Comparison of Matrix Multiplication Algorithms

Anand Patel 2561034

Due: 25 August 2025

1 Introduction

1.1 Background

Matrix multiplication is a core operation in computing, with applications in graphics, machine learning, and high-performance computing. This report analyzes three square matrix-multiplication algorithms from CLRS (Ch. 4) and compares their empirical runtimes.

1.2 Theoretical complexity

We study the following algorithms on $n \times n$ matrices:

1. **Square-Matrix-Multiply** (naive triple loop): $\Theta(n^3)$.
2. **Square-Matrix-Multiply-Recursive** (divide and conquer):

$$T(n) = 8\,T\Big(\frac{n}{2}\Big) + \Theta(n^2) = \Theta(n^3).$$

3. **Strassen's algorithm** (7 recursive products):

$$T(n) = 7\,T\Big(\frac{n}{2}\Big) + \Theta(n^2) = \Theta\big(n^{\log_2 7}\big) \approx \Theta(n^{2.81}).$$

The goal is to examine how these asymptotic bounds are represented in real runtime measurements.

2 Implementations

2.1 Implementation approach

- **Language.** C, for predictable performance and explicit memory control.
- **Data layout.** Matrices are stored as row-major 1D arrays (`int*`). Element (i, j) is accessed via `idx(i,j,n) = i*n + j`.

- **Submatrices.** Quadrants are addressed with pointer arithmetic and stride parameters, avoiding copies when recursing.
- **Value type.** `int` values are used to avoid floating-point overhead.
- **Temporaries (Strassen).** Each recursion level allocates the ten S matrices, the seven P matrices, and two scratch buffers, and frees them after combination.

2.2 Naive Algorithm

Algorithm 1: Naive Square Matrix Multiply

```
1 for i ← 1 to n do
2 |   for j ← 1 to n do
3 |   |   C[i, j] ← 0;
4 |   |   for k ← 1 to n do
5 |   |   |   C[i, j] ← C[i, j] + A[i, k] · B[k, j];
6 |   |   end
7 |   end
8 end
```

Implementation Details:

- Direct use of the mathematical definition: $C[i,j] = \sum_{k=0}^{n-1} A[i,k] \times B[k,j]$
- Initializes result matrix elements to zero before accumulation
- Uses the `idx` macro for 1D array indexing
- No special cases or optimizations

Complexity Analysis:

- Time: $O(n^3)$ - Each of the three loops runs n times
- Space: $O(1)$ - Only uses input/output matrices, no additional space
- Cache behavior: Good locality when accessing matrix C and A by rows

2.3 Recursive Algorithm

The recursive algorithm divides each matrix into four quadrants and computes the result through eight recursive multiplications:

Algorithm 2: Recursive Square Matrix Multiply

Input: Matrices A, B of dimension $n \times n$
Output: Matrix $C = A \cdot B$

if $n = 1$ **then**
  $C[1,1] \leftarrow C[1,1] + A[1,1] \cdot B[1,1]$;
  return
end
Divide A, B, and C into four $\frac{n}{2} \times \frac{n}{2}$ blocks:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \quad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Recursive calls:

$$\begin{aligned} C_{11} &\leftarrow C_{11} + \text{Recursive}(A_{11}, B_{11}) + \text{Recursive}(A_{12}, B_{21}) \\ C_{12} &\leftarrow C_{12} + \text{Recursive}(A_{11}, B_{12}) + \text{Recursive}(A_{12}, B_{22}) \\ C_{21} &\leftarrow C_{21} + \text{Recursive}(A_{21}, B_{11}) + \text{Recursive}(A_{22}, B_{21}) \\ C_{22} &\leftarrow C_{22} + \text{Recursive}(A_{21}, B_{12}) + \text{Recursive}(A_{22}, B_{22}) \end{aligned}$$

Implementation Details:

