Performance Benchmark of matrix multiplication

Sofía Travieso García - GitHub
University of Las Palmas de Gran Canaria
sofia.travieso101@alu.ulpgc.es

1 Abstract

This paper presents a comparative performance analysis of three distinct matrix multiplication optimization strategies: Strassen's algorithm (algorithmic optimization, $O(n^{2.807})$), Tiling/Blocking (cache optimization), and Sparse Matrix-Vector Multiplication (SpMV) (data structure optimization). Benchmarks were executed across C, Java, and Python using matrices up to 2048×2048. The proposed methods were evaluated based on execution time, memory usage, and scalability. Results show that the Sparse approach, leveraging 99% sparsity, yields orders of magnitude speedup and minimal memory footprint. The findings highlight the significant memory overhead of Strassen's recursive approach, especially in C, in contrast to the cache-efficient Tiling method.

2 Keywords

- Matrix Multiplication Optimization
- Strassen Algorithm
- Cache Tiling
- Sparse Matrices (SpMV)
- High-Performance Computing

3 Introduction

Matrix multiplication is fundamental in numerous computational domains, including machine learning and scientific simulation. While the basic approach is $O(n^3)$, real-world performance is significantly influenced by optimizations. This paper focuses on two principal methods for dense matrices—algorithmic reduction via Strassen's algorithm and hardware optimization via Tiling—and contrasts them with the structure-based optimization inherent in Sparse Matrix-Vector Multiplication (SpMV). The aim of this paper is to quantify the practical differences in execution time, memory footprint, and scalability potential across implementations in C, Java, and Python.

4 Problem Statement

The problem addressed is to determine the relative computational efficiency and resource requirements of optimized matrix multiplication techniques compared to a structure-based approach (SpMV). Specifically, the study evaluates: Strassen's algorithm ($O(n^{2.807})$); Tiling optimization ($O(n^3)$ with improved cache utilization); and Sparse Matrix-Vector Multiplication (SpMV). The comparison is formalized across varying matrix sizes ($N \times N$) in C, Java, and Python to identify performance bottlenecks and optimal approaches for Big Data computation.

5 Methodology

5.1 Implementation Details

The benchmarking codes were extended from the previous naive $O(n^3)$ implementations to include three optimized approaches:

- **Strassen's Algorithm:** Implemented recursively with a suitable cutoff (where basic multiplication is faster). Strassen's algorithm reduces the number of multiplications from eight to seven, achieving $O(n^{\log_2 7})$.
- **Tiling (Cache Optimization):** Implemented using a block size ($B = 64$) to improve data locality and reuse within the CPU cache hierarchy, minimizing expensive main memory access.
- **Sparse SpMV:** The multiplication simulates a sparse matrix of 99% sparsity. This approach only performs computations on non-zero entries, utilizing specialized data structures (like CSR) for minimal storage.

5.2 Experimental Setup

All experiments were conducted on a personal laptop with the following specifications:

- **Device:** Lenovo Legion 5 15IAH7H
- **Processor:** 12th Gen Intel® Core™ i7-12700H @ 2.30 GHz (14 cores, 20 threads)
- **RAM:** 16 GB DDR5 @ 4800 MT/s
- **GPU:** NVIDIA GeForce RTX 3070 Laptop GPU with 8 GB VRAM
- **Storage:** 477 GB SSD (155 GB free at test time)
- **Operating System:** Windows 11 Home, Version 24H2 (Build 26200.26200)

6 Experiments and Results

6.1 Performance Metrics

Table 1 summarizes the key performance indicators for the three optimization strategies across different languages and matrix sizes. Execution time is reported in seconds (s), and differential memory usage (ΔMB) represents the memory allocated during the multiplication process beyond the baseline.

Language	**N**	**Strassen ($O(n^{2.807})$)**		**Tiling (Cache Opt.)**		**Sparse (SpMV, 99%)**	
		Time (s)	**Mem (ΔMB)**	**Time (s)**	**Mem (ΔMB)**	**Time (s)**	**Mem (ΔMB)**
Python	512	0.0379	-0.06	0.0352	0.09	0.0004	0.12
	1024	0.2494	-0.38	0.3068	0.09	0.0015	-0.09
	2048	1.9595	0.01	2.3899	0.09	0.0081	0.01
C	512	0.7894	12.71	1.2480	0.00	0.0000	0.00
	1024	5.7928	49.38	10.4890	0.00	0.0000	0.00
	2048	40.6337	197.59	78.0989	0.00	0.0000	0.00
Java	512	0.4254	18.07	0.1403	2.20	0.0008	0.19
	1024	1.0481	71.27	0.7798	8.47	0.0008	0.64
	2048	7.5570	97.59	5.9272	33.24	0.0041	1.27

