DSAA Project Report of Group 2

This is the report for the project of DSAA course, written by Group 2, with following members.

- HUANG Guanchao, SID 11912309 from SME
- ZHENG Shuhan, SID 11712401 from PHY
- LI Yuru, SID 11911035 from EIE
- TIAN Yuqiong, SID 11911039 from EIE

The complete resources of our work in this project, including Python source codes, testing scripts, experiment raw data, figures and report can be retrieved at our GitHub repo

Introduction

Matrix multiplication (MM) is a practical application of linear algebra. It is a useful tool in many fields, including mathematics, physics and electrical engineering. For example, we can calculate the path between two places and solve the profit problem of goods in real life. Moreover, matrix multiplication is of great use in cryptology.

Previously, people used traditional matrix multiplication which was based on the definition of MM to calculate the product of two matrices. The traditional matrix multiplication algorithm contains three circulations. Concretely, $C_{ij} = A_{ik}B_{kj}$, each index $(i,\ j,\ k)$ runs from 1 to n. Therefore, the time complexity of which is $\Theta\left(n^3\right)$, where n is the length of the square matrix. The time complexity of traditional method is large, hence, with the extensive application of MM, optimization of MM becomes more and more important. Without considering the degree of matrix density, how to effectively reduce the number of arithmetic multiplication in MM is a major direction for optimization.

The earliest MM optimized algorithm was proposed by German mathematician Volker Strassen in 1969 and was named Strassen's algorithm. Its main idea is to replace multiplication by the combination of addition and subtraction. The result is obtained by piecing some indirect terms together and using the addition and subtraction on these indirect terms to cancel out part of the terms.

Strassen's algorithm embodies two different locality properties because its two basic computations exploit different data locality: matrix multiplication has spatial and temporal locality, and matrix addition has only spatial locality. This method contains 7 MMs and 22 MAs. For those 7 MMs, the runtime is

$$2\pi\left(\left\lceil\frac{m}{2}\right\rceil\left\lceil\frac{n}{2}\right\rceil p + mn\left\lceil\frac{p}{2}\right\rceil + \left\lceil\frac{m}{2}\right\rceil\left\lfloor\frac{n}{2}\right\rfloor\left\lceil\frac{p}{2}\right\rceil\right),$$

where π is the efficiency of MM, and $1/\pi$ is simply the **floating point operation per second** (FLOPS) of the computation. For 22 MAs (18 matrix additions and 4 matrix copies), the runtime is

$$\alpha\left[5\left\lceil\frac{n}{2}\right\rceil\left(\left\lceil\frac{m}{2}\right\rceil + \left\lceil\frac{p}{2}\right\rceil\right) + 3mp\right],$$

where α is the efficiency of MA.

Strassen's algorithm also has **layout effects**. That is, the performance of MA is not affected by a specific matrix layout or shape as long as we can exploit the only viable reuse: spatial data reuse. We know that data reuse (spatial/temporal) is crucial for matrix multiply. In practice, ATLAS and GotoBLAS cope rather well with the effects of a (limited) row/column major format reaching often 90% of peak performance. Thus, we can assume for practical purpose that π and α are functions of the matrix size only.

For Strassen's algorithm, the time complexity is $O\left(n^{\lg 7}\right)$. For a two order matrix multiplication, we need to spend $8 \times (2^3)$ of runtime with obvious matrix multiplication algorithm, but we just need $7 \times \left(2^{\lg 7}\right)$ by using Strassen's algorithm, the time complexity is decreased. However, the space complexity of Strassen's algorithm may be increased, since the more spaces are needed to save the submatrix.

After Strassen came up with this algorithm, more and more optimized algorithms were proposed by different people. For example, Pan's algorithm was proposed in 1981[1], time complexity of which is reduced to $O\left(n^{2.494}\right)$. Later, Andrew Stothers proposed a new algorithm in his paper in 2010[2], the time complexity of which is $O\left(n^{2.374}\right)$. Then, in 2014, François Le Gall[3] simplified

Stanford's algorithm [4] and the time complexity was reduced to $O\left(n^{2.3728639}\right)$, which is the most optimized algorithm for matrix multiplication so far.

In this project, we will mainly focus on the Strassen's algorithm. We will apply it to higher order matrix multiplication to make a comparison between the traditional matrix multiplication and the Strassen's algorithm and discuss more details about it. Further more, in this project, we also tried finding the crossover point for these two methods.

Background

Usefulness of Adaptive Strassen's Algorithm

The writers of the provided paper, Paolo D'Alberto and Alexandru Nicolau talked about an easy-to-use adaptive algorithm [5] which combines a novel implementation of Strassen's idea with matrix multiplication from different systems like ATLAS. The Strassen's algorithm has decreased the running time of matrix multiplication significantly, by replacing one discrete matrix multiplication with several matrix additions. However, for modern architectures with complex memory hierarchies, the matrix additions have a limited in-cache data reuse and thus poor memory-hierarchy utilization.

The first benefit of their algorithm authors listed is that, they divide the MM problems into a set of balanced sub-problems without any matrix padding or peeling. Second, in addition, their algorithm applies Strassen’s strategy recursively as many times as a function of the problem size. Third, they store matrices in standard row or column major format so that they can yield control to a highly tuned matrix multiplication. The authors' algorithm applies to any size and shape matrices. These are the advantages of adaptive algorithm provided by the authors of the given paper.

Implementation of the Adaptive Method

The concrete implementation of the adaptive algorithm is realized by several steps. To begin with, authors declared several notations and computations. As the authors suggests, their algorithm reduces the number of variables initialization and assignment as well as the number of computations because of a balanced division process.

For matrix $C = A \times B$, where $\sigma(A) = m \times n$, decompose A to four small matrices A_0, A_1, A_2 and A_3. The size of four small matrices are

$$\begin{cases} \sigma(A_0) = \left\lceil \frac{m}{2} \right\rceil \times \left\lceil \frac{n}{2} \right\rceil \\ \sigma(A_1) = \left\lceil \frac{m}{2} \right\rceil \times \left\lfloor \frac{n}{2} \right\rfloor \\ \sigma(A_2) = \left\lfloor \frac{m}{2} \right\rfloor \times \left\lceil \frac{n}{2} \right\rceil \\ \sigma(A_3) = \left\lfloor \frac{m}{2} \right\rfloor \times \left\lfloor \frac{n}{2} \right\rfloor \end{cases}$$

The dimension is the same for matrix B. When conducting matrix addition, we expand the scope of matrix addition to different sizes. For example, we define matrix $X = Y + Z$, if the size of Y and Z is not the same, then we expand the size of X to the largest of them and the redundant part of X is set to be 0. After that, the adaptive algorithm begins. Several matrix additions and multiplications are performed and are put into practice to several systems. Through this process, the authors found that the same algorithm applied to different systems can have different results.

Experiment Design and Results

Generally, the adaptive algorithm is based on the Strassen's method and have some advanced operations. The Strassen's method has its run-time advantage when the matrix size is quite big, but this advantage cannot be exhibited when the size is not big enough, since it adds a large quantity of matrix additions compared to the traditional standard matrix multiplication. For instance, the size has to be greater than about 1000 by 1000 for Strassen's method begins to show its superiority. However, speedups up to 30% are observed over already tuned MM using this hybrid approach.

Possible Applications

The adaptive Strassen's algorithm can be useful in some real-world problems involving matrix multiplication, especially for those matrix of size big and not fixed. For practical application, this algorithm can be applied to many industrial problems. For instance, the circuits equations in EDA which include several unknown parameters can be solved quickly using matrix divisions, equivalent to matrix multiplications as well.

In addition, the signal processing can make good use of the algorithm as well, for the input and output signals' relationship can be expressed in matrix. With the advantage of changeable size and comparatively fast speed, the adaptive method has great potential in real-world problems.

Theoretical Analysis

Time Complexity of Standard Matrix Multiplication

From the pseudocode we can know that there are three for-loop cycles so that the time complexity for standard matrix multiplication is $\Theta\left(n^3\right)$.

Time Complexity of Strassen Algorithm

From the recurrence relationship we know that the run-time complexity satisfies

$$\begin{cases} T(n) = O(1), n = 2 \\ T(n) = 7T\left(\frac{n}{2}\right) + \Theta\left(n^2\right), n > 2 \end{cases}$$

Then we can calculate the time complexity for Strassen's algorithm.

$$\begin{aligned}
T(n) &= 7T\left(\frac{n}{2}\right) + O\left(n^2\right) \\
&= 7\left\{7T\left(\frac{n}{2^2}\right) + O\left[\left(\frac{n}{2}\right)^2\right]\right\} + O\left(n^2\right) \\
&= 7^2T\left(\frac{n}{2^2}\right) + 7O\left(\frac{n^2}{4}\right) + O\left(n^2\right) \\
&= 7^2\left\{7T\left(\frac{n}{2^3}\right) + O\left[\left(\frac{n}{2^2}\right)^2\right]\right\} + 7O\left(\frac{n^2}{4}\right) + O\left(n^2\right) \\
&= 7^3T\left(\frac{n}{2^3}\right) + 7^2O\left(\frac{n^2}{4^2}\right) + 7O\left(\frac{n^2}{4}\right) + O\left(n^2\right) \\
&= 7^3T\left(\frac{n}{2^3}\right) + 7^2O\left(\frac{n^2}{4^2}\right) + 7O\left(\frac{n^2}{4^1}\right) + 7^0O\left(\frac{n^2}{4^0}\right) \\
&= \cdots \\
&= 7^kT\left(\frac{n}{2^k}\right) + 7^{k-1}O\left(\frac{n}{4^{k-1}}\right) + 7^{k-2}O\left(\frac{n}{4^{k-2}}\right) + \cdots + 7^2O\left(\frac{n^2}{4^2}\right) + 7O\left(\frac{n^2}{4}\right) + 7^0O\left(\frac{n^2}{4^0}\right)
\end{aligned}$$

