Strassen's Algorithm’s Potential for Large Matrix Multiplication

Evan Delanty
CPSC 450
Fall 2024

Summary
The problem I focused my research and implementation on is computing matrix multiplication for nxn matrices in under cubic time. I implemented the classic naive algorithm for matrix multiplication, which utilizes three nested loops to perform n^3 scalar multiplications. To compare to this baseline, I implemented Strassen's algorithm, a divide-and-conquer approach which ends up performing in $O(n^{log7})$ time, being just below cubic time at $O(n^{2.81})$ time. I used Java as my programming language, and decided to create a custom Matrix class which stores the data utilizing a flattened-one-dimensional array. To compare the two algorithms, I created a test which generates incrementally larger matrices from $2x2$ to $2048x2048$, comparing the time taken to perform matrix multiplication in milliseconds (ms). Unexpectedly, Strassen’algorithm performed dramatically worse compared to the naive approach, which is odd considering the faster complexity. The solution is to perform a slight modification to the recursive stop and to utilize the naive approach as a substitute for the dot product traditionally performed in Strassen’s algorithm [3].

1. ALGORITHM SELECTED
Matrix multiplication in Linear Algebra is generally introduced as an early and simple topic alongside addition and subtraction, but its use-cases extend much further in computer science. Calculating the multiplication for large matrices with a size n greater than multiple thousands can be time consuming for the cubic approach, which is why Strassen's algorithm, created by Volker Strassen and initially published in 1969 [2], comes into importance when improving computing time. Although I didn’t use Strassen’s paper much for this project, I wanted to focus on the applicable approach to Strassen's algorithm utilizing the pseudocode/directions and documentation from [1]. Note that pseudocode for Strassen’s algorithm wasn’t directly given, but the directions provided can be translated to what I wrote below.

```
Algorithm: Strassens
Input: Two $n \ x \ n$ matrices A and B where $n$ is of a power of 2.
Result: Resulting Matrix C
if $n = 1$ then
    $C = A_{11} \cdot B_{11}$
    return C
// partition A and B into $n/2 \ x \ n/2$ submatrices
$A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, B_{12}, B_{21}, B_{22}$, respectively,
// now create additional $n/2 \ x \ n/2$ matrices $S_1, S_2..., S_{10}$
$S_1 = B_{12} - B_{22}$,
$S_2 = A_{11} + A_{12}$,
$S_3 = A_{21} + A_{22}$,
$S_4 = B_{21} - B_{11}$,
$S_5 = A_{11} + A_{22}$,
$S_6 = B_{11} + B_{22}$,
$S_7 = A_{12} - A_{22}$,
$S_8 = B_{21} + B_{22}$,
$S_9 = A_{11} - A_{21}$,
$S_{10} = B_{11} + B_{12}$.
// now create additional $n/2 \ x \ n/2$ matrices $P_1, P_2..., P_7$
$P_1 = Strassens(A_{11}, S_1)$,
$P_2 = Strassens(S_2, B_{22})$,
$P_3 = Strassens(S_3, B_{11})$,
$P_4 = Strassens(A_{22}, S_4)$,
$P_5 = Strassens(S_5, S_6)$,
$P_6 = Strassens(S_7, S_8)$,
$P_7 = Strassens(S_9, S_{10})$.
// now create and update the submatrices of C
$C_{11} = P_5 + P_4 - P_2 + P_6, C_{12} = P_1 + P_2$,
$C_{21} = P_3 + P_4, C_{22} = P_5 + P_1 - P_3 - P_7$.
return C
```

