Task 2: Optimization and Sparse Matrix Multiplication Benchmark

Pablo Mendoza Rodríguez

November 2025

Abstract

This work extends a previous cross-language matrix multiplication benchmark by focusing on algorithmic optimizations and sparse matrix representations. We implement and compare several dense algorithms (baseline cubic, cache-optimized blocked multiplication, and a loop-unrolled variant), as well as a sparse matrix–dense matrix product using the CSR format. We evaluate execution time, memory usage, and scalability as matrix size and sparsity vary, and we relate our findings to practical considerations on modern multi-core architectures. In addition to random sparse matrices, we also run an experiment on the SuiteSparse matrix `mc2depi`, as requested in the assignment.

1 Introduction

Matrix multiplication is a fundamental operation in scientific computing, machine learning and numerical simulation. In Task 1 we analyzed how the execution model of different languages (Python, Java and C++) affects the performance of a classical $\mathcal{O}(n^3)$ dense matrix multiplication algorithm. In this second task, we shift the focus from language-level differences to algorithmic optimizations and the impact of sparsity.

The main goals of this work are:

- To compare different dense matrix multiplication kernels implemented in C++: a baseline triple-loop algorithm, a blocked/tiling version and a manually loop-unrolled variant.
- To implement a sparse matrix–dense matrix multiplication using a Compressed Sparse Row (CSR) representation and to study how the sparsity level affects performance.
- To analyse the trade-offs in terms of execution time, memory footprint and scalability, and to identify the regimes where sparse representations are advantageous.

2 Methodology

All dense algorithms operate on $n \times n$ matrices stored in contiguous `std::vector<double>` buffers in row-major order.

2.1 Dense Algorithms

Baseline (`dense_ijk`). The baseline implementation uses the classical triple-nested loop for matrix multiplication. The loop order is i–k–j, which iterates over rows of A and C and scans rows of B sequentially. This ordering matches the row-major layout and provides good cache locality for both B and C.

Blocked / tiled (`dense_block`). The blocked kernel partitions the matrices into square tiles of size $B \times B$ (with a default block size of $B = 64$). The computation is restructured as a triple loop over blocks, followed by inner loops over elements inside each tile. The goal is to keep small sub-blocks of A, B and C in cache while performing many multiply–accumulate operations, thus reducing cache misses for large matrices.

Loop-unrolled (`dense_unroll`). The loop-unrolled variant keeps the same logical algorithm as the baseline, but manually unrolls the innermost loop over the column index by a factor of four. This reduces the number of loop-control instructions and aims to expose more instruction-level parallelism to the compiler.

2.2 Sparse Representation and Algorithm

To exploit sparsity, we adopt the Compressed Sparse Row (CSR) format for an $n \times n$ matrix A. A CSR matrix is defined by three arrays:

- `row_ptr` (size $n + 1$) stores prefix sums of non-zero counts per row, so that non-zeros in row i lie in indices `row_ptr[i]` to `row_ptr[i+1]-1`.
- `col_idx` stores the column index of each non-zero.
- `values` stores the corresponding non-zero values.

We generate random sparse matrices by including each potential entry independently with a given density $d \in (0, 1]$, defined as the fraction of non-zero entries. The sparse kernel computes

$$C = A_{\text{CSR}} B,$$

where A is stored in CSR format and B is a dense $n \times n$ matrix. For each row i of A, the algorithm iterates over its non-zero entries a_{ik}, multiplies a_{ik} by row k of B, and accumulates the result into row i of C. The complexity of this kernel is $\mathcal{O}(n \cdot \text{nnz\_per\_row}) \approx \mathcal{O}(n^2 d)$.

2.3 Experimental Setup

All benchmarks are implemented in C++17 and compiled with `clang++ -O3 -std=c++17`. The experiments are executed on the same machine used in Task 1 (64-bit macOS; CPU, core count and RAM can be summarised in a small table if required).

