CS 124 Programming Assignment 2

HUID 51248651

03/24/2017

Theoretical Analysis Note: Throughout my analysis I use the implementation of strassens such that if n is odd, then it is padded out with 0s to a $n+1 \times n+1$ matrix when strassens is performed on it.

Let $m(n)$ be cost of multiplying two n by n matrices by the conventional method. Is this case n multiplications are performed n^2 times, while $(n-1)$ additions are performed n^2 times, so $m(n) = n^2(n+(n-1)) = 2n^3 - n^2$. Then we have the cost of adding two n by n matrices together, $a(n)$, involves n^2 additions, so $a(n) = n^2$. With those two relationships established, we can begin our theoretical analysis of the ideal n_0 (ie if $n \leq n_0$, then perform naive multiplication instead of continuing Strassen's). Furthermore suppose that when it is theoretically equivalent to revert to the base case, and to continue with Strssen's, I want to choose n_0 such that I revert to the base case for the sake of simplicity. I will prove that the ideal n_0 occurs when the cost of performing naive matrix multiplication (call this $C_1(n)$) is equivalent to the cost of performing a Strassen reduction once while performing naive multiplication on the submatrices (ie perform 7 naive multiplications on matrices of size $n_0/2$ or $(n_0+1)/2$) then add together via the strassen relationships (call this cost $C_2(n)$):

Proof: Suppose that the ideal n_0 were less than the equivalence point between $C_1(n)$ and $C_2(n) \rightarrow C_2(n_0) > C_1(n_0)$ The since $C_1(n) - C_2(n)$ is negative up until a crossover point, then positive afterward, since on the whole the cost of naive matrix multiplication is increasing more quickly than the cost of one step of Strassen's then regular multiplication. Then we have that the ideal n_0 is some fixed value, so we can consider the way that our algorithm would function on $n_0 + 1$. Then we have that cost $(C(n))$ of $n_0 + 1$ is $C_2(n_0 + 1) \geq C_1(n_0 + 1)$, since because we are above n_0, but only by 1, the program will elect to do one step of Strassen's then compute, and because n_0 is chosen below the equivalence point we have that it is at least as expensive to perform a step of strassen's then compute then to just compute. Thus we are getting a suboptimal outcome for the computation of $C(n_0 + 1)$, since either it is costing more than a known lesser cost $(C_1(n_0 + 1))$, or the cost ie equivalent to the cost of non-recursive computation, which we would prefer to perform, if possible. Thus, we have that it is suboptimal to have $n_0 <$ than the equivalence point between C_1 and C_2.

Next, suppose that the ideal n_0 were greater than the equivalence point between $C_1(n)$ and $C_2(n)$. Then we have that $C(n_0) = C_1(n_0) > C_2(n_0)$, so we are getting a suboptimal computational cost on $C(n_0)$. Thus we have that n_0 must be $\leq$ the cross over points.

Thus by parts 1 and 2 of the above we have pigeonholed n_0 to be equivalent to the unique point at which

$C_1(n) = C_2(n)$ and $n > 0$, or if this point yield some $n \in \mathbb{Z}\backslash\mathbb{N}$, we take its floor.

Mathematically, in the case of multiplying two $n \times n$ matrices where $n\%2 = 0$ the cost of performing Strassen's once then multiplying naively is

$$C_2(n) = 7(2(n/2)^3 - (n/2)^2) + 18(n/2)^2$$

while the cost of just multiplying naively is

$$C_1(n) = 2n^3 - n^2$$

Setting the two equal we get that

$$C_2(n) = C_1(n) \to 7(2(n/2)^3 - (n/2)^2) + 18(n/2)^2 = 2n^3 - n^2 \to$$

$$\frac{7}{4}n^3 - \frac{7}{4}n^2 + \frac{9}{2}n^2 = 2n^3 - n^2$$

$n > 0$ so we can divide both sides by n^2 to get that

$$\frac{7}{4}n - \frac{7}{4} + \frac{9}{2} = 2n - 1 \to 7n + 11 = 8n - 4 \to$$

$$n = 15 \to \boxed{n_0 = 15}$$

This means when implementing Strassen's it is optimal to perform standard multiplication if given a matrix of even dimension of dimension less than or equal to 15.

Then in the other case of two $n \times n$ matrices such that $n\%2 = 1 \to$ n is odd, we have that if Strassen's is performed our matrix will be padded out to dimension $n + 1$ when the multiplication is performed. This means that our relation is slightly different, with

$$C_1(n) = 2n^3 - n^2 \quad C_2(n) = 7\big(2\big(\frac{n+1}{2}\big)^3 - \big(\frac{n+1}{2}\big)^2\big) + 18\big(\frac{n+1}{2}\big)^2$$

Setting the two equal, we get that

$$C_1(n) = C_2(n) \leftrightarrow 7\big(2\big(\frac{n+1}{2}\big)^3 - \big(\frac{n+1}{2}\big)^2\big) + 18\big(\frac{n+1}{2}\big)^2 - (2n^3 - n^2) = 0$$

Solving this equation numerically for n using Mathematica yields that $n = 37.1699 \to$ we take its floor to get that in the odd case that

$$\boxed{n_0} = 37$$

Thus it is optimal to perform standard multiplication on a given matrix of odd dimension n if $n \leq 37$.

