Final Project Report
MA-UY 4424 - Spring 2023

Strassen's Method and Fast Matrix Multiplication

Kevin Zhou
Junwen Zhong

Supervised by
Prof. Tyler Chen

Contents

1 Introduction

Strassen's method is an algorithm for large matrix multiplication. It is faster than the normal matrix multiplication algorithm.

In the algorithm, we use the divide and conquer technique. Suppose that we have two matrices $\mathbf{A}$ and $\mathbf{B}$, and each of them has size $(2^n, 2^n)$. In every step, we divide $\mathbf{A}$ into 4 sub-matrices $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, $\mathbf{d}$; we also divide $\mathbf{B}$ into 4 sub-matrices $\mathbf{e}$, $\mathbf{f}$, $\mathbf{g}$, $\mathbf{h}$:

$$\mathbf{A} = \begin{bmatrix} \mathbf{a} & \mathbf{b} \\ \mathbf{c} & \mathbf{d} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{e} & \mathbf{f} \\ \mathbf{g} & \mathbf{h} \end{bmatrix}$$

And using the submatrices, we can get:

$$\mathbf{p_1} = \mathbf{a}(\mathbf{f} - \mathbf{h}), \quad \mathbf{p_2} = (\mathbf{a} + \mathbf{b})\mathbf{h}, \quad \mathbf{p_3} = (\mathbf{c} + \mathbf{d})\mathbf{e}, \quad \mathbf{p_4} = \mathbf{d}(\mathbf{g} - \mathbf{e}),$$
$$\mathbf{p_5} = (\mathbf{a} + \mathbf{d})(\mathbf{e} + \mathbf{h}), \quad \mathbf{p_6} = (\mathbf{b} - \mathbf{d})(\mathbf{g} + \mathbf{h}), \quad \mathbf{p_7} = (\mathbf{a} - \mathbf{c})(\mathbf{e} + \mathbf{f})$$

Then suppose that $\mathbf{C} = \mathbf{AB}$. We can do this matrix multiplication by:

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} \mathbf{p_5} + \mathbf{p_4} - \mathbf{p_2} + \mathbf{p_6} & \mathbf{p_1} + \mathbf{p_2} \\ \mathbf{p_3} + \mathbf{p_4} & \mathbf{p_1}\mathbf{p_5} - \mathbf{p_3}\mathbf{p_7} \end{bmatrix}$$

We do the above operations in each recursive step. In each step, the side length of the matrices is halved. In the base step (or the last recursive step), the size of matrices is $(1, 1)$.

The main idea of this project is to research Strassen's method and implement the algorithm. First, we did a run-time analysis on both Strassen's method and the normal method. Then, we implemented the algorithms with a Python program. We timed how long a matrix multiplication takes using both Strassen's method and the normal method. We tried matrices of different sizes, from $(2, 2)$ to $(2^{12}, 2^{12}) = (4096, 4096)$. We plotted the results in the same plot and compared them. We compared the results of the run-time analysis and the Python program to verify that Starssen's method can be faster than the normal method of matrix multiplication for large matrices.

Additionally, we tried to combine Strassen's method and the normal method for matrix multiplication. We set the size of matrices at the base step of Strassen's method to $(2, 2)$, $(4, 4)$, or $(8, 8)$. With this improvement, we accomplished a revised Strassen's method which is much faster than the basic Strassen's method.

2 Implementation

2.1 Algorithm run-time analysis

2.1.1 The normal method

The normal method, or the naive algorithm, for matrix multiplication simply loops through all the entries of the output and computes each one. Because the output matrix has the size $n \times n$, to loop through all entries of the matrix alone would take $O(n^2)$. For each entry, the calculation requires traversing a single row of the input matrix $\mathbf{A}$ and a single column of the input matrix $\mathbf{B}$ simultaneously. For each entry in the row/column, we perform 1 multiplication, and eventually we add all the products together. Therefore, to compute one entry within the output matrix takes n multiplications, and $(n-1)$ additions.