As explained in full depth in [1], the time complexity of Strassen’s algorithm can be broken down into a simple recurrence relation. Starting from the base case, if $n = 1$, you can simply perform a single dot product and return the resulting matrix *C*, which takes $O(1)$ time. If $n > 1$, you must partition the input matrices, *A* and *B*, and the output matrix, *C*, into $n/2 \ x \ n/2$ submatrices which takes $O(1)$ time. Now you must create additional $n/2 \ x \ n/2$ matrices, $S_1, S_2..., S_{10}$, as a combination of additions and subtractions from the previously created submatrices. Alongside these newly created matrices, you must initialize new matrices, $P_1, P_2..., P_7$, where all 17 matrices can be computed and initialized in $O(n^2)$ time. Now as a combination of products from the original submatrices and S_1, S_2..., S_{10}, you can compute the values for P_1, P_2..., P_7, which takes $7T(n/2)$ time [1]. Finally, you can create and update the four result submatrices, C, using different additions and subtractions of the previously made P_1, P_2 ..., P_7 matrices, taking $O(n^2)$ time. Using this information, you can setup the recurrence relation $T(n) = 7T(n/2) + O(n^2)$. Utilizing the divide-and-conquer master method approach learned in class, you can solve for the time complexity, shown below.

Given $a = 7,\ b = 2,\ d = 2$, so, $b^d = 4$.

Therefore, $a > b^d = 7 > 2^2$, so, $O(n^{log_b a}) = O(n^{log_2 7})$.

We are given the final time complexity of $O(n^{log7})$.

2. IMPLEMENTATION

For my implementation I used Java as my primary programming language. In order to represent a matrix, I created a custom class called Matrix, which utilized a flattened one-dimensional array to mimic a two-dimensional array, alongside primitive integers to save the rows and columns of a created Matrix object. I used the type "double" to save the data entries for the matrices, as there is no reason they should be limited to integers. I created a few member functions to return the count of rows and columns, and provided the methods to perform matrix addition and subtraction with two Matrix objects. Of course, you may use set(int row, int col, double data) and get(int row, int col) to set data to a certain row and column, and then retrieve that data. I used no other pre-developed libraries or data structures for my implementation of either the naive approach or Strassen's algorithm.

In terms of the implementation of Strassen's algorithm, I followed the steps listed in [1] in detail, and even listed the steps from the book as comments in my implementation. My code deviated in a few ways, as the book didn't give exact pseudocode, and these deviations were in the form of the many helper functions that were not provided to me and a modification of the base case. The main methods I developed were createSubMatrix(), alongside the matrix addition and subtraction methods mentioned above. Additionally, I was forced to create a modified base case, where, at a certain point, Strassens would simply have to use the naive approach to matrix multiplication to speed up the process for the smaller submatrices generated. I was practically forced to implement it this way, as on my machine it was impossible to get Strassens to run better than the naive approach without this modification [3]. As for the other two, they were just as simple as iterating through each row and column and adding (or subtracting) the respective data and returning the result.

During my implementation, I struggled quite a bit with all the parts of the pseudocode that were only glanced over at a high level. For example, creating submatrices and splitting up C to join later were tricky to implement. Ensuring these steps were implemented without impacting the algorithm's runtime was challenging, but with some focused problem-solving and iterative coding, I was able to make it work effectively, especially with some unit testing.

To prove the correctness of both my implementations, I used a few strategies. First, I made sure that upon an invalid input (e.g., a matrix not $n\ x\ n$ or n not being a power of 2) the test should assert an IllegalArgumentException throw. Second, I wanted to cover a minimum input edge case where $n\ is$ the size *1*. Finally, I simply wanted to confirm correctness on a small matrix where, n, is size *2,* and assert the expected result of matrix multiplication. Additionally, I created a few small unit tests for the helper methods I created, generally following the same strategies used for the naive and Strassen's algorithms. I made sure to test whether or not a matrix, A, when multiplied by the identity matrix, I, returned A. I also provided a test a large n of the size *512* to make sure Strassen's algorithm didn't break for a large n.

