CS 124, P2: The Matrix Recursed

Joe Kahn, Zac Bathen

Due March 25th, 2016

1 Introduction

In this report, we first perform mathematical analysis for optimal combination of Strassen with traditional matrix multiplication. Next, we perform experimental analysis on our implementation of the algorithm and compare it to the result from the theoretical analysis. Finally, we discuss the process we went through during our implementation, including difficulties we encountered, optimizations we considered and rejected, and optimizations we included. There are considerations to both time and space complexity throughout the report.

2 Theoretical Analysis

We set out to find a theoretical cutoff point for recursing through Strassen's algorithm before switching over to traditional matrix multiplication. To start this analysis, it is important to write down runtimes for each of the algorithms individually. For this analysis we assume that both addition and multiplication of 2 integers are unit time operations. We also assume a square, n by n matrix.

First, traditional multiplication: we need to iterate through the matrix, taking the dot product of n rows and n columns. So we have n^2 dot products. Each dot product includes n multiplications and $n-1$ additions. This gives us the runtime equation:

$$C(n) = (2n-1)n^2 \tag{1}$$

Next, look at Strassen's algorithm. For this analysis, we assume that the method uses lazy padding (as we do in our implementation) at each step. This is what causes the presence of ceiling functions in the analysis.

We will first write out a recurrence. Strassen's ultimately uses 7 multiplications on matrices of size $\lceil n/2 \rceil$, and performs an additional 18 matrix additions and subtractions, also on matrices of size $\lceil n/2 \rceil$. This generates the recursion

$$T(n) = 7T(\lceil n/2 \rceil) + 18(\lceil n/2 \rceil)^2 \tag{2}$$

But we need to go a step further and find the optimal crossover point, or the point where it stops being optimal to continue recursing with Strassen's algorithm and becomes better to simply multiply the submatrices using the traditional method. For this, we want to combine the equations into the runtime of our final algorithm, which runs Strassen's until the optimal cutoff point and then finishes using traditional matrix multiplication.

$$T(n) = 7 \min C(\lceil n/2 \rceil), T(\lceil n/2 \rceil) + 18(\lceil n/2 \rceil^2)$$

Our goal now is to find the cutoff point, so we want to find n when $T(n) \geq C(n)$. Use the condition $T(n) = C(n)$. Since we are using this condition, the minimization in our runtime function crystallizes (the arguments are equal), and we can use whichever we like. We have a simple expression for $C(n)$, so we will simply plug this in for the minimization and set the result equal to $C(n)$.

At $C(n) = T(n)$

$$(2n-1)n^2 = 7(2(\lceil n/2 \rceil) - 1)\lceil n/2 \rceil^2 + 18(\lceil n/2 \rceil^2) \tag{3}$$

However, now we must split the equation into 2 cases: n even and n odd.

<u>n even</u>

If n is even, solving equation 3 becomes a simple algebra problem:

$$(2n-1)n^2 = 7(2(n/2) - 1)(n/2)^2 + 18((n/2)^2)$$

The result is 15. However, as explained above, we are looking for the case where $T(n) \geq C(n)$, and we know that n is even, so we actually need to increase the cutoff value by 1.

Thus, for matrices of even dimension, the cutoff value for multiplying is $n = 16$.

<u>n odd</u>

In this case, we will need to pad. For the analysis, re-express n as $2k+1$. We will pad one row, so $\lceil n/2 \rceil$ becomes $\lceil \frac{2k+1}{2} \rceil = k+1$. Plug this into equation 3.

$$(2(2k+1)-1)(2k+1)^2 = 7(2(k+1)-1)(k+1)^2 + 18((k+1)^2)$$

The result of solving for k is $k = 18.085$. This gives $n = 2k+1 = 37.17$. We want to find the next higher odd number.

Thus, for matrices of odd dimension, the cutoff value for multiplying is $n = 39$.