Therefore, we have

$$\begin{aligned}
T(n) &\leq 7^kO(1) + cn^2\left(\frac{7}{4}\right)^{k-1} + cn^2\left(\frac{7}{4}\right)^{k-2} + \cdots + cn^2\left(\frac{7}{4}\right)^2 + cn^2\left(\frac{7}{4}\right)^1 + cn^2\left(\frac{7}{4}\right)^0 \\
&\leq 7^kc + cn^2\left[\left(\frac{7}{4}\right)^{k-1} + \left(\frac{7}{4}\right)^{k-2} + \left(\frac{7}{4}\right)^2 + \left(\frac{7}{4}\right)^1 + \left(\frac{7}{4}\right)^0\right] \\
&\leq c7^{\lg(n/2)} + cn^2\frac{1\left[1-(7/4)^k\right]}{1-7/4} \\
&\leq c7^{\lg(n/2)} + cn^2\frac{4}{3}\left[\left(\frac{7}{4}\right)^k - 1\right] \\
&\leq c7^{\lg(n/2)} + cn^2\frac{4}{3}\left[\left(\frac{7}{4}\right)^{\lg(n/2)} - 1\right] \\
&\leq c7^{\lg(n/2)} + cn^2\frac{4}{3}\left[\frac{4}{7}\left(\frac{7}{4}\right)^{\lg n} - 1\right] \\
&\leq \frac{c}{7}7^{\lg n} + cn^2\left[\frac{16}{21}\left(\frac{7}{4}\right)^{\lg n} - 1\right] \\
&\leq \frac{c}{7}n^{\lg 7} + cn^2\left[\frac{16}{21}n^{\lg(7/4)} - 1\right] \\
&\leq Cn^{\lg 7} - Cn
\end{aligned}$$

Since $\lg 7 > 2$, we can obtain the time complexity for Strassen's algorithm is

$$O\left(n^{\lg 7}\right) \approx O\left(n^{2.81}\right)$$

Crossover Point Estimation

The crossover point means the size of matrix when the runtime of Strassen algorithm starts to be faster than the standard matrix multiplication.

From the time complexity we know that the Strassen's method is faster than the standard matrix multiplication, however, in practice, for small matrices, Strassen's method has a significant overhead and conventional MM results in better performance. To overcome this, several authors in the provided paper point at hybrid algorithms, by deploying Strassen's method in conjunction with conventional matrix multiplication. For our project, we explored the crossover point in practical ways, running the Python code for both Strassen's method and standard matrix multiplication. The concrete result is given in the empirical analysis, and is explained in different illustrations.

Moreover, for square matrices, there is something special. Combining the performance properties of both matrix multiplications and matrix additions with a more specific analysis for only square matrices, that is, $n = m = p$. Based on this, the equation

$$\left\lfloor \frac{m}{2} \right\rfloor \left\lceil \frac{n}{2} \right\rceil \left\lceil \frac{p}{2} \right\rceil \leq \frac{\alpha}{2\pi} \left[5 \left\lceil \frac{n}{2} \right\rceil \left(\left\lceil \frac{m}{2} \right\rceil + \left\lceil \frac{p}{2} \right\rceil \right) + 3mp \right]$$

can be simplified. And from this equation, we can find that there exists a **recursion point** n_1 for Strassen's algorithm and

$$n_1 = 22 \frac{\alpha}{\pi}.$$

Methodology

Strassen Algorithm

Comparing to the obvious matrix multiplication, the Strassen's Algorithm replaces matrix multiplication into the matrix addition. In this algorithm,

- Operand matrices are divided into some submatrices and some other submatrices are defined to be the basic operated matrices, on which addition or subtraction of are conducted.
- Repeat these procedures on all submatrices to obtain the resulting submatrices, which we define as P.
- After this, the submatrices of the product of the original operand matrices are obtained.
- Finally, combine all these submatrices to get the result.

Since the Strassen's Algorithm replaces the one separated matrix multiplication with several new matrix additions, it can significantly reduce the running time of matrix multiplication. The pseudocode for Strassen's method used in two-ordered matrix can be written as follows:

```
Strassen(A, B)
    S1 = B12 - B22
    S2 = A11 - A12
    S3 = A21 + A22
    S4 = B21 - B11
    S5 = A11 + A22
    S6 = B11 + B22
    S7 = A12 - A22
    S8 = B21 + B22
    S9 = A11 - A21
    S10 = B11 + B12
    P1 = Strassen(A11, S1)
    P2 = Strassen(A11, B22)
    P3 = Strassen(S3, B11)
    P4 = Strassen(A22, S4)
    P5 = Strassen(S5, S6)
    P6 = Strassen(S7, S8)
    P7 = Strassen(S9, S10)
    C11 = P5 + P4 - P2 + P6
    C12 = P1 + P2
    C21 = P3 + P4
    C22 = P5 + P1 - P3 - P7
    return C
```

For n-ordered matrix multiplication, the pseudocode of Strassen's algorithm is :

```
Strassen(A, B)
    n = A.row
    Let C be a new matrix
    if n == 1 to n
        c11 = a11b11
    else partition A, B and C
        P1 = Strassen(A11, B12 - B22)
        P2 = Strassen(A11 + A12, B22)
        P3 = Strassen(A21 + A22, B11)
        P4 = Strassen(A22, B21 - B11)
        P5 = Strassen(A11 + A22, B11 + B22)
        P6 = Strassen(A12 - A22, B21 + B22)
        P7 = Strassen(A11 - A21, B11 + B12)
        C11 = P5 + P4 - P2 + P6
        C12 = P1 + P2
        C21 = P3 + P4
        C22 = P5 + P1 - P3 - P7
    return C
```

According to the recurrence relation, the running time for Strassen's method is $T(n) = 7T(n/2) + \Theta(n^2)$. According to the theoretical analysis above, the time complexity for Strassen's method is $O(n^{\lg 7})$. Compared to the time complexity of traditional matrix multiplication $O(n^3)$, we can calculate that when n larger than n_1 which is the **crossover point**, which is discussed later, the time of Strassen's algorithm is significantly lower than the standard matrix multiplication.

Standard Matrix Multiplication

For the standard matrix multiplication, the running time is $\Theta(n^3)$, and the pseudocode is given below:

```
SQUARE-MATRIX-MULTIPLY(A, B)
    n = A.rows
    let C be a new n * n matrix
    for i = 1 to n
        for j = 1 to n
            Cij = 0
            for k = 1 to n
                Cij = Cij + aik * bkj
    return C
```

Experiment Design

Class `Matrix`

A data structure for matrix. The matrix implementation is suitable for dense matrices. A class `Matrix` is defined in the implementation.

This class `Matrix` represent a matrix in row-major order. The class provides a constructor and methods to get and set the element at any row column index.

> For an n by n matrix the first row will be stored in an array at index `0` to index `n - 1`, the next row is at `n` to `2 * n - 1`, and so on.

Attributes:

- `row`: `int`, default `1`
- `col`: `int`, default `1`
- elements: `list`, default `[0.0]`

Methods:

- `__init__(self, elements, row=None, col=None):` Generate a `Matrix` object
- `__str__(self):` Return a row * col matrix-like `String`.
- `__getitem__(self, item):` Return elements in a `Matrix` object.
- `__add__(self, other):` If `other` is a `Matrix`, perform matrix addition, else perform addition with a number element-wisely. Return a `Matrix` object.
- `__sub__(self, other):` If other is a `Matrix`, perform matrix subtraction, else perform subtraction with a number element-wisely. Return a `Matrix` object
- `__sizeof__(self):` Return the number of elements in a `Matrix` object.
- `dimension(self):` Return row and column numbers of a `Matrix` object.

Data Storage

Matrix elements are stored in a list following row-major order.

For instance, a 2 by 2 matrix A is stored as a list $[A_{11}\ \ A_{12}\ \ A_{21}\ \ A_{22}]$.

Indexing

The indexing rule of an element in a `Matrix` object follows conventions in math.

That is, `Matrix[i, j]` is the element in the `i`th row and `j`th column (`i` and `j` run from 1 to `Matrix.row` and `Matrix.col` respectively).

Other Operations

Function `adaptive_add()` and `adaptive_minus()`

- `adaptive_add(a, b, target_row, target_col):`
  Given target matrix size, perform matrix addition of Matrix a and b.
  The function is called by `strassen_matrix_multiply()`.
  Return a `Matrix` object with the size of `target_row`*`target_col`.
- `adaptive_minus(a, b, target_row, target_col):`
  Given target matrix size, perform matrix subtraction of `Matrix` a and b.
  The function is called by `strassen_matrix_multiply()`.
  Return a `Matrix` object with the size of target_row*target_col.

