Optimization of Strassen's Matrix Multiplication Algorithm

Shreshth Rajan

March 2024

1 Implementation and Optimization

1.1 Matrix Multiplication Basics

The conventional matrix multiplication algorithm implemented the standard row-column method, where the dot product of corresponding row and column vectors computes each entry in the product matrix. This approach serves as a benchmark to underscore the efficiency gains provided by Strassen's algorithm.

1.2 Strassen's Algorithm Framework

Strassen's algorithm uses a recursive divide-and-conquer paradigm, enabling multiplication of large matrices by decomposing them into smaller sub-matrices. My implementation incorporates an adaptive padding function to handle matrices with odd dimensions, adding zeros as necessary to suit the requirements of the algorithm. This ensures the final product matrix aligns with the dimensions of the original matrices.

1.3 Addressing Odd Dimensionality

The initial padding strategy was refined from augmenting the matrix to the nearest power of two, to merely the nearest even number. This optimization was critical in reducing computational overhead, particularly for matrices just above a power of two in dimension.

1.4 Optimizations

1.4.1 Memory Optimization

I pre-allocated the result matrix to avoid dynamic resizing and iterated through it in an order more conducive to effective cache utilization.

1.4.2 Loop Order Reevaluation

Loop iteration, particularly in nested loops, was reordered to exploit data locality fully. The revised loop order ‘i, k, j‘ leverages sequential memory access patterns, crucial for performance improvement.

1.4.3 Minimizing Memory Access

Memory access within the innermost loop was minimized by holding the value of $matrix_a[i, k]$ in a temporary variable, reducing repeated and costly memory access operations.

1.4.4 Parallelization of Strassen’s Subproblems

To exploit concurrent execution capabilities, Strassen’s seven matrix subproblems, P1 to P7, were calculated in parallel using Python’s ‘ThreadPoolExecutor‘. This approach is particularly beneficial when dealing with large matrices, reducing overall computation time significantly.

1.4.5 The Cross-Over Point

(Detailed below) An analytical cross-over point was determined for switching from Strassen’s algorithm to the naive approach, optimizing performance based on matrix size. This cross-over point was empirically determined and represents a balance between the recursive overhead of Strassen’s algorithm and the computational simplicity of the naive method.

2 Crossover Point Analysis

2.1 Identifying Cross Over

The analysis began by defining the work functions for the standard matrix multiplication ($c_N(n)$) and for Strassen’s algorithm ($c_S(n)$) for matrices of size $n \times n$. In standard matrix multiplication, I compute each entry of the resulting matrix by performing n multiplications and $n-1$ additions. Thus, the work for each entry is $2n-1$ operations, and for the entire matrix of n^2 entries, I have:

$$c_N(n) = n^2(2n-1) = 2n^3 - n^2. \tag{1}$$

For Strassen’s algorithm, which divides each computation into seven smaller matrix multiplications of size $\frac{n}{2}$ and recombines them with additional additions and subtractions, the recursive relation is expressed as:

$$c_S(n) = 7c_S\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2. \tag{2}$$

Here, $c_S(1)$ represents the base case of a single multiplication operation. To determine the crossover point n_A, I equate $c_N(n_A)$ with $c_S(n_A)$ and solve for n_A, giving us the matrix size at which the two algorithms perform equivalently:

$$2n_A^3 - n_A^2 = 7c_S\left(\frac{n_A}{2}\right) + \frac{9}{2}n_A^2. \tag{3}$$

By simplifying the above equation and considering the base case $c_S(1) = 1$, I can analytically determine n_A as the smallest integer satisfying the equation:

$$n_A^2(n_A - 15) = 0. \tag{4}$$

Thus, I find $n_A = 15$ as the analytical crossover point, which means that for matrices of size less than 15, standard multiplication is more efficient.

