARRAY_H

// class declaration

template<class index, class anytype>
class Array
{
        public:
                Array(index);
                Array(const Array<index,anytype> &);
                ~Array();
                Array &operator=(const Array<index,anytype> &);
                void assign(index, const anytype &);
                anytype retrieve(index);                        //return element value
                anytype &operator[](index);

        private:
                anytype *arrayData;                             //points to array data
                int baseIndex, hiIndex;
                BOOLEAN outRange(index);                        //internal range checking
};

// constructor, allocate space for new object

template<class index, class anytype>
Array<index, anytype>::Array(index size )
{
        arrayData = new anytype[size];                          //allocate space
        assert(arrayData != 0);                                 //check the allocation
        baseIndex = 1;
        hiIndex = size;
}
```

```
// copy constructor, enables argument passing, return value
template<class index, class anytype>
Array<index, anytype>::Array(const Array<index, anytype> &initarray)
{
        baseIndex = initarray.baseIndex;                    //assign lower bound
        hiIndex = initarray.hiIndex;                        //assign upper bound
        arrayData = new anytype[hiIndex];                   //allocate space
        assert(arrayData !=0);                              //check the allocation

        for(index i= baseIndex; i<=hiIndex; i++)
          arrayData[i] = initarray.arrayData[i];             //copy element by element
}

template<class index, class anytype>
Array<index, anytype>::~Array()
{
        delete [] arrayData;
}

// overload operator=, enables assignment between objects
template<class index, class anytype>
Array<index,anytype> &Array<index,anytype>::
                        operator=(const Array<index,anytype> &source)
{
        anytype *ptr1, *ptr2;

        if((loIndex!=source.baseIndex) || (hiIndex!=source.hiIndex))
        {                                                   //prevents self assignment
                delete [] arrayData;
                baseIndex = source.baseIndex;
                hiIndex = source.hiIndex;
                arrayData = new anytype[hiIndex];           //space allocation
                assert(arrayData != 0);
        }
        ptr1 = arrayData;
        ptr2 = source.arrayData;

        for(int i=baseIndex; i<=hiIndex;  i++)              //assign element value
        {
                *ptr1 = *ptr2;
                 ptr1++;
                 ptr2++;
        }

        return *this;                                       //enable concatenation
}
```

```
// overload operator[], enables  array object using []
template<class index, class anytype>
anytype &Array<index, anytype>::operator[](index i)
{
        assert(!outRange(i));                          //index out range! abort
        return(arrayData[i]);
}


// checking array object index range
template<class index, class anytype>
BOOLEAN Array<index, anytype>::outRange(index i)
{
        if((i<baseIndex) || (i>hiIndex))               //i beyond range
        {
                cerr<<"Index "<<i<<" out of range"<<endl;
                return(TRUE);
        }
        else
                return (FALSE);
}


// enables value assignment to object element
template<class index, class anytype>
void Array<index, anytype>::assign(index i, const anytype &val)
{
        assert(!outRange(i));                          //if out of range abort
        arrayData[i] = val;
}


// enables retrieving element value from array object
template<class index, class anytype>
anytype Array<index, anytype>::retrieve(index i)
{
        assert(!outRange(i));
        return(arrayData[i]);
}

#endif
```

```
//=========================================================
//  naive.h head file
// This is the head file for straightforward algorithm of matrix multiplication, it simply
// performs the desired operation and return data like time, number of operations.
//=========================================================

#include<iostream.h>
#include<time.h>
#include<iomanip.h>
#include "matrix.h"
#ifndef  NAIVE_H
#define  NAIVE_H


template<class index,class anytypc>
class Naive
{
        public:
                Naive();
                void Mult(Matrix<index, anytype>&,Matrix<index, anytype>&);

        private:
                int N;
};

//initialization
template<class index,class anytype>
Naive<index,anytype>::Naive()
{
        N=0;
}


template<class index,class anytype>
void Naive<index, anytype>::Mult(Matrix<index, anytype>&A,
                                Matrix<index, anytype>&B)
{
        long int Add=0, Mul=0, Sub=0;
        clock_t start, end;