The following is the code of `adaptive_add(a, b, target_row, target_col)`. The code of `adaptive_minus(a,...)` is similar.

```python
def adaptive_add(a, b, target_row, target_col):
    """
    Given target matrix size, perform adaptive matrix addition
    :type a: Matrix
    :type b: Matrix
    :type target_col: Integer
    :type target_row: Integer
    :return: Matrix
    """
    arow = a.row
    acol = a.col
    brow = b.row
    bcol = b.col
    s = [0] * (target_col * target_row)
    for i in range(target_row):
        for j in range(target_col):
            flag = False
            if 0 <= i < arow and 0 <= j < acol:
                s[i * target_col + j] = a[i + 1, j + 1]
                flag = True
            if 0 <= i < brow and 0 <= j < bcol:
                s[i * target_col + j] = s[i * target_col + j] + b[i + 1, j + 1]
                flag = True
            if not flag:
                s[i * target_col + j] = 0
    c = Matrix(s, target_row, target_col)
    return c
```

Function `square_matrix_multiply()`

`square_matrix_multiply(a, b):`

Given `Matrix` a and b, perform standard matrix multiplication. `a.col` must equal to `b.row`. Return a `Matrix` object.

Function `strassen_multiply()`

`strassen_multiply(a, b, n=None):`

Given `Matrix` a and b, perform an improved version of Strassen's algorithm.

The `adaptive_add()` and `adaptive_minus()` are called in `strassen_multiply()` to tackle arbitrary matrix size inputs.

The algorithm is based on the paper published by Paolo D’Alberto and Alexandru Nicolau in 2007.

Return a `Matrix` object.

Function `random_matrix_gen()`

This function is for generating matrices for testing.

```python
def random_matrix_gen(n):
    """
    Generates a random matrix of size n by n, the elements are randomly from -1 to 1 float number.
    :param n: the size of the matrix
    :type n: int
    :return: The generated random matrix
    :rtype Matrix
    """
    elements = []
    for i in range(n * n):
        elements.append(random.uniform(-1, 1))
    return Matrix(elements, n, n)
```

Empirical Analysis

Testing Platform

The specifications of our main testing platform is as follows:

- Hardware
  - AMD Ryzen 9 3900X, with 12 cores, 24 threads, running at 3.8GHz, **maximum turbo frequency 4.6GHz**
  - ADATA DDR4 3200MHz 16GB × 4, **running at 2666MHz**, quad channel
  - Gigabyte X570 Gaming X
- Software
  - Windows 10 Professional 20H2
  - Python 3.9.1
  - PyCharm 2020.3, Professional Edition
  - Visual Studio Community 2019

> Special thanks to SUN Jiachen, for providing us with this powerful testing platform.

Some other lightweight tests are conducted on our own platform Surface Pro 6, the specifications are shown below.

- Hardware
  - Microsoft Surface Pro 6 1796
  - Intel Core i7-8650U, with 4 cores, 8 threads, 1.99GHz, running at 2.11GHz, **maximum Turbo frequency 4.2GHz**
  - 8GB of RAM, dual channel, **running at 1867MHz**
- Software
  - Windows 10 Professional 20H2
  - WSL2, Kali Linux
  - Python 3.9.1, running in WSL2
  - PyCharm 2020.3 Professional Edition
  - Visual Studio Code, with Pylance engine
  - Visual Studio Community 2019

Performance Benchmark

According to Paolo D'Alberto and Alexandru Nicolau, the crossover point (or the recursion point) for square matrices case is expressed as

$$n_1 = \frac{\alpha}{\pi}.$$

Therefore, it is natural for us to try to find this ratio by running benchmark, in order to approximate the recursion point. To simulate the real floating point addition and multiplication in MM, we conducted this benchmark by initializing two matrices, and do addition and multiplication by indexing the elements in the matrices.

```python
import time
from Matrix import random_matrix_gen

n = 8192
m1 = random_matrix_gen(n)
m2 = random_matrix_gen(n)

time1 = time.time()
for i in range(n * n):
    s = m1[i] + m2[i]
time2 = time.time()
print(time2 - time1)
```

Similarly, the test for multiplication is written in another module, so that we can utilize analyzer provided by Visual Studio or PyCharm to evaluate the performance.

Firstly, we run the performance benchmark, to figure out the run time of `float` addition and `float` multiplication.

We tested the additions and multiplications between the elements of two matrices of size 8192 by 8192, in total 2^{26} floating point numbers, the total runtime of which is shown below.

Addition	Multiplication	Ratio α/π
89.12498784065247	87.74823570251465	0.984552568572583

Which is obviously an absurd result, since generally, the runtime for addition cannot be longer than multiplication. In this case, the ratio is of sheds no light on the estimation of crossover point.

In this experiment, the larger size of the matrices is, the more convincing the result is. However, too large matrices would lead to unacceptable runtime for a single test. After doing several brief test on multiplication of different matrices size, taking the pace of our work into consideration, we chose 512 by 512 matrices for finding the crossover point.

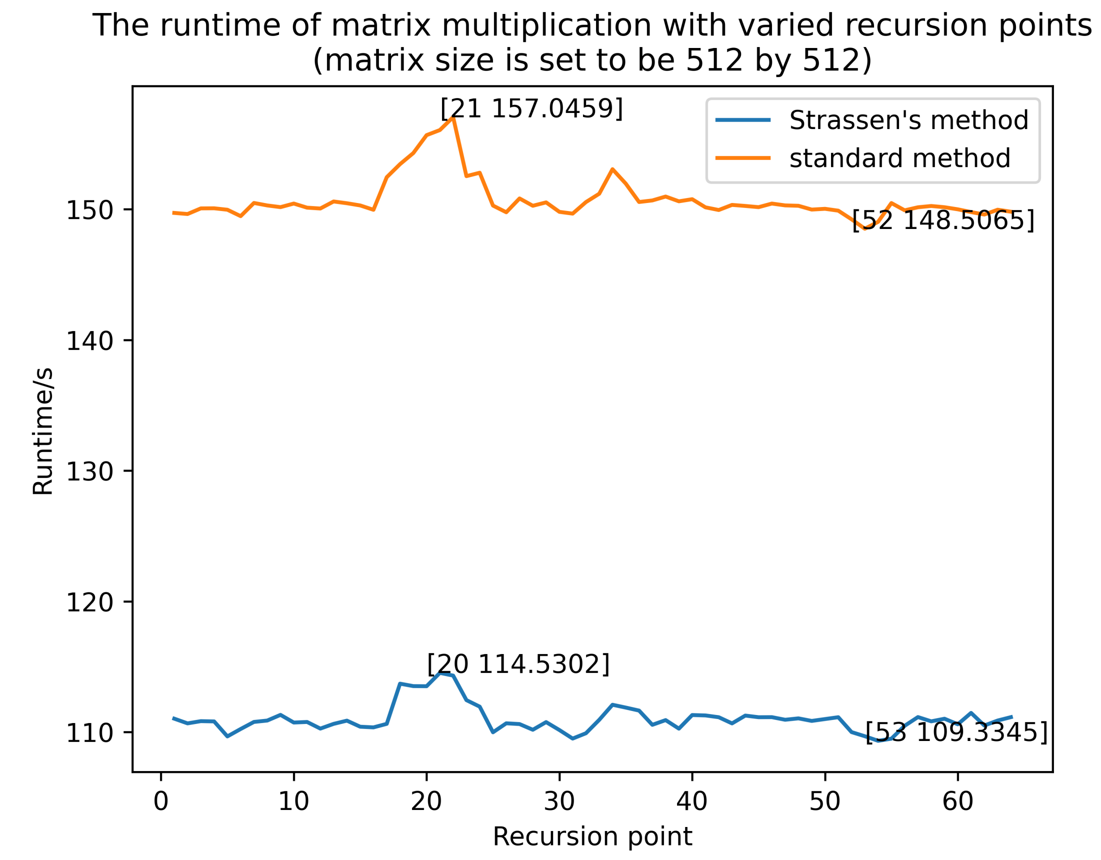
Crossover Point Searching

For searching for the crossover point, we used `openpyxl` package to collect the data in Excel file. In the script, `n` is the matrix size we consider, `s` is the start point for searching, `e` is accordingly the end point, `r` means for each recursion point, tests are conducted twice and take average.

```python
from Matrix import *
import time
import openpyxl

n = 512
s = 1
e = 64
r = 2

print("Matrix generation starts")
m1 = random_matrix_gen(n)
print("Matrix 1 generated")
m2 = random_matrix_gen(n)
print("Matrix 2 generated")

wb = openpyxl.load_workbook('../analysis/data.xlsx')
print("workbook", wb.sheetnames, "loaded")
ws = wb['crossover_point']

for i in range(s, e + 1):
    strassen = 0
    square = 0
    ws['A' + str(i)] = i
    print("The recursion point is set to be ", i)

    for j in range(1, r + 1):
        print("test ", j)

        time1 = time.time()
        c = strassen_multiply(m1, m2, i)
        time2 = time.time()
        print("The run time of Strassen's method is", time2 - time1)
        strassen = time2 - time1 + strassen

        time1 = time.time()
        c1 = square_matrix_multiply(m1, m2)
        time2 = time.time()
        print("The run time of brutal method is", time2 - time1)
        square = time2 - time1 + square

    print("average strassen", strassen / r)
    ws['B' + str(i)] = strassen / r
    print("average square", square / r)
    ws['C' + str(i)] = square / r
    print("==========")

wb.save('../analysis/data.xlsx')
```

