Performance Benchmark of Matrix Multiplication

Richard Raatz: L0NC8J5G
https://github.com/RGR-repo-cloud/ULPGC_BigData/tree/main/Task_2

Big Data (40386) - 3. year
Universidad de Las Palmas de Gran Canaria

Submission date: 2025-11-22

Contents

List of Figures

List of Tables

1 Introduction

Matrix multiplication is a fundamental, computationally expensive operation in Big Data processing, driving applications like machine learning and simulations [1]. The standard $\mathscr{O}(n^3)$ complexity becomes a critical bottleneck as data scales, demanding advanced optimization techniques.
This paper investigates and compares various optimization strategies for efficient matrix multiplication implemented in C. The primary goal is to demonstrate how these techniques can significantly reduce execution time and memory usage when handling increasingly large matrices.
The performance of the Basic Algorithm (baseline) is compared against four advanced approaches:

- Optimized Dense Implementations:
  - Loop Unrolling [2]
  - Cache Blocking (Tiling) [3]
  - Strassen's Algorithm [4]
- Sparse Implementation:
  - Compressed Sparse Row (CSR) Format [5]

Performance is evaluated based on Execution Time, Memory Usage, and the Maximum Matrix Size handled efficiently.
In the remainder of this paper, first, the different optimizations techniques are presented, then a short overview of the implementation of the suite utilized for the benchmarking is given. For the benchmarking the methodology is described first, and finally, the results are presented.

2 Optimization Techniques

2.1 Basic Matrix Multiplication (Baseline)

The basic matrix multiplication algorithm uses three nested loops to compute $C_{ij} = \sum_k A_{ik}B_{kj}$. This implementation serves as the performance baseline due to its clear, direct mathematical definition. While straightforward, its $\mathscr{O}(n^3)$ time complexity makes it highly inefficient for large matrices, as it involves $2n^3$ floating-point operations, neglecting opportunities for cache reuse or instruction-level parallelism [1].

2.2 Loop Unrolling

Loop unrolling is a compiler optimization technique that reduces the overhead associated with loop control structures (e.g., branch checks and index increments). It replicates the body of the innermost

loop several times, allowing the processor to execute multiple matrix element computations per loop iteration. This also helps expose Instruction-Level Parallelism (ILP), keeping the CPU's execution units busy and improving the utilization of registers [2].
This implementation employs 4-way unrolling on the innermost k-loop, executing four multiply-accumulate operations per iteration instead of one. The technique includes a cleanup loop to handle matrix dimensions not divisible by the unroll factor.

2.3 Cache Blocking

Cache blocking (or tiling) is a memory optimization technique designed to exploit the principle of locality. It reorganizes the matrix multiplication to operate on small, fixed-size sub-blocks (tiles) that are carefully chosen to fit entirely within the CPU's faster L1 or L2 cache. By performing all necessary computations on a block before moving to the next, it maximizes data reuse and drastically reduces costly access to main memory, significantly improving performance for large matrices [3].
This implementation uses a six-nested-loop structure where the outer three loops iterate over blocks and the inner three loops process elements within each block.

2.4 Strassen's Algorithm

Strassen's algorithm is a recursive, divide-and-conquer algorithmic improvement that reduces the asymptotic complexity of matrix multiplication from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^{\log_2 7}) \approx \mathcal{O}(n^{2.807})$. It achieves this by strategically computing the product of two $n \times n$ matrices using only seven recursive multiplications of $(n/2) \times (n/2)$ sub-matrices, instead of the standard eight. While it offers superior asymptotic performance, the high constant factors and increased overhead make it practical primarily for very large matrices [4].
This implementation partitions matrices into quadrants and computes seven intermediate products (M1-M7) using matrix additions, subtractions, and recursive Strassen calls, then combines these to form the result quadrants. The algorithm includes a threshold mechanism (64×64) that switches to basic multiplication for small matrices to avoid recursion overhead, and restricts operation to square, power-of-two matrices where the theoretical complexity advantage is realized.

2.5 Sparse Matrix Multiplication (CSR Format)

Sparse matrix multiplication using the Compressed Sparse Row (CSR) format focuses on exploiting matrices where the majority of elements are zero (sparse). Instead of storing and processing all n^2 elements, the CSR format only stores the non-zero elements and their corresponding indices. This approach completely eliminates arithmetic operations involving zeros, which drastically reduces both the memory footprint and the computational cost, making it essential for Big Data applications with

high sparsity [6].
This implemtation stores sparse matrices using three arrays: values (non-zero elements), column indices (column positions), and row pointers (starting indices for each row).

3 Architecture and Implementation

This project is primarily written in the language C. It maintains strict separation between production code and benchmarking infrastructure: core algorithms reside in src as standalone, optimized implementations without measurement overhead, while the benchmarks directory contains dedicated performance evaluation tools including precision timing, memory profiling, and statistical analysis frameworks. The system employs a modular design where each algorithm can be independently verified, tested, and benchmarked through unified interfaces defined in include. Performance measurement encompasses execution time analysis across multiple matrix sizes, memory usage tracking, maximum matrix size determination, and sparse-dense performance comparisons across varying sparsity levels. The framework generates comprehensive CSV datasets and automated visualization through Python plotting scripts, enabling systematic analysis of algorithmic trade-offs, scalability characteristics, and cache performance effects across different optimization techniques.

4 Benchmarking Methodology

4.1 Dense Matrix Performance Evaluation

The dense matrix benchmarking methodology evaluates four optimization techniques (basic, loop unrolling, cache blocking, and Strassen) across standardized matrix sizes of 256×256, 512×512, 1024×1024, and 2048×2048. Each algorithm undergoes three independent runs per matrix size to ensure statistical reliability, with performance metrics including execution time measured using high-precision wall-clock timing and memory consumption tracked through Linux /proc/self/status RSS monitoring. The framework employs a warmup phase to stabilize CPU caches before measurement, takes the minimum execution time across runs to reduce system noise, and calculates comprehensive statistics including mean, standard deviation, and speedup ratios relative to the basic algorithm baseline.

4.2 Maximum Matrix Size Analysis

This benchmark determines the largest matrix size each algorithm can handle within a one-second timeout constraint, starting from 800×800 matrices and employing adaptive step sizes that become progressively larger (16 -> 32 -> 64 elements) as matrices exceed cache boundaries. The methodology accounts for memory allocation failures and execution timeouts, providing insights into scalability limitations and memory efficiency characteristics of each optimization approach. This analysis reveals the practical upper bounds of algorithmic performance and identifies where different optimization strategies become computationally prohibitive.

