B503 - Analysis of Algorithms

Project: Fast Matrix Multiplication Final Report

ABU SALEH MD NOMAN
amdnoman@indiana.edu

Introduction

The Naïve matrix multiplication algorithm uses running time of $O(n^3)$ (infact it is m*n*p if the matrices in question are m*n and n*p in size respectively). In three loops it uses 8 multiplications and 4 additions to multiply 2*2 square matrices. Strassen devised an algorithm which requires 7 multiplications and 18 additions which was improved by Winograd and reduced to 7 and 15 respectively. Reducing the number of multiplication can come handy since it requires $O(n^3)$ time compared to $O(n^2)$ required for addition. Strassen's algorithm divides the initial matrix into smaller submatrices and recursively applies the addition and multiplications until it reaches the base case. Since we are interested in submatrices instead of individual numbers this is a huge improvement in terms of running time and CPU memory. The basic idea is, in each step of the recursion we break down a $n * n$ matrix to four submatrices of size $\frac{n}{2} * \frac{n}{2}$ and continue to do so until the base case is reached. In the base case we return the result using Naïve algorithm.

In general, the running time for Strassen's algorithms is $O(n^{\log_2 7})$. Strassen's algorithm achieves its lower complexity using a divide-and-conquer approach. However there are couple of trade-offs to achieve this improvement. First, if the division goes down to single element in base case, the recursion overhead becomes significant and reduces performance. This overhead can be overcome by stopping the recursion early using suitable cut-off point and performing a conventional matrix multiplication on submatrice. Second, the division step must efficiently handle odd-sized and non-square matrices because the classical Strassen's algorithm assumes both the input matrices to be of size $n * n$ where n is an even number. Generalized matrices can be handled with some tweaking which involve static or dynamic padding of zeroes to the original matrices to make them even and square.

Later in this report I tried to analyze the algorithms in brief and illustrate the experimental results with the varying cut-off points and matrix sizes.

Algorithm

As discussed earlier, the classical and naïve algorithms are used in my implementation. The classical cubic time algorithm is as follows:

Input: *An* $n * p$ *matrix* A *and a* $p * m$ *matrix* B

Output: $n * m$ *matrix* C *such that* $C = AB$

For $i = 1$ *to* n *do*

  For $j = 1$ *to* m *do*

    Set $c_{ij} = 0$

      For $k = 1$ *to* p *do*

        Set $c_{ij} = c_{ij} + a_{ik}b_{kj}$

      End for

  End for

End for

Whereas the Strassen's algorithm can be viewed as:

Input: *Two* $n * n$ *matrices* A *and* B

Output: $n * n$ *matrix* C *such that* $C = AB$

C *can be decomposed into quadrants as follows*:

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

The four quadrant matrices are computed as follows:

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

$$P_1 = (A_{11} + A_{22}) \bullet (B_{11} + B_{22})$$
$$P_2 = (A_{21} + A_{22}) \bullet B_{11}$$
$$P_3 = A_{11} \bullet (B_{12} - B_{22})$$
$$P_4 = A_{22} \bullet (B_{21} - B_{11})$$
$$P_5 = (A_{11} + A_{12}) \bullet B_{22}$$
$$P_6 = (A_{21} - A_{11}) \bullet (B_{11} + B_{12})$$
$$P_7 = (A_{12} - A_{22}) \bullet (B_{21} + B_{22})$$

The recursive definition of divide and conquer approach suggests:

$$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2), \textit{since there are } 8 \textit{ submatrices of size } \frac{n}{2} * \frac{n}{2}$$

However, this results into $T(n) = O(n^3)$ which is no improvement. But in Strassen's algorithm it uses 7 multiplications and therefore the definitions revises:

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$$

Where $T(n) = O(n^{log7})$, although it does not seem much a difference but it is handy because of the exponent.

Managing Issues and fixes

i. Recursion cut-off point

The seven products showed in the algorithm can be computed by recursively calling Strassen's algorithm on submatrices of half dimension, and at some point(called the *recursion truncation point)* restoring to naïve algorithm where the splitting no longer is profitable. If one were to estimate running time by counting arithmetic operations, the recursion truncation point would be around 16~24. However, if the size and nature of matrices are varied this cut-off point can fluctuate. So, it is difficult to set one truncation point which facilitates all size of matrices. It is observed that for square matrices input, the truncation point works well around 2^k, where k=4~7. I have used 2^7 in my experiment.

ii. Handling Generalized Matrices

There are couple of ways to address this problem:

- Pad out A and B by adding zeros to the bottom rows and right-most columns, to get square matrices of size $= 2^k$. Then run Strassen's algorithm on the larger arrays; then pull out the answer C from the large result matrix. **(static padding).** In my implementation, I have used this approach.
- For square matrices: At each recursive stage: if n is even, split the arrays into quarters in the usual way and recur. If n is odd, make it even by adding an extra row and column of zeros. Then split into same-sized quarters and recur. Strip off the row/column from the result C before returning. **(dynamic padding)**

