Matt Leifer and Manav Khandelwal 3/23/17
Programming Assignment #2: Strassen

1 Analytic Results

The conventional matrix multiplication algorithm takes $n^2(n+n-1) = 2n^3 - n^2$ steps because to take the dot product of one row with one column requires n multiplications and $n-1$ additions and there are n^2 dot products.

$$T(n) = \begin{cases} 7 \cdot T(\lceil \frac{n}{2} \rceil) + 18 \cdot (\lceil \frac{n}{2} \rceil)^2 & n > n_0 \\ 2n^3 - n^2 & n \leq n_0 \end{cases}$$

We want to find the smallest n_0 such that $T(n_0) < 2(n_0)^3 - n^2$, a.k.a the crossover point where we want to go from using the "hybrid Strassen" to using conventional algorithm. We set the two values equal to each other and solve (assuming that the recursive call of T will use the conventional multiplication, because that is the definition of a crossover value.

$$7 \cdot \left(2(\lceil \frac{n}{2} \rceil)^3 + (\lceil \frac{n}{2} \rceil)^2\right) + 18(\lceil \frac{n}{2} \rceil)^2) < 2n^3 - n^2$$

To remove the ceiling, we have to consider two cases:

1.1 Even

If n is even, we have the following relationship:

$$\frac{14n^3}{8} + \frac{11n^2}{4} < 2n^3 - n^2$$

$$\frac{15n^2}{4} < \frac{n^3}{4}$$

$$15 < n$$

This means that for values greater than 15, in the even case, mixed Strassen's is better than the conventional matrix multiplication algorithm. That means that we can say that for all even values of n less than 16, we should use the conventional matrix multiplication algorithm instead of Strassen's.

1.2 Odd

If n is odd, we have the following relationship:

$$14(\frac{n+1}{2})^3 + 11(\frac{n+1}{2})^2 < 2n^3 - n^2$$

When we solve for n_0, we get a value of 37.17 which means that our crossover value (since it is odd) is $n_0 = \mathbf{37}$ and so that for odd values of n less than or equal to 37, we should use the traditional matrix multiplication algorithm.

One small note is that in our implementation of Strassen's, in order to avoid an extra 18 memory allocations by creating new matrices when adding, we required 19 additions instead of 18. This, however, does not change the crossover point for either the even or odd cases. In practice, the time saved by not doing all those memory allocations was tremendous (see below) but the analysis did not take into account memory management and so that's why the analysis doesn't change significantly.

2 $O(n^3)$ Matrix Multiplication Implementation

In order to optimize the convention $O(n^3)$ matrix algorithm, I made sure to access the rows and columns of the matrices I was multiplying in a cache efficient way. Essentially this means that the function that multiplies these matrices accesses the entries in these matrices that are phyiscally close together in memory. As per a post on Piazza, I consulted `https://cs61.seas.harvard.edu/wiki/images/0/0f/Lec14-Cache_measurement.pdf` from Lecture 14 of CS61, specifically slide 30, to program this matrix multiplication. The cache optimized version of this algorithm performed particularly well for large matrices. Here is a table comparing the time in seconds it took for the non optimized and optimized conventional algorithms to compute various matrix products. For small matrices, the caching didn't matter but for $n \geq 256$, the caching made a vast difference. All further timings will use the optimized verion.

Size	Optimized	Non-optimized
1	0.000001	0.000001
2	0.000001	0.000001
4	0.000001	0.000002
8	0.000005	0.000006
16	0.000037	0.000031
32	0.000178	0.000115
64	0.001135	0.000827
128	0.007922	0.008746
256	0.059985	0.060686
512	0.493521	0.612233
1024	4.094844	18.983677
2048	34.155533	183.490784
4096	454.431335	1496.175171

3 Strassen Implementation

One of the simplest optimizations I made in implementing strassen's algorithm is that instead of passing the entire matrix through a function, I just pass a pointer to it and so all the data in that matrix doesn't have to be passed to the function (I'm passing by reference, not value). The second optimization I made was with regards to how my implementation of Strassen's algorithm handles matrices of odd dimensions and dimensions that are not a power of 2. If the dimensions of a matrix were not a power of 2, then I could just pad the entire matrix with zeroes to the right and below to make the dimensions of the matrix a power of 2 and multiply those. So, a 5x5 matrix would be padded to be 8x8. However, if the cross over point isn't 1, padding up to the next power of two is not the most efficient padding scheme that can be implemented. Instead, what I do is dynamically add an extra row below and an extra column of zeros to the current matrix if the matrix has odd dimensions. Thus, the padding only occurs if it's absolutely necessary.