We evaluate the following configurations:

- **Dense algorithms.** Matrix sizes $n \in \{512, 1024, 1536, 2048\}$, using the three kernels `dense_ijk`, `dense_block` and `dense_unroll`.
- **Sparse CSR (random).** Matrix sizes $n \in \{5000, 10000\}$ and densities $d \in \{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$ for the `sparse_csr` kernel.
- **Sparse CSR (SuiteSparse matrix `mc2depi`).** One additional experiment uses the real matrix `mc2depi` from the SuiteSparse collection, stored in Matrix Market format. The matrix is loaded into CSR and multiplied by a dense matrix B with a small number of columns to keep memory usage under control.
- **Repetitions.** For each (n, d) and algorithm we perform 5 runs (or 3 for `mc2depi`) and report the mean and standard deviation of execution time.

Execution time is measured using `std::chrono` high-resolution timers. Memory usage (in MB) and CPU utilisation (in %) are estimated using `getrusage`. On macOS, `ru_maxrss` is reported in bytes, so it is converted to MB by dividing by 1024^2, ensuring correct values for the memory plots.

3 Results

3.1 Dense Multiplication: Time and Memory vs Size

Figure 1 reports the average execution time of the three dense kernels as a function of the matrix size n. Figure 2 shows the corresponding average memory usage.

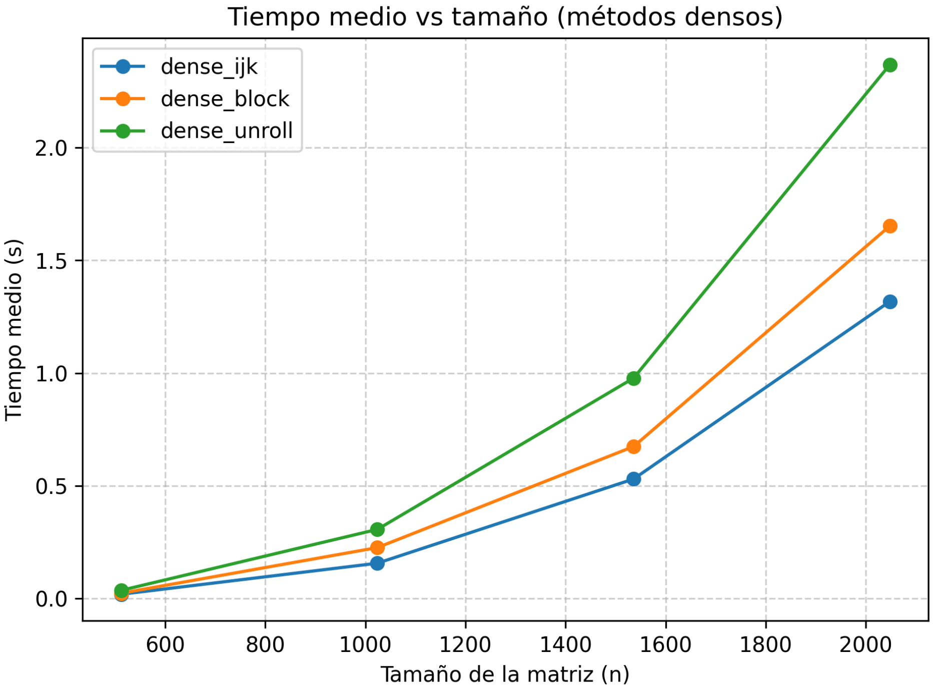

Figure 1: Average execution time of dense algorithms as a function of matrix size n.

All three time curves exhibit a clear cubic trend: when n grows by a factor of two, the runtime increases by roughly one order of magnitude, which is consistent with the $\mathcal{O}(n^3)$ complexity of dense matrix multiplication. The naive `dense_ijk` kernel is consistently the fastest, followed by the blocked version (`dense_block`), while the manually unrolled variant (`dense_unroll`) is the slowest for all tested sizes.

The memory plot shows that the footprint increases approximately as $\mathcal{O}(n^2)$, as expected from storing three dense $n \times n$ matrices (A, B and C). The differences between algorithms are small because they share the same data structures; the additional overhead of blocking and unrolling is negligible compared to the cost of the dense matrices themselves.

3.2 Sparse CSR Multiplication: Time and Memory vs Density

Figures 3 and 4 show the average execution time of the `sparse_csr` kernel as a function of the density d for matrix sizes $n = 5000$ and $n = 10000$, respectively. The horizontal axis is represented on a logarithmic scale.