Experimental Result

i. Setup

In the experiment, two input matrices of varying sizes were taken. First, they were randomly initialized using integer values within -16 to 16. Then from *test.cpp* the *Product()* function was called. Inside the *Product.cpp* there are two implementation of functions *Product_basic()* and *Prouct().* In the header *product.h* their definitions are given as:

```
void Product_basic(int *A, int Am, int An, int *B, int Bn,  int *C);
void Product(int *A, int Am, int An, int *B, int Bn,  int *C);
```

Inside the product function, there is logic to decide whether to call the naïve function or the strassen function. The basic idea is:

1. when Am, An, and Bn are all small (465 in the experiment), surely naïve algorithm will be the most rapid routine.
2. when Am, An, and Bn are all large and close to equal, surely Strassen's Algorithm will be most rapid.
3. the time for naïve algorithm is proportional to the product Am*An*Bn. This increases at a cubic rate so it is not the fastest algorithm when all the parameters are large. On the other hand, it increases only linearly with any single parameter, so it is likely to be the fastest when only one of the parameters is large.

After the resultant matrix is obtained, the running times are calculated and compared.
All the experiments were performed on a machine with a 2.5GHz Intel Core i7 CPU, 16GB of main memory and Microsoft Windows 8.1 as operating system. All techniques were implemented in C++.

ii. Performance Evaluation

Cut-off point=16

Type of Matrix	Am	An	Bn	Strassen Algorithm Runtime(sec)	Naive Algorithm Runtime(sec)
Square	100	100	100	.08	.01
	200	200	200	.04	.52
	300	300	300	3.71	.17
Rectangular	100	50	200	.51	.01
	200	10	300	2.85	.00

Cut-off point=32

Type of Matrix	Am	An	Bn	Strassen Algorithm Runtime(sec)	Naive Algorithm Runtime(sec)
Square	100	100	100	.03	.00
	200	200	200	.15	.03
	300	300	300	1.04	.12
Rectangular	100	50	200	.15	.00
	200	10	300	1.03	.00

Cut-off point=128

Type of Matrix	Am	An	Bn	Strassen Algorithm Runtime(sec)	Naive Algorithm Runtime(sec)
Square	100	100	100	.01	.00
	200	200	200	.08	.03
	300	300	300	.51	.12
Rectangular	100	50	200	.07	.00
	200	10	300	.51	.00
	10000	1	1000	1.96	.08
	3000	200	3000	17.01	23.06
	100	10000	100	1.03	1.07
	1000	1000	1000	5.17	9.58

The following figure shows for square matrices of smaller size, the naïve algorithm always produces better result.