4.3 Sparse Matrix Comparative Analysis

The sparse matrix evaluation methodology compares CSR format performance against the basic O(n^3) dense implementation across four sparsity levels (50%, 70%, 90%, and 95%) using 256×256 matrices. Each sparsity configuration undergoes statistical analysis with multiple runs, measuring both execution time and memory consumption for equivalent dense and sparse operations. The framework calculates performance metrics including computational speedup ratios, memory storage savings, and efficiency crossover points where sparse representation becomes advantageous, providing quantitative evidence for the practical benefits of sparse matrix techniques in high-sparsity scenarios.

5 Results

5.1 Performance on Dense Matrices

5.1.1 Execution Time

Matrix_Size	Algorithm	Avg_Time (s)	StdDev_Time (s)	Min_Time (s)	Max_Time (s)	Speedup
256	Basic	0.035132	0.000112	0.034974	0.035213	1.00
256	Loop_Unroll	0.038259	0.000065	0.038167	0.038306	0.92
256	Cache_Block	0.028227	0.000049	0.028166	0.028287	1.24
256	Strassen	0.009780	0.000030	0.009737	0.009802	3.59
512	Basic	0.782385	0.000307	0.782006	0.782758	1.00
512	Loop_Unroll	0.768468	0.003604	0.763507	0.771962	1.02
512	Cache_Block	0.529428	0.000042	0.529373	0.529474	1.48
512	Strassen	0.069331	0.000310	0.069006	0.069748	11.28
1024	Basic	7.288823	0.021518	7.260103	7.311896	1.00
1024	Loop_Unroll	7.168095	0.001071	7.166679	7.169268	1.02
1024	Cache_Block	4.255916	0.001014	4.254541	4.256955	1.71
1024	Strassen	0.487165	0.000973	0.486407	0.488539	14.96
2048	Basic	75.594854	0.197834	75.396437	75.864884	1.00
2048	Loop_Unroll	71.911675	0.023294	71.887429	71.943111	1.05
2048	Cache_Block	67.755450	0.009582	67.747691	67.768951	1.12
2048	Strassen	3.435487	0.002994	3.431946	3.439268	22.00

