Optimized Matrix Multiplication and Sparse Matrices: Dense Blocking, Random Sparsity and mc2depi Matrix

Fabio Nesta Arteaga Cabrera

November 22, 2025

Project Repository:
`https://github.com/NestX10/performance_bechmark_of_matrix_multiplication`

Abstract

This report presents the second individual assignment on matrix multiplication for the Big Data course: optimized dense matrix multiplication and sparse matrices. The work is implemented entirely in Python, using NumPy and SciPy, and focuses on three aspects: (i) cache-aware optimizations of dense matrix multiplication, (ii) sparse matrix multiplication for synthetic matrices with random sparsity, and (iii) a benchmark on the `mc2depi` matrix from the Williams sparse matrix suite. For all experiments, we measure execution time and peak memory usage while increasing the matrix size, and we examine how structure and sparsity affect performance. The code produces CSV files and unified plots so that the experiments are fully reproducible.

Keywords— matrix multiplication, sparse matrices, cache blocking, CSR, benchmarking, mc2depi.

1 Introduction

Matrix multiplication is a core building block in scientific computing, data analytics, and machine learning. In Big Data contexts, performance and scalability of matrix operations are crucial because matrix sizes can grow quickly and resource usage becomes a limiting factor. The second individual assignment of the course focuses on optimized approaches to matrix multiplication and on the use of sparse matrices.[1]

The project is driven by two main objectives. First, we study several dense kernels for $C = A \times B$ using a common Python-based infrastructure that measures time and memory while varying the matrix size. Second, we explore sparse matrix multiplication using the compressed sparse row (CSR) format on both synthetic matrices (with random sparsity) and on the `mc2depi` matrix.

2 Background and Related Work

Many optimizations for dense and sparse linear algebra have been studied in the high-performance computing community. Classical dense optimizations include cache blocking, loop reordering, etc. For sparse matrix-vector and matrix-matrix kernels, performance can be highly sensitive

[1] Assignment description in *"Individual Assignments: Matrix Multiplication"*.

to sparsity structure (nonzeros per row, bandwidth, clustering), and numerous works have proposed tuning frameworks and cache-aware formats.

The `mc2depi` matrix, used in this work, originates from an epidemiology application and appears in the Williams sparse matrix suite used to study sparse matrix-vector multiplication. Its structure (very large dimension and few nonzeros per row) makes it representative of challenging real-world sparse problems.

3 Assignment Description

The Task 2 objectives are:

- Implement at least two optimized versions of the matrix multiplication algorithm and compare them with a basic approach.
- Implement and evaluate matrix multiplication with sparse matrices and analyse how sparsity (percentage of zero elements) and structure affect performance.
- Test different matrix sizes and report execution time, memory usage, maximum matrix size handled, and a comparison between dense and sparse approaches.

This report addresses these objectives with:

1. Four dense matrix algorithms implemented in Python: a naive triple loop, a cache-aware version, several blocked (tiled) versions, and NumPy used as a correctness and performance reference.
2. CSR-based sparse multiplication for synthetic matrices with a random sparsity pattern and for matrices that mimic the structure of `mc2depi`.
3. An experiment on the `mc2depi` epidemiology matrix.

4 Methodology

All implementations are written in Python 3. The project is organised in the following modules (folder structure as shown in Figure 1 placeholder):

- `dense_ops.py`: dense matrix generation and algorithms.
- `sparse_ops.py`: sparse matrix generators and loader for `mc2depi`.
- `benchmark.py`: timing and memory-measurement helpers.
- `results_saver.py`: CSV writer for dense, sparse, and `mc2depi` results.
- `benchmark_plotter.py`: unified plotter that reads the latest CSV and produces figures.
- `config.py`: central configuration (matrix sizes, block sizes, number of runs, output directories).
- `main.py`: entry point that orchestrates all Task 2 benchmarks.