        Matrix<int, int> C(A.colsize, B.rowsize);
        start = clock();
        C = A * B;
        end = clock();
        C.statistic(Add, Sub, Mul);
        cout<<"Straightforward algorithm\n"<<endl;
        cout<<"\tMatrix A\t\t\t"<<A.rowsize<<" x "<<A.colsize<<endl;
        cout<<"\tMatrix B\t\t\t"<<B.rowsize<<" x "<<B.colsize<<endl;
        cout<<"\tMatrix C\t\t\t"<<C.rowsize<<" x "<<C.colsize<<endl;
        cout<<"\tNumber of additions\t\t"<<Add<<endl;
        cout<<"\tNumber of subtractions\t\t"<<Sub<<endl;
        cout<<"\tNumber of multiplications\t"<<Mul<<endl;
        cout<<"\tTotal time\t\t\t"<<float(end-start)/CLOCKS_PER_SEC
            <<" second"<<endl;
        cout<<endl;
        addition = 0;
        subtraction = 0;
        multiplication = 0;

}

#endif
```

```
//=================================================================
// Winogrd.h file
// This is the head file for Winograd's algorithm for matrix multiplication, it provides
//  implementation of the algorithm.
//=================================================================

#include<iostream.h>
#include<iomanip.h>
#include<time.h>
#include "array.h"
#include "matrix.h"

#ifndef WINOGRAD_H
#define WINOGRAD_H

template<class index, class anytype>
class Winograd
{
        public:
                Winograd();
                void Win(Matrix<index, anytype>&, Matrix<index, anytype>&);

        private:
                int wino;
};

template<class index, class anytype>
Winograd<index, anytype>::Winograd()
{
        int wino=0;
}


template<class index, class anytype>            // implementing the algorithm
void Winograd<index, anytype>::
                Win(Matrix<index, anytype> &A, Matrix<index, anytype> &B)
{
        long int Add_count=0;                               //initializing counters
        long int Mul_count=0;
        long int Sub_count=0;
        clock_t start, end;

        Matrix<int, int> C(A.rowsize, B.colsize);           //define product
        Array<int, int> X(A.rowsize), Y(B.colsize);         //define two array

        start = clock();
```

```
for(int i=1; i<=A.rowsize; i++)          //calculating X[i] from each row
{
        X[i] = 0;
        int k;
        for(k = 1; k<=(A.colsize)/2; k++)
                X[i] = X[i] + A(i, 2*k-1)*A(i,2*k);

        Add_count = Add_count + (k-2);      //recording the statistic
        Mul_count = Mul_count + (k-1);
}

for(int j=1; j<=B.colsize; j++)                 // calculating Y[j] from each col
{
        Y[j] = 0;
        int k;
        for(k=1; k<=(B.rowsize)/2; k++)
                Y[j] = Y[j] + B(2*k-1, j)*B(2*k,j);

        Add_count = Add_count + (k-2);      //recording the statistic
        Mul_count = Mul_count + (k-1);
}

for(int i=1; i<=A.rowsize; i++)                 //calculating element of C[i,j]
{
         for(int j=1; j<=B.colsize; j++)
        {
                C(i,j) = 0;
                int k;
                for(k=1; k<=(A.colsize)/2; k++)
                   C(i,j) = C(i,j)+(A(i,2*k-1)+B(2*k,j))*(A(i,2*k)+B(2*k-1,j));

                C(i,j) = C(i,j) - X[i] - Y[j];                  //even size

                Add_count = Add_count + (3*(k-1)-1); // accumulating counters
                Sub_count = Sub_count + 2;
                Mul_count = Mul_count + (k-1);

                if(A.colsize%2 != 0)                          //odd size
                {
                        C(i,j) = C(i,j) + A(i,A.colsize)*B(B.rowsize,j);
                        Add_count = Add_count +1;
                         Mul_count = Mul_count + 1;
                }
        }
}
```

