Performance benchmark report

Jorge Cubero Toribio

November 2025

1 Introduction

This report investigates performance optimization techniques for matrix multiplication. Three implementations are compared:

1. **Basic Pure Python Implementation**: a triple nested loop using lists of lists.
2. **Dense NumPy Multiplication**: highly optimized BLAS-based dense matrix multiplication using the `@` operator.
3. **Sparse SciPy Multiplication**: compressed sparse row (CSR) representation with 99% sparsity.

The main goal is to evaluate the impact of optimizations and sparsity on:

- Execution time,
- Memory usage,
- Scalability with matrix size,
- Performance under different sparsity levels.

2 Methodology

2.1 Implementation Details

- The **Basic** version uses explicit loops, serving as a baseline for complexity.
- The **Dense** version uses NumPy arrays with contiguous memory layouts.
- The **Sparse** version uses SciPy's CSR format, which stores only non-zero values along with their indices.

2.2 Experimental Setup

Parameter	**Description**
CPU	Local host (Python execution environment)
Software	Python 3.x, NumPy, SciPy
Matrix Sizes	256, 512, 1024, 2048, 4096
Sparsity Levels	99% (for size test), densities 0.001–0.3 for sparsity sweep
Timing	`time.perf_counter()`
Memory	`nbytes` for dense, CSR data+indices+indptr for sparse

Table 1: Experimental setup parameters.

3 Results

3.1 Dense vs. Sparse Multiplication (99% sparsity)

Matrix Size	NumPy Dense (s)	SciPy Sparse (s)
256	0.0293	0.00018
512	0.00217	0.00047
1024	0.0143	0.00171
2048	0.1303	0.0132
4096	0.8694	0.0944

Table 2: Execution time comparison for dense and sparse matrices at 99% sparsity.

Matrix Size	Dense Memory (bytes)	Sparse Memory (bytes)
256	524,288	8,888
512	2,097,152	33,504
1024	8,388,608	129,932
2048	33,554,432	511,512
4096	134,217,728	2,029,652

Table 3: Memory usage comparison between dense and sparse matrices.

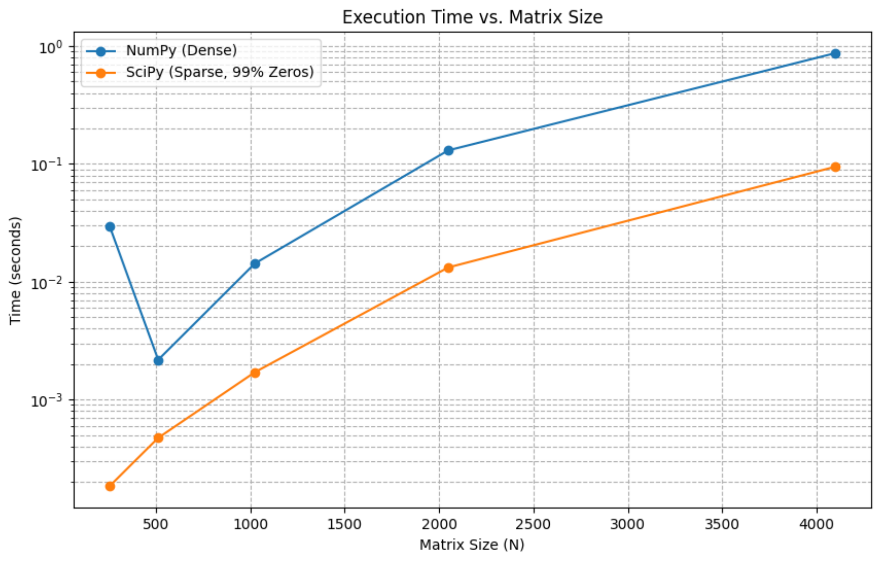

Figure 1: Execution Time vs. Matrix Size (log scale). NumPy dense vs. SciPy sparse multiplication.