For both matrix sizes, the runtime increases almost monotonically with the density. For very sparse matrices ($d = 10^{-4}$ and 10^{-3}) the execution time is close to zero, whereas for $d = 10^{-2}$ it becomes clearly noticeable, and for $d = 10^{-1}$ it grows by one to two orders of magnitude. This behaviour is consistent with the complexity of the CSR kernel, which scales as $\mathcal{O}(n^2 d)$: as more entries become non-zero, the number of multiplications and accumulations increases proportionally.

Figures 5 and 6 illustrate the corresponding memory usage; again, the density is shown on a logarithmic scale.

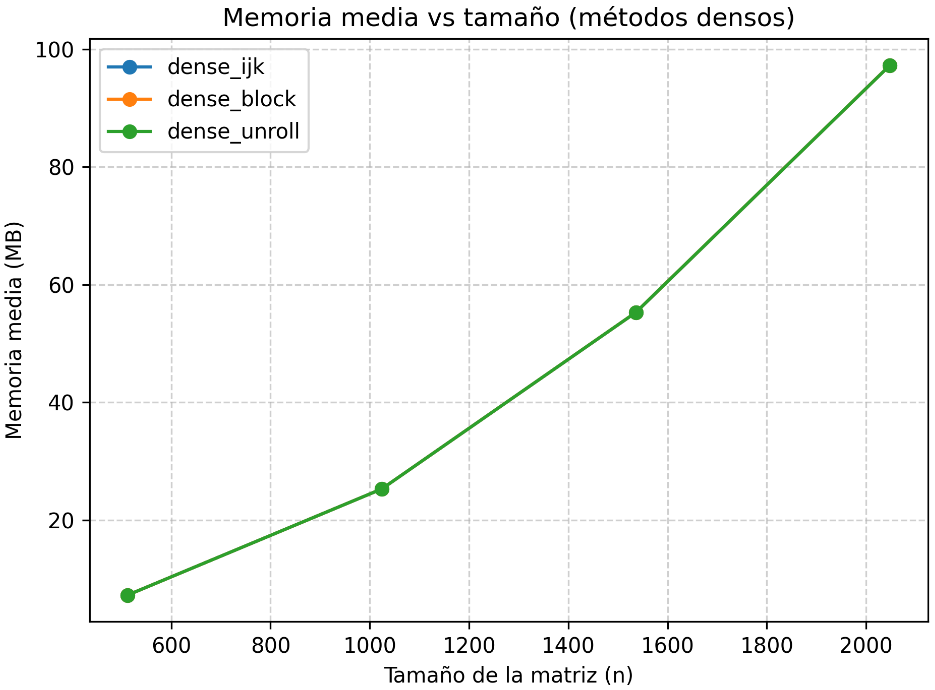

Figure 2: Average memory usage of dense algorithms as a function of matrix size n.

The memory curves are almost flat for low densities and only show a noticeable increase at $d = 10^{-1}$. For $n = 5000$ the average usage remains almost constant for $d \leq 10^{-3}$, increases moderately at 10^{-2}, and grows more at 10^{-1}. For $n = 10000$ we observe a similar pattern: the total footprint is dominated by the dense matrices B and C, which always require $\mathcal{O}(n^2)$ storage and do not depend on the sparsity of A, whereas the CSR structure adds only $\mathcal{O}(\text{nnz})$ on top of that.

3.3 Experiment with the SuiteSparse Matrix `mc2depi`

The assignment explicitly requests using a real sparse matrix from the SuiteSparse collection. For this purpose we include an additional experiment using `mc2depi`. The matrix is loaded from its Matrix Market file into CSR format and multiplied by a dense matrix B with a small number of columns. The reported density is on the order of 10^{-6}, which means that only a tiny fraction of the n^2 entries are non-zero.

Even though the dimension of `mc2depi` is several orders of magnitude larger than the dense matrices used in the previous subsections, the CSR kernel is still able to perform the multiplication on a single machine. A dense representation of `mc2depi` would be completely infeasible in terms of memory, so this experiment clearly illustrates the advantage of sparse formats for genuinely sparse real-world problems.