As a result, the complete algorithm requires n^3 multiplications and $(n-1)n^2$ additions, which indicates the runtime of the normal method for matrix multiplication is $O(n^3)$.

2.1.2 Strassen's method

The Strassen's method first divide the input matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n \times n}$ into 8 submatrices of size $\frac{n}{2} \times \frac{n}{2}$. Then, it uses 7 recursive calls along with some basic arithmatics on the submatrices to compute the final output. Because adding and subtracting two matrices with size $\frac{n}{2} \times \frac{n}{2}$ takes $O(n^2)$, the total runtime for the non-recursive part of the algorithm is cn^2 for some constant c.

Therefore, if we define $T(n)$ to be the number of operations required for computing the product of two $n \times n$ matrices using Strassen's algorithm, we can write the runtime in the following form:

$$T(n) \leq 7 \cdot T(\frac{n}{2}) + cn^2 \qquad T(1) = 1$$

(Since we are doing a total of 18 additions and subtractions on matices of size $\frac{n}{2} \times \frac{n}{2}$ within one recursive call of the algorithm, we can take $c = \frac{9}{2}$).

Using the Master Method, we can show the runtime of Strassen's method is $O(n^{\log_2 7}) \approx O(n^{2.8})$

2.2 Python program

First, we import the packages we need. We use *numpy* to generate the matrices we need and do the matrix operations like division. We use *matplotlib.pyplot* to plot the results of the program. And we use *time* to get the algorithm runtime.

```
import numpy as np
import matplotlib.pyplot as plt
import time
```

Then we define a function for the normal method for square matrix multiplication. It should be the same as numpy matrix multiplication function $A@B$ or $np.matmul(A, B)$.

```
#Normal method for square matrix
def multiply(A, B):
  n = A.shape[0]
  result = np.zeros((n, n))
  for i in range(n):
    for j in range(n):
      for k in range(n):
        result[i][j] += A[i][k]*B[k][j]
  return result
```

For the Strassen's method, we first define a function for matrix split. This function splits a matrix into 4 sub-matrices.

```
#Split matrix
def split(matrix):
    row, col = matrix.shape
    row2, col2 = row//2, col//2
    return matrix[:row2, :col2], matrix[:row2, col2:], matrix[row2:, :col2],
   matrix[row2:, col2:]
```

After this, we define the function for Strassen's method. There is a recursion in the function. The base case, as mentioned in the introduction section, is for two $(1, 1)$ matrices, and we do simple multiplication for them. For matrices of larger sizes, we first split them into sub-matrices

using the $split(matrix)$ function. Then, we compute the 7 products (p_1 to p_7) recursively. Then we put the results into the result matrix **c** using $np.vstack()$ function.

```
# Strassen's method
def strassen(x, y):
    # Base case when size of matrices is 1*1
    if len(x) == 1:
        return x*y
    # Splitting the matrices into quadrants. This will be done recursively
    # until the base case is reached.
    a, b, c, d = split(x)
    e, f, g, h = split(y)
    # Computing the 7 products, recursively (p1, p2...p7)
    p1 = strassen(a, f - h)
    p2 = strassen(a + b, h)
    p3 = strassen(c + d, e)
    p4 = strassen(d, g - e)
    p5 = strassen(a + d, e + h)
    p6 = strassen(b - d, g + h)
    p7 = strassen(a - c, e + f)
    # Computing the values of the 4 quadrants of the final matrix c
    c11 = p5 + p4 - p2 + p6
    c12 = p1 + p2
    c21 = p3 + p4
    c22 = p1 + p5 - p3 - p7
    # Combining the 4 quadrants into a single matrix by stacking horizontally
    and vertically.
    c = np.vstack((np.hstack((c11, c12)), np.hstack((c21, c22))))
    return c
```