- Uses pointer arithmetic to access submatrices without copying data
- Stride parameters handle non-contiguous submatrix access
- Accumulates results with `+=` to handle multiple contributions to each element
- Base case handles single element multiplication

Complexity Analysis:

- Time: $T(n) = 8T(n/2) + O(n^2) = O(n^3)$ from Master Theorem
- Space: $O(\log n)$ for recursion stack depth
- Despite divide-and-conquer approach, same asymptotic complexity as naive

2.4 Strassen's Algorithm

Strassen's algorithm reduces the number of recursive multiplications from 8 to 7 through algebraic manipulation:

Algorithm 3: Strassen Matrix Multiply

Input : Matrices $A, B \in \mathbb{R}^{n \times n}$ (with n a power of two)
Output: Matrix $C = A \cdot B$

if $n = 1$ **then**
  $C[1,1] \leftarrow A[1,1] \cdot B[1,1]$; **return**
end
Partition A into blocks $A_{11}, A_{12}, A_{21}, A_{22}$ (each $(n/2) \times (n/2)$);
Partition B into blocks $B_{11}, B_{12}, B_{21}, B_{22}$;
Partition C into blocks $C_{11}, C_{12}, C_{21}, C_{22}$;
$S_1 \leftarrow B_{12} - B_{22}$;
$S_2 \leftarrow A_{11} + A_{12}$;
$S_3 \leftarrow A_{21} + A_{22}$;
$S_4 \leftarrow B_{21} - B_{11}$;
$S_5 \leftarrow A_{11} + A_{22}$;
$S_6 \leftarrow B_{11} + B_{22}$;
$S_7 \leftarrow A_{12} - A_{22}$;
$S_8 \leftarrow B_{21} + B_{22}$;
$S_9 \leftarrow A_{11} - A_{21}$;
$S_{10} \leftarrow B_{11} + B_{12}$;
$P_1 \leftarrow \text{Strassen}(A_{11}, S_1)$;
$P_2 \leftarrow \text{Strassen}(S_2, B_{22})$;
$P_3 \leftarrow \text{Strassen}(S_3, B_{11})$;
$P_4 \leftarrow \text{Strassen}(A_{22}, S_4)$;
$P_5 \leftarrow \text{Strassen}(S_5, S_6)$;
$P_6 \leftarrow \text{Strassen}(S_7, S_8)$;
$P_7 \leftarrow \text{Strassen}(S_9, S_{10})$;
$C_{11} \leftarrow P_5 + P_4 - P_2 + P_6$;
$C_{12} \leftarrow P_1 + P_2$;
$C_{21} \leftarrow P_3 + P_4$;
$C_{22} \leftarrow P_5 + P_1 - P_3 - P_7$;

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \quad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}.$$

Implementation Details:

- Allocates 19 temporary matrices (10 S-matrices, 7 P-matrices, 2 temporaries) per recursion level
- Follows CLRS formulation exactly with no optimizations
- Recursion continues down to $n = 1$ base case
- Extensive memory allocation/deallocation overhead

Complexity Analysis:

- Time: $T(n) = 7T(n/2) + O(n^2) = O(n^{\log_2 7}) \approx O(n^{2.81})$
- Space: $O(n^2)$ at each level $\times$ $O(\log n)$ levels
- Constant factor overhead dominates for real-world matrix sizes

2.5 Padding Strategy

Since the recursive and Strassen algorithms require matrices with power-of-2 dimensions, a padding strategy is implemented for arbitrary-sized inputs:

Algorithm 4: Pad-To-Power-Of-Two

Input : Matrix $M \in \mathbb{R}^{n \times n}$
Output: Padded matrix $P \in \mathbb{R}^{m \times m}$ where $m = \min\{2^k \mid 2^k \geq n\}$

```
m ← 1;
while m < n do
    m ← 2 · m
end
Allocate P as an m × m zero matrix;
for i ← 1 to n do
    for j ← 1 to n do
        P[i, j] ← M[i, j];
    end
end
return P;
```