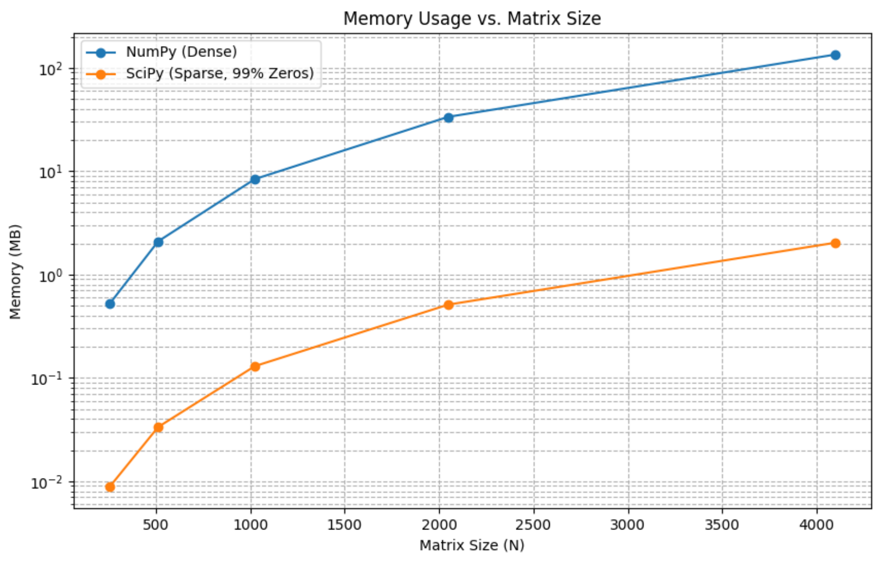

Figure 2: Memory Usage vs. Matrix Size (log scale). Dense matrices scale quadratically, while sparse matrices grow linearly with non-zero entries.

Observations:

- Sparse matrices are 10–100× times faster at high sparsity (99% zeros).
- Memory use is dramatically reduced — at 4096×4096, sparse uses about 1.5% of dense memory.
- Dense multiplication scales as $O(N^3)$ in time and $O(N^2)$ in memory.

3.2 Effect of Sparsity on Performance (2048×2048)

Density (% non-zero)	Sparsity (% zeros)	Sparse Time (s)	Memory (MB)
0.1	99.9	0.0034	0.0585
1	99	0.0088	0.512
5	95	0.114	2.52
10	90	0.267	5.04
20	80	0.882	10.07
30	70	1.856	15.11

Table 4: Sparse performance vs. density for 2048×2048 matrices. Dense baseline: 0.1992 s.

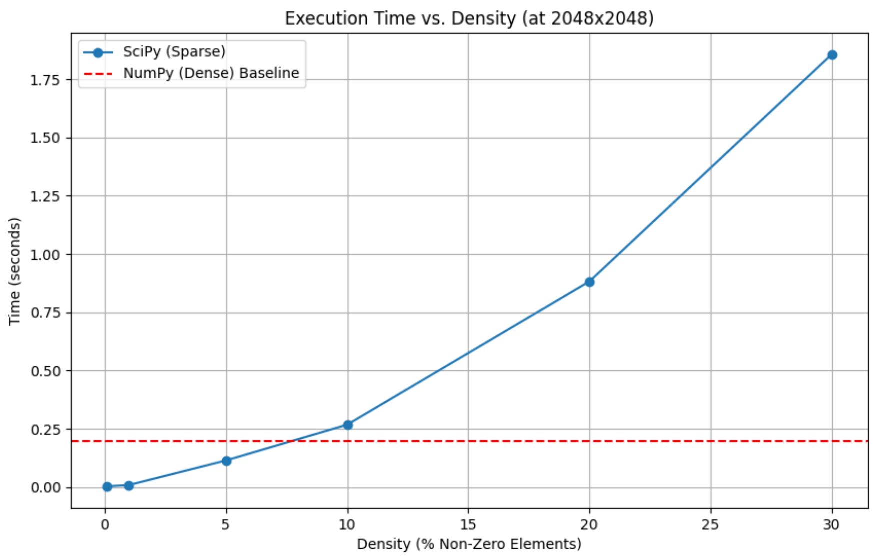

Figure 3: Execution Time vs. Density for 2048×2048 matrices. Sparse multiplication surpasses dense until roughly 10% density.

Analysis:

- Sparse multiplication is faster than dense up to about 10% density (≈90% sparsity).
- Memory grows linearly with the number of non-zeros.
- Beyond ∼10% density, sparse indexing overhead dominates.

3.3 Scalability and Maximum Efficient Size

Method	Max Size Efficiently Tested	Limiting Factor
Basic (Python)	128×128	CPU time ($O(N^3)$)
NumPy Dense	4096×4096	Memory (134 MB per matrix)
SciPy Sparse (99%)	4096×4096	None observed

Table 5: Maximum efficiently handled matrix sizes.

4 Discussion

- Dense multiplication benefits from cache locality and vectorized BLAS operations.
- Sparse multiplication is advantageous when most elements are zero, since the time complexity approximates $O(kN)$, where k is the average non-zero count per row.
- The break-even point between dense and sparse occurs at roughly 10% density.
- For dense workloads, performance becomes memory-bandwidth limited.

5 Conclusions

- Sparse matrices achieve major speed and memory improvements for sparsity levels above 90%.

- Dense NumPy multiplication remains optimal for dense or moderately sized matrices.
- The pure Python baseline is educational but impractical for performance computing.
- Overall, sparse matrix methods are critical for large-scale or memory-constrained applications.

6 Future Work

Future extensions may include:

- Implementing block-based (tiled) and Strassen algorithms.
- Parallelization via Numba or OpenMP.
- GPU acceleration (e.g., CuPy, PyTorch).
- Testing larger matrices ($N > 10{,}000$) or distributed systems.

All deliverables are located at: `https://github.com/JorsuCT/Big_Data/tree/main/IndividualAssignment/task2`