When considering non-even values of n, I account for padding, and the recurrence for Strassen's algorithm is adjusted accordingly:

$$c_S(n) = 7c_S\left(\frac{n+1}{2}\right) + 18\left(\frac{n+1}{2}\right)^2. \tag{5}$$

For odd-sized matrices, I then find an analytical crossover point n_A^{odd} by setting $c_N(n_A^{odd})$ equal to $c_S(n_A^{odd})$ and solving for n_A^{odd}. Through this process, I estimate n_A^{odd} to be approximately 37, implying that for odd-sized matrices up to 37, standard multiplication is likely to be more efficient than Strassen's algorithm, given the increased work due to padding and division of subproblems.

In meta, the theoretical analysis identifies two crossover points—one for even-sized matrices and one for odd-sized matrices—beyond which Strassen's algorithm becomes more efficient than the naive approach.

2.2 Experimental Analysis

My methodology for finding the empirical crossover point involved a sequence of matrix multiplication trials across a range of matrix sizes. I aimed to identify the specific size where the hybrid Strassen-traditional multiplication method outpaces the conventional approach in execution time. As aforementioned, this size defines the crossover point.

2.3 Experimentation and Findings

In practice, I conducted timed multiplication trials for matrix sizes within an interval suspected to contain the crossover point, based on the analytical predictions. The interval's lower limit was set just above the theoretical crossover, while the upper limit was selected to guarantee the superiority of Strassen's method.

2.3.1 Procedure

For each matrix size n within the interval:

1. Generate two random $n \times n$ matrices.

2. Compute their product using the traditional $O(n^3)$ algorithm, timing the operation.
3. Apply Strassen's algorithm with a single recursion level, timing this as well.
4. Evaluate a hybrid algorithm that combines one recursion level of Strassen's method with traditional multiplication on the submatrices.
5. Record and compare execution times.

Multiple trials were ran for each n and averaged to produce the data on hand. My results are as follows (details omitted for sake of space):

Matrix Size n	**Naive Time**	**Strassen Time**
n = 58	0.189	0.191
n = 60	0.193	0.195
n = 62	0.209	0.210
n = 64	0.200	0.198
n = 66	0.214	0.206
n = 68	0.218	0.209
n = 70	0.224	0.212

Table 1: Comparison of Naive and Strassen Matrix Multiplication Times

2.4 Conclusion

The experimental analysis identified the crossover point at $n = 64$. This framework confirmed the limitations of the theoretical model, suggesting that constant-time operations are not a practical assumption for Strassen's algorithm.

3 Application: Triangles in Undirected Graphs

In this section, I delve into an application of the optimized matrix multiplication program by estimating the count of triangles in an undirected graph. The probability of an edge's presence is denoted as p, with p taking values in the set $\{0.01, 0.02, 0.03, 0.04, 0.05\}$. I employ an adjacency matrix to represent the graph. Due to the undirected nature of the graph, the adjacency matrix is symmetric. This symmetry was exploited by generating only the upper triangular portion of the $n \times n$ matrix, then mirroring it across the diagonal to obtain the complete matrix, ensuring all diagonal elements are zero to exclude self-loops.

Upon constructing the adjacency matrix, my application of Strassen's algorithm cubes the matrix. The trace of the cubed matrix—divided by 6—yields the total count of triangles in the graph, as mentioned in the instructions. The empirical results are compared against theoretical expectations in Table 2.

Probability (p)	Actual Value	Theoretical Value ($p^3 \times 10243$)
0.01	171	178.43
0.02	1394	1427.46
0.03	4862	4817.69
0.04	11280	11419.71
0.05	22910	22304.13

Table 2: Comparison of actual triangle counts to theoretical expectations.

The theoretical value represents the expected number of triangles in a graph with 1024 vertices, while the actual value is the count obtained through the experimental procedure. The consistency between the actual and expected figures corroborates the soundness of the approach. However, some discrepancies are observed, attributable to several factors:

- Variability inherent to any probabilistic model, with each generated graph being a sample from a broader distribution that should converge on the expected value, though individual instances may deviate.
- Finite size of the graphs in contrast to the infinite size assumed in theoretical models, affecting the accuracy of theoretical predictions, particularly for smaller-sized graphs.
- Assumption of edge independence, which may not hold perfectly in practice, with the possibility that the existence of certain edges influences the presence of others, deviating from the ideal of a random graph.