CS 124 Programming Assignment 2 Writeup

30939506 and 20907541

March 24, 2017

Abstract

Our goal for this assignment is to implement a modified version of Strassen's algorithm that switches to conventional matrix multiplication after certain threshold. We analytically determined the estimated cross-over point to be $n_0 = 16$ for matrices that are powers of 2. For odd matrices, we determined that it's $n_0 = 37$. After significant tests, we saw that depending on the size of the matrix, there is a range of the best cross-over point. We discovered that the threshold tends to hover around 90 - 110 for higher dimensions.

Algorithm Overview

For the first part of the tasks, we had to compute our estimated n_0, the cross-over point, between Strassen's algorithm and the Standard matrix multiplication. From lecture 8, Strassen's algorithm takes 7 multiplications of matrices with $\frac{n}{2}$ size, and there are 10 additions and subtractions in order to compute P_1 to P_7. Also, there are 8 additions and subtractions in order to find the appropriate terms (AE + BG, AF + BH, etc); altogether there are 18 subtractions and additions for the Strassen's algorithm. We then have the following recurrence:

$$\begin{aligned} S(n) &= 7S(\frac{n}{2}) + 18(\frac{n}{2})^2 \\ &= 7S(\frac{n}{2}) + \frac{9n^2}{2} \end{aligned} \tag{1}$$

The work that is required for the Standard matrix multiplication algorithm is n^3 multiplications. It takes $n-1$ additions for each element and since there are n^2 elements, we have $n^2(n-1)$ additions. The recurrence equation for the standard algorithm is $T(n) = n^3 + n^2(n-1) = 2n^3 - n^2$. Next, we figure out the cross-point n_0 by combining the two algorithms, essentially solving for $S(n_0) = T(n_0)$. We can write:

$$S(n_0) = 7min(S(\frac{n_0}{2}), T(\frac{n_0}{2})) + \frac{9{n_0}^2}{2}$$
$$min(S(\frac{n_0}{2}), T(\frac{n_0}{2})) = T(\frac{n_0}{2})$$

The reason why $min(S(\frac{n_0}{2}), T(\frac{n_0}{2})) = T(\frac{n_0}{2})$ is because we are trying to find the n_0 where Strassen will be worse than Standard, and switch to Standard. With this we can write:

$$
\begin{aligned}
T(n_0) &= S(n_0) \\
T(n_0) &= 7T(\frac{n_0}{2}) + \frac{9{n_0}^2}{2} \\
2{n_0}^3 - {n_0}^2 &= 7(2(\frac{n_0}{2})^3 - (\frac{n_0}{2})^2) + \frac{9{n_0}^2}{2} \\
2{n_0}^3 - {n_0}^2 &= 7(\frac{{n_0}^3}{4} + \frac{{n_0}^2}{4}) + \frac{9{n_0}^2}{2} \\
2{n_0}^3 - {n_0}^2 &= \frac{7{n_0}^3 - 7{n_0}^2}{4} + \frac{18{n_0}^2}{4} \\
2{n_0}^3 - {n_0}^2 &= \frac{7{n_0}^3 + 11{n_0}^2}{4} \\
8{n_0}^3 - 4{n_0}^2 &= 7{n_0}^3 + 11{n_0}^2 \\
n_0 &= 15
\end{aligned}
\tag{2}
$$

We want our cross-over point to be the next closest integer to an even number, and we get n_0 = 16. This cross over point only holds when the dimension is a power of 2. For odd numbers we get a different cross-over point. In this scenario we can represent the Strassen recurrence equation as:

$$S(n_0) = 7S(\lceil\frac{n_0}{2}\rceil) + 18(\lceil\frac{n_0}{2}\rceil)^2$$

We can say that $\lceil\frac{n_0}{2}\rceil = x + 1$, and so then we have $n_0 = 2x + 1$. Now substituting into the equations we get:

$$S(n_0) = 7S(x + 1) + 18(x + 1)^2$$

Next we solve for n_0 in $S(n_0) = T(n_0)$:

$$
\begin{aligned}
7S(\lceil\frac{n_0}{2}\rceil) + 18(\lceil\frac{n_0}{2}\rceil)^2 &= 2{n_0}^3 - {n_0}^2 \\
7T(x + 1) + 18(x + 1)^2 &= 2(2x + 1)^3 - (2x + 1)^2 \\
7(2(x + 1)^3 - (x + 1)^2) + 18(x + 1)^2 &= 16x^3 + 20x^2 + 8x + 1 \\
14x^3 + 53x^2 + 64x + 25 &= 16x^3 + 20x^2 + 8x + 1 \\
0 &= 2x^3 - 33x^2 - 56x - 24
\end{aligned}
\tag{3}
$$

Then $x \approx 18$, which implies $n_0 = 2x + 1 = 37$. This basically means that if we are multiplying odd matrices, we want to keep using Strassen until it hits a dimension of 37. Having found the theoretical values, we now move on to determining the experimental n_0.