3. PERFORMANCE TESTS

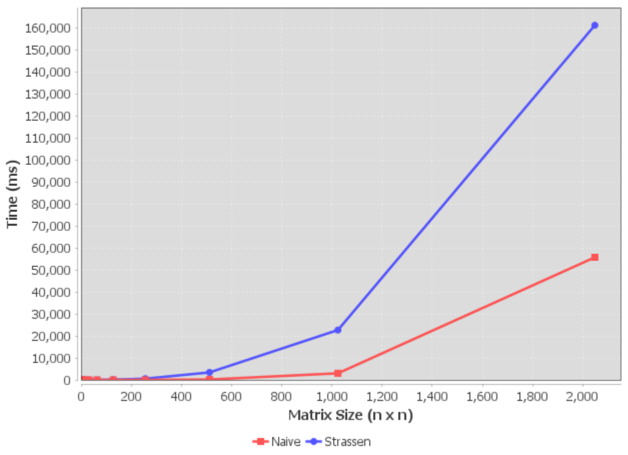
For performance testing, it's important to measure the potential for performance on multiplying matrices of the same size n with random data in each entry of input matrices A and B. I limited the random values to be in the range *0.0-100.0*, as I didn't want too much variance in each data entry, so I wouldn't accidentally create an additional variable to watch out for. The test starts with n being *2*, then increments by a step of $n * 2$ once the multiplication is computed and the resulting matrix C is returned. This continues until n is size *2048*, which marks the end of the plot.

I ran this test for both implementations, each time adjusting the recursive stopping condition, referred to in my code as STRASSENS_LIMIT. This value, represented as n in the captions of plots 1–4, determines the base case threshold for switching to the naive approach, which is run instead of the dot product for matrices sized $n = 1$, which is the traditional base case. I evaluated the performance across different recursive stopping points and analyzed the resulting speeds in the next section.

These tests were run on an Intel i5-10600k using Windows 11 Home utilizing Maven as my way to compile and run the Java code. To build and run these tests, you must install Maven and configure the PATH to be able to utilize it. Run the command "mvn compile" in your command line, then "mvn exec:java" to run my performance tests. Additionally, you may run "mvn test" to run unit tests for my various methods.

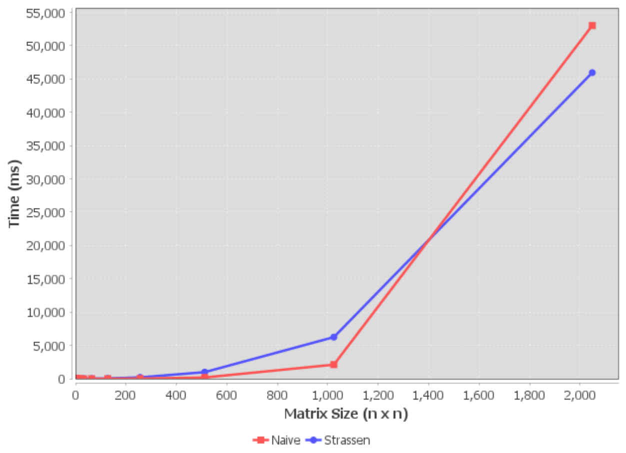
4. EVALUATION RESULTS

Plot 1. Performance Test With Traditional Strassens



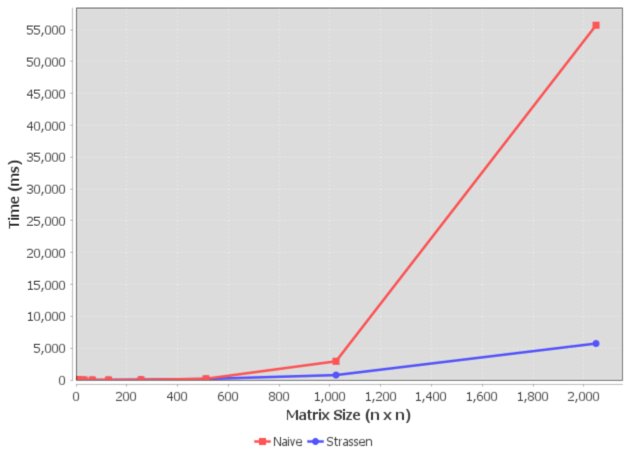
It's clear just from plot 1 that the traditional implementation of Strassen's on my machine runs 189.4% slower than the naive algorithm for size n of $2048x2048$, but as you will see from the next results, changes quickly with a modified base case and use of the naive algorithm in Strassen's.