Data storage In my program, I always stored matrices as two dimensional java arrays, which allowed for $O(1)$ access time for some fixed element in the array. Upon further consideration, using lists would have been an alternative, but would have added substantial complexity to my programming without a substantial reduction in the amount of memory being allocated (would've had made padding more efficient, but number of values needing to be stored in memory at any given point of time would be unchanged).

Optimizing caching on standard matrix multiplication Given that the standard matrix multiplication of a matrix A with $B \rightarrow AB$ is computed through three nested loops: The outside loop being the rows of A, the second loop being the columns of B, and the inner loop traverse over the row of A and down the column of B. Done naively, the innermost loop traverses over $B[*, x]$, which is inefficient traversal over memory. Then since the ordering of the outer two loop is arbitrary, I reordered my matrix multiplication such that the outermost loop traverses over the columns of B, at which point I store the entire column of B with which I am concerned in the inner 2 loops in a temporary array. This way within my inner loops (second loop traverses rows of A, third loop moves over given row) I am only traversing over the second index of my two dimensional array, and over my temporarily stored array. This yields superior caching performance as I am no longer moving around as much in memory when I perform my computation, since by the naive approach I was computing loops of the form $\beta[*, k]$ n^2 times, where $n = dimA = dimB$, but now I am only computing such loops n times in my optimization.

Optimizing Strassen's algorithm In order to make my Strassen's algorithm run as efficiently as possible, and given that the number of computations performed by Strassen's is fixed, I optimized by minimizing the amount of memory that Strassen's had to assign when run, ie how many new arrays Strassen's had to take the time to create in memory. Note: I wrote my program in Java. In order to minimize the amount of new memory assigned at each run, if my matrix was odd dimensional, I only padded it out with zeros to a matrix of dimension $n + 1$. This way, if my crossover point n_0, were, say 25, then if I were given a 49 dimensional matrix, I would only ever pad it out to 50, instead of all the way to 64, which would needlessly increase my run time. Furthermore, I modified my strassen function (called strass2n) and my matrix addition and subtraction functions such that I had the ability to pass the matrix in which the output is stored and returned as a passed in parameter to the function call. This way, I could use previously created arrays to store the outputs of my operations once I no longer required the information originally stored in the arrays. Lastly, I didn't ever save matrices $C_{11}, C_{12}, C_{21},$ and C_{22}, rather I computed each entry in my resultant matrix when it came up in my nested loop based on the appropriate sum of components in M_1 through M_7. In this manner, on each call of Strassens, I reduced the number of new arrays I create in memory from a naive 38 to 15, thereby improving the run time of my strassen algorithm. Furthermore, in the case of n being odd, instead of creating an entirely new padded out matrix, I instead added in extra zeros to my A_{11} through B_{22} matrices at the time of computation thereof, thus saving myself 1 array assignment in memory of size $n + 1 \times n + 1$ from a totally naive approach.

Experimental Crossover Point I computed the experimental crossover point based on the same paradigm as above, that is the ideal n_0 occurs when the cost of running naive matrix multiplication is equivalent to the cost of running strassen's once and then performing naive multiplication at the first recurssive step. I proved the correctness of this method for finding n_0 above, so it once again correct here. In the case of actual computation, "cost" is measured as the time it takes to run the algorithm. Then since n_0 is different for even and odd matrices, I first created a loop that started with $i = 12$ (to avoid false positives due to

”noise” in my computer”, able to do so since $n_0 = 15$ is the theoretical crossover point, and therefore a real life lower bound on the crossover point), and then while the time to multiply naively was less than the time to perform 1 level of strassens, I incremented i by 2. I then saved the i value of the crossover point, and then average this crossover point over $10,000$ iterations of my loop to get that my experimental crossover point over even dimensional matrices is $\boxed{58}$, whereas for odd dimensional matrices (same process, except starting with $i = 13$), I get that the ideal crossover point is $\boxed{73}$. The difference between the even and odd crossover points is then 15, which is less than the difference of 22 that I found in my analytical calculation of n_0. This makes sense, because what makes my experimental crossover points higher than my analytical ones is the amount of memory allocation and manipulation required by Strassen's which is not factored into my theoretical cost analysis; this extra cost is distributed roughly evenly over the even and odd cases of Strassen's, and since the run time of Strassen's is a cubic function of n, we get that adding cost to both cases actually reduces the overall difference between the cross over point of the even and odd cases.

Discussion of difficulties I initially attempted to analytically calculate the crossover point through complete recurrence relation for the number of operations required to run Strassens. While this was a nice idea in theory, in practice I was left with a relation and run time that was a function of both n and n_0, which made solving for the optimal n_0 difficult, if not impossible. In terms of programming, I struggled to reduce the number of new matrices initialized by Strassen's at each step, and during my first attempt at doing so I ruined my entire program by accidentally overwriting the information stored in matrices whose values I stilled need to calculate off of later in my program. This gave me some strange results, which took me a while to parse through. Lastly, creating a shell file and makefile for my java program to make it fit the run requirements was a new experience for me, and ultimately an educational one, though I struggled to correctly implement my shell file.