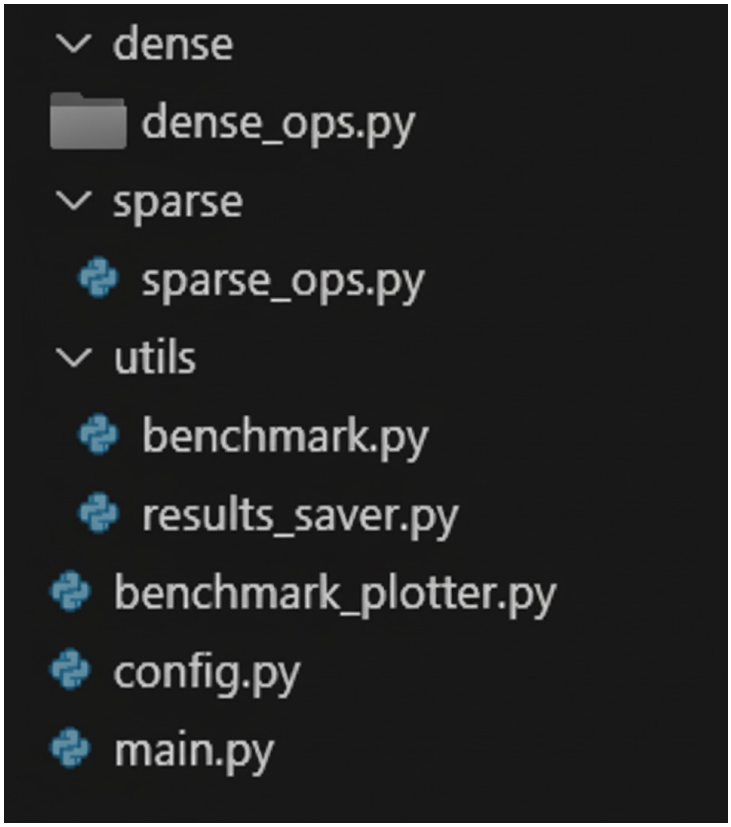

Figure 1: Project layout for Task 2 (placeholder figure).

4.1 Dense Matrix Multiplication

We consider three explicit dense algorithms for square matrices of size n. All work with Python lists-of-lists as the underlying representation, and all use double-precision (`float`) values. The set of sizes is defined in `config.py` as

$$\texttt{DENSE\_SIZES} = \{32, 64, 128, 256, 512, 1024, 2048\},$$

and the block sizes for tiling are

$$\texttt{BLOCK\_SIZES} = \{16, 32, 64, 128\}.$$

4.2 Maximum Matrix Size Handled

For dense matrices, preliminary tests with sizes larger than $n = 2048$ quickly became impractical: at 2048×2048, the naive kernel already needs about 606 s and around 580–600 MB of peak memory. Increasing the size further (e.g. to 4096×4096) would imply runtimes of many hours and memory consumption clearly above the available RAM on the test machine, so $n = 2048$ is considered the maximum feasible size for dense experiments.

For sparse matrices, the CSR kernels themselves are capable of handling much larger problems, especially for very sparse patterns (such as the epidemiology-like matrices). However, in order to keep the comparison fair and to be able to plot dense and sparse curves on the same axes, the synthetic sparse experiments were restricted to the same maximum size $n = 2048$ used in the dense benchmark. The only exception is the `mc2depi` experiment, where much larger sizes are explored thanks to the extremely low number of nonzeros per row.

Naive triple loop (`Original`). A straightforward implementation with three nested loops over i, j, and k, computing

Cache-aware (`Cache`). A re-ordered loop nest designed to improve spatial locality for the right-hand operand and to reduce cache misses by iterating over contiguous segments of memory when possible.

Tiled (`Tiling`). A blocked version that partitions the matrices into $b \times b$ tiles and performs the multiplication at tile granularity. The block size b is taken from `BLOCK_SIZES`. For each matrix size, we benchmark all block sizes and record their time and memory usage.

Additionally, a NumPy implementation using `np.matmul` (`@`) is used as a reference for correctness and as a strong baseline. It is treated as one more method in the dense benchmark.

4.3 Sparse Matrices and CSR Multiplication

All sparse experiments use the compressed sparse row (CSR) format from SciPy. The sparse kernel multiplies two CSR matrices A and B by computing $C = A \times B$ via SciPy's optimized routines.