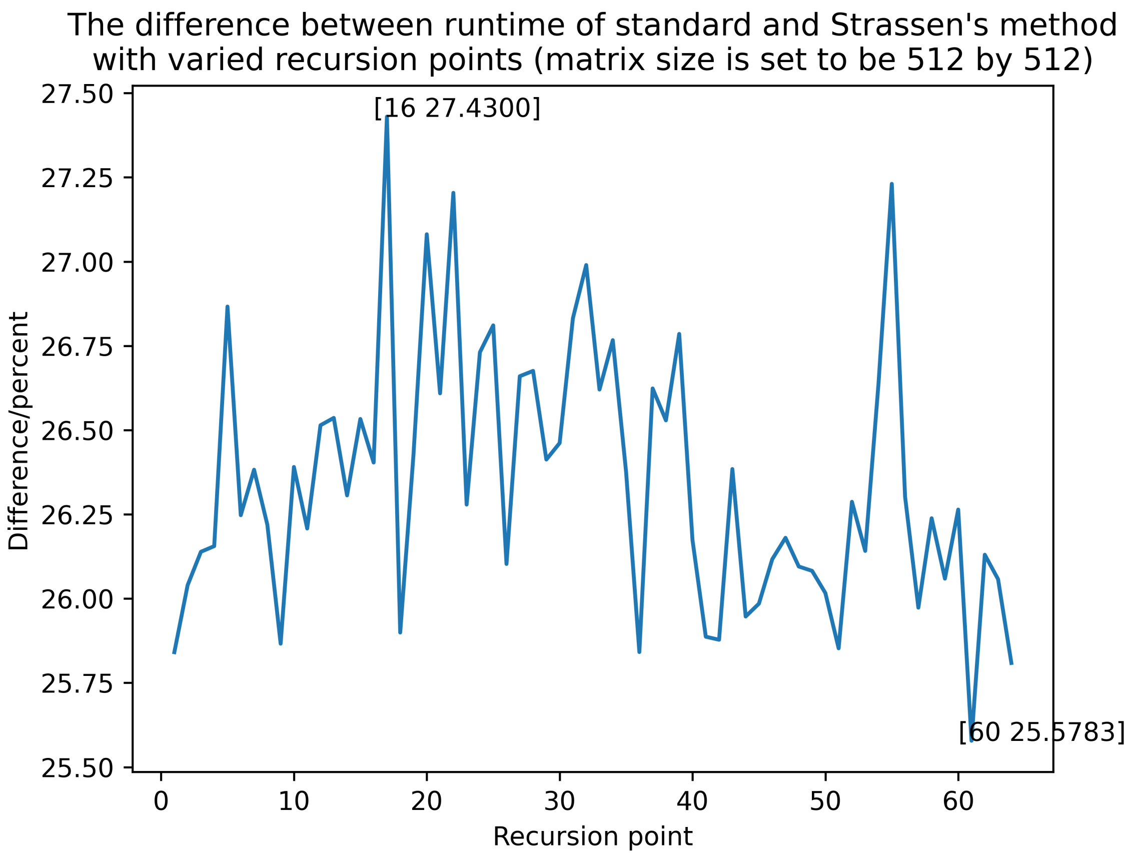
To make the condition as close as possible, `square_matrix_multiply()` is executed again every time `strassen_matrix_multiply()` runs in the loop, even theoretically the recursion point has no affect on the run time of `square_matrix_multiply()`.

The plot of the runtime with respect to the selection of recursion point is shown below.



From the figure, it is clear that, for 512 by 512 matrix multiplication, Strassen's method is stably faster then the standard method for about 40 seconds, even if the recursion point is set to be 1. In general, though the runtime of the two methods has fluctuations, but the tendency is synchronous, hence we may consider these changes as a consequence of the load of the computer and operating system.

More intuitively, the plot of the difference is shown below.



In which the difference percentage is defined as

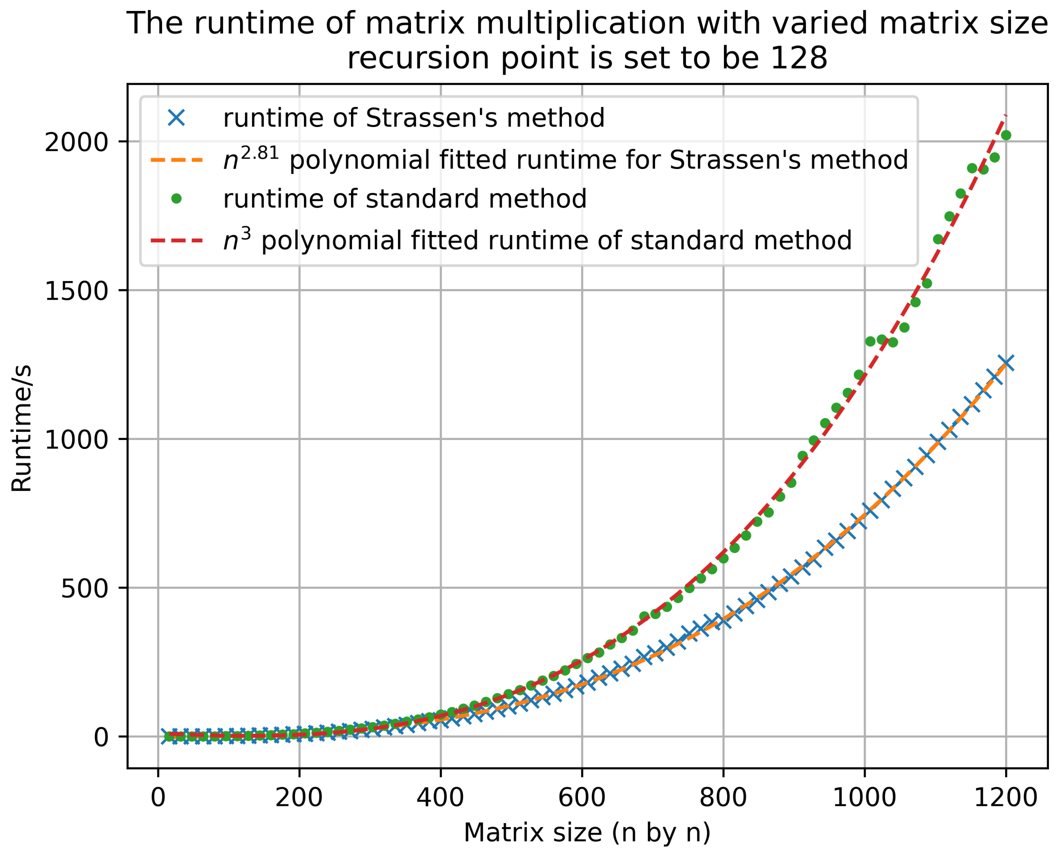
$$\text{difference percentage} = \frac{\text{runtime of standard method} - \text{runtime of Strassen's method}}{\text{runtime of standard method}}$$

No obvious regular pattern could be found from the statistics. Therefore, the affect on the runtime of the Strassen's multiplication due to the choice of recursion point can be neglected.

Algorithm Performance

For Varied Matrix Size

The runtime of Strassen's multiplication and its $n^{2.81}$ polynomial fitted graph, along with that of standard matrix multiplication graph is shown below.



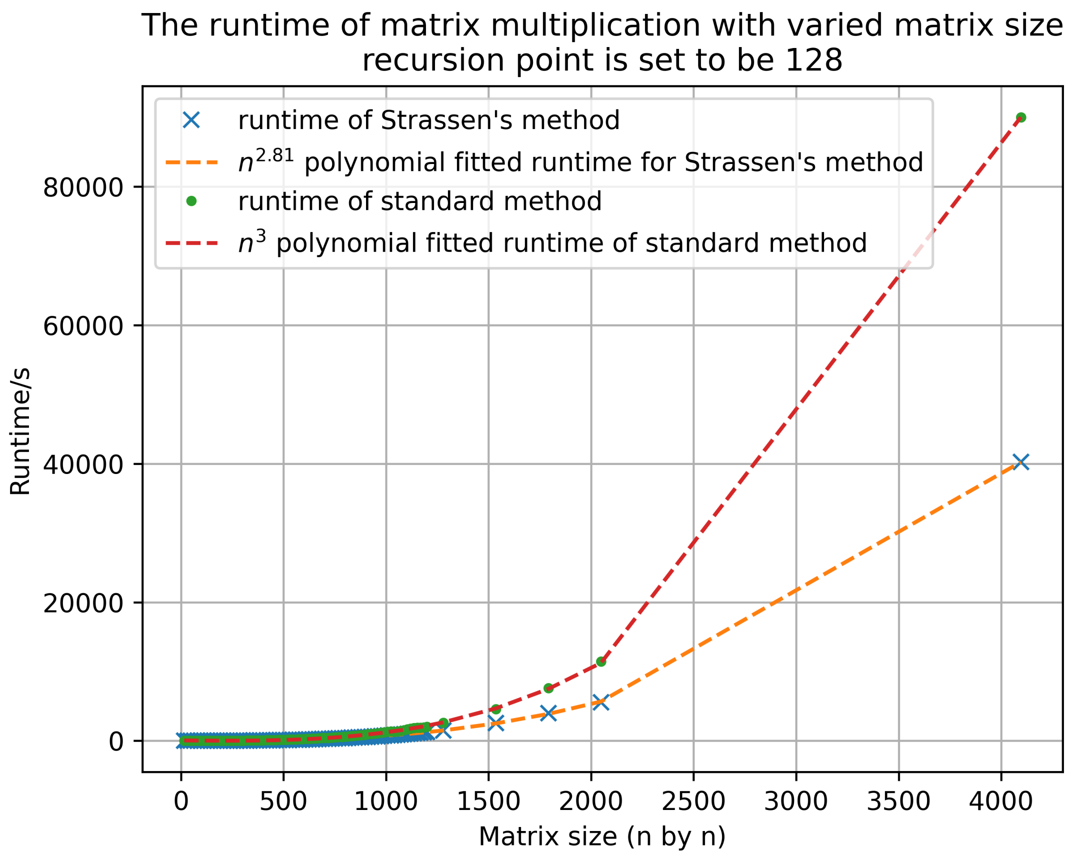
In this figure, the runtime of our Strassen's multiplication can be perfectly fitted with

$$T = 3.0306 \times 10^{-6} n^{2.81} - 0.0001 n^2 + 0.0333 n - 2.7749.$$

The runtime of our standard matrix multiplication can be perfectly fitted with

$$T = 1.1190 \times 10^{-6} n^3 + 1.9060 \times 10^{-4} n^2 - 1.0860 \times 10^{-1} n + 1.0460.$$

Adding a small amount of results from larger matrices, the conclusion is still roughly consistent.

