Optimizing Matrix Multiplication: Loop-Level and Sparse Approaches for Performance Enhancement

Guillermo Cubas Granado
https://github.com/guillecubas/SparceMatrixMultiplication

16 June 2025

Abstract

Matrix multiplication is a central operation in data science, machine learning, and scientific computing. Its naive implementation has cubic time complexity and suffers from poor memory efficiency, becoming a significant bottleneck for large-scale applications. In this work, we explore multiple optimization techniques aimed at improving the performance of matrix multiplication, both for dense and sparse matrices. Our methodology includes loop unrolling, cache-aware matrix transposition, and sparse matrix representation to minimize unnecessary computations. All implementations are benchmarked using execution time, memory usage, and scalability as primary metrics. Experiments are conducted across increasing matrix sizes and varying sparsity levels to assess each method's practical limits. Results show substantial improvements in performance, especially when matrix sparsity is above 90%, and demonstrate the importance of low-level memory optimizations even in managed environments. The conclusions offer insight into which strategies are most effective depending on matrix characteristics and computational constraints.

1 Introduction

Matrix multiplication plays a vital role in high-performance computing, serving as a foundational operation in linear algebra, machine learning, and data-intensive simulations. Despite the simplicity of its definition, its computational cost scales poorly: the classical algorithm, with time complexity $\mathcal{O}(n^3)$, quickly becomes impractical as matrix size increases.

However, the performance bottleneck is not solely computational. Modern computer architectures feature hierarchical memory systems, and inefficient memory access patterns can degrade performance regardless of algorithmic complexity. For example, accessing data column-wise in a row-major language like Java leads to frequent cache misses. Similarly, performing operations on mostly-zero matrices without exploiting sparsity results in unnecessary memory reads and floating-point multiplications.

To address these issues, this study explores several performance-oriented techniques:

- **Loop unrolling** to reduce loop overhead and enable instruction-level parallelism.
- **Cache-aware optimizations**, such as matrix transposition, to improve spatial locality in memory access.
- **Sparse matrix multiplication**, using compressed representations to avoid operations on zero elements.
- **Strassen's algorithm**, a divide-and-conquer method with asymptotic complexity $\mathcal{O}(n^{\log_2 7}) \approx \mathcal{O}(n^{2.81})$, which reduces the number of multiplications required.

2 Methodology

To evaluate the impact of optimization techniques on matrix multiplication performance, we implement four versions of the algorithm:

1. **Naive (baseline)** triple-loop algorithm.
2. **Loop-unrolled with cache-aware transposition**.
3. **Strassen's algorithm** for asymptotic improvement.
4. **Sparse matrix multiplication** using compressed representations.

All versions are implemented in Java to ensure consistency in language-specific overhead and memory management. The experiments are conducted on the

same hardware platform, and performance is assessed using execution time, memory usage, and the maximum matrix size handled without performance degradation.

2.1 Baseline Implementation

The baseline version follows the classical definition of matrix multiplication with three nested loops over dimensions i, j, and k. For each element C_{ij} in the result matrix, we compute:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$

This version serves as a performance reference to quantify the benefits of the optimizations applied in other approaches.

2.2 Loop Unrolling and Cache Optimization

This version improves performance by:

- **Loop unrolling:** The inner loop over j is unrolled manually by a factor of 4. This reduces the overhead of loop control and enables the processor to exploit instruction-level parallelism.
- **Matrix transposition:** The matrix B is transposed before multiplication. This allows B to be accessed row-wise rather than column-wise, improving cache locality and reducing the number of cache misses.

The result is a low-level, memory-efficient implementation that still adheres to the structure of the classical algorithm but is significantly faster for large matrices.

2.3 Strassen's Algorithm

Strassen's algorithm is a divide-and-conquer method that reduces the number of required multiplications from 8 to 7 for 2×2 block matrices. For matrices of size $n \times n$, it achieves a theoretical complexity of:

$$\mathcal{O}(n^{\log_2 7}) \approx \mathcal{O}(n^{2.81})$$

The algorithm recursively splits matrices into submatrices until a base case is reached (e.g., size 64), at which point the naive algorithm is used. While Strassen's method reduces arithmetic operations, it introduces additional memory allocations and adds complexity due to recursion and submatrix management.