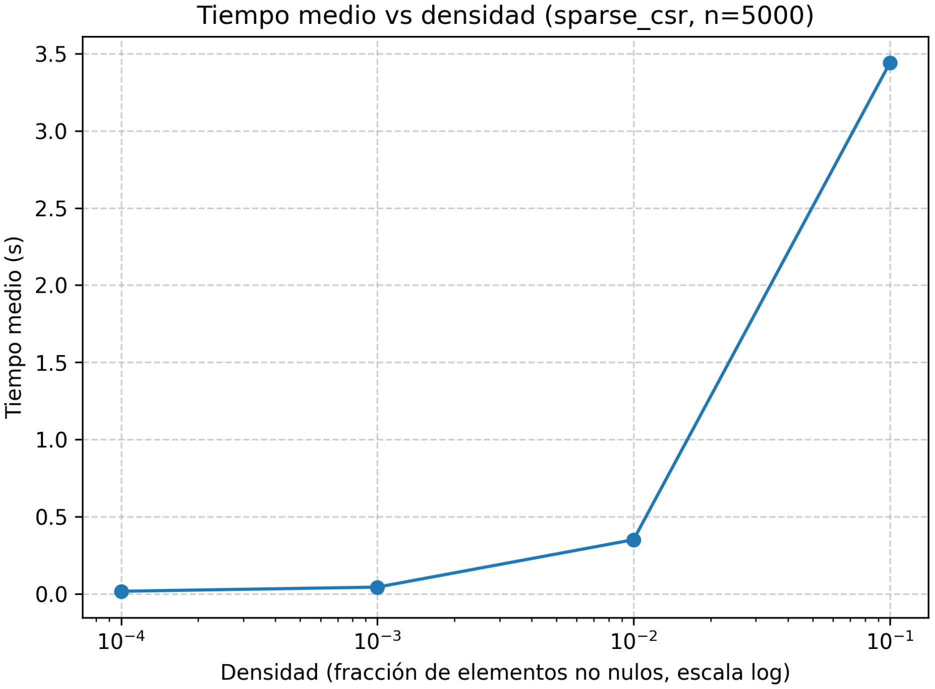

Figure 3: Average execution time versus density for the CSR kernel, $n = 5000$.

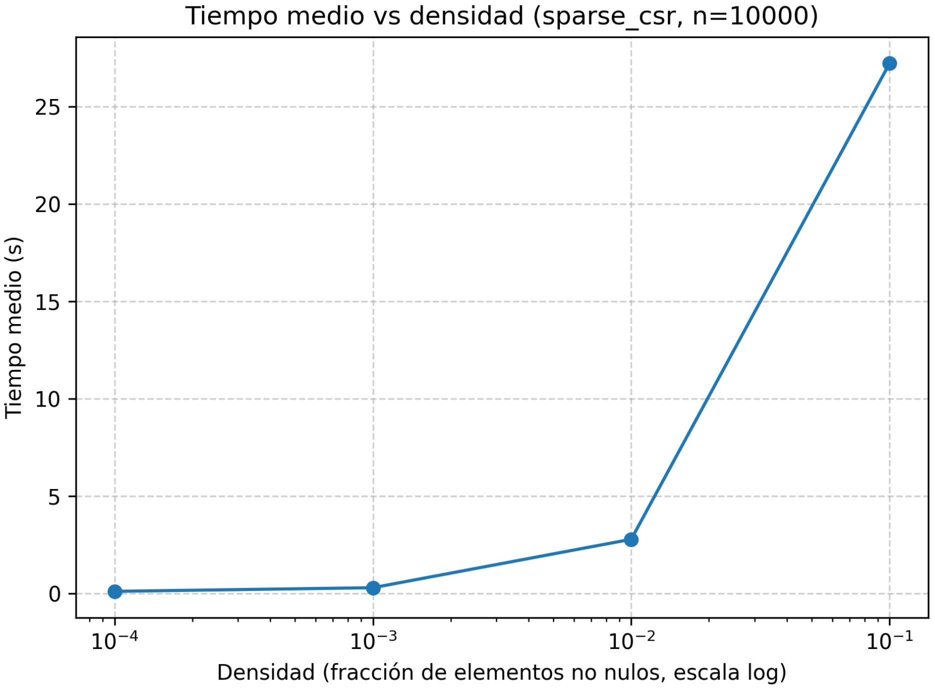

Figure 4: Average execution time versus density for the CSR kernel, $n = 10000$.