Implementation Considerations

Approaching the assignment, we had to decide if we wanted to represent matrices as dynamically sized matrices (int **c) or as double arrays (int c[d][d]). We decided to represent matrices as dynamically sized matrices, so that we are able to *malloc* memory on the heap to avoid memory issues. By placing onto the heap, we won't necessarily run out of memory quickly since the heap doesn't have strict size restrictions. We knew that as the dimension grew larger, putting the matrices on the stack will cause trouble. We created a function that allocates memory to create a new matrix based on the dimension. We also created a function that generates a matrix that is filled with random values from 0 to 49, for testing purposes. However, we mainly used the python scripts (discussed below) to generate random matrices.

Next, we implemented the conventional matrix multiplication; this was pretty straight forward. We made sure to use the correct indices so as to make the implementation as fast as possible. Now it was time to move on to handling the Strassen algorithm. We first had to make a function that would add matrices. We spent some time deciding whether to make this function return a matrix or be a void function. In the end we returned a matrix to make the Strassen algorithm easier. Additionally, we could just pass in the returned value into our calls of Strassen. We made a design choice to not create a subtraction function and instead just pass in a parameter that will negate the addition.

Next we moved on to implementing Strassen. Using our knowledge of pointers from CS61, we knew it wasn't necessary to have to initialize/create new matrices from A to H in order to compute P_1 to P_7. Using pointer addresses would result in the correct matrices. Next, we changed Strassen to return the result matrix. This made it so that it's easy to save the results after each Strassen call to the subsequent $P1$ to $P7$ products. After this, we needed to calculate the 4 sub-matrices that will make up our final matrix we return. At first we created temp variables and initialized by calling *newmatrix*, but we saw we could simply just use normal arithmetic with the matrices. This removed 8 calls to *malloc()* and 8 calls to *free()*. In the end, we were able to reduce our total calls each time Strassen is called to *malloc()* and *free()*, from 38 to 22, which significantly reduced memory usage and improved the run time.

Since we had Strassen working for dimensions that our powers of 2, it was necessary to pad matrices for other dimension sizes. Initially, we implemented a function that would return the nearest power of 2. This function worked well with small dimensions. However, for large dimensions, there would be a lot of extra padding. For instance, for dimension 513, the closest power of two would be 1024, so it would need $1024^2 - 513^2 = 785407$ extra 0's for padding. This seemed rather unnecessary, timely, and space consuming. We figured out instead that after we hit our cross over point in modified Strassen, the conventional matrix multiply didn't care about if dimensions were a power of 2. So our function would instead keep dividing our dimension by 2 until it was less than the threshold, then it would keep multiplying by 2 till we hit our dimension. We do this to ensure we can continuously keep halving our dimension successfully in the modified Strassen.

Python Script

The python scripts used in testing and gathering visualization data may no longer work, as the Strassen variant has been formatted to work with the pset specs. It had been modified for testing and data collection purposes.

- *compare_one_dim.py* was written to test the run-time of any of the algorithms - this could be done by changing the flag or the dimension value.
- *crossover.py* thoroughly tests every possible crossover point up to dimension 1024 and selects the crossover point that yields the fastest runtime for the Strassen variant. This was used to gather data points on the experimental n_0 value at higher dimensions.

- *r_crossover.py* is extremely similar to *crossover.py* but reduces the number of crossover points tested. This script allowed the identification of trends with regards to the experimental n_0.
- *generate_matrix.py* allows for the creation of any dimension matrix.
- *why.py* takes in crossover point data from a text file and generates a graph.

Results

After finishing the implementation, we thoroughly tested our algorithm. Python scripts were used to generate test files, graph data, and automate the testing process. We made sure to do a variety of tests such as generating random values with different ranges, testing multiple dimensions, and running multiple trials. We also used C to test and find the average run-time it took for various crossovers over different dimensions. We included these tests in a separate file named *testing.txt*.

Modified Strassen vs Conventional

We wanted to see how our modified Strassen algorithm compared against regular Strassen and standard multiplication for certain dimensions. For lower dimensions, standard multiplication will be slightly faster than our modified algorithm; this is trivially noticed. Additionally, without modifying Strassen, it will take a lot longer to multiply matrices. In fact, the amount of time it takes to multiply matrices seems to grow exponentially, until it seems to plateau for dimensions above 750. As the dimensions grow larger, our modified algorithm begins to perform better than standard.