3 Experimental Analysis

We performed experimental analysis in 2 different ways.

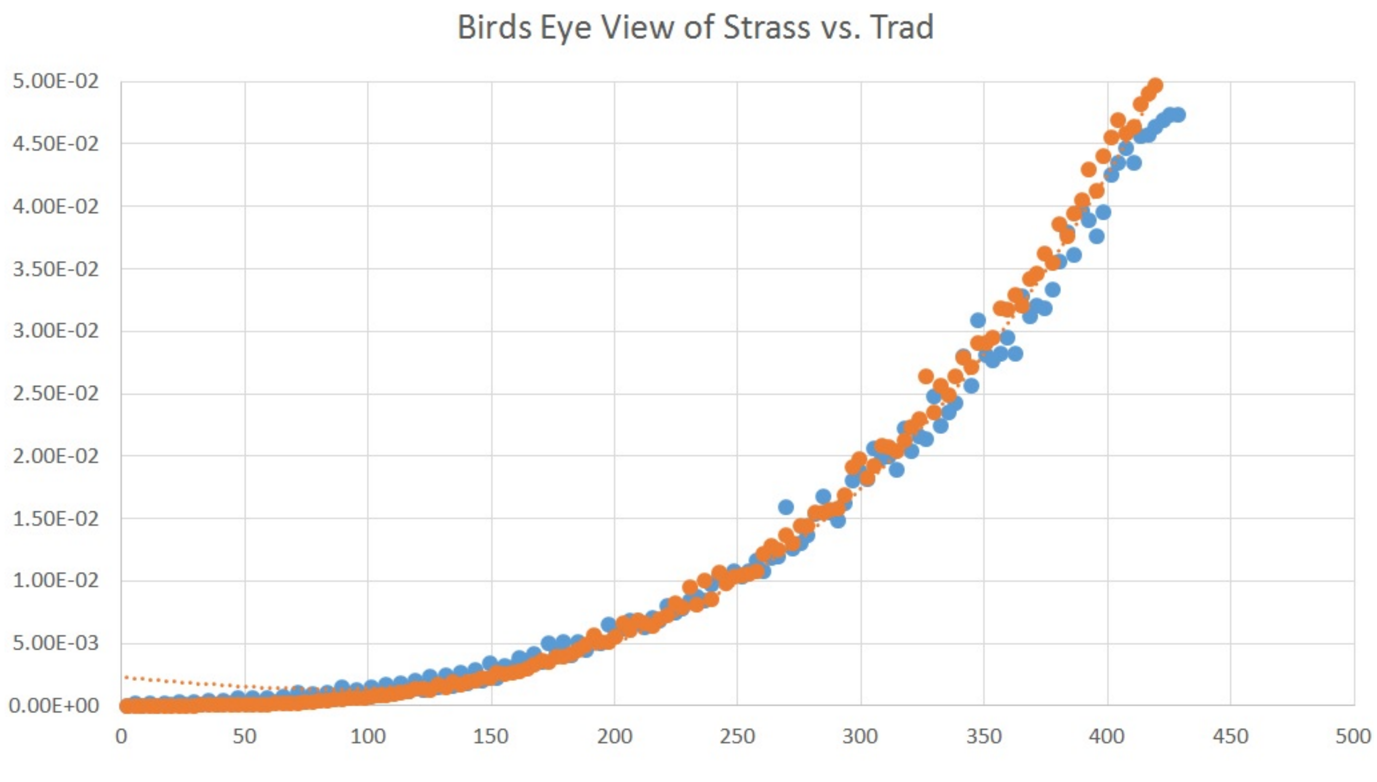
3.1 Straight Find The Min

First, we wrote a simple function that iterated through a series of matrix dimensions and ran the algorithm on randomly-generated matrices of that size. For each size, we iterate through different cutoff points and compare the runtimes. At each dimension, we took the average of 5 trials and stored the time and cutoff if it was a new minimum. Interestingly we found that nearly every even trial showed a local minimum time around 115 and each odd trial showed a local minimum around 215. However, for larger matrices, this was rarely the global minimum. We believe that this is likely due to ordinary variance. One simple way to avoid the data being skewed in this way was to stop the recursion when we found that for k consecutive runs, the runtime was higher than the previous runtime. This due to the convex nature of the timing curve, as by definition the optimal cutoff point will produce a minimum time and cutoff points on either side of it will result in higher times. However, we were not perfectly confident in our criteria to find the local minimum, so moved on to other analysis.