Synthetic sparse matrices are generated using a routine `generate_random_sparse_matrix(n, sparsity)`, which draws each entry as nonzero with probability $1 - s$, where s is the target sparsity (fraction of zeros). In order to keep the experiment tractable but still demanding, we test a range of sparsity patterns with s set to $0.50, 0.75, 0.90, 0.95$, and 0.99, corresponding to $50\%, 75\%, 90\%, 95\%$, and 99% zeros.

For each generated matrix pair (A, B), the benchmark reports:

- matrix size n,
- sparsity s (fraction of zero entries),
- total number of nonzeros nnz,
- average nonzeros per row (nnz/n),
- mean execution time and peak memory.

The sparse sizes for the synthetic patterns are configured in `config.py` as

$$\texttt{SPARSE\_SIZES} = \{32, 64, 128, 256, 512, 1024, 2048\}.$$

4.4 `mc2depi` Matrix

The `mc2depi` matrix is loaded from the Williams sparse matrix collection via `load_or_download_mc2depi` in `sparse_ops.py`. We compute its sparsity and nonzeros per row, and we perform several CSR matrix-matrix multiplications to measure time and peak memory. The `mc2depi` results are stored in the CSV file in the sparse section and are summarised in a dedicated table.

4.5 Measurement Procedure

Timing and memory measurement are central to the assignment. The benchmarking helper functions perform:

- **Multiple runs.** Each configuration is executed a fixed number of runs, with `NUM_RUNS_DENSE = 3`, `NUM_RUNS_SPARSE = 3`, and `NUM_RUNS_MC2DEPI = 3` defined in `config.py`. We report the mean execution time.
- **Peak memory.** During each run, the process memory is sampled (using `psutil` inside `benchmark.py`) and the peak resident memory is recorded; the maximum across runs is reported as "memory peak".
- **Correctness checks.** Before benchmarking, small matrices are multiplied with all dense kernels and compared against NumPy results to ensure numerical correctness.

5 Results

This section presents the quantitative results stored in the CSV and in the plots generated from it.

5.1 Dense Matrix Multiplication

5.1.1 Execution Time (Table)

Table 1 shows the mean execution time for the four main dense methods and all matrix sizes.

Table 1: Mean execution time (s) for dense multiplication by method and matrix size.

Method	Size: 32	64	128	256	512	1024	2048
Original	0.001	0.011	0.092	0.744	6.944	67.409	605.597
Cache	0.002	0.014	0.108	0.843	6.989	56.892	460.522
Tiling (b=64)	0.002	0.013	0.110	0.901	7.394	59.603	481.100
NumPy (ref.)	0.000	0.000	0.001	0.004	0.014	0.063	0.257

The three Python-loop kernels scale roughly as $O(n^3)$ and remain within the same order of magnitude for all sizes. At $n = 2048$, the naive `Original` kernel needs about 606 s, whereas the cache-aware version reduces this to 461 s. The tiled kernel with block size 64 is slightly slower than the cache-aware kernel in this configuration but stays very close.

NumPy is consistently orders of magnitude faster: for $n = 2048$ it completes the multiplication in about 0.26 s, which is approximately a factor of 2350× faster than the naive implementation.

