Performance Benchmark of Matrix Multiplication Approaches

Big Data Course Project

Victor Blanco
GitHub Repository

November 2025

1 Introduction

Matrix multiplication is a core operation in many scientific, engineering, and big data applications. As dataset sizes grow, it becomes increasingly important to optimize both the execution time and memory usage of matrix operations. In this study, we benchmark multiple matrix multiplication strategies for dense and sparse matrices of various sizes and sparsity levels.

2 Methods

We evaluated four main approaches:

- **Naive multiplication:** A straightforward triple-nested loop implementation. In previous experiments, the pure naive method was extremely slow; for example, multiplying 512×512 matrices took over 100 seconds. To overcome this, we applied `Numba` just-in-time compilation, which parallelizes loops and speeds up execution significantly.
- **Blocked multiplication:** Divides the matrices into smaller blocks to exploit CPU cache locality. Each block multiplication is parallelized using `ThreadPoolExecutor`, which allows safe multithreading and accelerates computation for large matrices.
- **Strassen's algorithm:** A recursive divide-and-conquer approach that reduces the number of multiplications compared to the naive method. A crossover threshold is used, below which standard multiplication is applied, balancing recursion overhead and efficiency.
- **Sparse multiplication:** For low-density matrices, many multiplications involve zeros. By representing the matrix in compressed sparse row (CSR)

format, we avoid unnecessary multiplications and reduce memory footprint. When `SciPy` is unavailable, a custom sparse multiplication routine is used.

3 Results

3.1 Dense Matrices

Table 1 shows execution time and peak memory usage for dense matrices of increasing size.

Table 1: Dense matrix multiplication performance

Size	Method	Time (s)	Memory (MB)
256	Naive	0.0023	0.50
256	Blocked	0.0181	1.02
256	Strassen	0.0021	1.63
512	Naive	0.0287	2.00
512	Blocked	0.1031	3.43
512	Strassen	0.0445	6.50
1024	Naive	0.1108	8.00
1024	Blocked	1.1268	16.79
1024	Strassen	0.1798	26.00
2048	Naive	1.9493	32.00
2048	Blocked	7.4743	101.55
2048	Strassen	1.2981	104.00
4096	Naive	26.9892	128.00
4096	Blocked	94.6001	677.44
4096	Strassen	9.5709	416.00

3.2 Sparse Matrices

Sparse matrices with low density achieve substantial improvements in both execution time and memory usage. Table 2 summarizes the results for selected sizes and densities.

4 Analysis of Results

From the dense matrix results, we observe that:

- **Naive multiplication:** Efficient only for small matrices. Despite JIT optimization via Numba, its performance quickly degrades for larger sizes, as seen in 4096×4096 matrices.

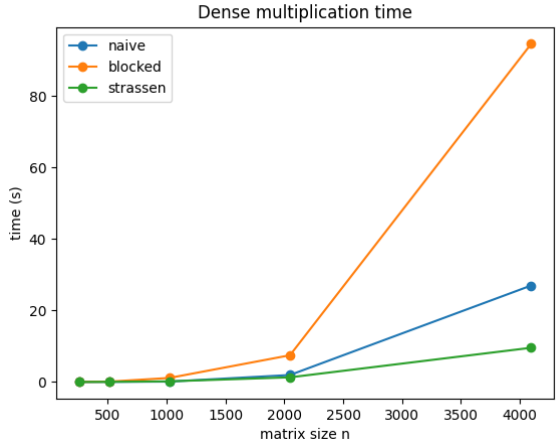

Figure 1: Execution time for dense matrix multiplication.

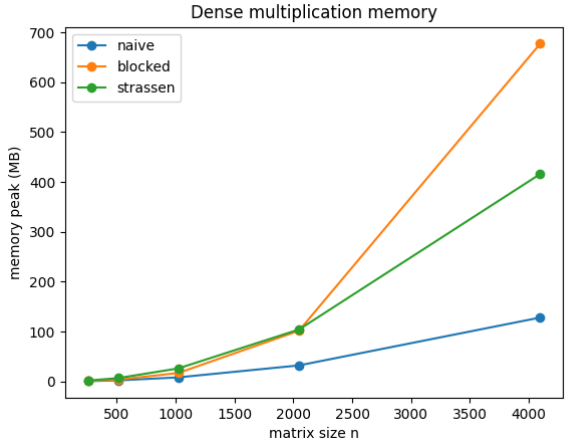

Figure 2: Peak memory usage for dense matrix multiplication.