Perhaps, however, the most significant optimization I made that actually doubled the speed of my implementation of Strassen's and reduced the amount of memory allocation and deallocation that had to be done was in how I added and subtracted matrices. If I am adding matrix B to matrix A, instead of allocating a new block of memory for the resulting matrix C and filling C_{ij} with $A_{ij}+B_{ij}$, I update the values in matrix A so that A_{ij} gets reassigned to be $A_{ij}+B_{ij}$. Doing this saved me from allocating 18 new matrices on any given run of Strassen's algorithm. There is one drawback and that is that adding matrices in this way forced me to make 1 extra addition. So my implementation uses 19 additions and not the usual 18 and you can show that the cross over point derived theoretically above is unchanged even if $T(n) = 7T(\lceil \frac{n}{2} \rceil) + 19 \cdot (\lceil \frac{n}{2} \rceil)^2$. Despite this extra addition, the time saved is significant for matrices of all sizes (the cross over point here is when $n = 1$). The column called 'State Change' in this table gives the runtime for my implementation of Strassen's algorithm when is used the form of matrix addition that updates one of the matrices being added together in place. The Malloc column gives the times for Strassen's algorithm for the addition that allocates an entirely new block of memory to add two matrices.

Size	State Change	Malloc
1	0.000001	0.000001
2	0.000008	0.000015
4	0.000056	0.000069
8	0.000233	0.000373
16	0.001507	0.001761
32	0.007833	0.013492
64	0.058018	0.098547
128	0.400884	0.674828
256	2.875862	4.776727
512	20.106550	34.681881
1024	140.375183	243.241043

Just to show that this method of adding produces the same results as the set of operations in the regular Strassen, here is a step-by-step walkthrough of the additions. When I write $A' = A + B$, I don't mean that I've allocated a new block of memory for A' that would defeat the purpose of all this, all that is meant to do is help keep track of whether a matrix has been updated or not. A $\cdot$ indicates that Strassen's is called.
$F' = F - H$
$P_1 = A \cdot F'$
$A' = A + B$
$P_2 = A' \cdot H$
$C' = C + D$
$P_3 = C' \cdot E$
$G' = G - E$
$P_4 = D \cdot G'$
$P_5 = ((A' - B) + D) \cdot (E + H) = (A + B - B + D) \cdot (E + H) = (A + D) \cdot (E + H)$
The $(A' - B)$ is the one extra subtraction I make.
$A'' = A + D$
$E' = E + H$
$P_6 = (B - D) \cdot (G' + E') = (B - D) \cdot (G - E + E + H) = (B - D) \cdot (G + H)$
$B' = B - D$
$G'' = G + H$
$P_7 = (A'' - C') \cdot (E' + F') = (A + D - C - D) \cdot (E + H + F - H) = (A - C) \cdot (E + F)$
Then to actually calculate the product, I did the following:
Top Right $= AF + BH = P1' = P1 + P2$
Bottom Left $= CE + DG = P3' = P3 + P4$
Top Left $= AE + BG = P6 + P5'$, where $P5' = P5 + (P4 - P2)$
Bottom Right $= CF + DH = (P5' + P1') - (P7 + P3') = P5 + P4 - P2 + P1 + P2 - P3 - P4 - P7 = P5 + P1 - P3 - P7$

4 Results

To experimentally find the crossover point I guessed at what I thought it might be until I found a value for n_0 such that for n greater than n_0, Strassen's would run faster than the traditional. Based on my implementation it seems that this point occurs somewhere in the **low 80's**. However, given values for particularly small matrices, the time it takes to run Strassen's and the conventional algorithm is subject to a lot of noise and variablity and so it's hard to pinpoint exactly where the optimal cross over is. Included in the file **data_1.csv**, **data_2.csv**, and **data_3.csv** are three tables that show the times comparing Strassen's and the conventional algorithm with a cross over point of **83** because that was a point that gave me some of the best results. A 1 in the rightmost column indicates that Strassen's was faster than the traditional algorithm and 0 for the reverse. By the time the size of the matrices grows to 120, Strassen's algorithm is just about always faster than the traditional method barring a few sporadic sizes. There are some instances in which the conventional algorithm does perform better in one file than Strassen and in the other file it's reversed and that I attribute just to the noise that occurs when timing something on a computer. Odd matrices,

by and large, were more likely to run slower for Strassen's than even ones because they would have to be padded with an extra row of zeros which inflated their size.

As we can see from the results, the crossover point in practice is far larger than the crossover point theoretically (meaning the mixed Strassen's algorithm is not faster until we reach a higher value of n_0). This makes sense because, in practice, there are a lot of inefficiencies to deal with. Our ability to reduce the number of matrices allocated in Strassen's helped significantly with this, as seen above, but Strassen's still requires passing pointers to many different sub-matrices and allocating new blocks of memories for the recursive calls to the function. This is the reason we have a cross-over point in the first place (because more additions takes times), and while Strassen's is *asymptotically* faster than the conventional algorithm, at low enough ns in practice the constants are too large to justify its use.

For matrices that had dimensions that were a power of 2, a cross over point of 32 optimized the matrix multiplication - making the cross over point 64, didn't change anything significantly. One possible reason for why the cross over point was the optimal cross over point is that there was no need to spend any time or extra thought to pad the matrices correctly with zeros. Since these are matrices of even dimensions them theoretically optimal crossover point would be when the dimensions were 16x16, but because of the memory allocation is was actually 32. I tested the matrix multiplication functions by writing a function gen_matrix that generates a random $n \times n$ integer matrix with integers between -9 and 9.