Plot 2. Performance Test With n = 2



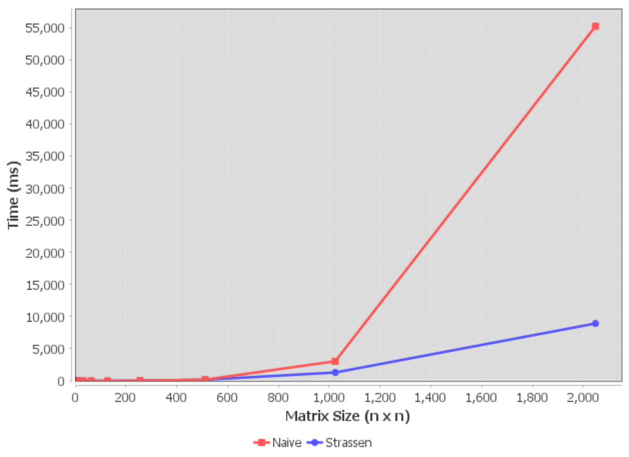
If we look at the timings from Table 1 and 2 below, you can see that for sizes $2x2$ - $1024x1024$ for $n = 2$ Strassen's algorithm performs worse than the naive approach, but presumably for any size larger (if padding were to be used for sizes not of the power of 2) and clearly at $2048x2048$ Strassen's algorithm begins to outperform the naive approach as expected.

Plot 3. Performance Test With n = 128



If we increase the recursion stop and begin to use the naive algorithm for submatrices smaller than size *128*, you can see from this plot above and the tables below that Strassen's performance is faster or on par even for smaller matrices with size $n < 512$, which was definitely not the case previously. This improvement is likely due to the reduced overhead from fewer recursive calls, which balances out the cost of switching to the naive algorithm at an earlier stage. This adjustment allows Strassen's algorithm to capitalize on its efficiency in dividing the problem while mitigating the diminishing returns of recursion and memory allocation for small submatrices. As a result, it strikes a more effective balance between the overhead of Strassen's recursive structure and the computational simplicity of the naive approach.

Plot 4. Performance Test With n = 512



Interestingly, you can see how a higher base case starts to have drawbacks at a certain size. For my machine, you can see in the plot and in the tables below that there is worse performance for a base case of $n <= 512$ than there is for $n <= 128$, this makes a lot of sense since we begin to use the slower $O(n^3)$ algorithm for matrices that would have benefited from Strassen's $O(n^{2.81})$ time complexity. The increased base case threshold reduces the number of recursive calls, which is beneficial up to a point. However, as we've seen, when the threshold becomes too high the algorithm shifts too much of the load to the slower naive approach. This clearly starts to erode the benefits from Strassen's algorithm, which was previously utilized with the limit being $n <= 128$.

This demonstrates the importance of finding an optimal balance for the base case threshold, as it directly influences how effectively Strassen's algorithm can leverage its asymptotic advantages. Different machines might cause this balance point to differ, emphasizing the need for tuning the recursive stop defined in my code for the specific hardware used.

Table 1. Timings (milliseconds) For Different Base Cases

Naive Approach

Matrix Size	**n = 1**	**n = 2**	**n = 128**	**n = 512**
$2x2$	0	0	0	0
$4x4$	0	0	0	0
$8x8$	0	0	0	0
$16x16$	0	0	0	0
$32x32$	1	1	1	2
$64x64$	6	4	5	5
$128x128$	3	2	4	6
$256x256$	22	24	22	54
$512x512$	178	183	185	192
$1024x1024$	3,003	2,110	2,927	3,068
$2048x2048$	55,697	53,033	55,690	55,228

Table 2. Timings (milliseconds) For Different Base Cases Strassen's Algorithm

Matrix Size	n = 1	n = 2	n = 128	n = 512
$2x2$	0	0	0	0
$4x4$	0	0	0	0
$8x8$	0	0	0	0
$16x16$	1	1	0	1
$32x32$	6	2	2	2
$64x64$	22	12	5	5
$128x128$	109	36	3	4
$256x256$	557	199	47	31
$512x512$	3,408	985	157	182
$1024x1024$	22,669	6,246	761	1,308
$2048x2048$	161,144	45,963	5,715	8,943