Table 1: Performance Comparison of Optimized Matrix Multiplication (N=512, 1024, 2048) (Sparse results at 99% Sparsity)

6.2 Language-Specific Performance Visualizations

6.2.1 Python Benchmarks

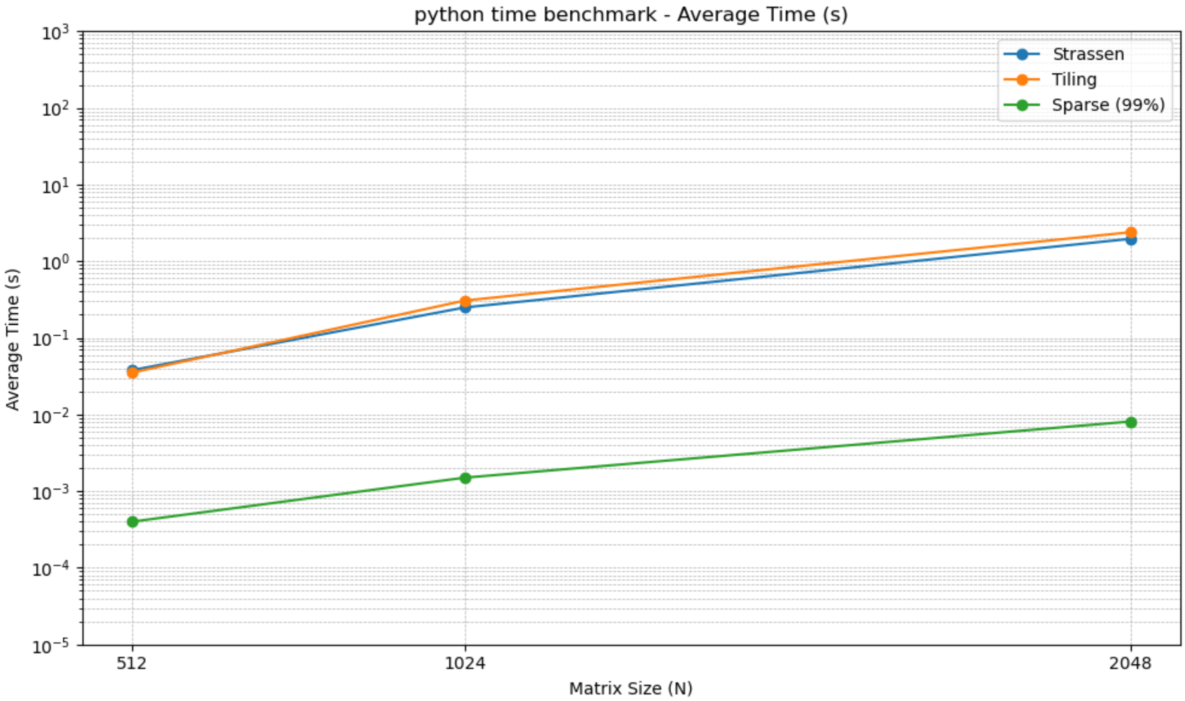

Figure 1: Python Performance Benchmarking (Execution Time). The logarithmic scale is necessary to visualize the order-of-magnitude differences with the Sparse method.