5.1.2 Execution Time (Figure)

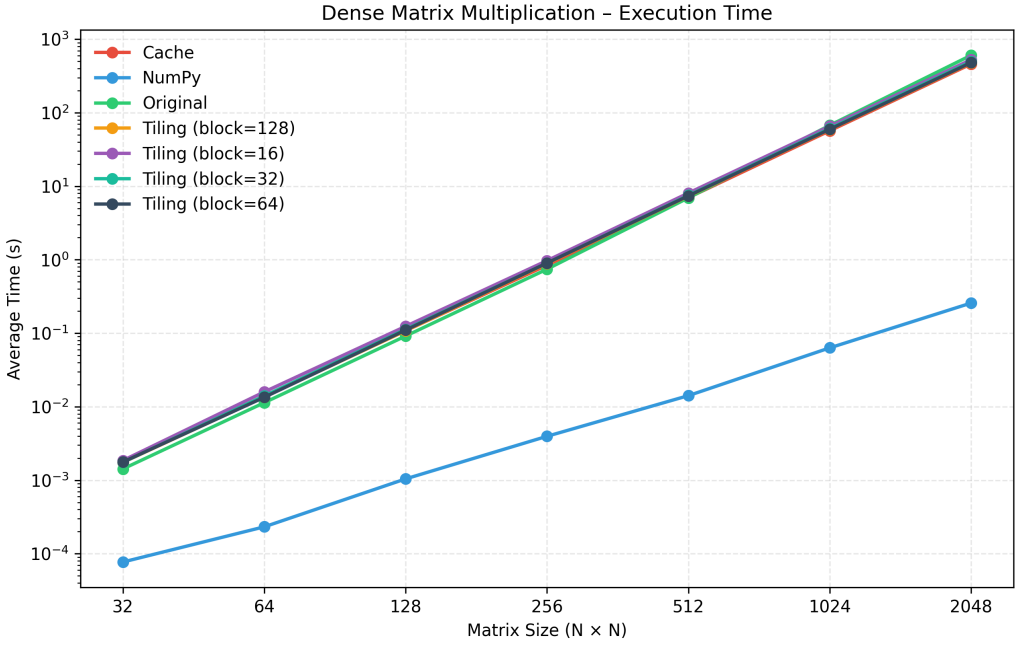

Figure 2: Dense matrix multiplication: execution time vs. matrix size for all methods (log–log scale).

Figure 2 confirms the cubic scaling: the lines for the loop-based kernels appear almost parallel in log–log scale, while the NumPy curve is much lower across all sizes.

5.1.3 Peak Memory (Table)

Table 2 reports the peak memory usage.

Table 2: Peak memory usage (MB) for dense multiplication by method and matrix size.

Method	Size: 32	64	128	256	512	1024	2048
Original	69.32	70.02	71.71	82.56	113.95	220.13	568.18
Cache	69.32	70.02	71.71	82.09	115.66	223.14	568.58
Tiling (b=64)	69.35	70.05	71.97	82.58	120.14	239.60	599.59
NumPy (ref.)	69.34	70.04	72.75	82.08	120.09	227.05	578.16

All dense methods require very similar amounts of memory because they store the same three $n \times n$ matrices plus small temporary buffers. Memory grows roughly quadratically with n, from about 70 MB at $n = 32$ to around 570–600 MB at $n = 2048$.

5.1.4 Peak Memory (Figure)

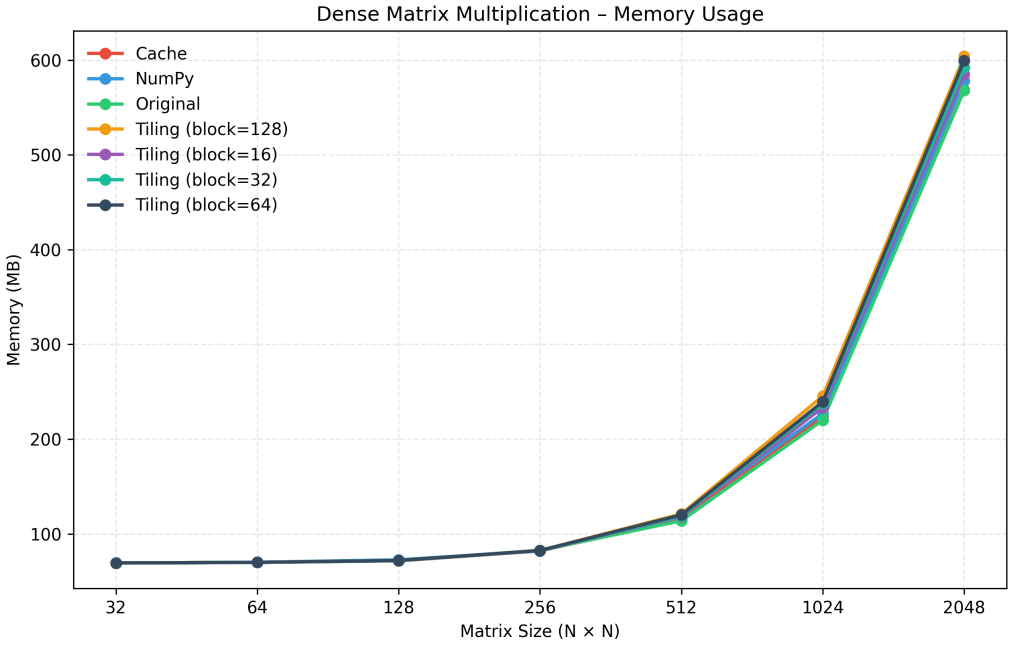

Figure 3: Dense matrix multiplication: peak memory vs. matrix size for all methods.

