Task 2 - Optimized Matrix Multiplication Approaches and Sparse Matrices
Big Data

Alberto Guedes

November 2025

Contents

Abstract

This report evaluates three approaches to square matrix multiplication on a single machine: a pure-Python naive triple-loop implementation, a blocked implementation that leverages NumPy's optimized dot within sub-blocks, and NumPy's BLAS-backed dot. Sparse matrix multiplication via SciPy's CSR backend is also benchmarked across multiple sparsity levels.

Wall-clock time, approximate CPU utilization, and memory delta were measured across repeated runs for dense sizes up to $n = 512$ and sparse matrices at varying densities. NumPy's dot is the fastest for dense matrices; the blocked approach reduces the gap relative to naive Python but remains slower than BLAS, while the naive implementation scales poorly. Sparse multiplication is efficient at low densities and scales with the number of non-zeros. Source code: GitHub Repository

1 Methodology

1.1 Implementations

- **Naive (Python)**: Triple-nested loops accumulating $C_{ij} = A_{ik}B_{kj}$ implemented in pure Python (see `src/algorithms/naive.py`).
- **Blocked**: Tiles A and B into blocks and applies `numpy.dot` per block to improve cache locality (`src/algorithms/blocked.py`).
- **NumPy dot**: Direct `numpy.ndarray.dot`, relying on the linked BLAS (`src/algorithms/numpy_dot.py`).
- **Sparse (SciPy)**: Converts inputs to CSR via `scipy.sparse.csr_matrix` and multiplies (`src/sparse/sparse_ops.py`).

1.2 Data Generation

For dense benchmarks, random dense matrices $A, B \in \mathbb{R}^{n \times n}$ are generated. For sparse benchmarks, a target number of non-zeros (nnz) is sampled from a given density $d = 1 - \text{sparsity}$, placed at random coordinates in A and B, and assigned values from a uniform distribution.

1.3 Measurement

Each function is measured for `repeats` runs. For every run the following are recorded: (1) elapsed wall-clock time using `time.perf_counter()`, (2) CPU percent via `psutil.cpu_percent()`, and (3) resident memory delta ΔRSS for the process. Per-setting averages across repeats are reported. The measurement harness resides in `src/utils/measure.py`. CSV artifacts are stored under benchmarks/. A companion notebook including data visualization (benchmark_results.ipynb).

1.4 Environment

Experiments were executed on a Fedora 43 machine with python 3.13, using NumPy, SciPy, pandas, seaborn, matplotlib, and psutil.

2 Results

2.1 Dense Matrices

Table 1 summarizes representative timings for $n = 256$ and $n = 512$. NumPy's BLAS-backed dot leads across sizes; the blocked method is within one order of magnitude; the naive implementation becomes orders of magnitude slower as n grows.

Table 1: Dense multiplication: average time (seconds) and relative slowdowns vs. NumPy.

Size n	Algorithm	Time [s]	Slowdown vs NumPy
256	NumPy dot	0.000335	1.0x
	Blocked	0.001763	$\approx$ 5.3x
	Naive	0.912070	$\approx$ 2724x
512	NumPy dot	0.001916	1.0x
	Blocked	0.013386	$\approx$ 7.0x
	Naive	7.944882	$\approx$ 4148x

2.2 Sparse Matrices

For sparse multiplication using CSR, time scales primarily with the number of non-zeros. For $n = 256$, Table 2 shows the average time as density increases.

Table 2: Sparse CSR multiplication (SciPy), $n = 256$: average time vs density $d = 1 - \text{sparsity}$.

Sparsity	Density d	Time [s]
0.995	0.005	0.000654
0.990	0.010	0.000693
0.950	0.050	0.001075
0.900	0.100	0.002068

The following plots have been made to properly visualize results and compare between matrix multiplication methods.

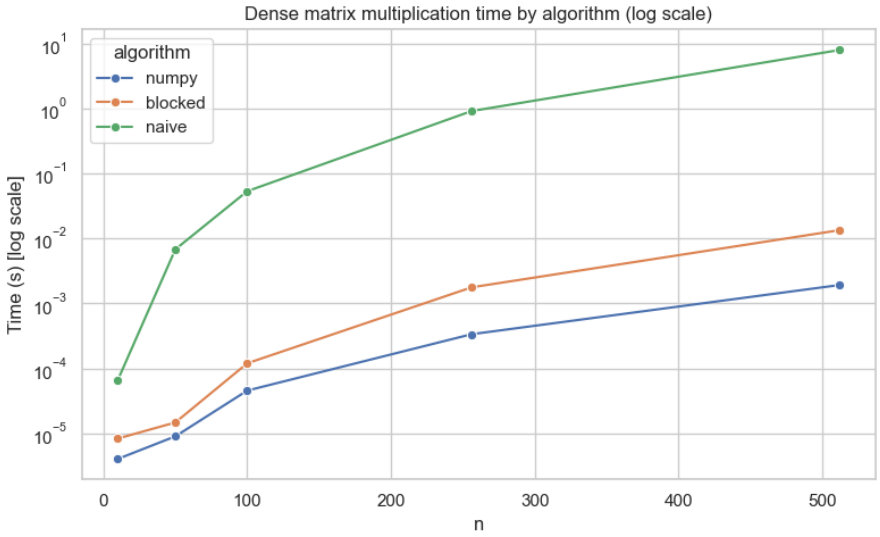

Figure 1: Benchmark results for Naive, Blocked and Numpy's BLAS implementations.