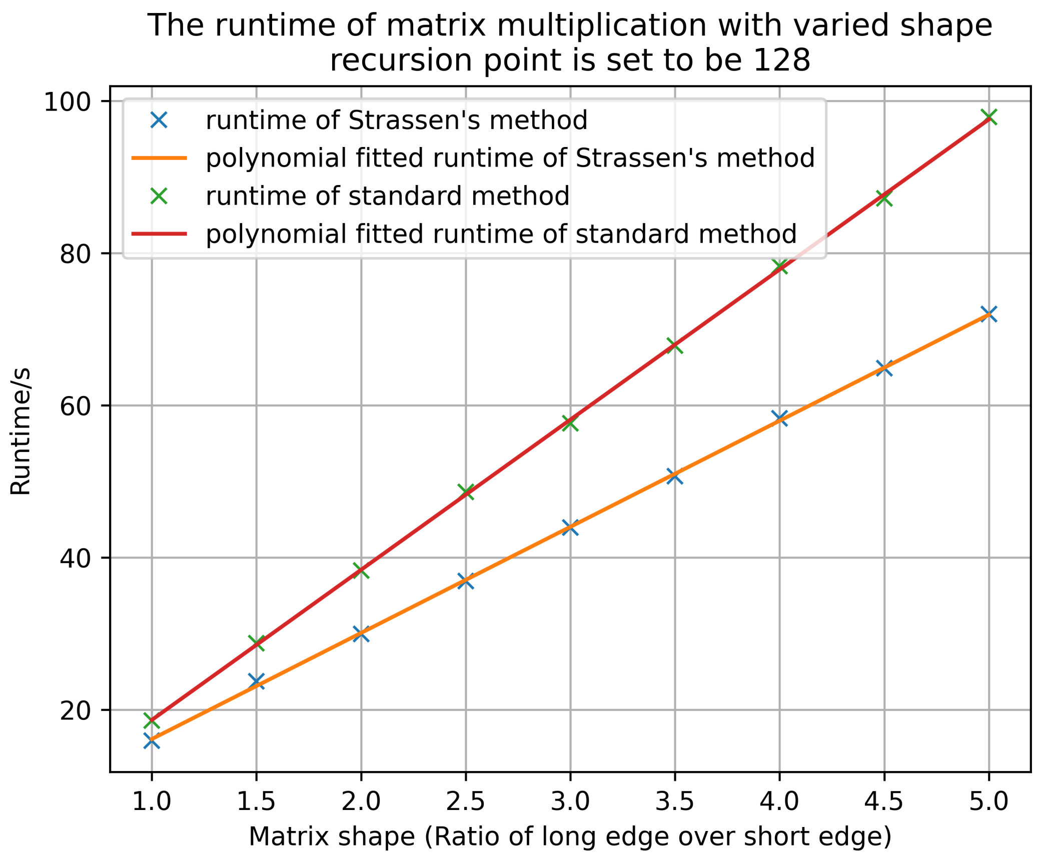


These results perfectly corresponds to the asymptotic estimation of time complexity in theoretical analysis.

For Varied Matrix Shape

Though adaptive Strassen's method, it should work best with square or almost square matrices. We also conducted a test for the performance of adaptive Strassen's method with varied matrix size.

In this experiment, the dimension of the matrices is from 256 by 256 to 256 by 1280, hence the ratio of the long edge over the short edge is from 1 to 5. The results are shown below.



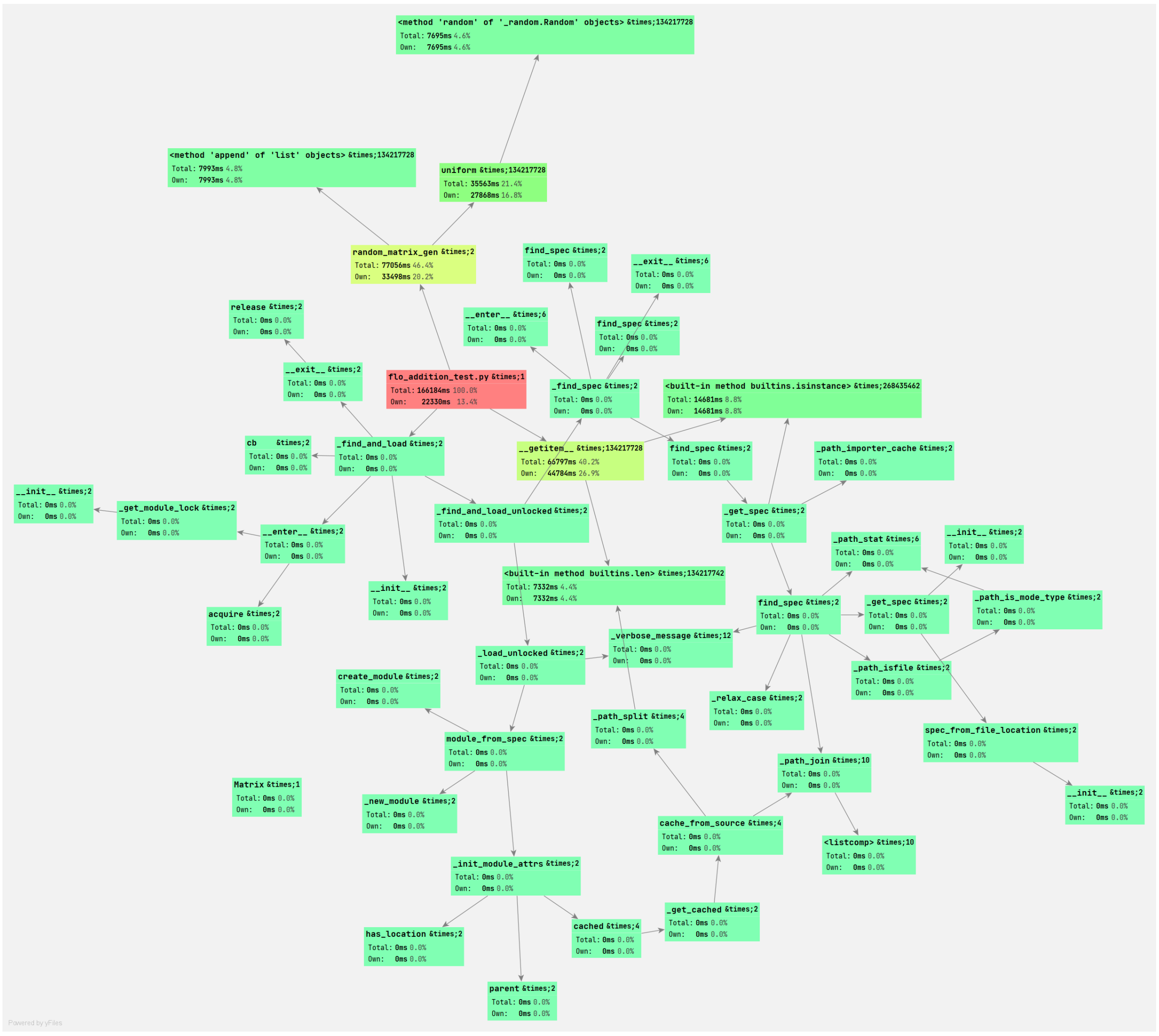
From the figure, it is clear that, the runtime of MM in two methods increases linearly as oen dimension of the matrix grows. However, the runtime of adaptive Strassen's method grows even slower. This contradicts our assumption.

More on FLO Benchmark

The results obtained from FLO benchmark is completely invalid, hence we did further research on it. The profiler tool in PyCharm enables us to analyze the CPU time and function calling tree. According to the report, most of the CPU time were spent on indexing the elements of the matrices.

For Addition Test

The call graph of addition test is shown below.