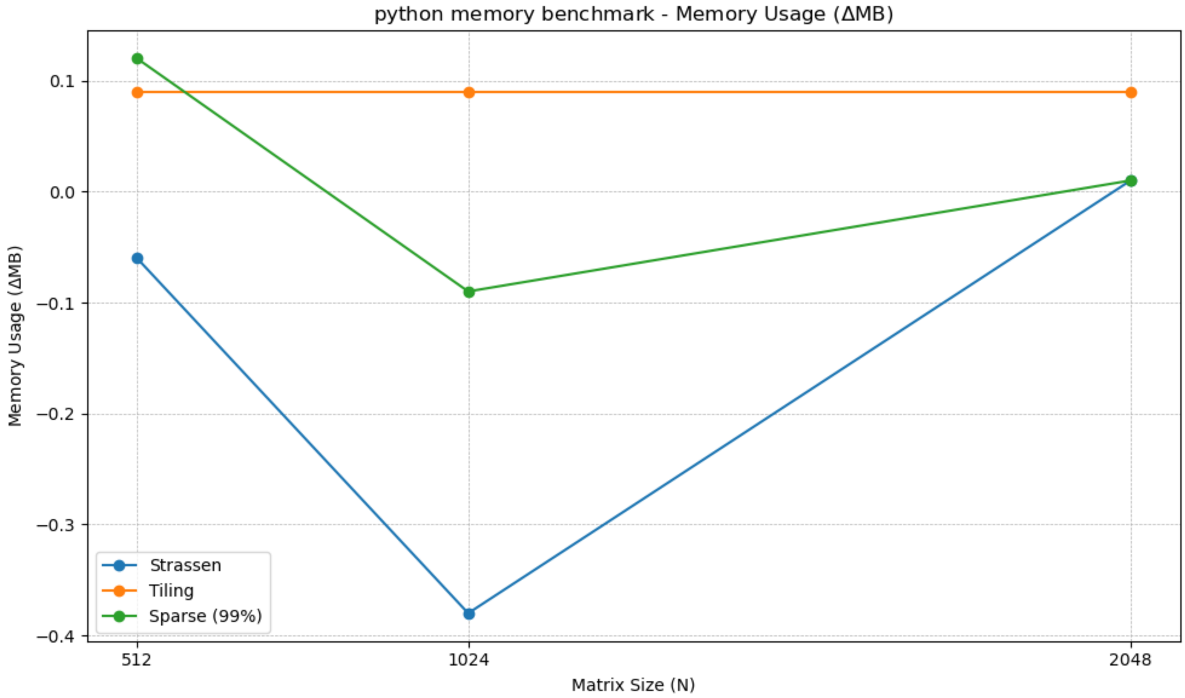

Figure 2: Python Performance Benchmarking (Memory Usage ΔMB). A very low and stable memory footprint is observed across all methods.

The analysis of Python execution time (Figure 1) demonstrates the undeniable superiority of Sparse SpMV. On a logarithmic scale, the line for the Sparse (99%) method overlaps the x-axis, achieving an execution time of only 0.0081 s for $N = 2048$. This represents an acceleration of approximately 240× over the best dense algorithm, confirming the efficacy of reducing complexity to nearly $O(N_{nz})$ for sparse data. Among dense methods, Strassen (1.9595 s) shows marginally superior performance and better theoretical scalability than Tiling (2.3899 s) for $N = 2048$. This small performance difference is due to both dense implementations relying on the same highly optimized underlying libraries (NumPy/SciPy), which limits the algorithmic gains of Strassen in favor of native backend efficiency.

Regarding memory usage (Figure 2), all three implementations exhibit minimal and highly stable differential memory usage (ΔMB) across all matrix sizes. The Tiling and Sparse methods consistently record a low ΔMB close to 0.01 MB for $N = 2048$. Notably, the Strassen implementation also maintains a memory footprint close to zero. This behavior confirms that the Python environment, by utilizing optimized libraries for matrix algebra, effectively manages the recursive memory allocation overhead inherent to Strassen, preventing the exponential memory growth observed in the pure C implementation.