Table 1: Execution time metrics across algorithms and matrix sizes

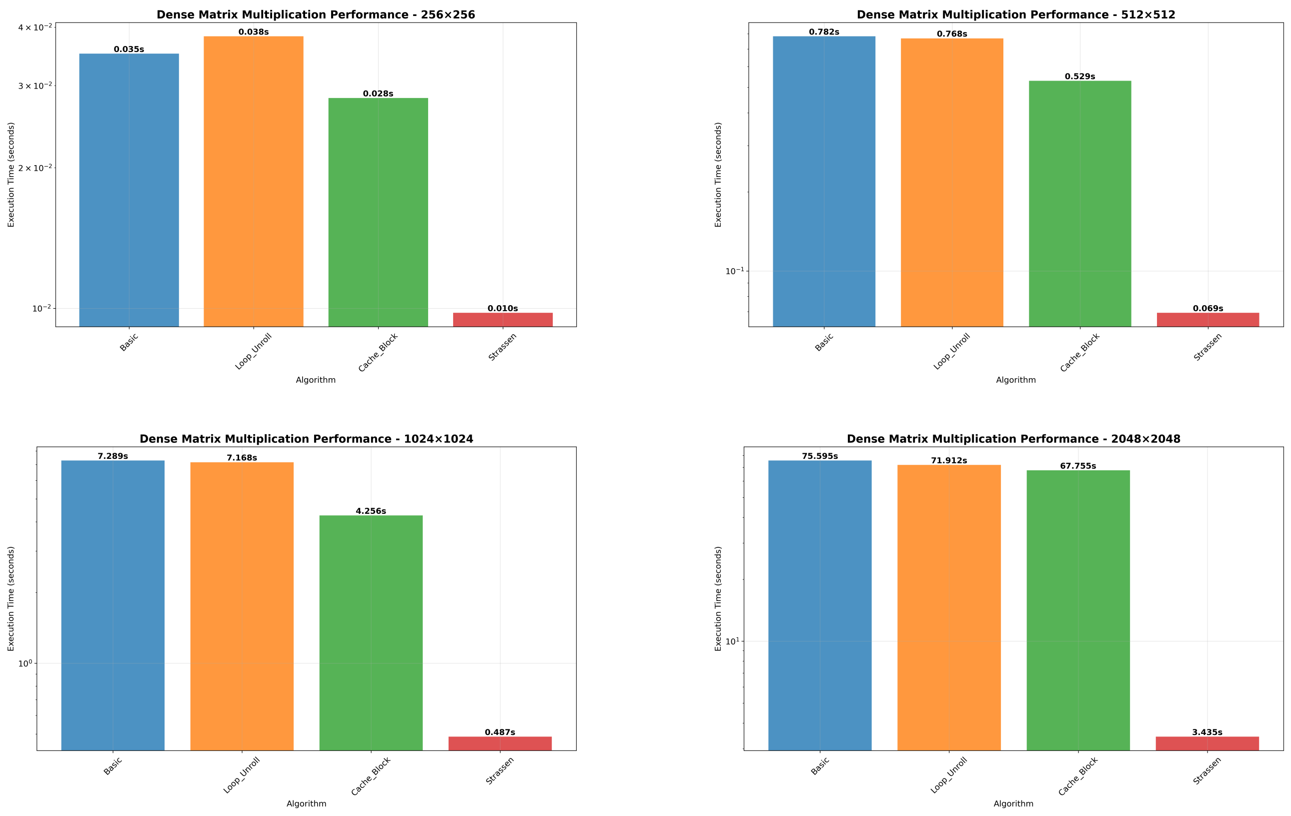

Figure 1: Average execution times across algorithms and matrix sizes

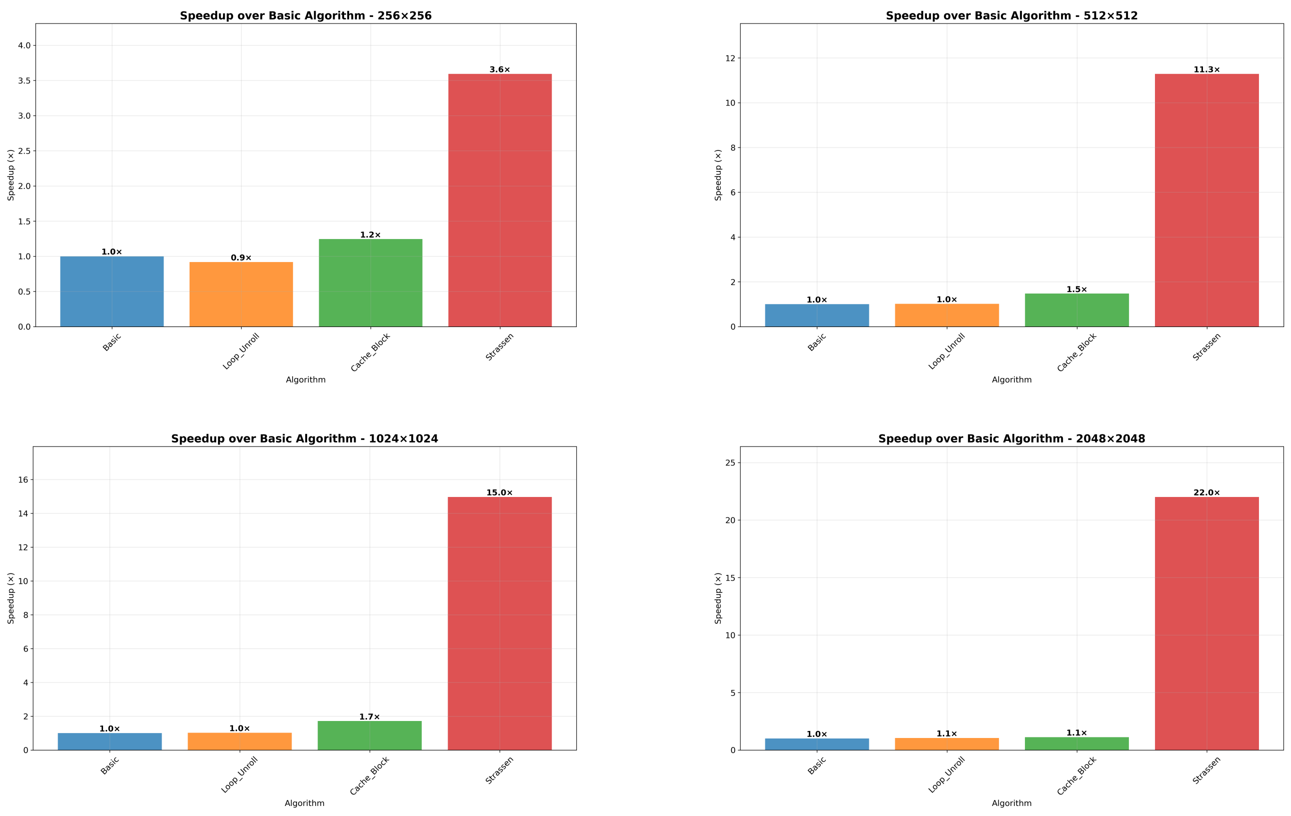

Figure 2: Speedups of execution times across algorithms and matrix sizes

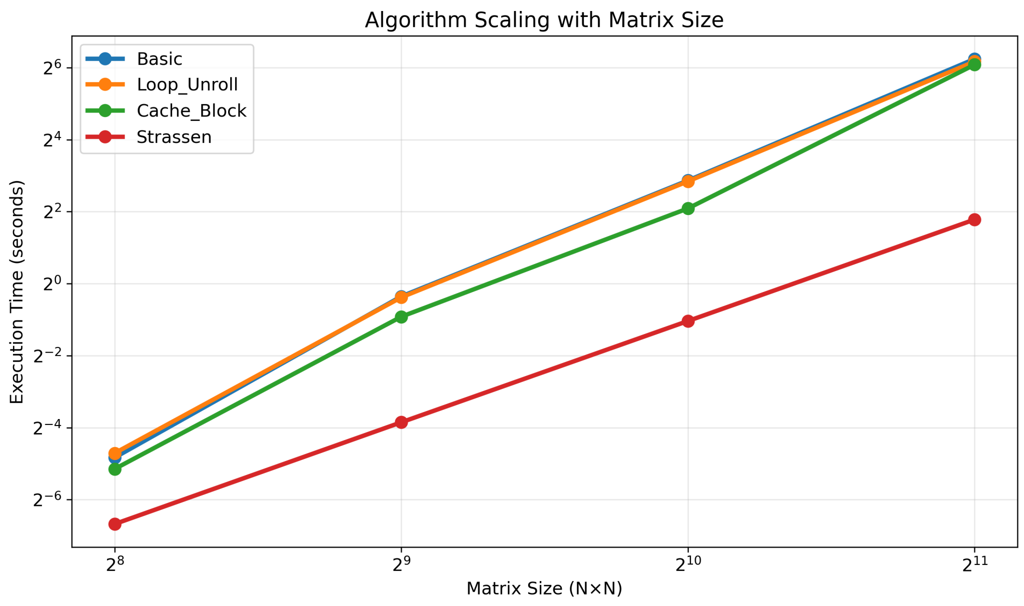

Figure 3: Scaling performance concerning execution time across algorithms

The benchmark results shown in table 1 reveal distinct performance characteristics across the four optimization approaches. Loop unrolling demonstrates minimal improvement, achieving only 1-5% speedup over the basic algorithm across all matrix sizes, indicating that the 4-way unrolling provides limited benefit on modern processors with sophisticated instruction pipelines. Cache blocking shows consistent moderate gains, delivering 24-71% speedup that increases with matrix size (1.24× at 256×256 to 1.71× at 1024×1024), demonstrating the growing importance of memory hierarchy optimization as matrices exceed cache capacity. However, cache blocking effectiveness diminishes at 2048×2048 (1.12× speedup), suggesting that even blocked access patterns struggle when working sets far exceed L3 cache.

Strassen's algorithm exhibits the most dramatic scaling behavior, with speedups increasing exponentially from 3.59× at 256×256 to 22.00× at 2048×2048, validating its superior $O(n^{2.807})$ complexity advantage over O(n^3) methods. The algorithm's performance improvement accelerates with matrix size, confirming that the recursive divide-and-conquer approach becomes increasingly beneficial for large matrices despite higher memory overhead (2-3× memory usage). These results demonstrate that while micro-optimizations like loop unrolling provide marginal gains, algorithmic improvements (Strassen) and memory access optimization (cache blocking) offer substantial performance benefits, with their effectiveness varying significantly based on matrix size and system cache characteristics.