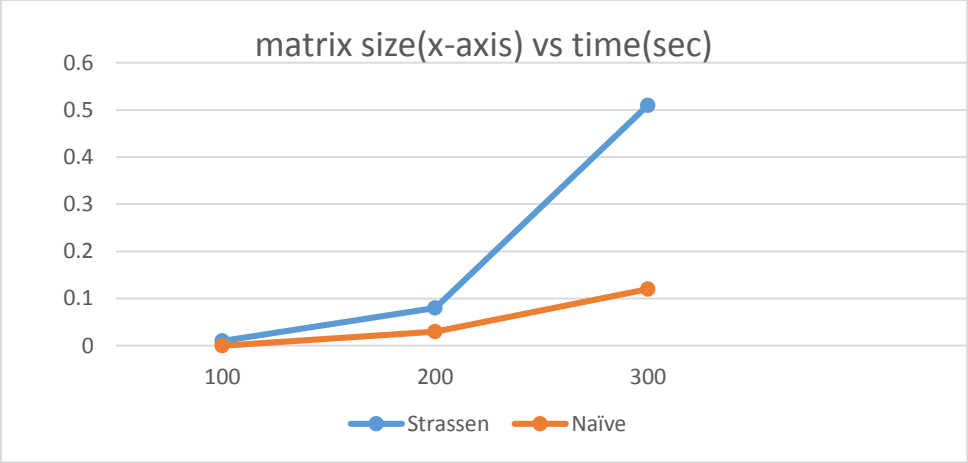


Figure 1for cut-off point 128

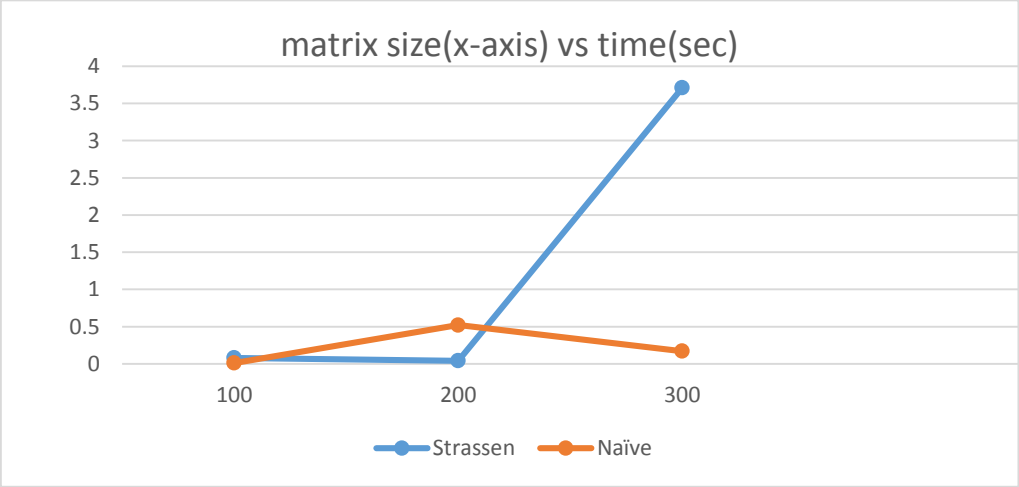
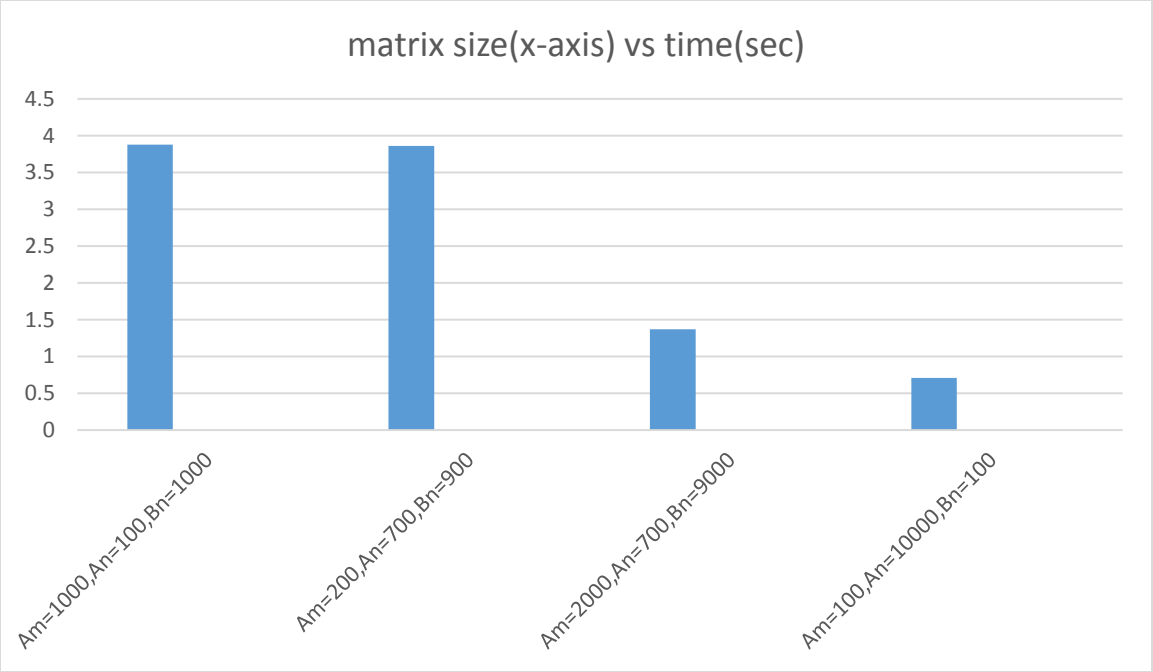


Figure 2for cutoff point 16

The same is also reflected for cut-off point 16 in Figure 2. However, the running time is worse compared to previous one, which is expected.

The running times for rectangular matrices are somewhat misleading. Following is the result depicted in bar chart. The running times are outcome of combination of naïve and strassen algorithm depending on the input matrice sizes and the corresponding logic implemented in the code.



Conclusion

Matrix multiplication is an important algorithm and usage is widespread. Strassen's algorithm for matrix multiplication achieves lower arithmetic complexity than the conventional algorithm. It is somewhat difficult to implement this algorithm since it uses divide and conquer strategy and therefore the issues of rectangular size matrices and a suitable cut-off point needed to be addressed. In this implementation, the larger square matrices perform better using Strassen algorithm. However, for rectangular matrices with padding the result is not satisfactory and there is much scope of improvement.

Reference

1. **Textbook**
2. **Codes provided by instructor**
3. **http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-3-divide-and-conquer-strassen-fibonacci-polynomial-multiplication/lec3.pdf**
4. **http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.47.5949&rep=rep1&type=pdf**
5. **http://src.acm.org/loos/loos.html**

6. http://www.answers.com/Q/What_is_the_source_code_for_Strassen's_matrix_multiplication_program
7. http://www.codeyourproject.com/ada/strassen-multiplication-program-order-n-n-matrix-c
8. http://www.dreamincode.net/forums/topic/116381-strassens-multiplication/
9. http://homes.soic.indiana.edu/classes/spring2015/csci/b503-pwp/index.html/project.html
10. http://www.cs.amherst.edu/ccm/cs20/divconproj.pdf