5. REFLECTION

This project focused on implementing and evaluating Strassen's algorithm for matrix multiplication, alongside the naive approach, to understand their relative performances and the impact of different optimizations. Strassen's algorithm is particularly interesting because it fundamentally challenges the traditional $O(n^3)$ complexity of matrix multiplication, reducing it to the just barely better asymptotically $O(n^{2.81})$. I thought that this aligned very well with our discussions and experience with how sometimes the theoretical time complexity is very dependent on the implementation at the end of the day.

I think the main takeaway from this project and a significant portion of the course is that creating an algorithm is, as expected, not obvious and extremely difficult. When I was reading [1] and [2] an important thought that was going through my head was how anybody could figure something like Strassen's algorithm out, and I don't think I'll have that answer for a while. It was stressful, and intriguing to find that Strassen's was outperformed by the naive algorithm with the original base case and dot product computation, and researching and trying to find out why that was the case was the best part of the project. It was challenging to be successful in my research and understanding, but I found that those challenges helped me learn.

If I had more time, I definitely would have made it so my implementation of Strassen's works on matrices that are nxn, but where n is not of a power of *2*. This would have given me more flexibility in my testing, and unfortunately I found the solution on how to do this a bit too late into the project to fully understand and implement. I hinted at this in my evaluation results, and from what I currently know you simply have to pad the matrix with 0's and then un-pad it before you return the resulting matrix C. In theory, this wouldn't be very hard, but the extra steps and documentation would have taken too long. I would definitely be interested in continuing this later though, as I'm curious to see where exactly Strassen's begins outperforming the naive approach for each base case modification, which would be where their plots intersect.

6. RESOURCES

[1] *Introduction to Algorithms*:

I based my entire pseudocode and implementation purely on this book. I found that it was much easier to understand why Strassen's algorithm works, and additionally helped me break down the steps of the algorithm found in Volker Strassen's paper which seemed very confusing on my first read, where the notation, in comparison, is very different and oddly complex. I also used this source to break down the time complexity, which it provided in full detail and explanation, but it was missing the work for the Master Method which I added in this paper. Although the pseudocode for the algorithm was not provided directly (it was actually a practice question for the section) the authors broke down the process of Strassen's algorithm into a condensed four step process that I was able to derive the implementation and write pseudocode from.

[2] *Gaussian elimination is not optimal:*

I didn't directly reference Strassen's original paper while implementing his algorithm, but I did read through it for the sake of understanding, and it was interesting to compare his notation to the one used in the modern book. It was much more mathematically heavy, and mentioned quite a few Linear Algebra topics which gave insight into how the algorithm was initially discovered.

[3] *A Modified Strassen Algorithm to Accelerate Numpy Large Matrix Multiplication:*

I found this research paper while trying to find out why my original implementation was resulting in slower times compared to the naive algorithm. It ended up being the solution to my problem, as there was mention of how at a certain point (the author called it recursiveCutOff) if the base case becomes $n <= recursiveCutOff$ you can return the "matMul" of A and B, which, in Python, is just traditional matrix multiplication. Once I modified my version to repeat this, I found the results you saw in my evaluation, where Strassen's becomes much more practical and quick compared to the naive approach under certain "recursiveCutOffs".

It also gave me insight on how I could have modified Strassen's algorithm further to work on matrices with a size n not to the power of *2*. I mention this in detail in my reflection regarding padding the matrix with *0*'s until its size n is of a power of *2*.

7. REFERENCES

[1] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction to Algorithms (4th. ed.). MIT Press, Cambridge, Massachusetts, 1990.

[2] V. Strassen. Gaussian elimination is not optimal. Numer. Math., 13:354-356, 196

[3] T. Breitzman. A Modified Strassen Algorithm to Accelerate Numpy Large Matrix Multiplication with Integer Entries. Gerudo Press, 2023.