<method 'random' of '_random.Random' objects> ×134217728 Total: 7695ms 4.6% Own: 7695ms 4.6%
<method 'append' of 'list' objects> ×134217728 Total: 7993ms 4.8% Own: 7993ms 4.8%
uniform ×134217728 Total: 35563ms 21.4% Own: 27868ms 16.8%
random_matrix_gen &times;2 Total: 77056ms 46.4% Own: 33498ms 20.2%
find_spec &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
__exit__ &times;6 Total: 0ms 0.0% Own: 0ms 0.0%
release &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
__enter__ &times;6 Total: 0ms 0.0% Own: 0ms 0.0%
find_spec &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
__exit__ &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
flo_addition_test.py &times;1 Total: 166184ms 100.0% Own: 22330ms 13.4%
_find_spec &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
<built-in method builtins.isinstance> ×268435462 Total: 14681ms 8.8% Own: 14681ms 8.8%
cb &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
_find_and_load &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
__getitem__ ×134217728 Total: 66797ms 40.2% Own: 44784ms 26.9%
find_spec &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
_path_importer_cache &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
__init__ &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
_get_module_lock &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
__enter__ &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
_find_and_load_unlocked &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
_get_spec &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
_path_stat &times;6 Total: 0ms 0.0% Own: 0ms 0.0%
__init__ &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
<built-in method builtins.len> ×134217742 Total: 7332ms 4.4% Own: 7332ms 4.4%
__init__ &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
acquire &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
find_spec &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
_get_spec &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
_path_is_mode_type &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
_verbose_message &times;12 Total: 0ms 0.0% Own: 0ms 0.0%
_load_unlocked &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
create_module &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
_path_isfile &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
_relax_case &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
_path_split &times;4 Total: 0ms 0.0% Own: 0ms 0.0%
module_from_spec &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
spec_from_file_location &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
_path_join &times;10 Total: 0ms 0.0% Own: 0ms 0.0%
Matrix &times;1 Total: 0ms 0.0% Own: 0ms 0.0%
_new_module &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
__init__ &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
cache_from_source &times;4 Total: 0ms 0.0% Own: 0ms 0.0%
<listcomp> &times;10 Total: 0ms 0.0% Own: 0ms 0.0%
_init_module_attrs &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
_get_cached &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
has_location &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
cached &times;4 Total: 0ms 0.0% Own: 0ms 0.0%
parent &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
Powered by yFiles

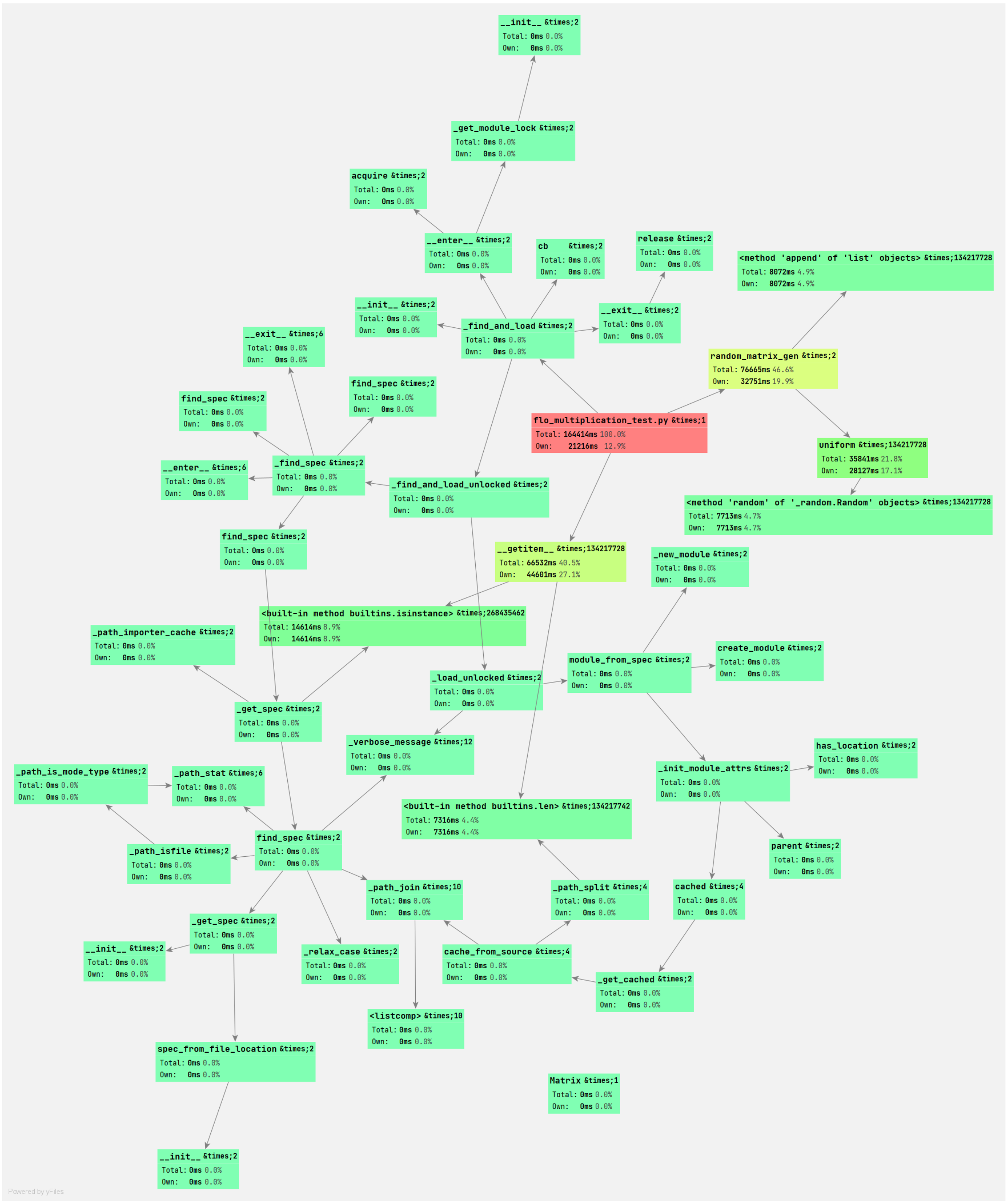
The most called functions are shown below (in order of time).

Name	Call Count	Time (ms)	Time percentage	Own Time (ms)	Own time percentage
`flo_addition_test.py`	1	166184	100.0%	22330	13.4%
`random_matrix_gen`	2	77056	46.4%	33498	20.2%
`__getitem__`	134217728	66797	40.2%	44784	26.9%
`uniform`	134217728	35563	21.4%	27868	16.8%
<built-in method `builtins.isinstance`>	268435462	14681	8.8%	14681	8.8%
<method `append` of `list` objects>	134217728	7993	4.8%	7993	4.8%
<method `random` of `_random.Random` objects>	134217728	7695	4.6%	7695	4.6%
<built-in method `builtins.len`>	134217742	7332	4.4%	7332	4.4%

> The rest calls are with negligible runtime.

For Multiplication Test

The call graph of multiplication test is shown below.



__init__ &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
_get_module_lock &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
acquire &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
__enter__ &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
cb &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
release &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
<method 'append' of 'list' objects> ×134217728 Total: 8072ms 4.9% Own: 8072ms 4.9%
__init__ &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
_find_and_load &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
__exit__ &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
__exit__ &times;6 Total: 0ms 0.0% Own: 0ms 0.0%
random_matrix_gen &times;2 Total: 76665ms 46.6% Own: 32751ms 19.9%
find_spec &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
find_spec &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
flo_multiplication_test.py &times;1 Total: 164414ms 100.0% Own: 21216ms 12.9%
uniform ×134217728 Total: 35841ms 21.8% Own: 28127ms 17.1%
__enter__ &times;6 Total: 0ms 0.0% Own: 0ms 0.0%
_find_spec &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
_find_and_load_unlocked &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
<method 'random' of '_random.Random' objects> ×134217728 Total: 7713ms 4.7% Own: 7713ms 4.7%
find_spec &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
__getitem__ ×134217728 Total: 66532ms 40.5% Own: 44601ms 27.1%
_new_module &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
<built-in method builtins.isinstance> ×268435462 Total: 14614ms 8.9% Own: 14614ms 8.9%
_path_importer_cache &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
create_module &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
module_from_spec &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
_load_unlocked &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
_get_spec &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
_verbose_message &times;12 Total: 0ms 0.0% Own: 0ms 0.0%
has_location &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
_path_is_mode_type &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
_path_stat &times;6 Total: 0ms 0.0% Own: 0ms 0.0%
_init_module_attrs &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
<built-in method builtins.len> ×134217742 Total: 7316ms 4.4% Own: 7316ms 4.4%
_path_isfile &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
find_spec &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
parent &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
_path_join &times;10 Total: 0ms 0.0% Own: 0ms 0.0%
_path_split &times;4 Total: 0ms 0.0% Own: 0ms 0.0%
cached &times;4 Total: 0ms 0.0% Own: 0ms 0.0%
_get_spec &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
__init__ &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
_relax_case &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
cache_from_source &times;4 Total: 0ms 0.0% Own: 0ms 0.0%
_get_cached &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
<listcomp> &times;10 Total: 0ms 0.0% Own: 0ms 0.0%
spec_from_file_location &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
Matrix &times;1 Total: 0ms 0.0% Own: 0ms 0.0%
__init__ &times;2 Total: 0ms 0.0% Own: 0ms 0.0%
Powered by yFiles

Name	Call Count	Time (ms)	Time percentage	Own Time (ms)	Own time percentage
`flo_addition_test.py`	1	164414	100.0%	21216	12.9%
`random_matrix_gen`	2	76665	46.6%	32751	19.9%
`__getitem__`	134217728	66532	40.5%	44601	27.1%
`uniform`	134217728	35841	21.8%	28127	17.1%
<built-in method `builtins.isinstance`>	268435462	14614	8.9%	14614	8.9%
<method `append` of `list` objects>	134217728	8072	4.9%	8072	4.9%
<method `random` of `_random.Random` objects>	134217728	7713	4.7%	7713	4.7%
<built-in method `builtins.len`>	134217742	7316	4.4%	7316	4.4%

> The rest calls are with negligible runtime.