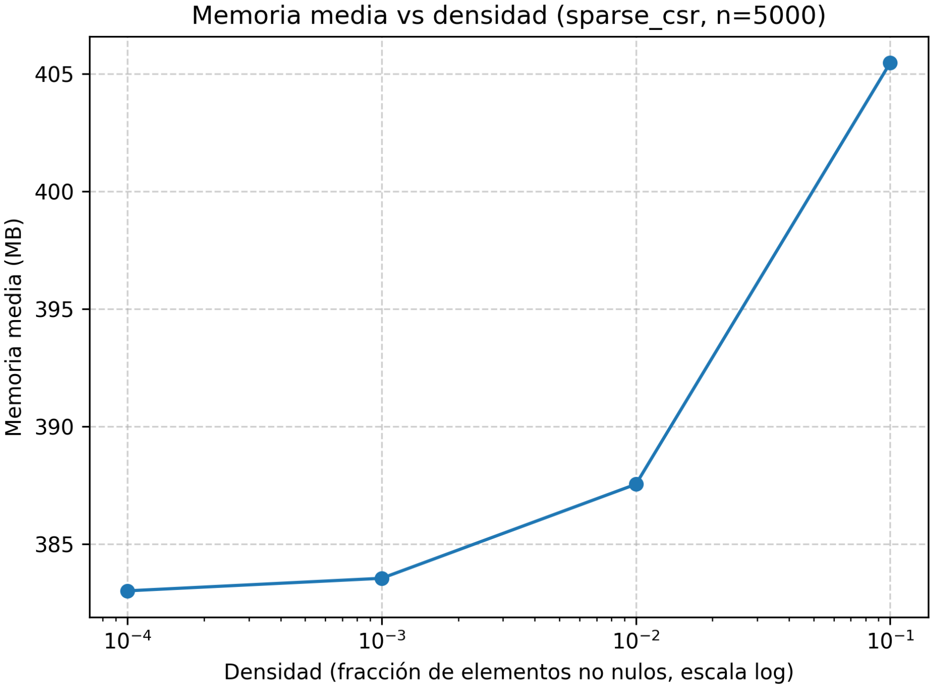

Figure 5: Average memory usage versus density for the CSR kernel, $n = 5000$.

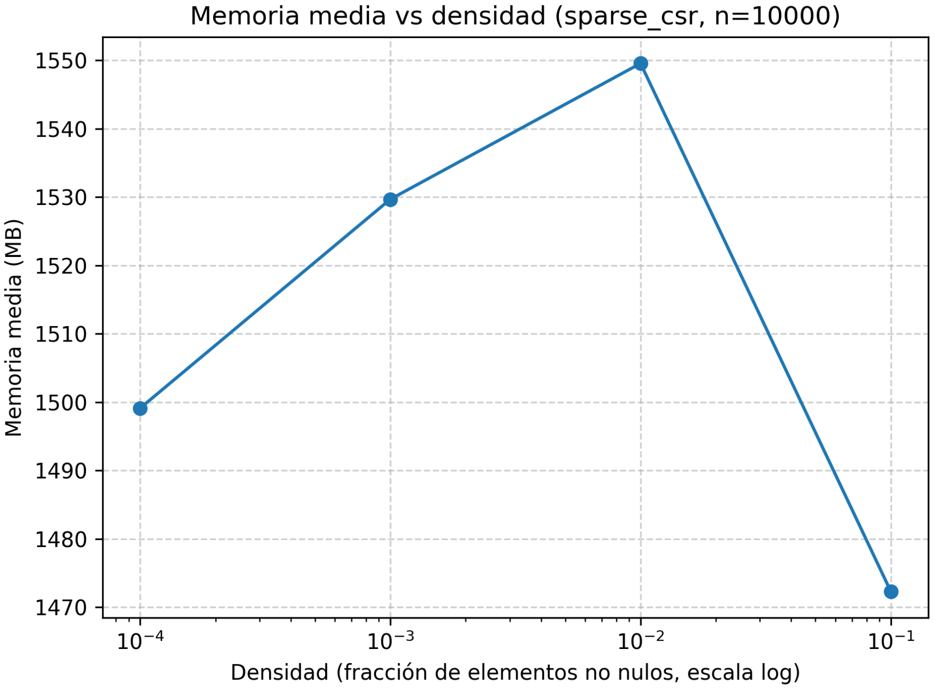

Figure 6: Average memory usage versus density for the CSR kernel, $n = 10000$.