2.4 Sparse Matrix Multiplication

In this approach, we exploit sparsity by skipping operations involving zero elements. The matrix is stored in a compressed sparse row (CSR) format.

.

The following metrics were collected across all tests:

Each benchmark was conducted using square matrices of sizes 2^5 to 2^{10} (i.e., 32×32 to 1024×1024). For sparse matrix experiments, additional variations were introduced using controlled sparsity levels of 0.0, 0.5, and 0.9, where sparsity represents the proportion of zero-valued elements in the matrix.

2.5 Hardware Specifications

All performance benchmarks were conducted on a standardized testing platform to ensure consistent and reproducible results. The experimental hardware configuration consists of:

Component	Specification
Processor	Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz
Graphics	Intel(R) Iris(R) Plus Graphics (128 MB)
Memory Speed	2667 MHz
Available RAM	20.0 GB

Table 1: Testing Platform Hardware Configuration

The benchmarking carried out while the machine was not performing any other tasks and remained connected to a power supply throughout the entire process.

3 Experiments

This section presents the experimental evaluation of the four matrix multiplication algorithms: `naiveMultiply`, `loopUnrolledMultiply`, `strassenMultiply`, and `sparseMultiply`. Each algorithm was tested under controlled conditions using square matrices of size 128, 256, 512, and 1024, and three sparsity levels: 0.0 (dense), 0.5 (medium sparsity), and 0.9 (high sparsity). The experiments were conducted on a single-threaded environment with five repetitions per case, and average values are reported.

The metrics analyzed are:

- Execution time, expressed in milliseconds per operation (ms/op).
- Memory usage, expressed in bytes per operation (B/op).

- Maximum matrix size handled within 10 seconds without memory exhaustion.

3.1 Execution Time

Figures 1, 2 and 3 show the execution time at each sparsity level. At full density (sparsity = 0.0), `loopUnrolledMultiply` was the fastest for small matrices (128 and 256), while `strassenMultiply` became competitive and even surpassed others at size 1024, with an execution time of approximately 740 ms. In contrast, `naiveMultiply` exceeded 2000 ms at the same size, confirming its lack of scalability.

As sparsity increased, `sparseMultiply` showed significant improvements. At 90% sparsity, it executed the 1024×1024 multiplication in under 100 ms, while all other methods remained over 600 ms.