Table 1: Comparing Matrix Multiplication Algorithms (seconds)

dim	Strassen Mod	Standard	Normal Strassen
50	0.000573	0.000502	0.014450
100	0.003780	0.003916	0.091204
200	0.030402	0.033413	0.641871
500	0.359128	0.431977	5.035976
750	1.163620	1.493466	32.559812
850	1.812890	2.273023	34.399298
1000	2.463080	3.492311	34.470516

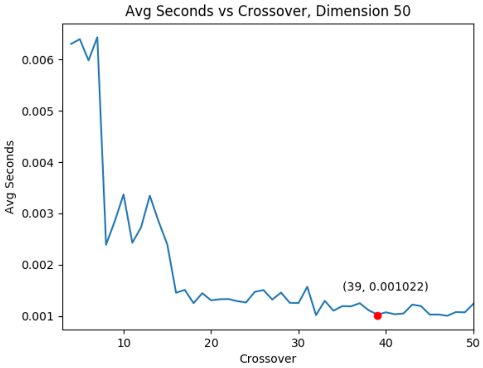
Crossovers and Timing

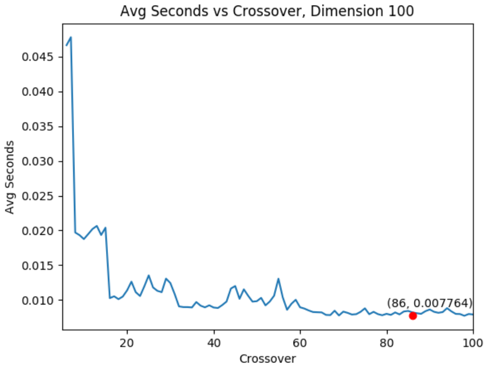
Below we tested for dimensions of 50, 100, 500, and 1000 to see the average number of seconds it takes to run that dimension against the cross-over points.

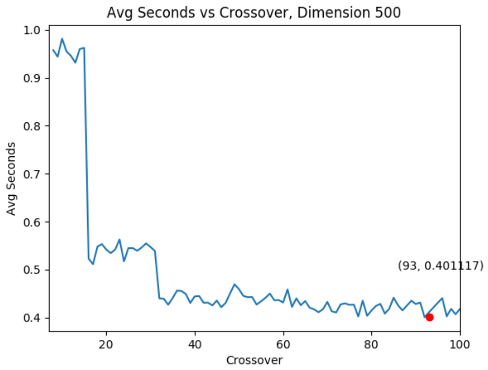
Table 2: Average Runtime (seconds) for Various Crossovers

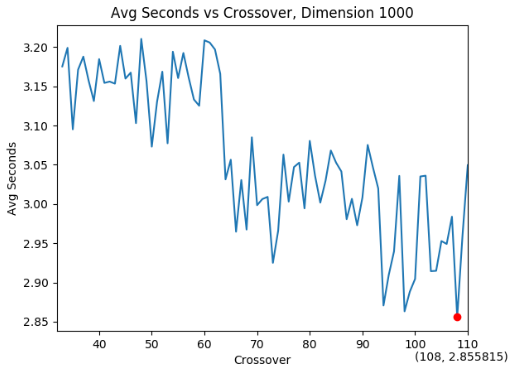
n_0	d = 50	d = 100	d = 500	d = 1000
5	0.006392	0.049580	2.353443	16.455410
10	0.003370	0.018762	0.981672	6.761160
15	0.002395	0.020398	0.962501	6.648398
20	0.001309	0.011392	0.962501	3.754576
25	0.001475	0.013545	0.544746	3.786594
30	0.001309	0.012445	0.546947	3.799339
35	0.001197	0.008937	0.440383	3.095093
40	0.001075	0.008928	0.443932	3.184594
45	0.001032	0.012016	0.435194	3.159743
50	0.001243	0.009833	0.459157	3.159743
55		0.013072	0.433526	3.160399
60		0.008960	0.431178	3.208519
65		0.008221	0.434180	3.056351
70		0.008333	0.432587	2.998436
75		0.008810	0.426286	3.062882
80		0.008022	0.414633	3.080380
85		0.008437	0.441265	3.052615
90		0.008640	0.428037	3.008268
95		0.008371	0.430847	2.909606
100		0.007923	0.417677	2.904465
105				2.952741
110				3.049450

Below is the plotted version of the data above so that it's easier to visualize. We just wanted to get a general sense of the convergence point and saw that it hovered around the 90 - 110 range. We will discuss more below about why we thought we were receiving such high numbers and why the graphs look the way they do.









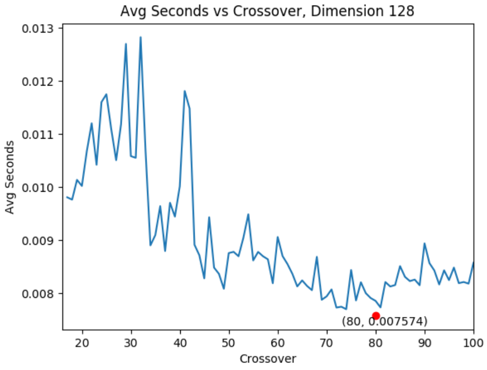
We also wanted to see what the average cross was for dimensions that are powers of 2 and for odd matrices.