5.2 Sparse Matrix Multiplication

5.2.1 Execution Time and Structural Metrics

Table 3 summarises the sparse CSR results for the largest tested matrix size ($N = 2048$) across all synthetic sparsity patterns.

Table 3: Sparse CSR multiplication: time, sparsity and structural metrics for the $N = 2048$ matrix size.

Size	Pattern	Sparsity (%)	nnz	nnz/row	Mean time (s)	Memory (MB)
2048	Random 50% zeros	50.00	2097152	1024.0	3.714193	170.18
2048	Random 75% zeros	75.00	1048576	512.0	0.950887	188.18
2048	Random 90% zeros	90.00	419430	204.8	0.223792	195.45
2048	Random 95% zeros	95.00	209715	102.4	0.112145	204.09
2048	Random 99% zeros	99.00	41943	20.5	0.006051	162.44

The random pattern has been tested across various matrix sizes, specifically $N \in \{32, 64, 128, 256, 512, 1024, 2048\}$. However, the discussion has been focused on the largest size, $N = 2048$, as it is the most significant for evaluating real-world, demanding workloads and clearly illustrates the impact of sparsity on performance. For this fixed matrix size of $N = 2048$, the performance of the random pattern is highly dependent on the sparsity level. The number of nonzeros per row varies drastically across the tested range, decreasing from 1024.0 at 50% zeros to just 20.5 at 99% zeros. As a consequence, the mean execution time spans several orders of magnitude: it reaches 3.71 s for the densest case (50% zeros) and drops sharply to only 6.05×10^{-3} s (or about 6.1 ms) for the sparsest case (99% zeros). Memory usage remains relatively consistent and modest, ranging between 162 MB and 170 MB across the tested sparsity levels.

5.2.2 Execution Time vs. Size (Figure)

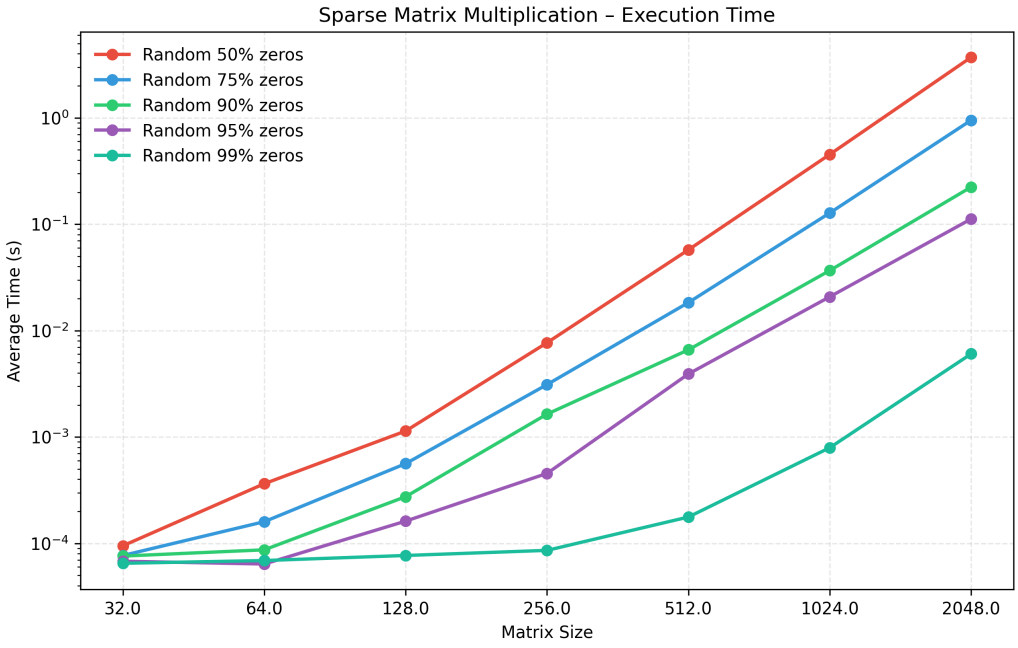

Figure 4: Sparse matrix multiplication: execution time vs. matrix size for each sparse pattern.