```
        end = clock();
        cout<<"Winograd's algorithm\n"<<endl;
        cout<<"\tMatrix A\t\t\t"<<A.rowsize<<" x "<<A.colsize<<endl;
        cout<<"\tMatrix B\t\t\t"<<B.rowsize<<" x "<<B.colsize<<endl;
        cout<<"\tMatrix C\t\t\t"<<C.rowsize<<" x "<<C.colsize<<endl;
        cout<<"\tNumber of additions\t\t"<<Add_count<<endl;
        cout<<"\tNumber of subtractions\t\t"<<Sub_count<<endl;
        cout<<"\tNumber of multiplications\t"<<Mul_count<<endl;
        cout<<"\tTotal time\t\t\t"<<float(end-start)/CLOCKS_PER_SEC
            <<" second"<<endl;

}

#endif
```

```
//==================================================================
// strassn.h head file
// This head file implements Strassen's algorithm, it also includes some functions
// necessary to perform the algorithm.
//==================================================================

#include<iostream.h>
#include<iomanip.h>
#include<time.h>
#include"matrix.h"

#ifndef STRASSN_H
#define STRASSN_H

int truncation;

template<class index, class anytype>
class Strassen
{
        public:
                Strassen();
                void stras(Matrix<index, anytype>&,Matrix<index,anytype>&);
                Matrix<index, anytype> strass(Matrix<index,anytype>&,
                                Matrix<index, anytype>&,int);

        private:
                int stra;
};

template<class index, class anytype>
Strassen<index,anytype>::Strassen()
{
        stra = 0;
}

template<class index, class anytype>    // pass matrices A and B to strassen's algorithm
void Strassen<index, anytype>::stras(Matrix<index, anytype> &A,
                                        Matrix<index,anytype> &B)
{
        long int N_Add = 0, N_Sub = 0, N_Mul = 0;
        clock_t start, end;

        Matrix<int, int> C(A.colsize, B.rowsize);
        start = clock();
        C = strass(A,B, A.colsize);
        end = clock();
        cout<<"Original"<<endl;
        C.statistic(N_Add, N_Sub, N_Mul);

        cout<<"Strassen’s algorithm\n"<<endl;
        cout<<"\tMatrix A\t\t\t"<<A.rowsize<<" x "<<A.colsize<<endl;
        cout<<"\tMatrix B\t\t\t"<<B.rowsize<<" x "<<B.colsize<<endl;
        cout<<"\tMatrix C\t\t\t"<<C.rowsize<<" x "<<C.colsize<<endl;
        cout<<"\tNumber of additions\t\t"<<N_Add<<endl;
        cout<<"\tNumber of subtractions\t\t"<<N_Sub<<endl;
        cout<<"\tNumber of multiplications\t"<<N_Mul<<endl;
        cout<<"\tTotal time\t\t\t"<<float(end-start)/CLOCKS_PER_SEC
            <<" seconds\n"<<endl;
        addition = 0;
        subtraction = 0;
        multiplication = 0;
}


template<class index, class anytype>          // strassen’s algorithm
Matrix<index,anytype> Strassen<index,anytype>::strass(Matrix<index,anytype>&A,
Matrix<index,anytype>&B,int n)
{
        if(n<=truncation)
        {
                Matrix<int,int> C(1,1);
                 return C = A*B;
        }
        else
        {
                Matrix<int, int> A11(n/2, n/2);          //defining block matrix
                Matrix<int, int> A12(n/2, n/2);
                Matrix<int, int> A21(n/2, n/2);
                Matrix<int, int> A22(n/2, n/2);
                Matrix<int, int> B11(n/2, n/2);
                Matrix<int, int> B12(n/2, n/2);
                Matrix<int, int> B21(n/2, n/2);
                Matrix<int, int> B22(n/2, n/2);
```