Algorithm 5: Unpad

Input : Matrix $P \in \mathbb{R}^{m \times m}$ (with $m \geq n$), target size n
Output: Matrix $U \in \mathbb{R}^{n \times n}$ extracted from the top-left block of P

```
Allocate U as an n × n zero matrix;
for i ← 1 to n do
    for j ← 1 to n do
        U[i, j] ← P[i, j];
    end
end
return U;
```

Implementation Details:

- Finds the smallest power of 2 greater than or equal to n n
- Allocates new matrix initialized to zeros using `calloc`
- Copies original matrix values to top-left corner
- Unpadding extracts the $n \times n$ result from the padded computation

Impact on Performance:

- Best case: $n = 2^k$ requires no padding
- Worst case: $n = 2^k + 1$ nearly doubles the work (e.g., 2049 $\rightarrow$ 4096)
- Additional memory allocation and copying overhead

3 Experimental Methodology

3.1 Test Environment

- **Hardware**: Apple M4 Pro (14 cores: 10 performance, 4 efficiency), 24GB unified memory
- **Compiler**: GCC 15.1.0 (Homebrew) with optimization level -O2
- **Operating System**: macOS Tahoe 26
- **Timing Method**: C `clock()` function measuring CPU time

3.2 Input Generation

Matrix Dimensions Selected:

- Powers of 2: 2^k for $k = 1, 2, ..., 12$ (2 through 4096)
- Near powers: $2^k - 3$ and $2^k + 5$ to test padding impact
- Range shows clear performance trends while within a reasonable runtime

Matrix Generation:

- Random integer values between 1 and 100
- Generated using `rand()` with time-based seed
- New random matrices generated for each trial

3.3 Timing Methodology

- 5 trials per algorithm per matrix dimension
- Matrix generation not included in timing measurements
- Only the multiplication operation is timed
- Results averaged across trials to reduce variance
- Padding operations excluded from Strassen and recursive timing
- All algorithms tested on identical input matrices per trial

4 Results and Analysis

4.1 Performance Overview

Table 1 shows average runtimes for the three algorithms on power-of-two sizes. As expected, the naive and recursive algorithms grow cubically, while Strassen has a better asymptotic bound but performs worse in practice due to constant-factor overhead.

Matrix Size	Naive (ms)	Recursive (ms)	Strassen (ms)
1024	133.4	141.2	1 221.5
2048	1 064.8	1 129.5	9 774.4
4096	168 080.8	174 742.3	503 366.2

Table 1: Average runtime for power-of-2 matrices (5 trials each).

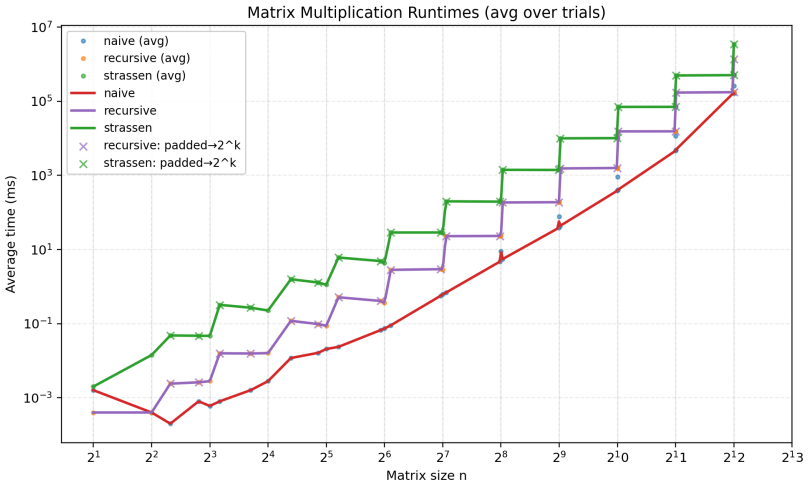

Figure 1: Matrix multiplication runtime comparison on log-log scale. The step patterns for recursive and Strassen algorithms indicate padding to power-of-2 dimensions.