5.2.3 Peak Memory vs. Size

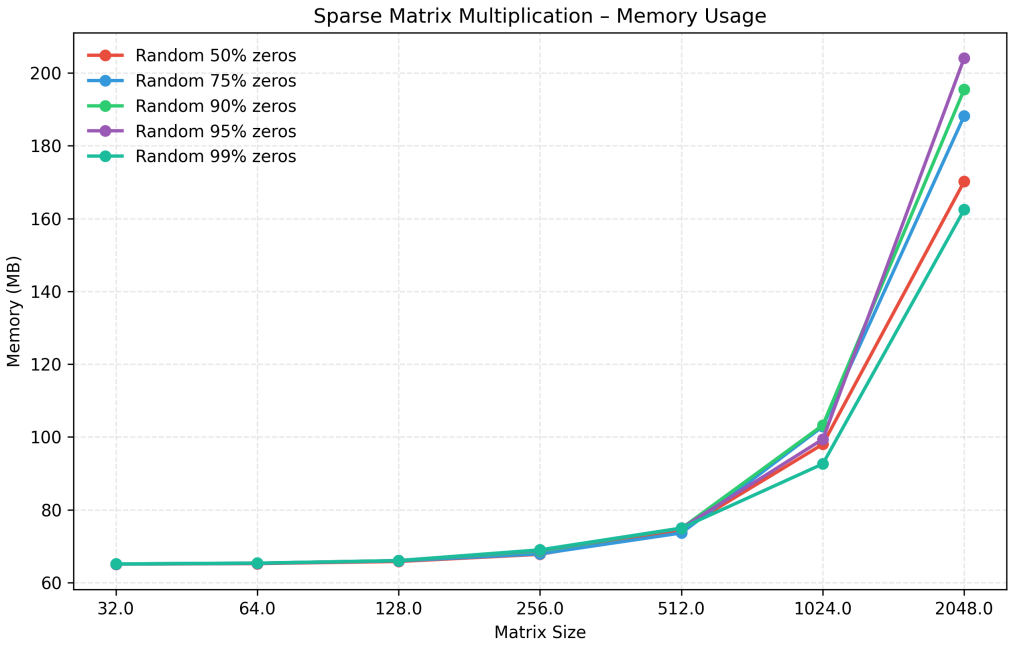

Figure 5: Sparse matrix multiplication: peak memory vs. matrix size for each sparse pattern.

The memory plots mirror the behaviour observed in Table 3: the random pattern quickly approaches the memory footprint of dense matrices.

5.3 `mc2depi` Experiment

5.3.1 Summary Table

Table 4 summarises the results for the full `mc2depi` matrix.

Table 4: Summary metrics for the full `mc2depi` matrix benchmark.

Size	Sparsity (%)	nnz	nnz/row	Mean time (s)	Memory (MB)
525825	99.99920	2100225	4.0	0.043490	161.65

No dedicated plots are generated for `mc2depi`; the analysis is based on this summary table and on comparisons with the synthetic patterns. The key observation is that, despite the huge matrix size (over half a million rows), the number of nonzeros per row remains fixed at 4, so the runtimes stay below 0.04 s and the memory footprint is well under 300 MB.

5.4 Dense vs. Sparse Comparison

Finally, we compare the average behaviour of dense and sparse approaches. The plotter computes the mean time and memory across dense methods and across sparse patterns for each matrix size where both experiments are available and generates the following figures.