5.1.2 Memory Usage

Matrix_Size	Algorithm	Avg_Mem (KB)
256	Basic	554
256	Loop_Unroll	512
256	Cache_Block	512
256	Strassen	2560
512	Basic	2048
512	Loop_Unroll	2048
512	Cache_Block	2048
512	Strassen	10752
1024	Basic	8192
1024	Loop_Unroll	8192
1024	Cache_Block	8192
1024	Strassen	43520
2048	Basic	32768
2048	Loop_Unroll	32768
2048	Cache_Block	32768
2048	Strassen	99974

Table 2: Memory usage metrics across algorithms and matrix sizes.

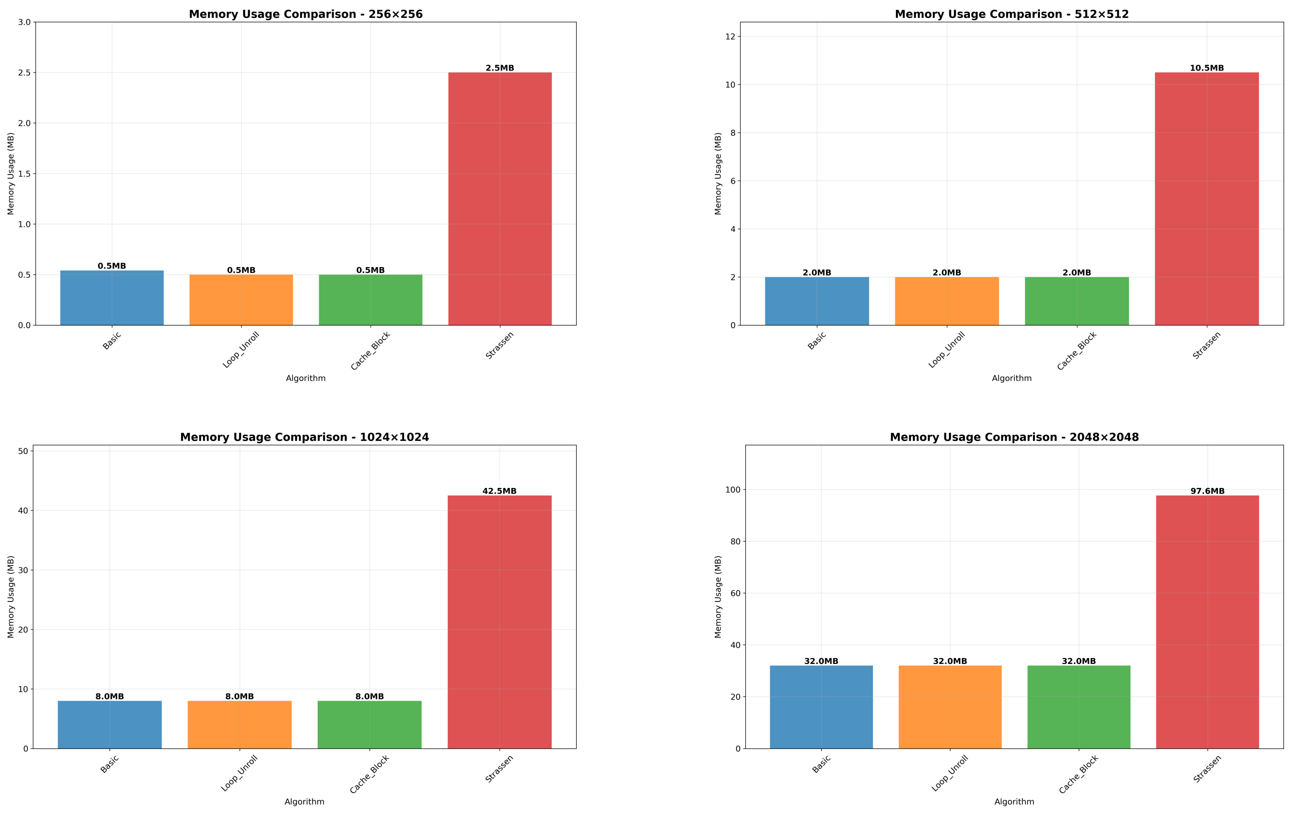

Figure 4: Average memory usage across algorithms and matrix sizes

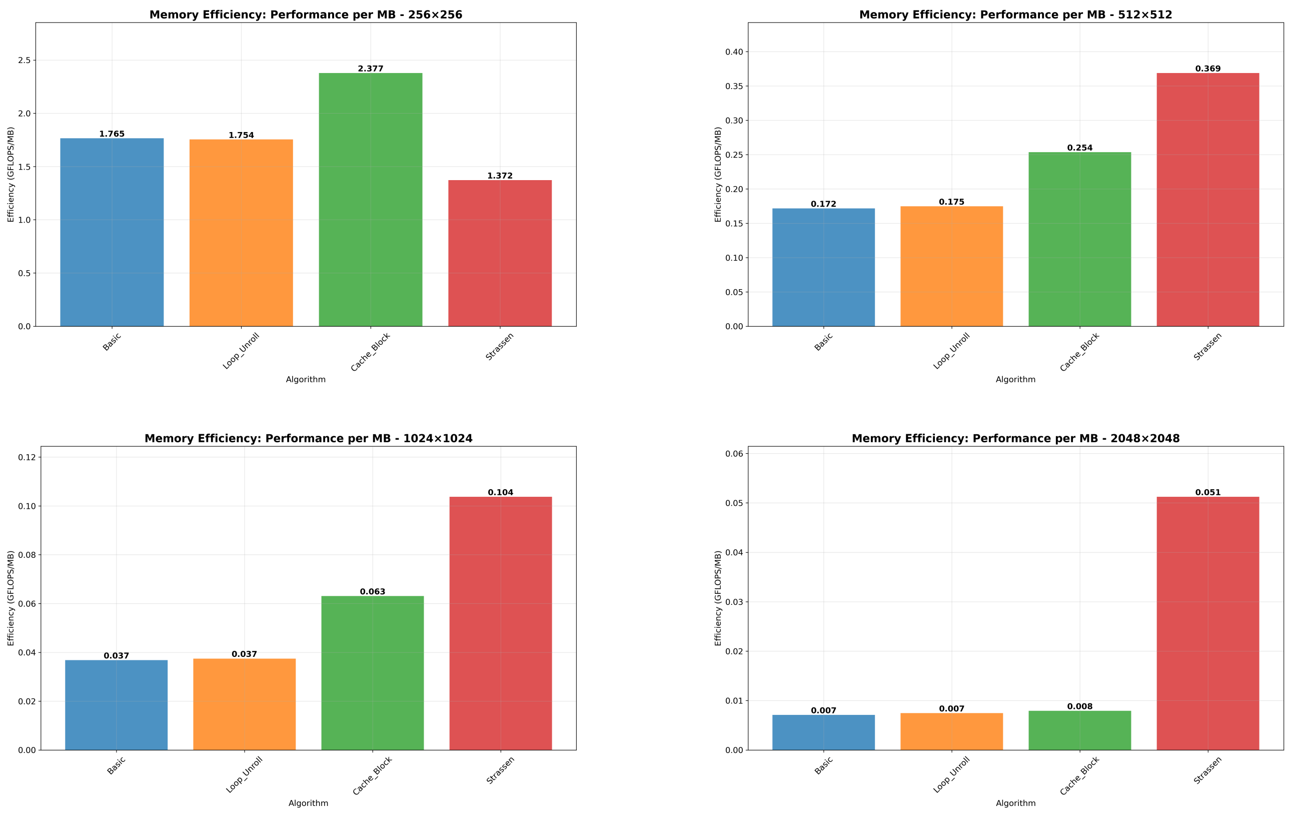

Figure 5: Average memory efficiency across algorithms and matrix sizes

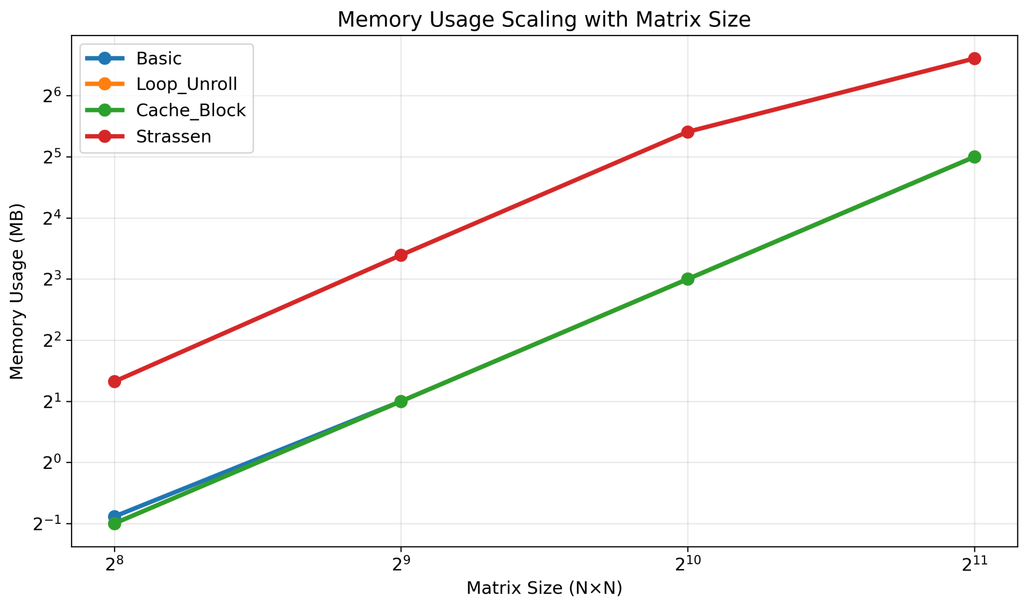

Figure 6: Scaling performance concerning memory usage across algorithms

The memory consumption patterns across dense matrix algorithms shown in figure 11 reveal important characteristics of each optimization approach. Basic, loop unrolling, and cache blocking algorithms demonstrate identical memory usage at each matrix size (554KB for 256×256, 2MB for 512×512, 8MB for 1024×1024, and 32MB for 2048×2048), confirming they operate with the same memory footprint of three matrices (A, B, and result). This consistent memory scaling follows the expected O(n²) growth pattern, where memory requirements quadruple as matrix dimensions double. Strassen's algorithm exhibits significantly higher memory overhead, consuming 2.5-3× more memory than other algorithms due to its recursive divide-and-conquer approach that creates multiple temporary submatrices and intermediate results. The memory usage grows from 2.5MB at 256×256 to nearly 100MB at 2048×2048, representing a substantial trade-off between computational speed and memory efficiency. This increased memory consumption reflects the algorithm's need to store seven intermediate products (M1-M7) and various temporary matrix additions/subtractions throughout the recursion tree.

The memory efficiency metric visualized in 5 shows Strassen achieves the highest efficiency (0.67 GFLOPS/MB at 1024×1024) despite using 3× more memory, because its 15× performance improvement outweighs the memory overhead. Cache blocking provides moderate efficiency gains (1.5× better than basic) while loop unrolling shows minimal improvement. Results demonstrate that algorithmic sophistication can deliver better memory efficiency even with higher absolute memory usage, as computational intensity improvements exceed storage costs.

The memory scaling plot 6 demonstrates perfect O(n²) scaling behavior for basic, loop unrolling, and cache blocking algorithms (quadrupling memory usage as matrix size doubles), while Strassen exhibits consistent 2.5-3× overhead across all sizes. All algorithms follow predictable log-log linear relationships, confirming theoretical memory complexity. Strassen maintains proportional scaling despite higher absolute usage, showing that recursive overhead remains bounded relative to problem size.

The results demonstrate that while micro-optimizations (loop unrolling) and cache-aware techniques (cache blocking) maintain minimal memory overhead, algorithmic improvements like Strassen come with considerable memory costs. This trade-off becomes critical for memory-constrained systems or when working with very large matrices, where the 3× memory overhead could limit the maximum processable matrix size despite Strassen's superior computational complexity.