Figure 2 provides additional insights into algorithm performance. The normalized growth rate plot (b) confirms the theoretical complexities: naive and recursive algorithms show nearly constant values when divided by n^3, validating their $O(n^3)$ behavior. Strassen's decreasing trend when normalized confirms its sub-cubic growth of $O(n^{2.81})$.

The slowdown factor plot (c) quantifies Strassen's overhead, showing it runs $9\times$ slower at $n = 1024$ but improves to only $3\times$ slower at $n = 4096$. The trend line suggests Strassen would achieve parity with naive around $n \approx 16,000$, well beyond our test range.

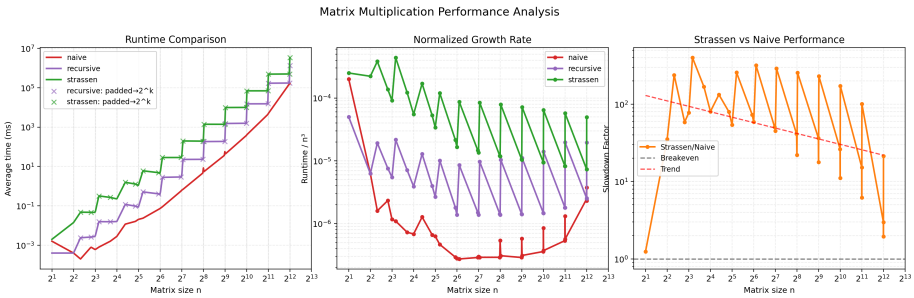

Figure 2: Extended performance analysis: (a) Runtime comparison with padding indicators, (b) Normalized growth rates showing constant factors, (c) Strassen slowdown factor demonstrating the crossover trend.

Key observations:

- The naive algorithm consistently outperforms the recursive version (6–8% faster), despite identical $O(n^3)$ complexity.
- Strassen is significantly slower in this range: up to $9\times$ slower at $n = 1024$.
- Step patterns in recursive and Strassen results match padding to the nearest 2^k.

4.2 Naive Algorithm

The naive algorithm exhibits smooth cubic growth, excellent cache locality (row-wise access in both A and C), and no padding overhead. Its simplicity makes it the best in practice with our testing range.

4.3 Recursive Algorithm

Despite equal asymptotic complexity, recursion adds overhead:

- Function call overhead accumulates over $\log_2 n$ recursion depth.
- Strided submatrix access reduces cache locality compared to the naive kernel.
- Padding introduces discontinuities at non-powers of two.

4.4 Strassen's Algorithm

While Strassen reduces multiplications from 8 to 7, overhead dominates:

- At $n = 4096$, Strassen is still $3\times$ slower than naive.

- Memory usage: each recursion level allocates 19 temporaries, but **peak live space is** $O(n^2)$, not $O(n^2 \log n)$.
- Total allocations for $n = 4096$: $\sum_{i=0}^{11} 19 \times 7^i = 19 \times \frac{7^{12}-1}{6} \approx 4.38 \times 10^{10}$ allocations

4.5 Padding Impact

Table 2 illustrates padding overhead. The worst case ($n = 2^k+1$) nearly doubles work. In our dataset, $n = 2049$ was padded to 4096, and runtime almost doubled compared to $n = 2048$.

Original Size	Padded Size	Work Increase	Runtime Impact
2045	2048	0.3%	Minimal
2049	4096	99.9%	Nearly doubles
4093	4096	0.2%	Negligible

Table 2: Padding overhead for non-power-of-2 matrices.

4.6 Theory vs Practice

Asymptotics explain growth rates but hide constants. In practice:

- Naive has fewer instructions and superior cache locality, so it dominates.
- Recursive is penalized by call overhead and strided memory access.
- Strassen trades multiplications for additions and memory traffic. The asymptotic $O(n^{2.81})$ is visible only beyond our tested range ($n \gtrsim 16{,}000$ by extrapolation).