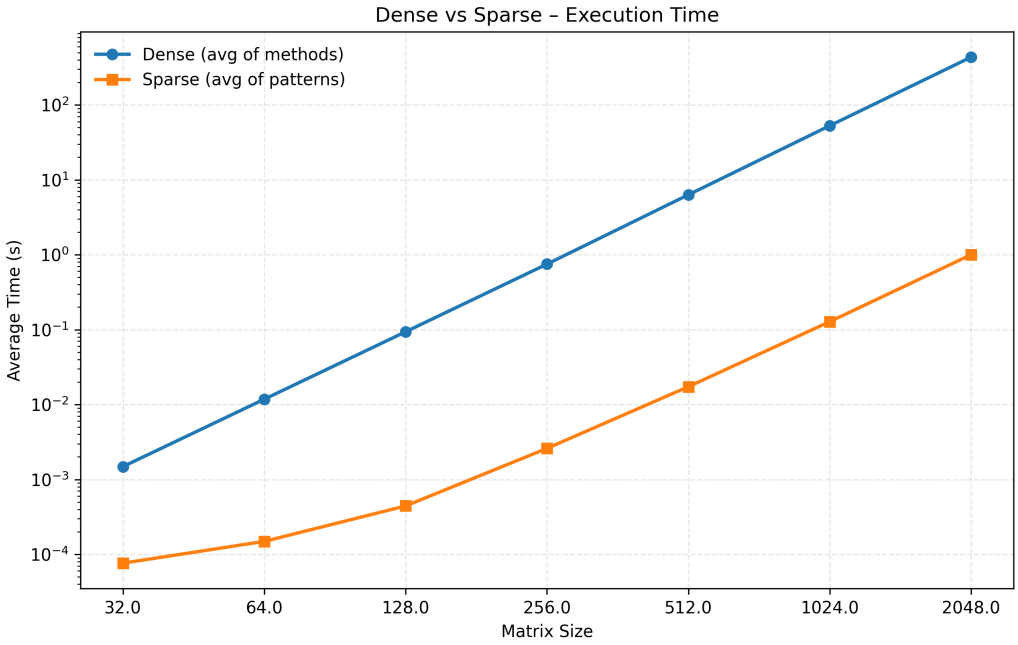

Figure 6: Dense vs. sparse matrix multiplication: mean execution time.

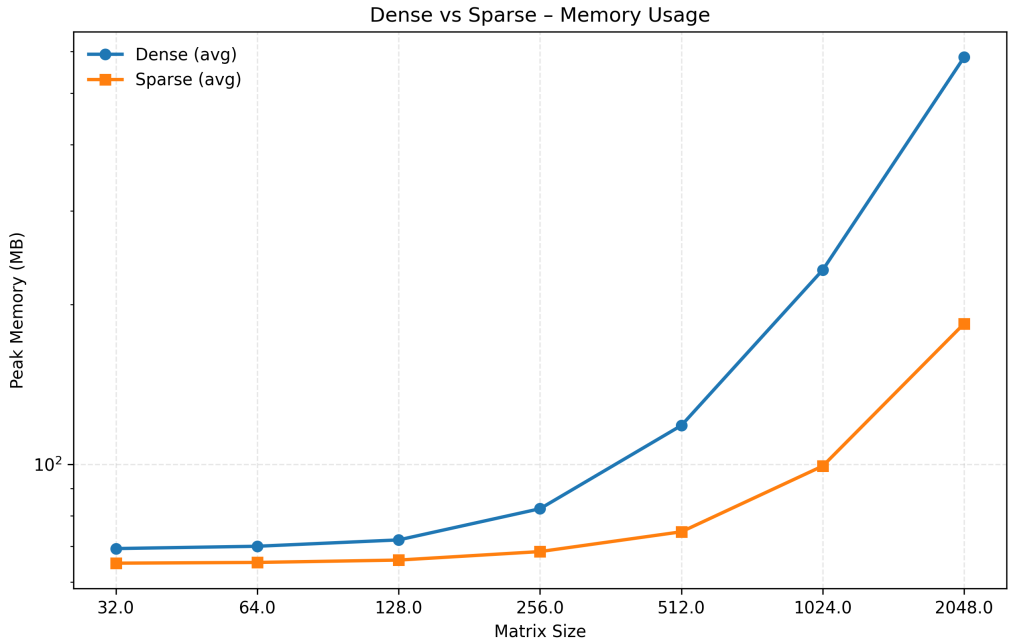

Figure 7: Dense vs. sparse matrix multiplication: mean peak memory.

In log–log scale, the dense curve grows much more steeply than the sparse curve. Around matrix order $n \approx 10^3$, dense multiplication already requires tens of seconds and over 200 MB of memory, whereas sparse multiplication for matrices with a small number of nonzeros per row runs in fractions of a second and uses only tens of megabytes.