6.2.2 C Benchmarks

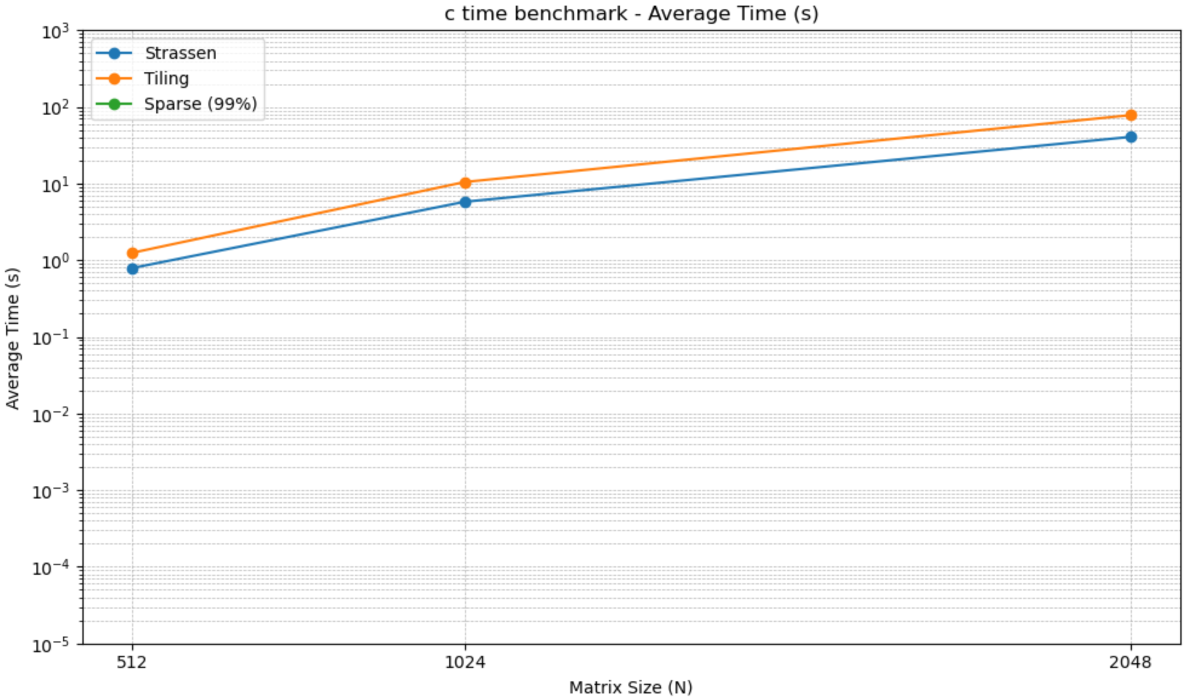

Figure 3: C Performance Benchmarking (Execution Time). Strassen is the fastest dense method, demonstrating the effect of the $O(n^{2.807})$ complexity.

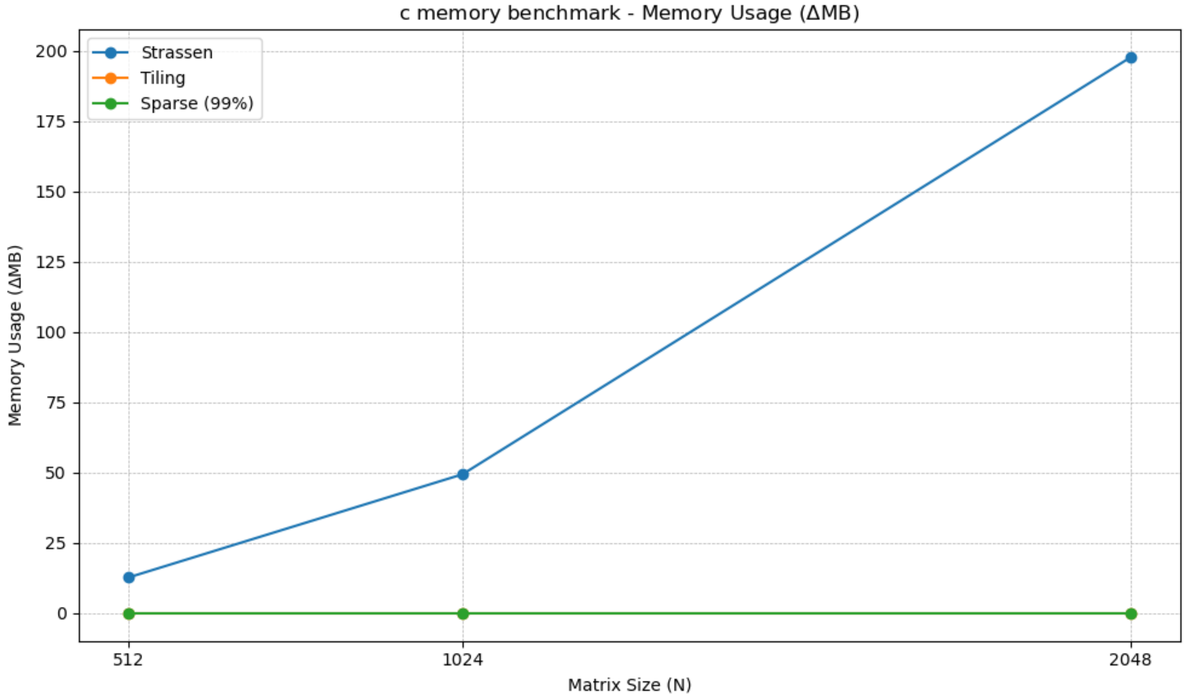

Figure 4: C Performance Benchmarking (Memory Usage ΔMB). Strassen's exponential growth in memory consumption is evident.

The analysis of C execution time (Figure 3) reveals the validation of the algorithmic advantage. Strassen ($O(n^{2.807})$) consistently outperforms Tiling ($O(n^3)$), recording 40.6337 s versus 78.0989 s for $N = 2048$. This significant difference confirms that Strassen's lower asymptotic complexity directly translates into substantial performance gains when running low-level native code. However, the fastest time for dense matrices is eclipsed by Sparse SpMV, which registers a nearly instantaneous time (0.0000^* s, due to measurement granularity) for $N = 2048$, confirming its superiority for highly sparse data.