4.7 Memory Layout Effects

Our 1D row-major layout favors the naive method, which accesses rows contiguously. Recursive and Strassen introduce non-contiguous sub-blocks, leading to cache misses. This explains why, despite the same big-O, recursive is slower than naive.

4.8 Memory Overhead in Strassen

Strassen's asymptotic improvement is offset by massive allocation churn:

- Peak live space: $O(n^2)$, but repeated alloc/free dominates runtime.
- Memory bandwidth, not arithmetic, is the bottleneck.

4.9 Crossover Point Estimation

Extrapolating log–log slopes shows Strassen would overtake naive only at matrix sizes well beyond $n = 16{,}000$ on this hardware. For all practical sizes, naive remains superior.

5 Discussion

5.1 Theory vs. Practice

The results highlight a fundamental lesson in algorithm analysis: asymptotic notation hides significant constant factors. While Strassen's $O(n^{2.81})$ is theoretically better than $O(n^3)$, our results show:

- Strassen's constant factor overhead dominates until very large n
- For $n = 4096$: Naive takes ≈ 69 billion operations, Strassen takes ≈ 58 billion multiplications but adds billions of additions and memory operations
- The "break-even" point where Strassen outperforms naive is not reached in our test range

5.2 Memory Overhead Analysis

Strassen's memory requirements are substantial:

- Memory per level: 19 matrices of size $(n/2^i)^2$ at recursion depth i
- Total allocations for $n = 4096$: $\sum_{i=0}^{11} 19 \times 7^i = 19 \times \frac{7^{12}-1}{6} \approx 2.6$ billion allocations
- Each allocation/deallocation has system overhead
- Memory bandwidth becomes the bottleneck, not computation

5.3 Recursion Depth Impact

- Recursion depth: $\log_2 n$
- Leaf nodes (base cases): $7^{\log_2 n} = n^{\log_2 7} \approx n^{2.81}$
- Function call overhead accumulates dramatically
- Stack space requirements: $O(\log n)$ but with large constants

5.4 Crossover Point Estimation

Based on the growth trends, we can estimate when Strassen might become practical:

- Current slowdown factor decreases as n increases
- Extrapolating the graph shows crossover around $n \approx 16,384$ to $32,768$
- Implementation without optimisations may never be practical for real use

6 Optimization Opportunities (Not Implemented)

6.1 Crossover Threshold

The most impactful optimization would be switching to naive multiplication below a threshold:

- Typical thresholds: 32-128
- Would remove billions of small recursive calls
- Maintains $O(n^{2.81})$ asymptotic complexity

6.2 Memory Pool Management

Pre-allocating temporary matrices could reduce allocation overhead:

- Single allocation of workspace at start
- Reuse matrices across recursion levels
- Eliminate malloc/free overhead
- Improve cache locality

6.3 Cache Optimization

Current implementation suffers from poor cache usage:

- Strided access patterns miss cache lines
- Small submatrices at deep recursion don't utilize cache blocks
- Tiling or blocking could improve memory bandwidth utilization
- Loop reordering in add/subtract operations could help

7 Conclusions

7.1 Key Findings

- All three algorithms demonstrate their theoretical complexity: $O(n^3)$ for naive/recursive, $O(n^{2.81})$ for Strassen
- Strassen's constant factor overhead makes it impractical without optimization
- Padding to power-of-2 creates significant performance penalties
- Memory allocation dominates Strassen's runtime, not arithmetic operations

7.2 Learnings

- Asymptotic notation abstracts away critical performance factors
- Implementation details can overwhelm algorithmic advantages
- "Faster" algorithms aren't always faster in practice

7.3 Areas for improvement

- Use naive multiplication for matrices smaller than $n = 10000$
- Implement crossover thresholds when using Strassen