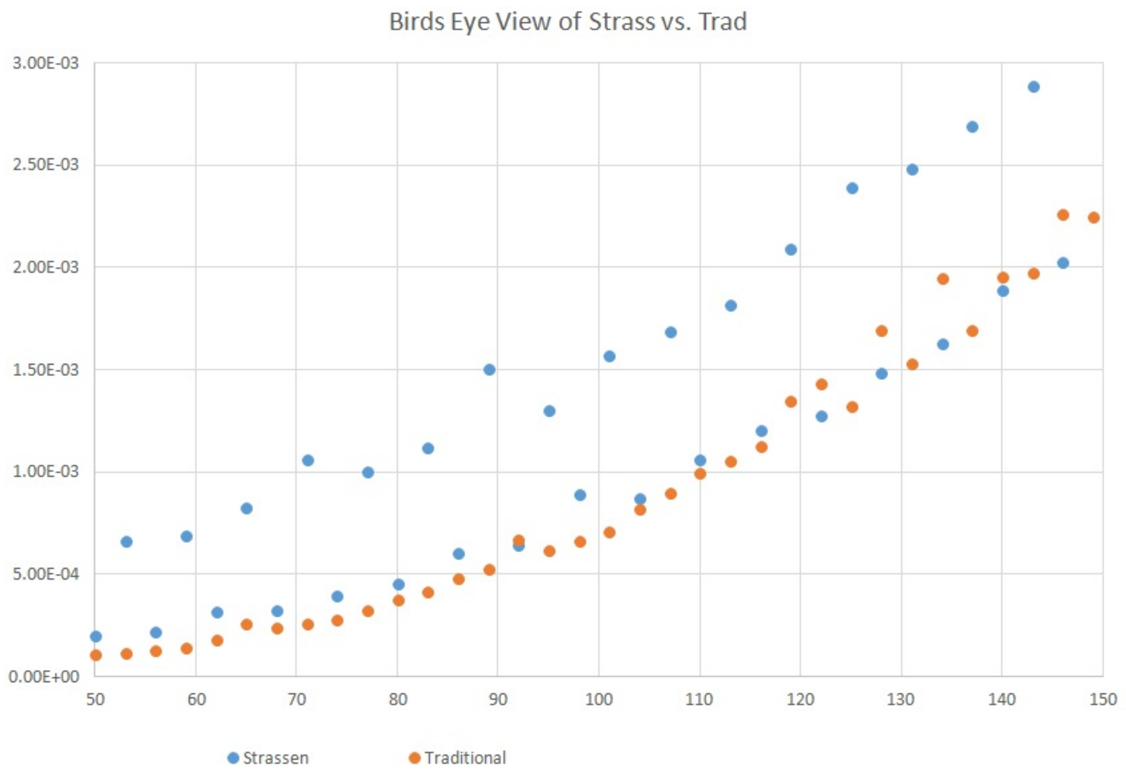
3.2 Graphing Trends and Finding Intersect

For the second type of analysis, we generated runtimes for a series of n values using the cutoff as $\lceil n/2 \rceil$. We calculated these data sets for both Strassen's and traditional. Essentially, the intersection of these graphs will give us the point where it stops being optimal to run Strassen's one more time, and starts being optimal to simply switch to traditional multiplication.