In the main function, we run the normal method and Strassen's method for matrices of sizes from $(2, 2)$ to $(2^{12}, 2^{12}) = (4096, 4096)$. We store the time value of two methods in two lists, *strassens* and *normals*. For each size, we use $np.random.randint(low = 1, high = 10, size = (n, n))$ function to generate three random matrices (A, B, C) with integers in range $[1, 9]$. Then, we run the normal method and Strassen's method for $A \times B$, $A \times C$, $B \times C$ and get the avearage run-time. We do so to reduce the deviation caused by *np.random*. When the program finish the multiplications, we calculate the proportion value by strassen time/normal time. If this value is smaller than 1, it means that Strassen's method is faster than the normal method. Then before the program goes to the next matrix size, it prints out the total time cost and the proportion

value.

```
max_power = 12
length = 2**max_power
theA = np.random.randint(low=1, high=10, size=(length,length))
theB = np.random.randint(low=1, high=10, size=(length,length))
theC = np.random.randint(low=1, high=10, size=(length,length))
#let computer warm up first
C = multiply(theA, theB)
#initializations
powers = np.arange(1,max_power+1)
strassens = np.ones(len(powers))
normals = np.ones(len(powers))
prop = np.zeros(len(powers))
for i in range(len(powers)):
    power = powers[i]
    n = 2**power
    A = theA[:n,:n]
    B = theB[:n,:n]
    C = theC[:n,:n]
    #strassen
    start = time.perf_counter()
    D = strassen(A, B)
    D = strassen(A, C)
    D = strassen(B, C)
    end = time.perf_counter()
    strassen_time = (end-start)/3
    strassens[i] = strassen_time
    #normal
    start = time.perf_counter()
    D = multiply(A, B)
    D = multiply(A, C)
    D = multiply(B, C)
    end = time.perf_counter()
    normal_time = (end-start)/3
    normals[i] = normal_time
    #result
    prop[i] = strassen_time/normal_time
    print("2^", power, "done, strassen time: ", strassen_time, ", normal time: "
    , normal_time, "Proportion: ", prop[i])
```

When all the calculations are done, we print out the time values in the lists. Then we make a plot using *plt.plot* function.

```
print(strassens)
print(normals)
print(prop)
plt.xlabel('size = 2^')
plt.ylabel('proportion')
plt.grid()
plt.plot(powers, prop, marker='o')
for i in range(len(powers)):
    plt.text(powers[i], prop[i], round(prop[i],2), ha='center', va='bottom')
```

2.3 Graphs and plots

Matrix size	Total time cost (seconds)	Proportion (Strassen's/normal)
(2,2)	0.00007	4.63
(4,4)	0.00033	5.14
(8,8)	0.00190	5.37
(16,16)	0.01233	4.33
(32,32)	0.07333	3.06
(64,64)	0.52531	2.70
(128,128)	3.98297	2.14
(256,256)	27.07797	2.13
(512,512)	198.90984	1.84
(1024,1024)	1466.90398	1.60
(2048,2048)	10833.39099	1.40
(4096,4096)	80529.96561	1.22