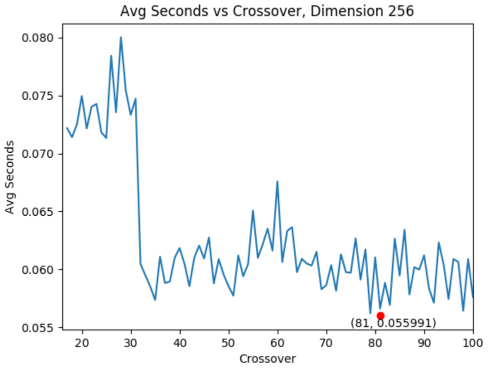
Table 3: Average Crossover for Powers of 2 dimensions

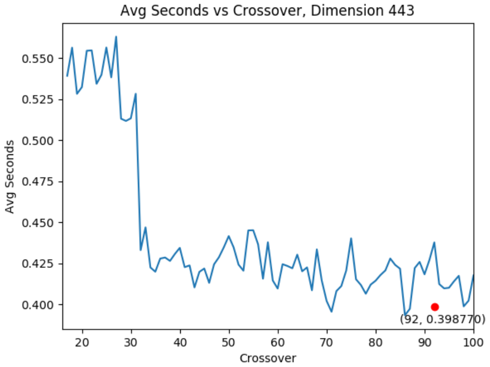
Dim	Avg Crossover
64	51
128	80
256	82
512	91
1024	105

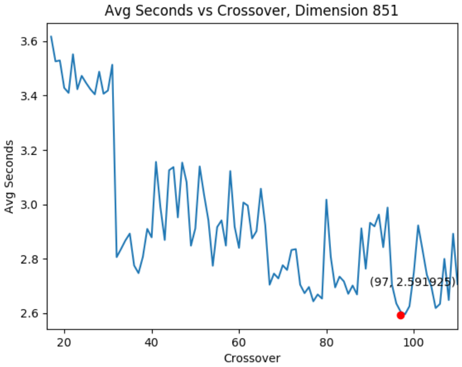
Table 4: Average Crossover for Odd Matrices

Dim	Avg Crossover
83	71
159	87
285	81
443	92
851	97









Comparing odd and even dimensions, we can see that there is no difference between the crossover points. We figured that this is due to us initially padding the matrices in the beginning (in main).

Difficulties

We wanted to make sure our implementation was working before optimizing the code. There were many memory issues for large dimensions. We knew this would happen and our laptops would keep running into memory warnings and would shut down. Eventually we were able to find the source of the problem; we malloced/freed less memory and updated our padding function, which decreased memory usage and significantly improved performance.

Discussion/Conclusion

In the first part of the task, we analytically computed the estimated cross-over points to be 16 for dimensions that are powers of 2, and 37 for odd matrices. After implementing the necessary functions, we discovered that the optimal crossover point is in the range 90 to 110 for high dimensions.

The theoretical floor is 37 for odd dimensions and 15 for dimensions with values that are powers of 2. When we mathematically solved for the cross-over point, we made assumptions about the run-time of memory allocation and de-allocation and ignored optimizations within the compiler and OS (such as caching) that can affect the experimental n_0, either by affecting the conventional algorithm or the matrix multiplication. Additionally, running the same function multiple times and averaging the results allows for outliers to affect experimental crossover thresholds. As discussed above, we were at first getting high cross-over points because of memory allocation. Even after optimization, the analytically-determined crossover points were not reached.

Taking a look at the graphs, one can see that for each of the dimensions we tested, the average number of seconds ran at that specific cross-over point begins to plateau. With this we concluded that any number in that range will do fine as a cross-over point, as factors outside our control introduced variance in the experimental n_0. For instance, with dimension 50, the data indicates a range of cross-over points is between 40 - 50. Generally, matrices of size 100 - 500 had crossovers between 85 - 95, and dimensions larger than 500 had crossovers between 95 - 110. The variance in crossover threshold results indicate that the algorithm produces consistent results; this is testament to the stability and testing that went in to the creation of this program.

From the graphs, it is clear that the cross-over point will eventually plateau with the higher dimensions. With regards to the run-time in seconds, there is also a distinct difference between the run-times of matrices with dimensions between the powers of 2, i.e. dimensions 257 through 512 vs 513 through 1024. This can be attributed to the padding at the beginning of the matrix versus other methods, such as padding the matrix to an even dimension (if necessary).