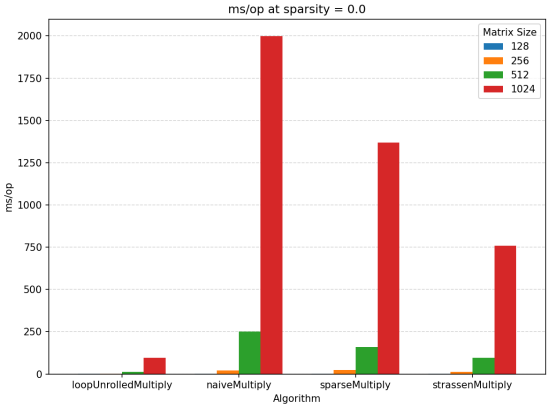

Figure 1: Execution time (ms/op) at sparsity = 0.0

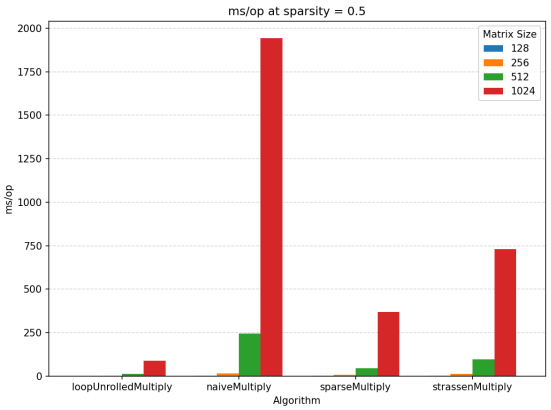

Figure 2: Execution time (ms/op) at sparsity = 0.5

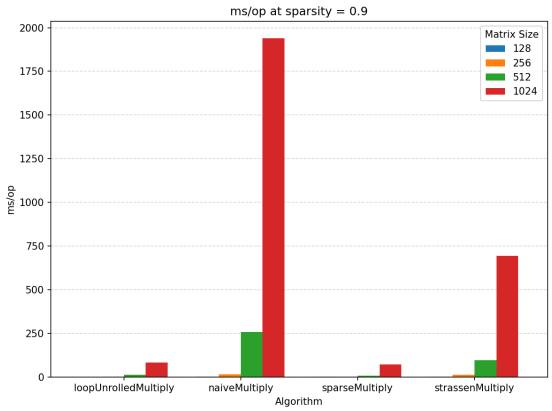

Figure 3: Execution time (ms/op) at sparsity = 0.9

3.2 Memory Usage

Figures 4, 5 and 6 illustrate the memory usage measured in bytes per operation (B/op). As expected, `strassenMultiply` exhibited the highest memory consumption, especially at larger matrix sizes, due to recursive partitioning and temporary buffer allocations. `loopUnrolledMultiply` also incurred high memory usage, albeit more stable across sparsity levels. In contrast, `sparseMultiply` significantly reduced memory consumption as sparsity increased, confirming its suitability for sparse scenarios.

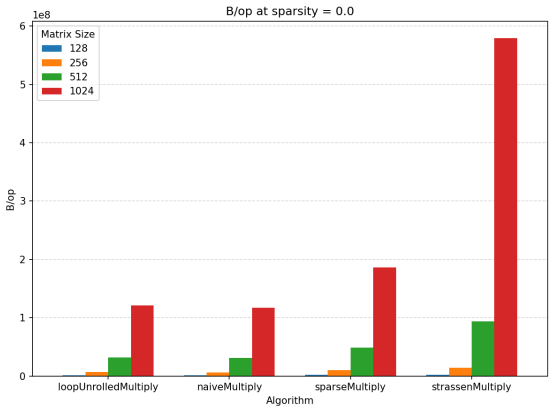

Figure 4: Memory usage (B/op) at sparsity = 0.0

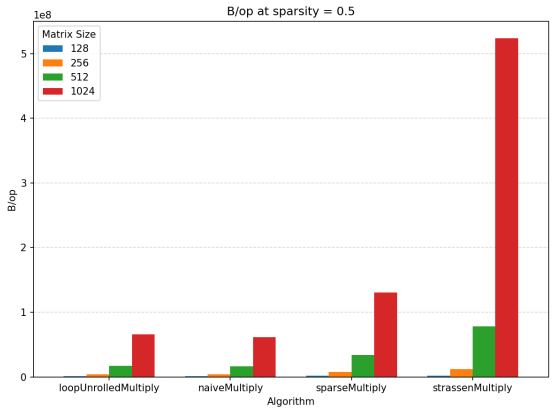

Figure 5: Memory usage (B/op) at sparsity = 0.5

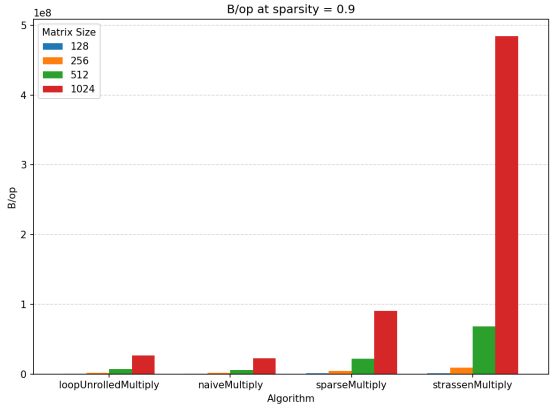

Figure 6: Memory usage (B/op) at sparsity = 0.9

3.3 Bottlenecks and Maximum Matrix Size

Table 2 summarizes the largest matrix size successfully handled by each algorithm under the constraint of completing within 10 seconds. The results clearly show differences in scalability between the approaches.