Table 1: The total time cost and the proportion value

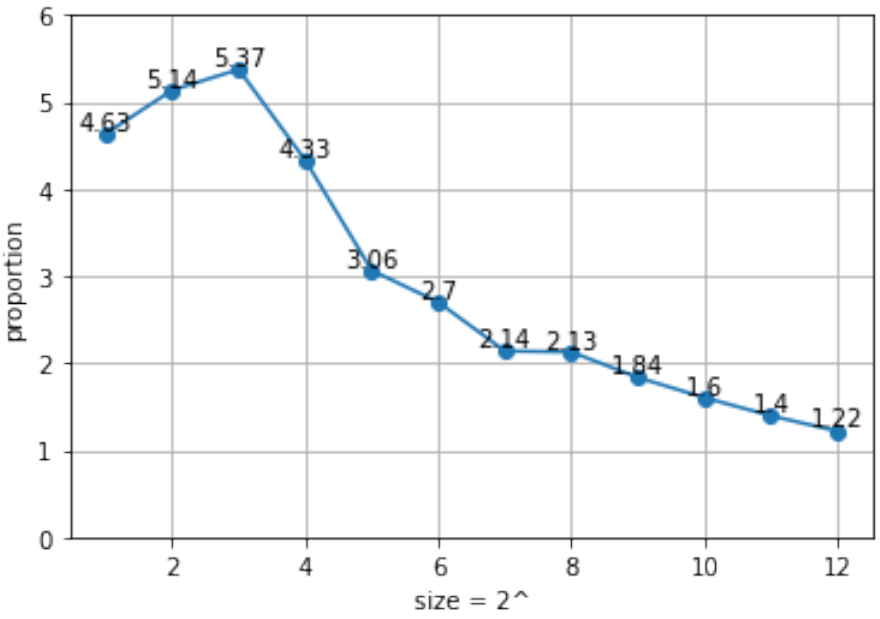

Figure 1: Proportion (Strassen's/Normal) vs Matrix size

2.4 Results

According to Table 1, the total time cost grows exponentially with regard to the size of matrices. This result accords with the algorithm runtime analysis result, that the runtime of the normal method is $O(n^3)$, and the runtime of Strassen's method is $O(n^{2.8})$.

According to Figure 1 and the last column of Table 1, the proportion value (strassen time/normal time) is high when the matrix is small, but the value get smaller as the size of matrices get larger. However, even the largest matrices in our implementation, which have size of $(4096, 4096)$, the proportion value didn't get below 1. The value 1.22 means that Strassen's method is about 20% slower than the normal method. Depending on the trend of the line, we expected Strassen's method to be faster than the normal method when matrix size is equal to or larger than either $(2^{13}, 2^{13}) = (8192, 8192)$ or $(2^{14}, 2^{14}) = (16384, 16384)$. Since the total time for running the program to the current stage is about 93000 seconds (more than one day), it is impossible for us to run the program to multiply matrices of size $(16384, 16384)$. It will cost more than 1 week!

3 Revised Strassen's method

3.1 Revised algorithm: change base step

According to the results from program implementation in the previous part, the proportion values (strassen time/normal time) for relatively small matrices are very large. For example, for matrix multiplication of $(8, 8)$ matrices, the Strassen't method is 437% slower than the normal method. Thus, we decided to combine Strassen's method and the normal method for matrix multiplication to get a revised Strassen's method.

In the unrevised version, we do divide and conquer in each recursive step, and the size of matrices in the base step is $(1, 1)$. In the revised version, we change the size of matrices in the base step to $(2, 2)$ or $(4, 4)$. And for the base step matrix multiplication, we use the normal method. Since the normal method is much faster than Strassen's method for relatively small matrices , this revised version will be much faster than the unrevised version as well, although the runtime complexity doesn't change.

3.2 Python program

In the function for the revised Strassen's method, everything is the same except the base step. The variable n here represents the size $(2^n, 2^n)$ for the base step matrix.

```
# Strassen's method revised
def strassen_revised(x, y, n):
    # Base case when size of matrices is (2^n)*(2^n)
    if len(x) == 2**n:
        return multiply(x,y)
    # Splitting the matrices into quadrants. This will be done recursively until
     the base case is reached.
    a, b, c, d = split(x)
    e, f, g, h = split(y)
    # Computing the 7 products, recursively (p1, p2...p7)
    p1 = strassen_revised(a, f - h, n)
    p2 = strassen_revised(a + b, h, n)
    p3 = strassen_revised(c + d, e, n)
    p4 = strassen_revised(d, g - e, n)
    p5 = strassen_revised(a + d, e + h, n)
```

```
    p6 = strassen_revised(b - d, g + h, n)
    p7 = strassen_revised(a - c, e + f, n)
    # Computing the values of the 4 quadrants of the final matrix c
    c11 = p5 + p4 - p2 + p6
    c12 = p1 + p2
    c21 = p3 + p4
    c22 = p1 + p5 - p3 - p7
    # Combining the 4 quadrants into a single matrix by stacking horizontally
    and vertically.
    c = np.vstack((np.hstack((c11, c12)), np.hstack((c21, c22))))
    return c
```