- **Blocked multiplication:** Improves cache utilization, but memory consumption increases significantly for very large matrices.
- **Strassen's algorithm:** Achieves the best trade-off between time and memory for large matrices due to reduced arithmetic complexity.

Sparse matrices show that:

- Execution time scales primarily with the density rather than matrix size, making sparse methods highly efficient for low-density matrices.

Table 2: Sparse matrix multiplication performance (selected sizes)

Size	Density	Time (s)	Memory (MB)
256	0.010	0.0006	0.50
256	0.050	0.0006	0.50
256	0.100	0.0007	0.50
512	0.010	0.0041	2.00
512	0.050	0.0086	2.00
512	0.100	0.0151	2.00
1024	0.010	0.0094	8.00
1024	0.050	0.0303	8.00
1024	0.100	0.0715	8.00
2048	0.010	0.0728	32.00
2048	0.050	0.3306	32.00
2048	0.100	0.6466	32.00
4096	0.010	0.5958	128.00
4096	0.050	2.6146	128.00
4096	0.100	5.0903	128.00

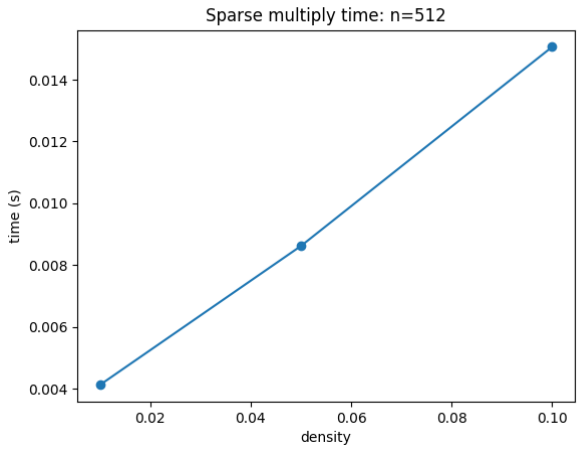

Figure 3: Execution time for sparse matrices of size 512 with varying density.

- Memory usage remains nearly constant for low densities, significantly lower than dense equivalents.
- For higher densities, sparse approaches converge towards dense performance.

Overall, the combination of algorithmic and hardware-aware optimizations is crucial for handling large-scale matrix operations in big data applications.

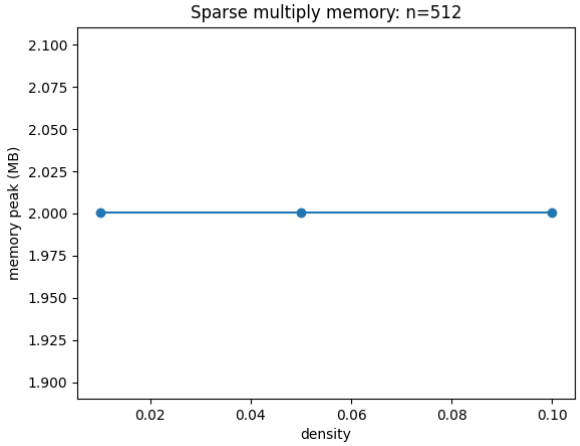

Figure 4: Memory usage for sparse matrices of size 512 with varying density.

5 Conclusions

Optimized matrix multiplication is vital in big data scenarios where large datasets are common. Our experiments indicate that:

- Algorithmic improvements (Strassen, blocked) and hardware acceleration (Numba, multithreaded blocks) can provide significant speedups over naive implementations.
- Sparse-aware computation dramatically reduces computation time and memory usage for low-density matrices, enabling efficient processing of extremely large datasets.
- Memory usage is often the limiting factor for blocked methods at extreme scales, while Strassen balances both memory and time efficiently.
- Future work could explore GPU acceleration and hybrid sparse-dense algorithms for further improvements.