4 Discussion

4.1 Dense Algorithms

The dense experiments highlight an important practical point: algorithmic optimisations must be evaluated empirically, as their benefits depend not only on asymptotic complexity but also on compiler behaviour, cache sizes and problem dimensions.

In principle, blocking should improve cache locality and therefore reduce execution time, whereas loop unrolling is expected to decrease loop overhead and expose more instruction-level parallelism. However, modern compilers already perform aggressive optimisations at `-O3`, including loop reordering, automatic unrolling and vectorisation. Our baseline `dense_ijk` kernel is compiled into highly optimised code.

The blocked implementation introduces additional control logic: three extra loops, bound checks and index computations. For the matrix sizes considered in this task ($n \leq 2048$), this overhead is not fully compensated by the improved cache reuse, and the kernel ends up being slightly slower than the baseline. The manually unrolled variant further complicates the loop body and can interfere with the compiler's own unrolling and vectorisation heuristics, resulting in the poorest performance among the three.

4.2 Sparse CSR Kernel

The sparse results confirm the expected dependence on density. For fixed n, the runtime of the CSR kernel grows almost linearly with the density d, reflecting the complexity $\mathcal{O}(n^2 d)$. When the matrix is extremely sparse ($d \leq 10^{-3}$), only a tiny fraction of entries are processed and the cost is negligible compared to dense multiplication. As d reaches 10^{-2} and 10^{-1}, the number of non-zeros becomes large enough that the cost approaches that of dense algorithms.

Memory usage exhibits a similar but less pronounced trend. The total footprint is dominated by the dense matrices B and C, which always require $\mathcal{O}(n^2)$ storage and do not depend on the sparsity of A. The CSR structure adds only $\mathcal{O}(\text{nnz})$ memory on top of that. Thus, for very low densities the additional memory is negligible, and only for $d = 10^{-1}$ does the CSR storage become comparable to the cost of storing a dense matrix.

Overall, these experiments show that CSR is highly effective for genuinely sparse matrices, providing substantial time and memory savings when the density is below 10^{-2}. For higher densities, the benefits of the sparse representation diminish and the kernel gradually approaches the cost of dense multiplication.

5 Conclusions

In this task we extended a previous language-level matrix multiplication benchmark by focusing on algorithmic optimisations and sparse matrices in C++. We implemented three dense kernels and a CSR-based sparse matrix–dense matrix multiplication, and evaluated them across varying matrix sizes and densities, including the real-world SuiteSparse matrix `mc2depi`.

The main conclusions are:

- All dense implementations exhibit the expected $\mathcal{O}(n^3)$ scaling, but the naive triple-loop kernel compiled with `-O3` is already highly efficient and generally outperforms the manually blocked and unrolled versions for the problem sizes considered.

- Manual optimisations such as tiling and unrolling can introduce overhead and interfere with compiler optimisations; their impact must therefore be assessed experimentally rather than assumed.

- The CSR-based sparse kernel scales approximately as $\mathcal{O}(n^2 d)$ and is extremely efficient when matrices are truly sparse. Both execution time and memory usage remain low for densities below 10^{-2}, but increase significantly when the fraction of non-zero entries approaches 10^{-1}.

- In terms of "maximum matrix size handled efficiently", dense kernels are comfortable up to the largest n tested here, while sparse CSR enables working with much larger problems such as `mc2depi`, which would be impossible to store densely.

Repository and Reproducibility. All source code, scripts and result files used in this task are available in the project repository:

- GitHub: `https://github.com/PabloMr05/matrix-bench`

References

[1] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix–vector multiplication on emerging multicore platforms. In *Proceedings of the 2009 Conference on High Performance Computing Networking, Storage and Analysis*, 2009.