In the main function, we run the revised Strassen's method for matrices of sizes from $(2, 2)$ to $(2^9, 2^9) = (512, 512)$. First, we run the normal method and store the time. Then, we run the revised Strassen's method for different base step matrix sizes $(1, 1), (2, 2), (4, 4)$. For $(1, 1)$ base matrix size, the algorithm is the same as unrevised Strassen's method. When the program finish the multiplications, we calculate the proportion value by strassen time/normal time. We also print out the time taken by each step and each size. When all the calculations are done, we print out the time values in the lists and make a plot of proportion values.

```
#start point (2^x): x range
x_range = [0,1,2]
max_power = 9
length = 2**max_power
theA = np.random.randint(low=1, high=10, size=(length,length))
theB = np.random.randint(low=1, high=10, size=(length,length))
theC = np.random.randint(low=1, high=10, size=(length,length))
#let computer warm up first
C = multiply(theA, theB)
#initializations
powers = np.arange(1,max_power+1)
normals = np.ones(len(powers))

#normal
print("normal: ")
for i in range(len(powers)):
    power = powers[i]
    n = 2**power
    A = theA[:n,:n]
```

```
    B = theB[:n,:n]
    C = theC[:n,:n]
    #normal
    start = time.perf_counter()
    D = multiply(A, B)
    D = multiply(A, C)
    D = multiply(B, C)
    end = time.perf_counter()
    normal_time = (end-start)/3
    normals[i] = normal_time
    print("2^", power, "done, normal time: ", normal_time)

#strassen revised
for i in range(len(x_range)):
    x = x_range[i]
    print("x =", x)
    powers_new = np.arange(x+1, max_power+1)
    strassens_new = np.zeros(len(powers_new))
    prop_new = np.zeros(len(powers_new))
    for j in range(len(powers_new)):
        power = powers_new[j]
        n = 2**power
        A = theA[:n,:n]
        B = theB[:n,:n]
        C = theC[:n,:n]
        #strassen
        start = time.perf_counter()
        D = strassen_revised(A, B, x)
        D = strassen_revised(A, C, x)
        D = strassen_revised(B, C, x)
        end = time.perf_counter()
        strassen_time_new = (end-start)/3
        strassens_new[j] = strassen_time_new
        #result
        prop_new[j] = strassen_time_new/(normals[j+x])
        print("2^", power, "done, strassen time: ", strassen_time_new)
    print(strassens_new)
    print(prop_new)
    plt.plot(powers_new, prop_new, marker='o', label="base={}*{}".format(2**x,
```

```
    2**x))
59      for k in range(len(powers_new)):
60          plt.text(powers_new[k], prop_new[k], round(prop_new[k],2), ha='center',
    va='bottom')
61  plt.xlabel('size = 2^')
62  plt.ylabel('proportion')
63  plt.grid()
64  plt.legend()
```

When all the calculations are done, we print out the time values in the lists. Then we make a plot using *plt.plot* function.

3.3 Graphs and plots

Matrix size	Normal method	Unrevised Strassen's method	Revised Strassen's method (2,2)	Revised Strassen's method (4,4)
(2,2)	0.00001	0.00013	-	-
(4,4)	0.00004	0.00033	0.00014	-
(8,8)	0.00028	0.00173	0.00053	0.00036
(16,16)	0.00217	0.01176	0.00357	0.00220
(32,32)	0.01718	0.07844	0.02552	0.01524
(64,64)	0.13288	0.51611	0.17340	0.10015
(128,128)	1.04454	3.58248	1.15614	0.70194
(256,256)	8.3106	25.06217	8.01112	4.87677
(512,512)	67.39664	176.47650	56.34925	34.10312