```
for(int i=1; i<=n/2; i++)            //filling elements of 4 block matrices
      for(int j=1; j<=n/2; j++)
      {
              A11(i,j) = A(i,j);
              B11(i,j) = B(i,j);
              A12(i,j) = A(i,j + n/2);
              B12(i,j) = B(i,j + n/2);
              A21(i,j) = A(i + n/2,j);
              B21(i,j) = B(i + n/2,j);
              A22(i,j) = A(i + n/2,j + n/2);
              B22(i,j) = B(i + n/2,j + n/2);
      }


Matrix<int, int> P1(n/2, n/2);
Matrix<int, int> P2(n/2, n/2);
Matrix<int, int> P3(n/2, n/2);
Matrix<int, int> P4(n/2, n/2);
Matrix<int, int> P5(n/2, n/2);
Matrix<int, int> P6(n/2, n/2);
Matrix<int, int> P7(n/2, n/2);

P1 = strass(A11+A22,B11+B22,n/2);
P2 = strass(A21+A22,B11,n/2);
P3 = strass(A11,B12-B22,n/2);
P4 = strass(A22,B21-B11,n/2);
P5 = strass(A11+A12,B22,n/2);
P6 = strass(A21-A11,B11+B12,n/2);
P7 = strass(A12-A22,B21+B22,n/2);

Matrix<int, int> C11(n/2, n/2);
Matrix<int, int> C12(n/2, n/2);
Matrix<int, int> C21(n/2, n/2);
Matrix<int, int> C22(n/2, n/2);

C11 = P1 + P4 - P5 + P7;
C12 = P3 + P5;
C21 = P2 + P4;
C22 = P1 + P3 - P2 + P6;

Matrix<int, int> C(n,n);             //C is composes of 4 blocks
```

```
          for(int i=1; i<=n/2; i++)      //join blocks together
                for(int j=1; j<=n/2; j++)
                {
                        C(i,j) = C11(i,j);
                        C(i,j+n/2) = C12(i,j);
                        C(i+n/2,j) = C21(i,j);
                        C(i+n/2,j+n/2) = C22(i,j);
                }

                return C;
        }
}

#endif
```

```
//==============================================================
// This is the drive
// This main defines objects of the straightforward, Winograd, and Strassen classes, and
// then executes the three algorithms by the objects on different size of matrices.
//==============================================================

#include "rand.h"
#include "naive.h
#include "matrix.h"
#include "strass.h"
#include "winograd.h"

main()
{
        int size;
        long int x;
        Rnumber R;                                  //defining objects
        Naive<int, int> N;
        Strassen<int, int> S;
        Winograd<int, int> W;

        size =1024;
        Matrix<int, int> A(size,size), B(size, size);

        R.SetSeed((long)time(NULL));

        for(int i=1; i<=size; i++)
                for(int j=1; j<=size; j++)
                {
                        x = int(R.NextRand()*10+1);
                        A(i,j) = x;
                }

        B = A;

        cout<<"\tPlease indicate truncation"<<endl;
        cin>>truncation;


        S.stras(A, B);                      //using Strassen's algorithm
        N.Mult(A, B);                       //using straightforward algorithm
        W.Win(A, B);                        //using Winograd's algorithm


        return 0;
}
```

2

VITA

XING ZHANG

Candidate for the Degree of

Master of Science

Thesis: CHOOSING A BETTER ALGORITHM FOR MATRIX MULTIPLICATION

Major Field: Computer Science

Biographical:

Personal Data: Born in HaiNai Provicnce of the P. R. China, the son of ZaiKei Zhang and PeiLei Qian.

Education: Received Bachelor of Science degree from South China Teacher's University, Guangzhou and Master of Project Management degree form Western Carolina University, Cullowhee, NC in July, 1985 and August, 1991, respectively. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in July, 2000.

Experience: Employed by General Bureau of State Farms of Guangdong province as a project analyst; a graduate research assistant by Western Carolina University; a computer technician by Fortress Systems International, Inc.; a technical support technician by IBM Corporation; a graduate teaching assistant by Computer Science Department of Oklahoma State University respectively.