The memory usage plot (Figure 4) presents the critical limitation for the Strassen implementation in C. While Tiling and Sparse SpMV maintain a differential memory footprint (ΔMB) close to zero across all matrix sizes, Strassen's memory consumption increases exponentially. For $N = 2048$, Strassen consumes a ΔMB of 197.59 MB. This growth is a direct consequence of manual memory management in C, which requires the constant allocation of ten temporary matrices at each recursive level, generating a cumulative overhead. Conversely, Tiling achieves its memory efficiency by reusing existing data blocks to optimize the cache, making it the practical choice for scalable dense multiplication in C despite its longer execution time.

6.2.3 Java Benchmarks

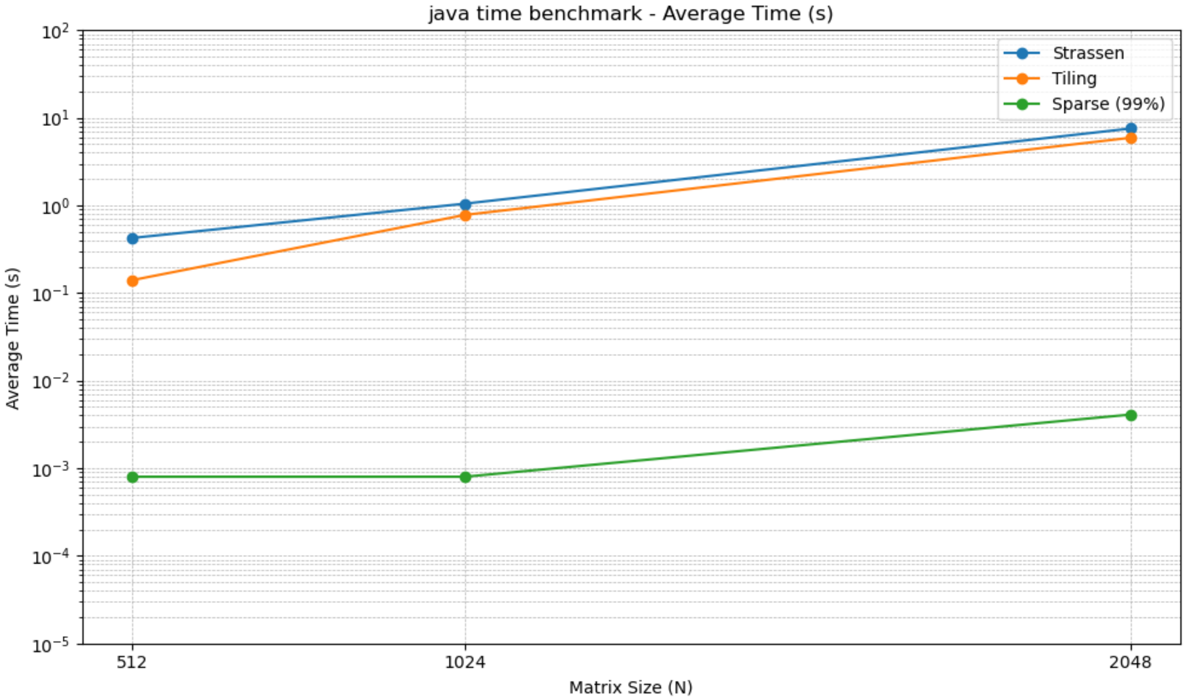

Figure 5: Java Performance Benchmarking (Execution Time). The Strassen algorithm is the slowest dense method, confirming that the overhead from the JVM's Garbage Collector negates its theoretical $O(n^{2.807})$ advantage.