Table 2: Time costs for different methods

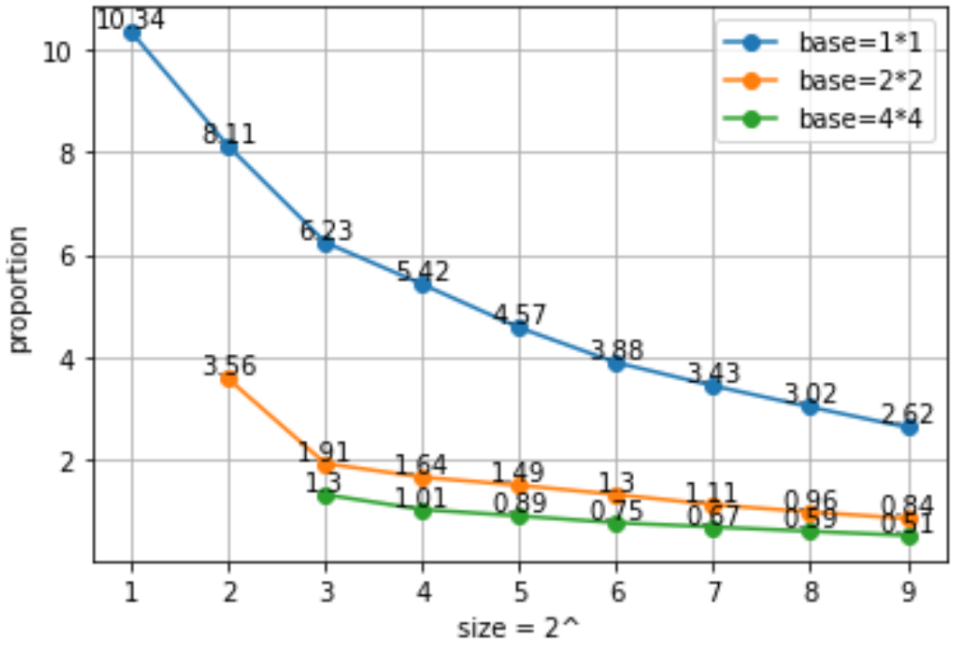

Figure 2: Proportion (Strassen's/Normal) vs Matrix size

3.4 Results

According to Table 2 and Figure 2, the time cost of the revised Strassen's method is much shorter than the revised version. On the figure, the line for the revised Strassen's method with $(2, 2)$ base step is way below the line for the unrevised Strassen's method. The line for the revised Strassen's method with $(4, 4)$ base step is even lower. When the base step matrix size is $(2, 2)$, the revised Strassen's method is faster than the normal method when the matrix size is equal to or greater than $(2^8, 2^8) = (256, 256)$. When the base step matrix size is $(4, 4)$, the revised Strassen's method is faster than the normal method when the matrix size is equal to or greater than $(2^5, 2^5) = (32, 32)$. This result accords with our expectations.

4 Conclusion

From the original implementation of the Strassen's matrix multiplication method, we can conclude that even though the Strassen's method is asymptotically faster than the normal matrix multiplication method, the normal method still has a better runtime for relatively small matrices. However, if we revise the algorithm and modify the base case Strassen's algorithm, so that when we reach some matrices with small enough sizes (e.g. 4×4, 8×8, etc.) we run the naive matrix multiplication method, instead of performing basic arithmetics on 1 matrices, the runtime of Strassen's algorithm would improve drastically, and the runtime is negatively correlated with the size of the matrix on which we decide to run the normal matrix multiplication method.

On the other hand, for the multiplication of matrices with a size that is large enough, Strassen's algorithm always performs better than the normal algorithm, for the asymptotic runtime of Strassen's method is close to, but ultimately less than, that of the normal method.

5 References

Divide and conquer: Set 5 (Strassen's matrix multiplication) (2023a) *GeeksforGeeks*.
Available at: https://www.geeksforgeeks.org/strassens-matrix-multiplication/
(Accessed: 09 May 2023).

Mahato, S. (2021) *Strassen's Matrix Multiplication Algorithm, Medium*
Available at: https://medium.com/swlh/strassens-matrix-multiplication-algorithm-936f42c2b344#:~:text=Strassen%20algorithm%20is%20a%20recursive,four%202%20x%202%20matrices
(Accessed: 09 May 2023).

Strassen's matrix multiplication (2022) *InterviewBit*.
Available at: https://www.interviewbit.com/blog/strassens-matrix-multiplication/
(Accessed: 09 May 2023).