5.1.3 Efficiency Limits

Algorithm	Max_Matrix_Size	Max_Elements	Memory_Requirement (MB)
Basic	864	746496	17.09
Loop_Unroll	864	746496	17.09
Cache_Block	992	984064	22.52
Strassen	864	746496	17.09

Table 3: Maximum matrix sizes concerning the 1 second execution time bound

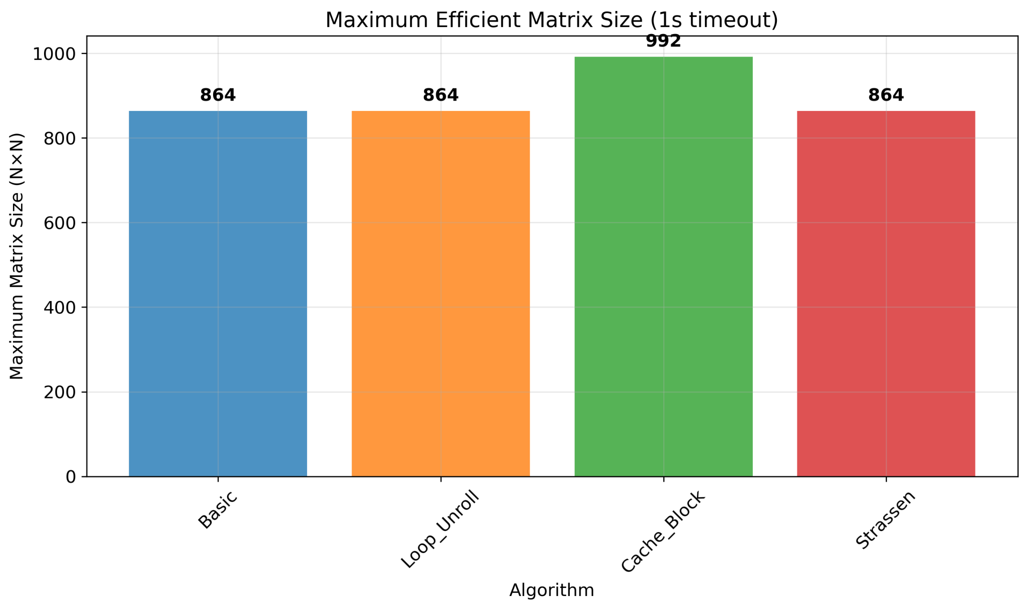

Figure 7: Maximum matrix sizes concerning efficincy across algorithms

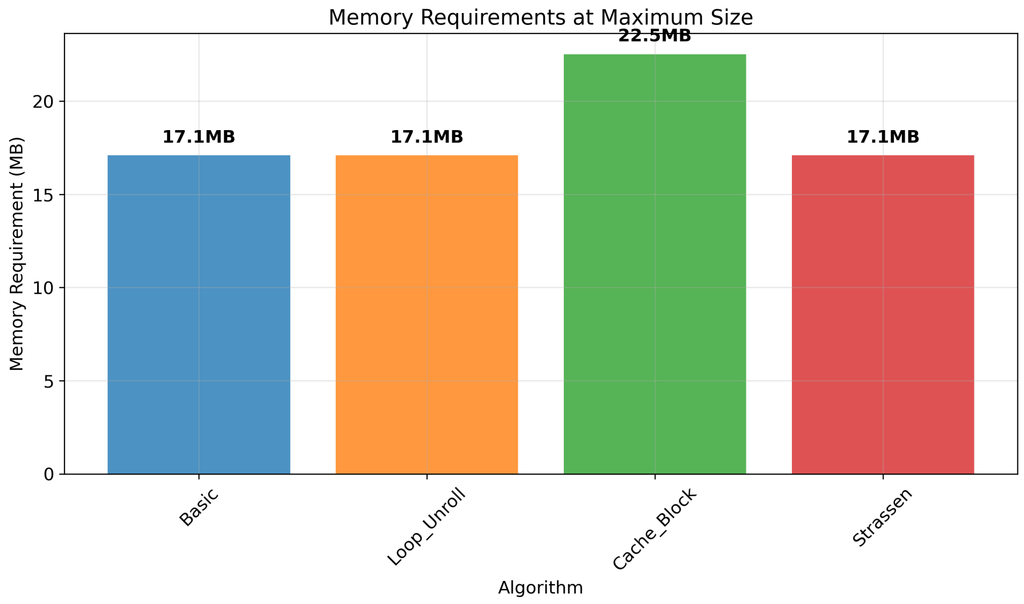

Figure 8: Memory requirements for maximum efficient matrix sizes across algorithms

The maximum size benchmark reveals cache blocking as the most scalable algorithm, handling 992×992 matrices compared to 864×864 for all other approaches under the 1-second timeout constraint as can be seen in figure 7. Basic, loop unrolling, and Strassen reach identical limits because Strassen falls back to basic multiplication for non-power-of-2 matrix sizes - the implementation restricts Strassen to exact powers of 2 (512, 1024, etc.) for simplicity, causing it to use basic algorithm at size 864. Cache blocking's 15% larger maximum size (992 vs 864) demonstrates the effectiveness of memory hierarchy optimization in extending practical matrix size limits.

Memory requirements shown in 8 remain modest across all algorithms, with cache blocking requiring only 22.52MB and others needing 17.09MB at their maximum sizes, indicating that computational time rather than memory capacity is the primary limiting factor. The results show that micro-optimizations provide no scalability advantage - loop unrolling offers no improvement over basic multiplication for large matrices. Strassen's power-of-2 restriction limits its practical applicability for arbitrary matrix sizes, while cache blocking emerges as the most practical optimization for extending usable matrix sizes through superior memory access patterns.

5.2 Performance on Sparse Matrices

5.2.1 Execution Time

Matrix_Size	Sparsity (%)	Format	Avg_Time (s)	StdDev_Time (s)	Min_Time (s)	Max_Time (s)	Speedup
256	50	Dense	0.037844	0.003333	0.035378	0.042556	1.00
256	50	Sparse	0.003548	0.000254	0.003285	0.003892	10.67
256	70	Dense	0.033800	0.000042	0.033744	0.033842	1.00
256	70	Sparse	0.001562	0.000008	0.001552	0.001569	21.64
256	90	Dense	0.033921	0.000131	0.033776	0.034092	1.00
256	90	Sparse	0.000570	0.000001	0.000568	0.000571	59.53
256	95	Dense	0.033645	0.000042	0.033586	0.033684	1.00
256	95	Sparse	0.000567	0.000014	0.000552	0.000586	59.31