Analysis for FLO Runtime

In the two tables above, `flo_addition_test.py` and `flo_multiplication.py` is the test script, hence takes the whole runtime. Function `random_matrix_gen` and `uniform` is for generating the random matrices. From the CProfiler statistics, we can see that, the efficiency of matrix elements addition and multiplication is nearly the same, including the detailed statistics for each function calls. The most of the runtime, that is to say, up to is 74.95% in addition and 75.82% in multiplication, setting aside the runtime of matrices generation, is spent on indexing the elements.

```python
def __getitem__(self, item):
    ...
    # for tuple input
    if isinstance(item, tuple):
        # both of the elements in the item are slice
        if isinstance(item[0], slice) and isinstance(item[1], slice):
            ...
            return Matrix(e, r, c)

        # one slice, one int
        # both of the elements in the item are int
        if isinstance(item[0], int) and isinstance(item[1], int):
            ...
            return self.elements[(item[0] - 1) * self.col + item[1] - 1]

    # else for int input
    if isinstance(item, int):
        if item > len(self.elements):
            raise (exceptions.ColumnOutOfBoundsException(item))
        return self.elements[item - 1]
```

Referring to the source code of `__getitem__` method, it is clear that, in order to support different indexing patterns, and to give a hint for users while for out of bounds exception, `isinstance` and `len` functions are called too many times.

Since the process of indexing the elements, or `__getitem__` function dominates the runtime of FLO, the efficiency advantage of floating point addition against multiplication is completely diluted. Thus, seemingly, floating point multiplication is as "fast" as addition, which is the reason why our Strassen's method is always faster than the standard matrix multiplication in spite of the choice of recursion point, even if which is set to be 1.

Conclusion

Performance Bottleneck

The performance of our algorithms is severely restricted by the data structure. More concisely, the performance bottleneck is the process of indexing the elements in matrix.

Balance between Performance and Ease of use

The reason the indexing process, or the `__getitem__` method is inefficient, is the tedious procedure for judging the form of indexing. Such practice offers convenience for user, since more complex indexing operations, such as slice, are supported through it. However, this also caused huge performance waste.

The balance between code performance and user friendliness is of great significance. Indeed, our data structure provides prominent ease of use for developers or users, however, the performance cost is too high for the purpose of this project.

Optimizations

Some optimizations can be made to our code to improve the performance.

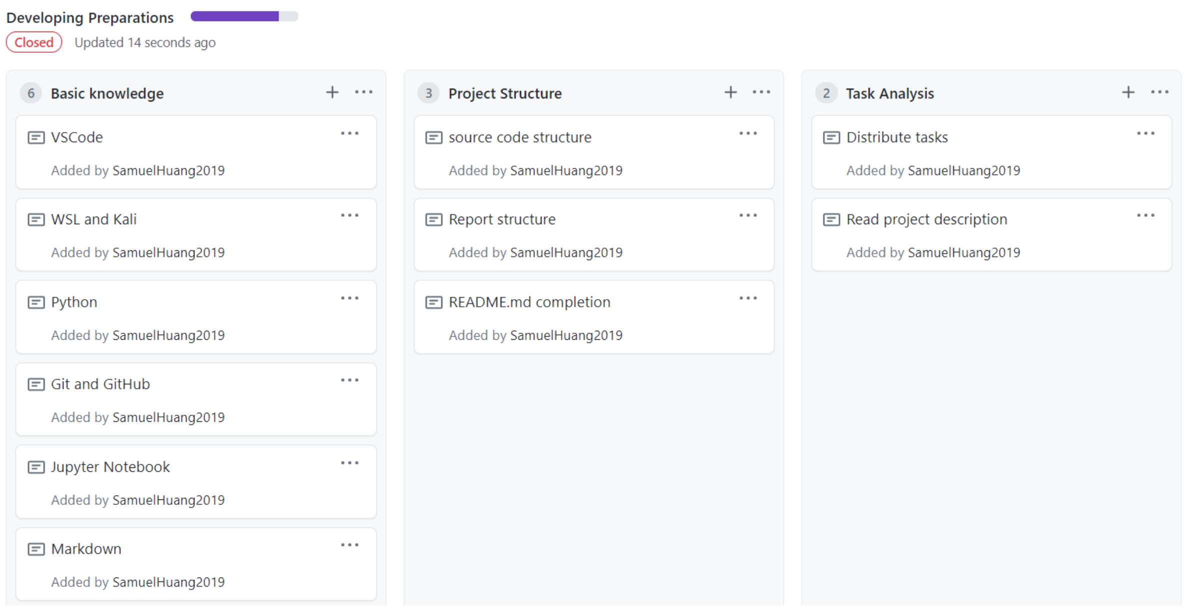
- The indexing of the elements can be less flexible, instead, using more operations in underlying system, in order to provide better performance
- Some judge sentences, such as verifying the dimension of the matrices can be commented out, since this project is only intended for testing the performance of the algorithm itself.
- Multi-thread, parallel run, or even SIMD in GPU acceleration may be introduced, and provides a huge improvement in performance.

Project Management

- HUANG Guanchao, SID 11912309 from SME
  - organizing the project team, work scheduling
  - `Matrix` data structure implementing
  - conducting algorithm test
  - experiment data analysis
  - Empirical Analysis and Conclusion part of report
  - creating PowerPoint slides, and delivered presentation on class
  - final adjust of the report
  - the repository of this project is under his GitHub account SamuelHuang2019
- ZHENG Shuhan, SID 11712401 from PHY
  - reading the paper, and offering theoretical instructions on adaptive Strassen's algorithm for us
  - `Matrix` data structure implementing
  - implementing two methods of multiplication, including function `square_matrix_multiply`, `adaptive_add`, `adaptive_minus` and `strassen_multiply`
  - Code Implementation part of the report
- LI Yuru, SID 11911035 from EIE
  - Introduction and Theoretical Analysis part of the report
- -TIAN Yuqiong, SID 11911039 from EIE
  Background and Theoretical Analysis part of the report

> SPECIAL THANKS TO:
>
> - SUN Jiachen, who provided testing platform for us.
> - HAN Zichen, who offered us professional advice on Python and other CS problems.

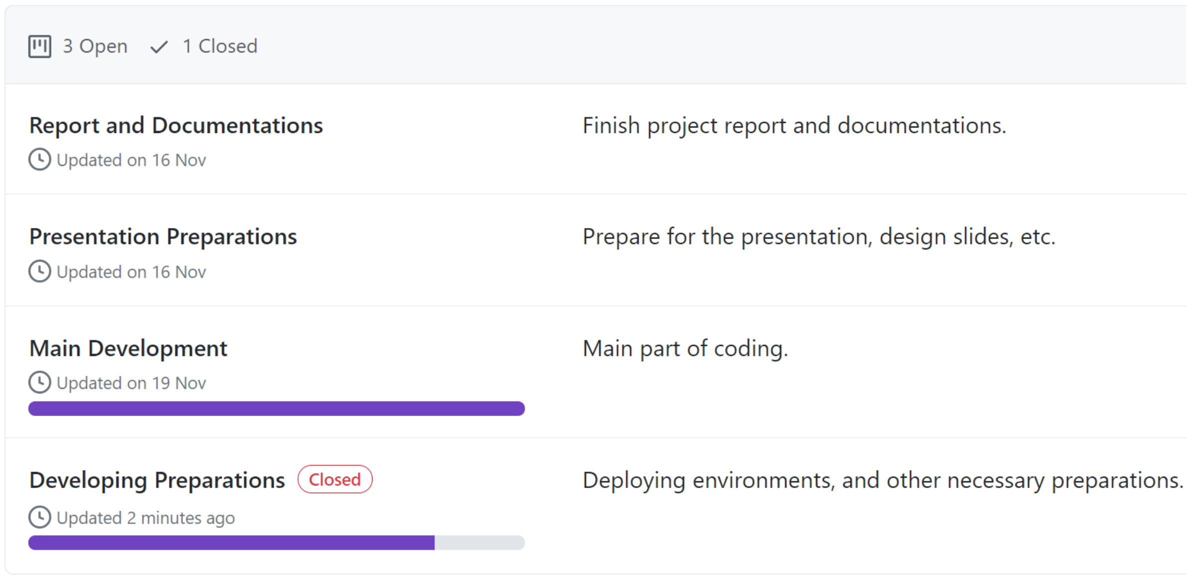
Our project is based on GitHub for version control and code cooperation, on which we created projects for different stages of our work.