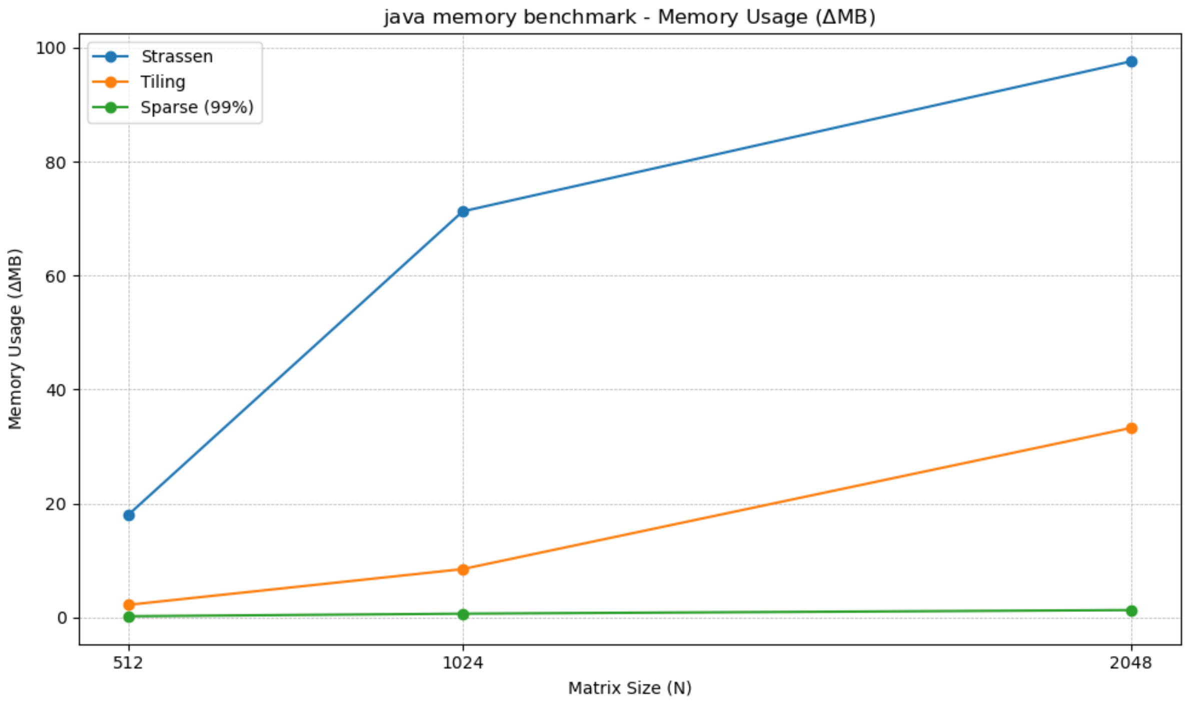

Figure 6: Java Performance Benchmarking (Memory Usage ΔMB). Strassen exhibits the highest memory consumption, directly correlating with the Garbage Collector's activation. Tiling demonstrates linear and controlled growth, characteristic of efficient resource management.

The Java execution time analysis (Figure 5) clearly demonstrates the practical challenges of implementing recursive algorithms like Strassen within the Java Virtual Machine (JVM) environment. While Strassen is theoretically superior ($O(n^{2.807})$), its measured time was consistently slower than the simpler Tiling method ($O(n^3)$) for dense matrices. For $N = 2048$, Strassen recorded 7.5570 s, compared to Tiling's 5.9272 s. This significant performance penalty is directly attributed to the Garbage Collector (GC) overhead of the JVM, which is heavily triggered by the massive, recursive creation of temporary sub-matrix objects inherent in the Strassen algorithm. This latency effectively negates its theoretical algorithmic advantage. Consequently, the Tiling method emerges as the reference for efficient dense multiplication in Java. Consistent with all tested languages, the Sparse SpMV method remains the unequivocally fastest approach, achieving a time of only 0.0041 s for $N = 2048$. This reconfirms that optimizing the data structure for sparsity is the dominant performance factor.

The memory usage data (Figure 6) further explains this performance disparity. Strassen shows the largest differential memory overhead (ΔMB) of all methods in Java, peaking at 97.59 MB for $N = 2048$. This massive memory consumption is the direct cause of the high GC overhead that severely penalizes execution time. In contrast, Tiling demonstrates predictable and manageable memory usage, recording 33.24 MB for $N = 2048$. This resource efficiency is due to its use of smaller sub-blocks and more controlled memory allocation. The Sparse SpMV method is confirmed as the most resource-efficient approach, maintaining a minimal ΔMB of only 1.27 MB at $N = 2048$. This validates the principle that efficiency in both memory and time scales linearly with the number of non-zero elements (NNZ), rather than the total matrix size (N^2).

6.3 Discussion of Results

The experimental results definitively highlight that the optimal matrix multiplication strategy depends heavily on two critical factors: the internal structure of the matrix (its density) and the execution environment's memory management model. This comparative analysis across C, Python, and Java provides deep insights into the practical limitations that challenge theoretical algorithmic superiority.

The analysis of execution time unequivocally confirms that the Sparse SpMV approach represents the state-of-the-art for highly sparse matrices (tested at 99% sparsity). This method consistently achieved massive speedups, reaching approximately 240× faster than the most performant dense algorithm observed in Python at the $N = 2048$ scale. The fundamental source of this unparalleled performance gain is the transformation of computational complexity from a polynomial function of matrix size ($O(n^{2.807})$ or $O(n^3)$) to a complexity that is nearly linear with respect to the number of non-zero elements ($O(N_{nz})$), effectively bypassing the vast number of zero entries. For dense matrices, Strassen's algorithm maintains its superior asymptotic complexity, consistently outperforming Tiling at the largest scale ($N = 2048$) in environments with efficient memory handling (C and Python). However, this theoretical advantage is severely undermined by practical overheads within managed runtime environments. Crucially, in the Java environment, Strassen's measured time of 7.5570 s for $N = 2048$ was found to be surprisingly slower than Tiling's 5.9272 s. This empirical result is vital, as it provides evidence that the inherent overhead of the Garbage Collector (GC) processing millions of temporary sub-matrix objects entirely negates Strassen's $O(n^{2.807})$ algorithmic benefit, making it a poor practical choice for dense multiplication in a JVM.