Table 4: Execution time metrics on sparse matrices

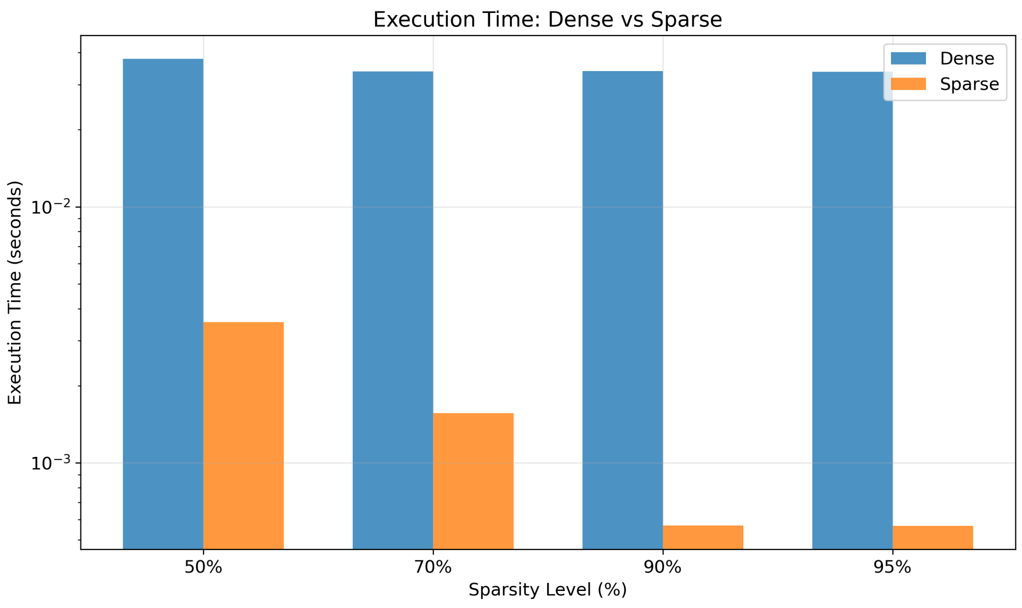

Figure 9: Average execution times of the basic dense algorithm and the CSR format algorithm across sparsity levels

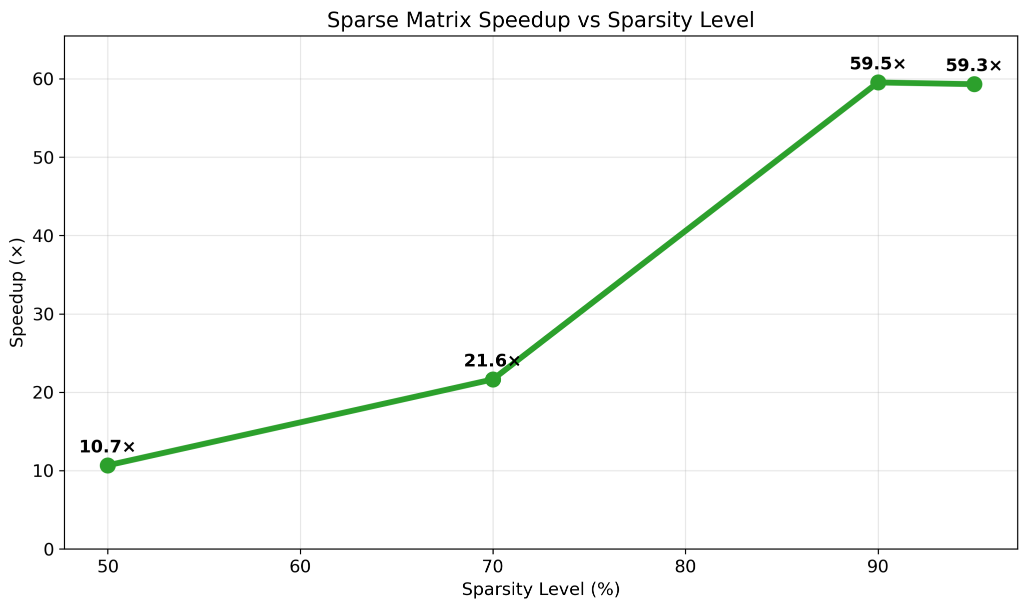

Figure 10: Average speedups of the CSR format algorithm compared to the basic dense algorithm across sparsity levels

The sparse matrix benchmark demonstrates dramatic performance improvements with increasing sparsity levels, as can be seen in 10. At 50% sparsity, CSR format delivers 11× speedup (0.0035s vs 0.038s) over dense multiplication, while higher sparsity levels show exponential gains: 70% sparsity achieves 22× speedup, and 90-95% sparsity reaches almost 60× speedup. The performance improvement follows the sparsity pattern because CSR multiplication only processes non-zero elements, effectively reducing the computational workload proportional to matrix density.
Dense matrix execution time shown in 9 remains constant (0.032s) regardless of sparsity level since it processes all elements, while sparse execution time decreases dramatically from 0.0035s at 50% sparsity to 0.0006s at 95% sparsity. The results reveal a clear breakeven point around 50% sparsity where CSR becomes advantageous, with benefits increasing exponentially at higher sparsity levels. This demonstrates that sparse matrix techniques become essential for matrices with >70% zeros, where they provide order-of-magnitude performance improvements by exploiting the underlying data structure to skip redundant zero operations.

5.2.2 Memory Usage

Matrix_Size	Sparsity (%)	Format	Avg_Memory (KB)	Storage_Elements	Runtime_Memory_Savings (%)
256	50	Dense	469	65536	0.0
256	50	Sparse	1194	65529	-154.6
256	70	Dense	469	65536	0.0
256	70	Sparse	1280	39283	-172.9
256	90	Dense	512	65536	0.0
256	90	Sparse	1152	12950	-125.0
256	95	Dense	512	65536	0.0
256	95	Sparse	768	6538	-50.0

