Introduction: In this assignment, we were asked to find (both experimentally and mathematically) the optimal crossover point for a hybrid implementation of Strassen's algorithm and traditional matrix multiplication. While we saw in class and in previous assignments that the divide-and-conquer approach (with Strassen's algebra trick) was asymptotically faster than the standard $O(n^3)$ algorithm, intuitively it was apparent that Strassen's would run slower on small matrices. This is because (while there are 7 rather than 8 matrix multiplications) there are 18 additions/subtractions. So the standard method will be faster up to a certain point.

Theoretical Crossover: But what is that certain point? The official recurrence for Strassen's runtime isn't specific enough to help us: $T(n) = 7T(n/2) + \Theta(n^2)$. There is a lot hidden away in that $\Theta(n^2)$. Examining the algorithm (included in L8.pdf), we can see that while the matrices we're adding are $\Theta(n)$ but in reality they are $n/4$ - each one is a quarter the size of the original matrix. This is a big deal for actual runtime, even though it's just a constant asymptotically. Similarly, we know that we have 18 add/subtract operations for all those size $n/4$ matrices, which gives us the final recurrence:

$$\mathbf{T(n) = 7T(n/2) + 18((n/4)^2)}$$

n	**T(n)**	$\mathbf{n^3}$	$\mathbf{T(n) < n^3}$?
1	1	1	FALSE
2	12	8	FALSE
4	99	64	FALSE
8	762	512	FALSE
16	5,619	4,096	FALSE
32	40,482	32,768	FALSE
64	287,979	262,144	FALSE
128	2,034,282	2,097,152	TRUE
256	14,313,699	16,777,216	TRUE
512	100,490,802	134,217,728	TRUE
1,024	704,615,259	1,073,741,824	TRUE
2,048	4,937,025,402	8,589,934,592	TRUE
4,096	34,578,052,179	68,719,476,736	TRUE

(Note - we are dealing with square matrices for this assignment). We can now examine the exact values of the theoretical recurrence vs the $\Theta(n^3)$ algorithm. We can see that the crossover occurs at $n = 64$: for any matrix that is 64x64 and below, you are better off using the traditional method. However, once you get into matrices of size 128x128 and above, the advantages of the reduced number of multiplications required by Strassen's begin to come across.

Experimental Crossover: Now for the interesting part - coding it up. Knowing the theoretical optimum, it became an interesting challenge to code up Strassen's such that it could achieve the desired results. In my first pass, I simply threw together the algorithm without regard for memory management or optimization - step one was to get it working. This was pretty straightforward, though for a while I was quite pleased my tests were all passing, only to discover I was testing the standard algorithm twice instead of Strassen's. Luckily it worked anyway, but I felt pretty dumb for a bit.

Stack to Heap: This naive implementation only got me so far, though. My crossover point was too high - 128 or 256. I was essentially making several copies of matrices, each copy taking $\Theta(n^2)$ steps. I knew I had to optimize, and the staff hint to look at memory allocation/deallocation as a source of slowdown proved quite helpful. I undertook a major overhaul of how matrices were implemented. Instead of creating and duplicating large data structures on the stack, I set them up as wrappers around an underlying raw chunk of numbers on the heap. By having objects that consisted solely of a few integers to keep track of starting indices and dimension, and maintaining pointers to the underlying data, I was able to speed the algorithm up enough to get the theoretical optimum: crossover 64.

Shared Pointers: Originally this was incredibly leaky, using upwards of a gigabyte for $n = 1024$. However, the wonders of shared pointers made quick work of that. Essentially, instead of having to keep track of and free every matrix manually, shared pointers employ the C++ equivalent of ARC (Automatic Reference Counting). When the last pointer to an object is removed, it is freed. This emulates the memory safety of a stack object, with all the benefits of the heap. In addition, you don't have to worry that your program could stop at any point to free up orphaned blocks, as with garbage cleaning (one of my least favorite paradigms). Therefore, I could be confident that the timing for each trial was fair when trying to find the optimum. One possible drawback to ARC is that you can lose memory when you have a pointer cycle, however using profiling/leak-checking tools I was able to verify that the program was no longer leaking memory.

"Square" Matrices: Another one from the staff - pad matrices with zeroes to make them square. Because Strassen's from class is designed to work with square matrices (dividing each square into four more squares), I had to find a way to make it work with non-square inputs. After a bit of thought, it made sense that padding with zeroes would work. If you are a two-dimensional creature, it's not that the other dimensions don't exist: they are just zero for you. So thinking of matrices as ways to describe n-dimensional space, adding zeroes was simply a more dimension-specific calculation for the same transformation. This same modification isn't really necessary for standard multiplication, which could affect performance and therefore the crossover point. If you have to change a 1025x1025 matrix to be 2048x2048, you are definitely going to see a big slowdown compared to the straightforward approach.

Closing Remarks: Another fun assignment. I particularly enjoyed learning more about memory management and shared pointers, and definitely feel that I'm improving with C++ - a language I've wanted to dive into for a while. More on topic, though, I gained a very thorough understanding of Strassen's algorithm. In addition, I feel that I am much better equipped to recognize theoretical vs. practical tradeoffs in algorithms. Awesome asymptotic performance doesn't necessarily translate into real-life gains, and it's important to understand your tools to make an intelligent decision on what makes sense in practice.

Sources:

`http://www.cplusplus.com/doc/tutorial/files/`

`http://www.cplusplus.com/reference/vector/vector/`

`http://www.cplusplus.com/forum/beginner/12409/`

`http://www.cplusplus.com/forum/general/13135/`

`http://stackoverflow.com/questions/23438393/new-to-xcode-cant-open-files-in-c`

`http://stackoverflow.com/questions/1538420/difference-between-malloc-and-calloc`

`http://stackoverflow.com/a/24916697`

`https://www.exploit-db.com/docs/28550.pdf`

`http://stackoverflow.com/a/1549960`

`http://stackoverflow.com/a/108360`

`http://stackoverflow.com/a/466256`

`http://stackoverflow.com/a/11587545`

`http://stackoverflow.com/a/26647753`

`https://lists.nongnu.org/archive/html/qemu-devel/2014-05/msg03631.html`

`http://stackoverflow.com/a/26734543`