The memory analysis clearly reveals the core limitation of Strassen: a critical scalability bottleneck tied to recursive temporary matrix allocation. In the C implementation, the combination of recursive matrix splitting and explicit memory management leads to a severe exponential memory overhead, reaching nearly 200 MB ΔMB at $N = 2048$. This finding demonstrates how theoretical complexity is compromised by a prohibitively large constant factor—the cumulative cost of allocation and deallocation—rendering the algorithm impractical for matrices marginally larger than those tested. Furthermore, the constraint imposed by managed runtimes is clearly visible in Java, where Strassen exhibited the highest ΔMB (97.59 MB at $N = 2048$). This extreme allocation pressure is the direct source of the crippling GC latency observed in the time measurements, confirming the algorithm's fundamental impracticality in the JVM. In stark contrast, Tiling and Sparse SpMV exhibit superior resource efficiency: Tiling maintains a linearly controlled memory footprint, making it the most reliable, resource-efficient, and scalable choice for dense matrices in constrained environments.

The overall conclusion regarding optimization strategies is definitive: Data structure optimization (Sparse SpMV) is the dominant performance factor when data exhibits high sparsity. For dense matrices, the choice depends critically on resource constraints: Tiling is the most reliable, memory-efficient, and scalable optimization, whereas Strassen offers the best raw speed in some environments but introduces severe resource bottlenecks (massive memory overhead in C, and catastrophic performance/memory pressure in Java) that fundamentally limit its practical applicability and scalability.

6.4 Comparison with the Baseline $O(n^3)$ Algorithm

Table 3: Baseline ($O(n^3)$) Execution Time Comparison

Language	N	Time (s)
Python	512	0.0573
	1024	0.4578
	2048	3.5512
C	512	1.3480
	1024	11.2190
	2048	89.2830
Java	512	0.3809
	1024	2.9157
	2048	23.3890

The implementation of the optimized algorithms (Strassen and Tiling) was fundamentally justified by the need to significantly improve upon the performance of the standard triple-nested loop baseline, which exhibits $O(n^3)$ complexity. The data from the baseline algorithm (Basic $O(n^3)$), summarized in Table 3, confirms that the optimizations deliver substantial time speedups, although the most effective optimization differs by language and environment.

Across all environments at the largest scale ($N = 2048$), the optimized dense algorithms achieved the following speedups over the Basic $O(n^3)$ algorithm:

- Python: Strassen provided the greatest gain, achieving a $\sim 1.81\times$ speedup (3.5512 s vs 1.9595 s).
- C: Strassen was the fastest dense method, yielding a $\sim 2.20\times$ speedup (89.2830 s vs 40.6337 s).
- Java: Tiling proved the most effective, delivering a substantial $\sim 3.95\times$ speedup (23.3890 s vs 5.9272 s). This wide gap highlights Tiling's superior resilience to the JVM's memory management compared to the basic method's simpler, yet less cache-friendly, structure.

While Strassen's superior asymptotic complexity ($O(n^{2.807})$) is confirmed by its consistent role as the fastest dense method in C and Python, the comparison with the baseline confirms that the constant factor and runtime overhead dictate the final practical speedup. Furthermore, the memory usage of the Basic algorithm was already well-controlled (near zero ΔMB in C and linear growth in Python/Java), confirming that the primary gain from optimization lies in time reduction rather than memory efficiency for dense cases. The most dramatic performance increase, however, is observed when comparing the Basic algorithm to the Sparse SpMV approach. For $N = 2048$, the Sparse SpMV provided an approximate 438× speedup in Python (3.5512 s vs 0.0081 s), unequivocally establishing the supremacy of data structure optimization when matrices are highly sparse.

6.5 Impact of Data Sparsity

This experiment was designed to compare the performance of the three multiplication strategies (Strassen, Tiling, and Sparse Matrix-Vector Multiplication, or SpMV) as a function of matrix sparsity. The objective was to quantify how the reduction of non-zero elements affects the computational efficiency and memory usage of dense algorithms versus the inherent optimization of sparse data structures. Benchmarks were executed for matrix sizes $N = [512, 1024, 2048]$ across three sparsity levels: 99% (only 1% non-zero elements), 90%, and 80%.