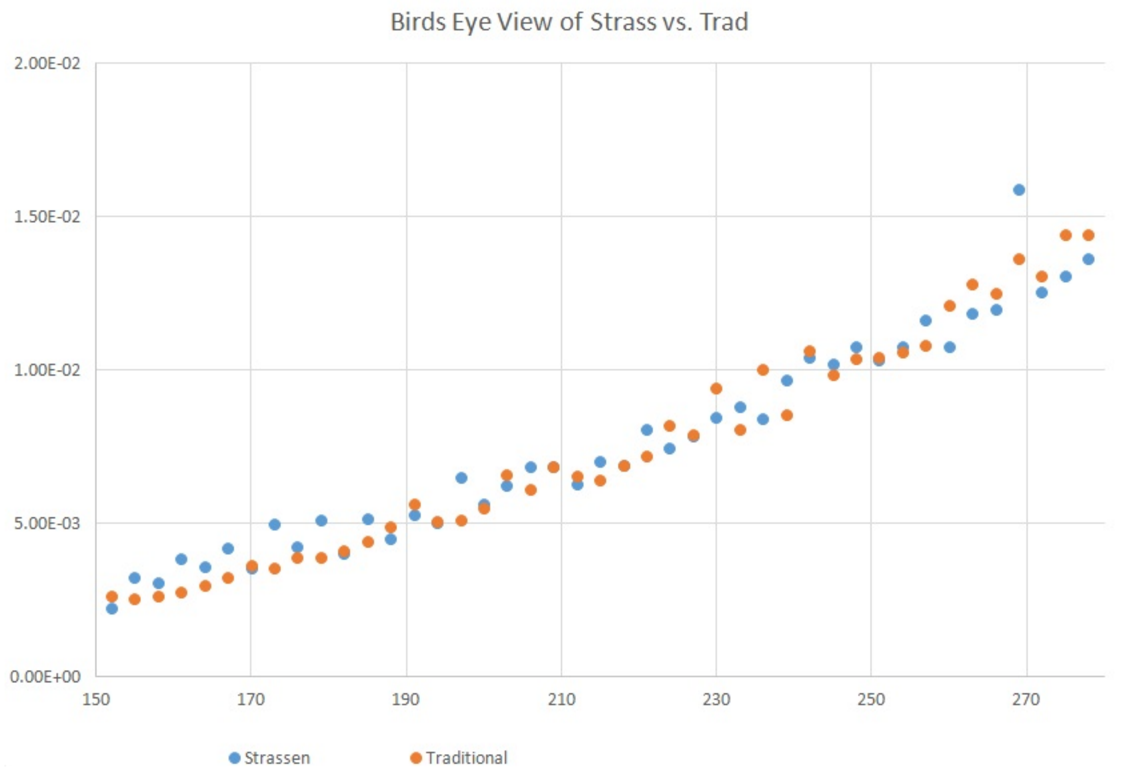
Here are the graphs we generated with this data:



As can be seen from the graph above, Strassen's is asymptotically faster. Let's find the cutoff point for even...



As we can see, the experimental cutoff for even is about 115. And for odd...



Here, we see a crossover of about 215.

Since the two tests generated simple results, we were left confident with our analysis.

4 Implementation

In this section, we discuss our specific implementation of the algorithm. Throughout the process, we attempted to be mindful about the order of steps, the space complexity (especially with regards to cache performance), and memory allocation.

4.1 Cache Optimization I: Smart For-Loops

By smartly selecting the for-loops for the traditional multiplication we were able to access contiguous blocks of memory as often as physically possible. (This as opposed to calculating dot products as we are taught in school.) We effectively build up the matrix product by taking partial dot products of rows against rows and incrementally filling the entries of the output matrix.

Accessing the cache (contiguous blocks of memory) for successive operations instead of jumping between contiguous blocks of memory (skipping between rows) is desirable, and this approach allowed us to do just that.