6 Discussion

6.1 Dense algorithms

The three loop-based dense algorithms exhibit the expected $O(n^3)$ scaling. The cache-aware version consistently outperforms the naive kernel (e.g. 56.9 s vs. 67.4 s at $n = 1024$), showing that simple loop reordering can yield noticeable speedups without changing the algorithmic complexity. The tiled kernel with block size 64 behaves similarly; its performance is close to the cache-aware version and slightly worse in this configuration, probably because the block size is not perfectly tuned to the hardware cache.

NumPy is by far the fastest method. At $n = 2048$ it is around $2350\times$ faster than the naive implementation. This reflects the benefits of compiled code, vectorization, and highly tuned BLAS libraries. However, all dense methods have a similar memory footprint, which grows quadratically with n and quickly becomes the limiting factor when increasing the matrix size further.

6.2 Sparse Matrix Analysis

The random pattern has been tested across various matrix sizes, specifically $N \in 32, 64, 128, 256, 512, 1024, 2048$. However, the discussion has been focused on the largest size, $N = 2048$, as it is the most significant for evaluating real-world, demanding workloads and clearly illustrates the impact of **sparsity** on performance.

For this fixed matrix size of $N = 2048$, the performance of the random pattern is highly dependent on the sparsity level. The number of nonzeros per row (nnz/row) varies drastically across the tested range, decreasing from 1024.0 at 50% zeros to just 20.5 at 99% zeros.

As a consequence, the mean execution time spans several orders of magnitude: it reaches **3.71** s for the densest case (50% zeros) and drops sharply to only $\mathbf{6.05 \times 10^{-3}}$ s (or about 6.1 ms) for the sparsest case (99% zeros). Memory usage remains relatively consistent and modest, ranging between **162** MB and **170** MB across the tested sparsity levels.

Summary of Performance Range at $N = 2048$

- **Densest Case (50% Zeros):** nnz/row = 1024.0, Time = 3.71 s.
- **Sparsest Case (99% Zeros):** nnz/row = 20.5, Time = 6.05×10^{-3} s.

7 Conclusion

This assignment implements and benchmarks several optimized dense kernels and CSR-based sparse matrix multiplications in a unified Python framework. For dense matrices, cache-aware and blocked algorithms improve locality over the naive triple loop but still exhibit cubic scaling and large memory requirements. NumPy, backed by optimized BLAS libraries, provides a strong baseline and dramatically outperforms hand-written Python loops.

For sparse matrices, the experiments demonstrate that both sparsity and structure are critical. Random 90% sparse matrices with many nonzeros per row behave almost like dense matrices in terms of cost, whereas matrices with only a handful of nonzeros per row (`mc2depi`) can be multiplied extremely fast and with moderate memory usage, even at very large sizes. The `mc2depi` matrix thus provides a practical reference point for large, highly sparse problems.

Overall, the results and code form a reproducible baseline for further optimizations, such as parallel or distributed versions in future assignments.

Environment and Platform

All experiments were executed on:

- CPU: *(Ryzen 7 6000 Series)*.
- RAM: *(16 GB)*.
- Operating system: *(Windows 11)*.

How to Reproduce

The project for Task 2 is a pure Python project. To reproduce the benchmarks and plots:

1. Move into the Task 2 directory (the folder that contains `main.py`, `config.py`, and the `dense`, `sparse`, and `utils` subfolders):

   ```
   cd "directory_name"
   ```

2. Run the main benchmark script:

   ```
   python main.py
   ```

   This produces a timestamped CSV file under the directory configured as `CSV_RESULTS_DIR in config.py`.

3. Generate all figures from the latest CSV:

   ```
   python benchmark_plotter.py
   ```

   The plotter reads the most recent CSV in `CSV_RESULTS_DIR` and writes the plots to the directory configured as `PLOT_RESULTS_DIR in config.py`.

4. If desired, matrix sizes, block sizes, and number of runs can be modified directly in `config.py` before running the benchmarks.

Disclaimer

Parts of the benchmarking and plotting code were developed with the assistance of an AI code assistant (ChatGPT).