Figure 1 (produced by `benchmark_results.ipynb`) visualizes mean time versus size on log-scaled axes for readability.

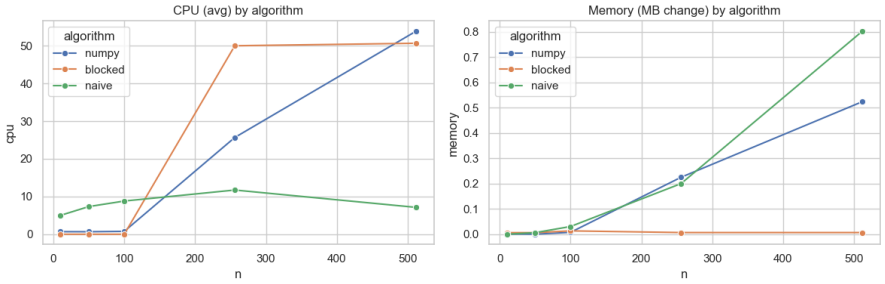

Figure 2: Benchmark results for Naive, Blocked and Numpy's BLAS implementations.

Figure 2 (produced by `benchmark_results.ipynb`) visualizes CPU time and memory usage versus size on log-scaled axes for readability.

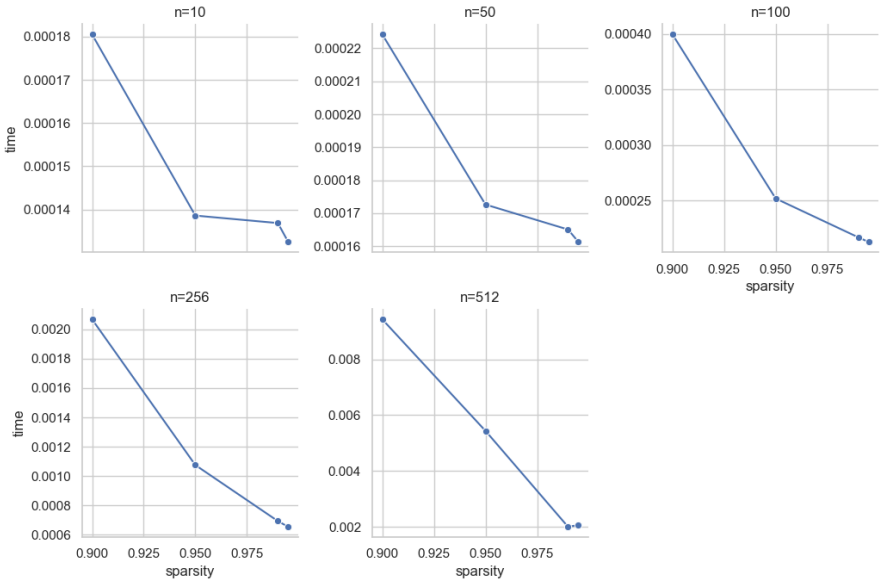

Figure 3: Benchmark results for Sparse matrices multiplication using CSR multiplication.

Figure 3 (produced by `benchmark_results.ipynb`) visualizes mean time versus size and sparsity on log-scaled axes for readability.

3 Discussion

- **Dense.** The BLAS-backed `numpy.dot` is optimal for dense multiplication due to highly optimized low-level kernels and favorable cache/memory utilization. Blocking improves locality compared to naive loops but still does not match BLAS throughput. The naive Python implementation incurs interpreter overhead and exhibits poor cache behavior, producing super-linear slowdowns as n increases.
- **Sparse.** For very sparse matrices (e.g., density $\leq 1\%$), CSR multiplication is extremely fast and scales approximately with nnz. As density increases, runtime grows; eventually, dense methods may become competitive when matrices are insufficiently sparse.
- **CPU/Memory metrics.** The CPU percentages collected with `psutil.cpu_percent()` are noisy (sampling and scheduler effects). Memory deltas were small for these runs but may rise with larger matrices or data conversions.

4 Conclusion

For dense matrices, `numpy.dot` (BLAS) provides the best performance. A blocked strategy can serve as a compromise when custom data flows are required but remains slower than BLAS. The naive Python approach is suitable only for didactic purposes or very small sizes. For sparse matrices, CSR-based multiplication (SciPy) is efficient at low densities and should be used when sparsity is significant.

Future work includes exploration of alternative block sizes, multi-threading/OpenMP settings for BLAS, and evaluation of other sparse formats (e.g., CSC, COO) tuned to specific sparsity patterns.

Reproducibility

Benchmarks were run with 10 repetitions per size using the scripts and programs in this repository. Raw CSVs are under benchmarks/. The notebook benchmark_results.ipynb computes aggregates and produces comparative plots.