Using variants on grade scale matrix multiplication, it took 73.47 seconds to multiply 2 2048x2048 matrices. With this optimization, it took only 6.02 seconds.

We hypothesize that this is one of the reasons our cutoff point for Strassen's was found to be higher than we had originally expected (i.e. much higher than our theoretical analysis). The impact of this optimization on regular multiplication appears to have been relatively larger than the impact of the optimizations we made to Strassen's. As such, Strassen's algorithm effectively has "more to compete with."

4.2 Padding

Padding is necessary to run Strassen's algorithm on any matrix that doesn't have a dimension equal to a power of 2. The obvious method of padding is to simply check your inputs at the beginning and add rows and columns of zeroes until you have matrices with dimensions that are powers of 2, then multiply them and "unpad" them at the end.

A more efficient approach is to "lazily pad." This means that at each step of the recursion you check to see if the matrices have odd dimension, and if so pad one row and one column to them, then proceed. If you pad, simply remove this extra row and column of zeroes before returning the result matrix.

This lazy padding is more efficient because it never requires the allocation of memory of a matrix larger than $n+1$ by $n+1$, whereas traditional padding could generate and work with matrices of with dimension equal to the next highest power of 2.

4.3 Clever Recombination

During the process of recombination into the answer matrix, the algorithm uses at most 4 of the 7 generated component matrices at a given time. For this reason, through clever ordering of the steps, we can minimize the amount of space needed, holding only 4 matrices in memory. Here is the ordering, where $m1 - m7$ are the component matrices and $C11, C12, C21, C22$ are the 4 corners of the answer matrix:
1. Generate $m1, m2, m3, m6$.
2. Compute $C22 = m1 - m2 + m3 + m6$
3. Overwrite $m6$ with $m4$
4. Compute $C21 = m2 + m4$
5. Overwrite $m2$ with $m5$
6. Compute $C12 = m3 + m5$
7. Overwrite $m3$ with $m7$
8. Compute $C11 = m1 + m4 - m5 + m7$

It might be possible to find an even more optimal ordering, but given that this only made a marginal improvement to runtime and memory usage we choose to favor simplicity over further (non-trivial) implementations. Note that calculating the new submatrix for each quadrant of C in one go requires fewer $O(n^2)$ passes during addition / subtraction.

5 Room For Improvement

5.1 Working Matrices

Note: These optimizations would be the next step if we had more time to work on this code. We put time into implementing each one, but were unable to ultimately make them work in time to submit the assignment. If you'd like to take a look at our efforts, the git repository we used for this project is the record of truth.

The main optimization that we attempted to employ was using working matrices withing our Strassen function. Rather than allocating memory for a matrix each time we need to store data within the recursive calls, then using the data and freeing the memory, we could instead generate a single global working matrix, with dimension equal to the dimension of the matrices being multiplied. This avoids additional memory allocation while moving up the stack, and de-allocation on the way down—instead the only allocation necessary to store the products computed recursively during Strassen's would have occurred in global scope, before running the algorithm.

5.2 In-line Matrix Operations

In theory, we determined that it would be possible to perform Strassen multiplication using only 3 memory allocations all the way through the program. The first 2 build the input matrices from the text file, and the third is passed into the Strassen function and used as a reference, but is also updated with the answer values. As the function recurses downward, it also needs to be passed in coordinates for the portion of the answer matrix it is solving for, and it edits and updates only that range with the correct values. While we determined that this would be better in theory, the improvement would have been extremely small, as we were able to use just a few more allocations and make the code and process much more readable and understandable.

5.3 Multi-threading

This problem could have also been done more optimally using multithreading, but we had neither the time nor experience to implement it.

6 Conclusion

This project was a great exercise for us in continuous improvement and optimization. Each time we wrote code that worked, we went back and thought through possible improvements, both in terms of time and space complexity. This process allowed us to ultimately create a product which we are very proud of, and one which is a result of extensive thought and effort (and less sleep than either of us have had in a semester).