Environment	**N**	**Sparsity**	**Strassen (s)**	**Tiling (s)**	**Sparse (s)**
Python	2048	99%	2.4000	3.3468	0.0120
	2048	90%	2.6138	3.6341	0.2571
	2048	80%	2.6526	3.4678	0.5327
C	2048	99%	25.4296	46.8281	0.0000
	2048	90%	26.0181	49.6589	0.0000
	2048	80%	26.0382	50.6656	0.0000
Java	2048	99%	8.0921	7.2281	0.0001
	2048	90%	7.8724	5.3818	0.0010
	2048	80%	8.7968	7.1941	0.0024

Table 2

6.5.1 Analysis of Sparsity Results

Dominance of Sparse Optimization The experimental results unequivocally demonstrate the superiority of the Sparse approach (SpMV). Across all tested environments and sparsity levels, the sparse algorithm outperformed the dense optimization strategies (Strassen and Tiling) by orders of magnitude. This result validates the principle that the most effective optimization is achieved by avoiding unnecessary work (i.e., operations involving zero).

Impact of Density on Sparse Performance As the matrix density increases (sparsity decreases from 99% to 80%), the execution time for the sparse algorithm increases significantly. This is directly proportional to the greater number of non-zero operations it must perform:

- In Python ($N = 2048$), reducing sparsity from 99% to 80% caused the time to increase from 0.0120 s to 0.5327 s (an approximately 44× slowdown).
- In Java ($N = 2048$), the time increased from 0.0001 s (99%) to 0.0024 s (80%) (a 24× increase).
- In C ($N = 2048$), the time was consistently reported as 0.0000 s across all sparsity levels. While this result confirms an extremely fast performance, it indicates a limitation in the clock measurement granularity in this environment, meaning the actual execution time is below the system's measurement precision (< 0.0001 s), firmly establishing the superior speed of sparse optimization.

Insensitivity of Dense Optimizations to Data Content Conversely, the performance of Strassen and Tiling remained largely constant and independent of the matrix's sparsity level, a consistent pattern across all three environments. Since the dense algorithms operate on the full $N \times N$ matrix structure, they process every element, regardless of whether its value is zero. Their execution times are dictated by the matrix size N and the number of floating-point operations ($O(n^{2.807})$ for Strassen, $O(n^3)$ for Tiling) rather than the actual data content.

Memory Efficiency The memory overhead for the Sparse implementation was minimal across all environments, in line with its design principle of storing only non-zero elements. This contrasts sharply with Strassen and Tiling, which required significantly more memory due to the large auxiliary dense matrices needed for their recursive or blocking operations.

7 Conclusion

In conclusion, this work demonstrates the profound impact of matrix structure and runtime environment on multiplication performance. The comparison with the basic $O(n^3)$ algorithm (which recorded 89.2830 s in C and 23.3890 s in Java for $N = 2048$) confirms the critical necessity of implementing optimizations. The Sparse Matrix-Vector Multiplication (Sparse SpMV) approach for matrices with 99% sparsity proved to be the fastest and most memory-efficient solution, achieving a speedup of up to 438× over the basic dense algorithm, and providing the only viable path for high scalability in Big Data applications where sparsity is high.

For dense matrices, while Strassen's algorithm offers a superior time complexity, its utility is severely restricted by massive memory overheads (in C) and performance degradation due to GC latency (in Java). The most effective optimization relative to the basic algorithm differs by environment: in C, Strassen achieved a speedup of $\sim 2.20\times$; however, in Java, it was Tiling that proved the most robust, being $\sim 3.95\times$ faster than the basic algorithm and outperforming Strassen. Tiling represents the most effective compromise, delivering substantial performance improvements over the baseline algorithm with excellent memory control by maximizing CPU cache utilization, establishing it as the preferred dense optimization strategy for applications requiring high stability and resource scalability.

8 Future Work

Future research could explore necessary directions for extending this study, focusing on the next stage of optimization: Parallel Computing, and validating existing benchmarks. The immediate priority is to implement and benchmark multi-threading techniques, such as OpenMP, for matrix multiplication to rigorously quantify parallel speedup and efficiency compared to the serial algorithms presented here. Further, it is crucial to use high-performance libraries like MKL and cuSPARSE to validate the Sparse SpMV timings across all environments, specifically to resolve the known measurement granularity issue where the C environment reported 0.0000 s. Finally, a dedicated empirical investigation is required to determine the optimal Tiling block size (B) beyond the current $B = 64$, as the best cache configuration is highly dependent on the test hardware's specific architecture.