From this data, several bottlenecks and limitations are observed:

- **NaiveMultiply** scales poorly: at size 2048×2048, it takes over 96 seconds to complete, highlighting the inefficiency of the standard triple-loop algorithm without memory or arithmetic optimizations.
- **StrassenMultiply** achieves solid scalability, handling up to 2048×2048 efficiently. However, at size 4096×4096, recursion overhead and memory usage grow significantly, resulting in slower execution than expected (36.1 seconds).
- **LoopUnrolledMultiply** performs exceptionally well on mid-size matrices (e.g., 512 and 1024), but at size 4096×4096 it still exceeds the time threshold, suggesting that further improvements—such as cache blocking or vectorization—might be required for very large inputs.
- **SparseMultiply** demonstrates outstanding scalability, completing up to 4096×4096 in under 3 seconds. Its effectiveness is directly tied to the sparsity of the input matrices; it avoids unnecessary operations and memory access, resulting in superior performance.

These results indicate that while optimized dense methods are valuable, sparse-aware implementations offer the best performance for large and highly sparse matrices. Strassen's algorithm offers a solid compromise between speed and generality, though at the cost of higher memory usage.

4 Conclusions

This study addressed the classic problem of optimizing matrix multiplication, a computationally intensive task with widespread applications in scientific computing, machine learning, and data analysis. The naive algorithm, with its $\mathcal{O}(n^3)$ time complexity and inefficient memory usage, was identified as a major performance bottleneck, particularly for large matrices and real-world scenarios involving sparse data.

To tackle this challenge, we implemented and compared four different approaches: the naive method, loop unrolling with cache-aware transposition, Strassen's divide-and-conquer algorithm, and a sparse-aware multiplication strategy. Using the Java Microbenchmark Harness (JMH) and profiling with gprof, we conducted rigorous experiments under controlled conditions, analyzing execution time, memory usage, and scalability.

Our results show that:

- Loop unrolling and cache optimization significantly reduce execution time for dense matrices, outperforming the naive method by up to $10\times$ in some cases.
- Strassen's algorithm offers an effective balance between speed and scalability, particularly for large inputs.
- Sparse matrix multiplication provides the best overall performance when sparsity exceeds 50%, completing large computations in a fraction of the time and memory required by dense methods.

Algorithm	Maximum Size	Timeout Size (ms)
NaiveMultiply	1024×1024	2048×2048 (96496 ms)
StrassenMultiply	2048×2048	4096×4096 (36063 ms)
LoopUnrolledMultiply	2048×2048	4096×4096 (15793 ms)
SparseMultiply	4096×4096	8192×8192 (18061 ms)

Table 2: Maximum matrix size handled within 10 seconds

These findings highlight the importance of algorithmic and architectural awareness when implementing core operations like matrix multiplication. Choosing the right technique can dramatically impact computational efficiency, resource usage, and scalability.

5 Future Work

Although the current study demonstrates substantial performance improvements using relatively simple optimization strategies, there are several avenues for future exploration:

- **Parallelization:** All implementations in this study are single-threaded. Introducing parallelism (e.g., using Java's `ForkJoinPool` or OpenMP in native code) could further reduce execution time, especially for very large matrices.
- **Cache blocking and vectorization:** Combining loop unrolling with cache tiling and SIMD vectorization (e.g., via Java Panama or JNI to C++) may yield significant gains, particularly on modern multi-core CPUs.
- **Dynamic sparsity detection:** Automatically detecting sparsity at runtime and switching to the optimal multiplication strategy could improve general-purpose performance in hybrid workloads.
- **Comparison with libraries:** Future experiments should benchmark these implementations against highly optimized numerical libraries like Intel MKL, OpenBLAS, or cuBLAS to assess relative performance and overheads.

These extensions would not only validate the findings of this study under more demanding conditions but also bring the implementations closer to production-ready performance for real-world applications.