Table 5: Memory usage on sparse matrices

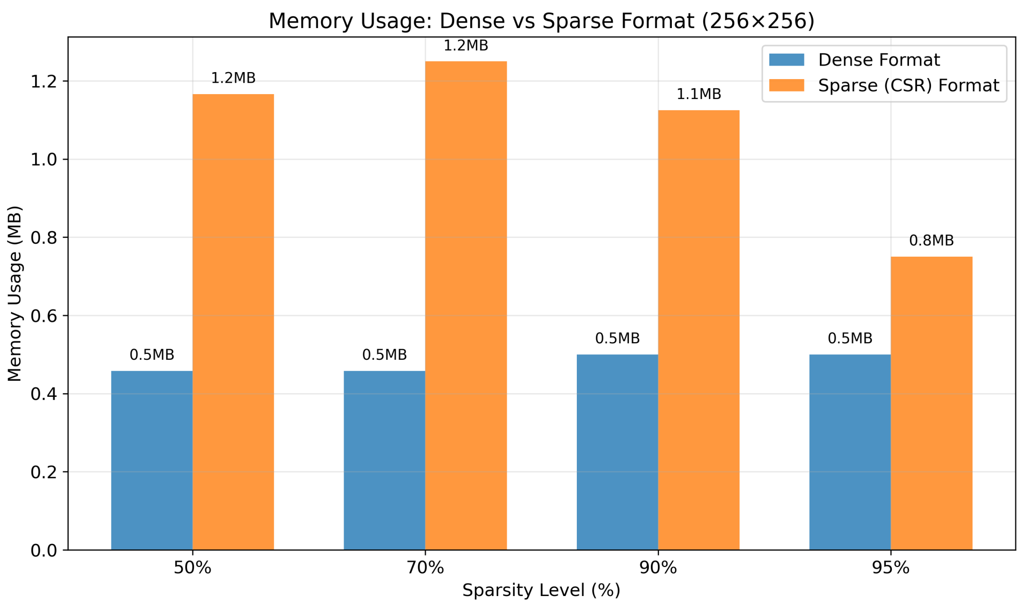

Figure 11: Average memory usage of the basic dense algorithm and the CSR format algorithm across sparsity levels

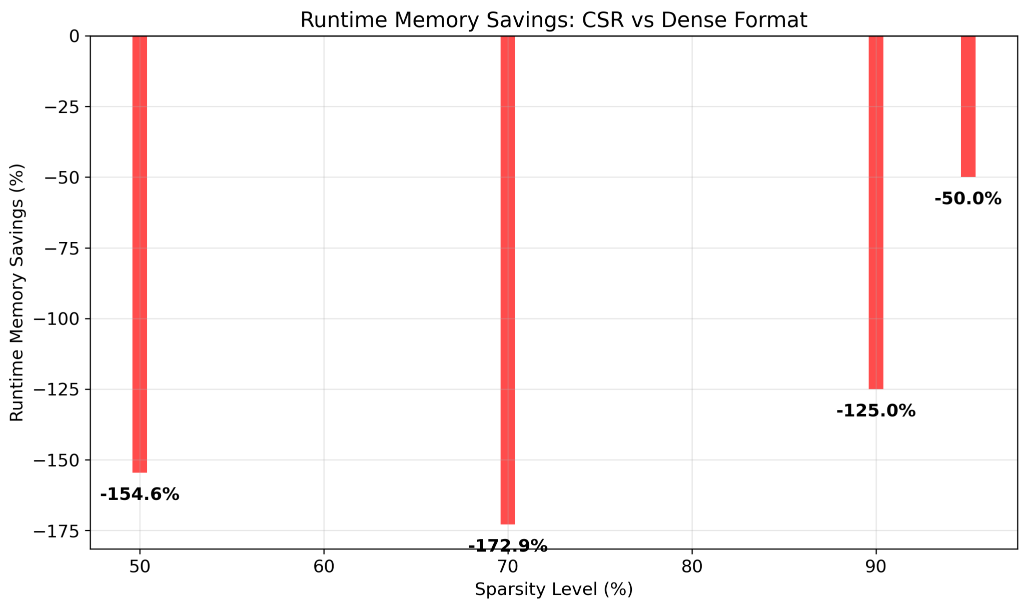

Figure 12: Average memory savings of the CSR format algorithm compared to the basic dense algorithm across sparsity levels

CSR format consistently uses more memory than dense matrices visualized in 11, with overhead ranging from 50% to 173% across sparsity levels. At 50% sparsity, CSR requires 154% more memory (1194KB vs 469KB), while at 95% sparsity, overhead reduces to 50% (768KB vs 512KB). This overhead comes from storing row pointers, column indices, and values needed for sparse operations. Memory overhead shown in 12 decreases as sparsity increases because fewer non-zero elements need storage, but CSR never achieves memory savings at these sizes. The 50-173% memory cost enables 11-60× performance improvements, representing a clear trade-off where computational speed comes at the expense of memory efficiency.

6 Summary and Conclusion

This investigation successfully implemented and evaluated multiple matrix multiplication optimization techniques, demonstrating significant performance variations across different approaches and problem characteristics. The comprehensive benchmark suite revealed that algorithmic improvements provide the most substantial gains, with Strassen's algorithm [4] delivering up to 22× speedup for large matrices despite increased memory overhead, while cache blocking [3] emerges as the most practical general-purpose optimization, consistently improving performance across all matrix sizes and extending maximum processable dimensions.

Loop unrolling [2] proved ineffective in modern computing environments, providing minimal improvements that highlight the diminishing returns of micro-optimizations on contemporary processors with sophisticated instruction pipelines. The maximum matrix size analysis confirmed computational complexity as the primary scalability constraint, with cache-optimized approaches demonstrating superior practical limits under time constraints.

Sparse matrix evaluation validates the critical importance of data structure selection for big data applications, where CSR format [5] transforms computational complexity from O(n^3) to effectively O(nnz), delivering order-of-magnitude improvements for matrices exceeding 70% sparsity. The analysis establishes clear optimization guidelines: Strassen for computationally intensive scenarios with power-of-2 matrices, cache blocking for general-purpose performance improvements, and sparse techniques for high-sparsity datasets typical in big data applications.

These findings demonstrate that effective big data processing requires matching algorithmic approaches to problem characteristics, emphasizing the importance of understanding both theoretical complexity and practical implementation constraints when designing high-performance computational systems for large-scale matrix operations.

List of sources

[1] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. 4th ed. Baltimore, MD: Johns Hopkins University Press, 2013.

[2] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 6th ed. Morgan Kaufmann, 2019.

[3] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Second. Society for Industrial and Applied Mathematics, 2002. DOI: `10.1137/1.9780898718027`. eprint: `https://epubs.siam.org/doi/pdf/10.1137/1.9780898718027`. URL: `https://epubs.siam.org/doi/abs/10.1137/1.9780898718027`.

[4] Volker Strassen. “Gaussian elimination is not optimal”. In: *Numer. Math.* 13.4 (Aug. 1969), 354–356. ISSN: 0029-599X. DOI: `10.1007/BF02165411`. URL: `https://doi.org/10.1007/BF02165411`.

[5] Samuel Williams et al. “Optimization of sparse matrix-vector multiplication on emerging multicore platforms”. In: *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. SC ’07. Reno, Nevada: Association for Computing Machinery, 2007. ISBN: 9781595937643. DOI: `10.1145/1362622.1362674`. URL: `https://doi.org/10.1145/1362622.1362674`.

[6] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. 2nd ed. Society for Industrial and Applied Mathematics (SIAM), 2003.