Developing Preparations

Closed Updated 14 seconds ago

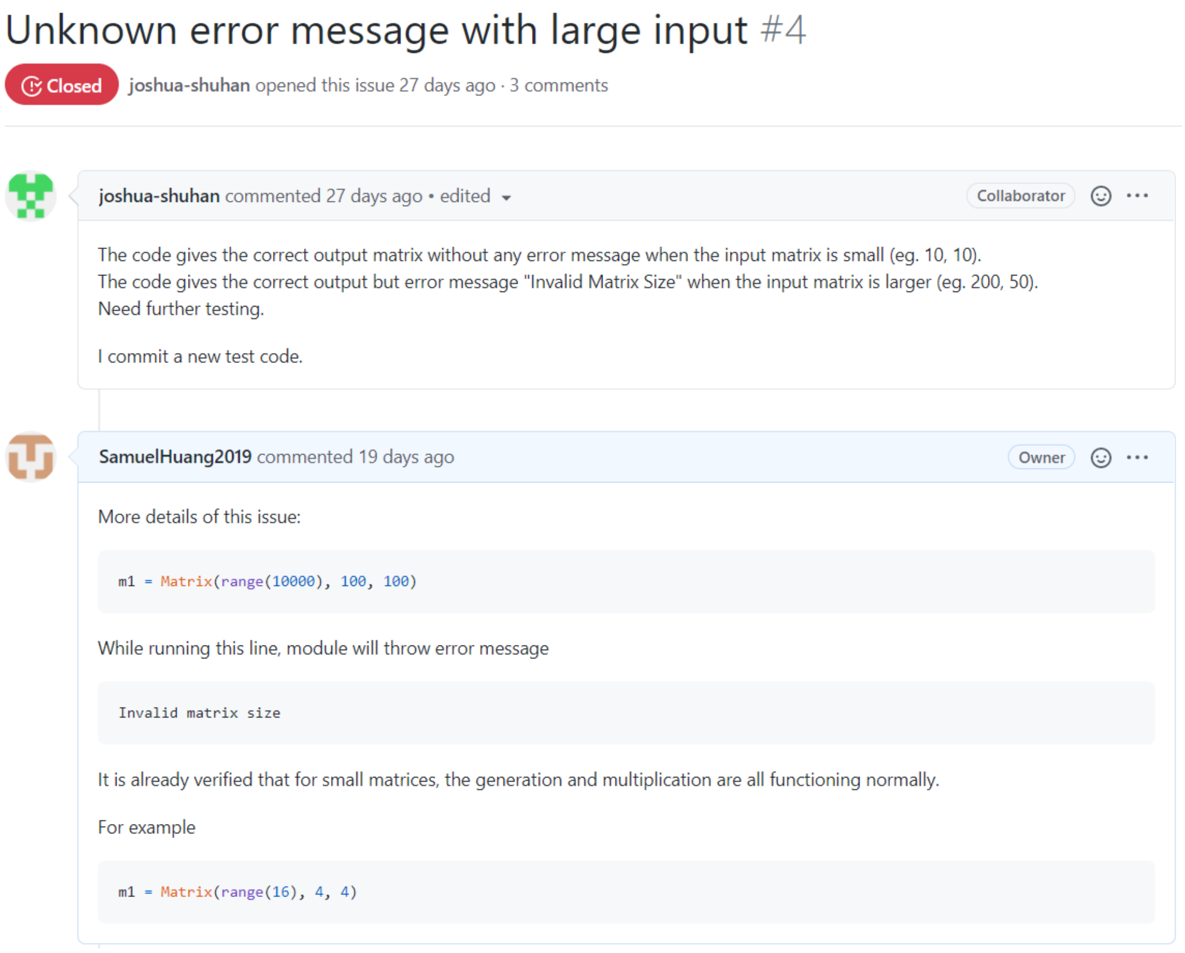
6 Basic knowledge	3 Project Structure	2 Task Analysis
VSCode Added by SamuelHuang2019	source code structure Added by SamuelHuang2019	Distribute tasks Added by SamuelHuang2019
WSL and Kali Added by SamuelHuang2019	Report structure Added by SamuelHuang2019	Read project description Added by SamuelHuang2019
Python Added by SamuelHuang2019	README.md completion Added by SamuelHuang2019	
Git and GitHub Added by SamuelHuang2019		
Jupyter Notebook Added by SamuelHuang2019		
Markdown Added by SamuelHuang2019		



3 Open ✓ 1 Closed

Report and Documentations Updated on 16 Nov	Finish project report and documentations.
Presentation Preparations Updated on 16 Nov	Prepare for the presentation, design slides, etc.
Main Development Updated on 19 Nov	Main part of coding.
Developing Preparations Closed Updated 2 minutes ago	Deploying environments, and other necessary preparations.

Also, we opened some issues while code debugging.



Unknown error message with large input #4

Closed joshua-shuhan opened this issue 27 days ago · 3 comments

joshua-shuhan commented 27 days ago • edited Collaborator

The code gives the correct output matrix without any error message when the input matrix is small (eg. 10, 10).
The code gives the correct output but error message "Invalid Matrix Size" when the input matrix is larger (eg. 200, 50).
Need further testing.

I commit a new test code.

SamuelHuang2019 commented 19 days ago Owner

More details of this issue:

```
m1 = Matrix(range(10000), 100, 100)
```

While running this line, module will throw error message

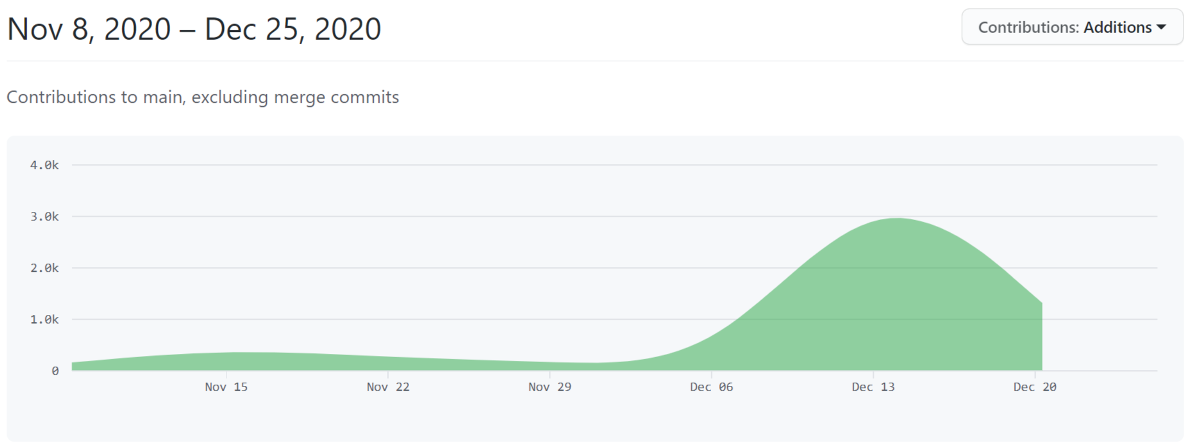
```
Invalid matrix size
```

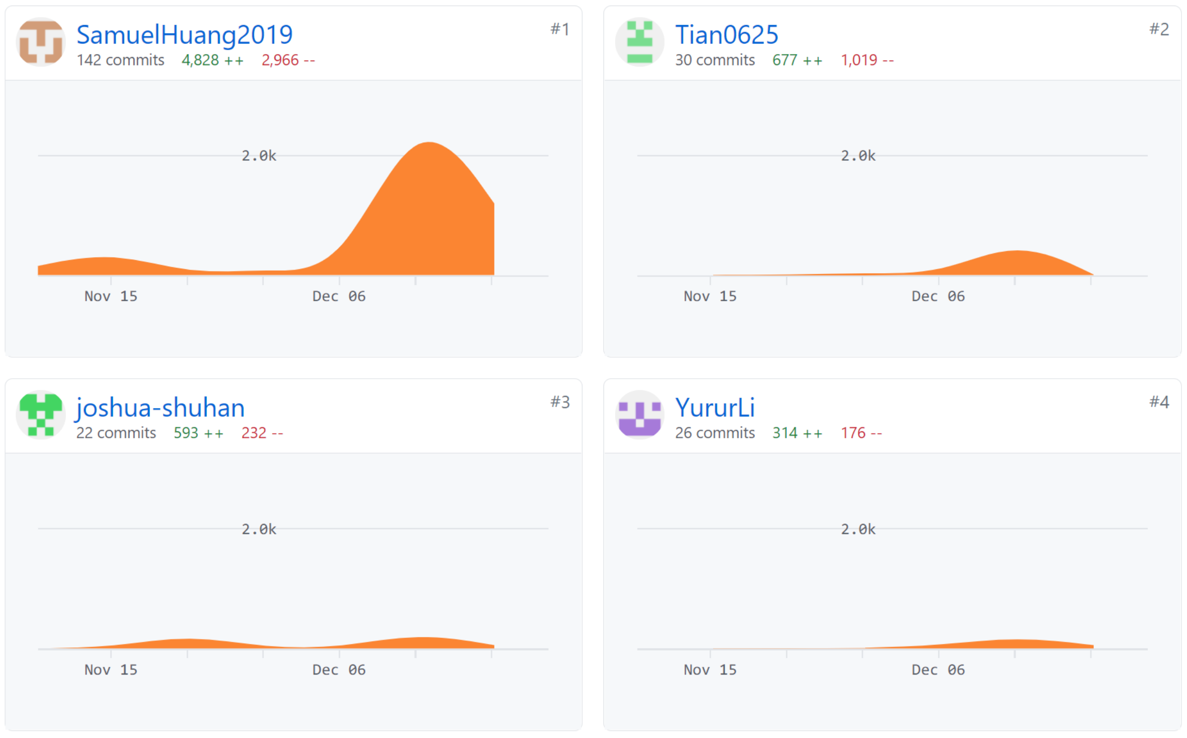
It is already verified that for small matrices, the generation and multiplication are all functioning normally.

For example

```
m1 = Matrix(range(16), 4, 4)
```

Until December 25th, 2020, the contributions insight is as shown below.





Open Source License

Reference

1. Pan (1984). How to Multiply Matrices Faster. Retrieved December 25, 2020, from SEMANTIC SCHOLAR. DOI:10.1007/3-540-13866-8 ↩

2. Stothers, A. J. (2010). On the Complexity of Matrix Multiplication. Retrieved December 25, 2020, from Edinburgh Research Archive. Web site: https://era.ed.ac.uk/handle/1842/4734. ↩

3. Francois Le Gall (2014). Powers of Tensors and Fast Matrix Multiplication. Retrieved December 25, 2020, from Cornell University. Web site: https://arxiv.org/abs/1401.7714D ↩

4. Williams, V. V (2014). Multiplying matrices in $O(n^{2.373})$ time. Retrieved December 25, 2020, from Stanford University. Web site: https://people.csail.mit.edu/virgi/matrixmult-f.pdf ↩

5. Paolo D'Alberto and Alexandru Nicolau. 2007. Adaptive Strassen's matrix multiplication. In Proceedings of the 21st annual international conference on Supercomputing (ICS '07). Association for Computing Machinery, New York, NY, USA, 284–292. DOI:https://doi.org/10.1145/1274971.1275010 ↩