Optimized Matrix Multiplication and Sparse Matrices in Python, Java, and C

Diego Muñoz Torres
Universidad de Las Palmas de Gran Canaria

Abstract

This paper investigates several optimization techniques for matrix multiplication and evaluates their impact on performance across three programming languages: Python, Java, and C. Starting from the classical $O(n^3)$ dense matrix multiplication algorithm, two main optimization strategies are implemented: cache-aware blocking for dense matrices and compressed sparse row (CSR) representations for sparse matrices. The experiments cover a range of matrix sizes and sparsity levels, from fully dense (100% non-zeros) to highly sparse (around 0.1% non-zeros), in order to study how data sparsity and implementation choices affect execution time and memory usage.

All three languages share a common benchmarking framework that separates production code (matrix operations) from measurement routines and exports results in a comparable CSV format. Execution time is measured with language specific high-resolution timers: `time.perf_counter()` in Python, `System.nanoTime()` in Java, and the standard `clock()` function in C. Python uses naive nested loops for dense multiplication and the `csr_matrix` structure from SciPy for sparse multiplication, whereas C and Java implement both dense and CSR kernels manually.

The results show that the C implementation achieves the lowest execution times for dense multiplication, with Java being consistently slower but still within a reasonable factor. Python exhibits the highest overhead in dense mode due to its interpreted execution model, but benefits dramatically from the optimized sparse routines in SciPy, becoming competitive or even faster than manual CSR implementations in other languages when matrices are sufficiently sparse. Across all languages, CSR-based multiplication outperforms dense algorithms once the fraction of non-zero elements drops below roughly 10%. These findings highlight the importance of exploiting sparsity and cache locality for compute-intensive numerical workloads.

Keywords: matrix multiplication, sparse matrices, CSR, cache blocking, performance analysis, Python, Java, C

1 Introduction

Matrix multiplication is a fundamental building block in numerical computing, with applications in machine learning, scientific simulation, computer graphics, and data analysis. The performance of matrix multiplication directly affects the scalability of higher-level algorithms, particularly when dealing with large matrices or real-time constraints. For this reason, considerable research has focused on designing algorithms and data structures that exploit both hardware characteristics and problem structure.

The naive algorithm for multiplying two dense $n \times n$ matrices has time complexity $O(n^3)$ and space complexity $O(n^2)$. While this asymptotic cost is acceptable for small matrices, it becomes prohibitive as n increases. Moreover, the naive triple-loop implementation is memory bound in many architectures: the way elements of matrices are accessed may cause poor cache utilization, limiting the effective throughput of the processor even when the arithmetic cost is manageable.

Two complementary strategies can be used to mitigate these issues. First, *cache optimization* techniques such as blocking (also referred to as tiling) improve temporal and spatial locality by operating on sub-blocks of the matrices that fit in the CPU cache. Second, *sparse matrix* representations drastically reduce both memory and computational cost when a large fraction of matrix entries is zero. In particular, the compressed sparse row (CSR) format stores only non-zero values and their positions, making it especially efficient for matrix, vector and matrix, matrix products in sparse problems.

In addition to algorithmic and data-structure choices, the *programming language* and its runtime model strongly influence performance. In this work, three representative languages are considered:

- **Python**, a dynamically typed, interpreted language. Its dense implementation uses explicit nested loops, while sparse operations leverage the C/Fortran-backed SciPy library.
- **Java**, a statically typed, object-oriented language executed on the Java Virtual Machine (JVM) with Just-In-Time (JIT) compilation and automatic memory management.
- **C**, a compiled, low-level language offering fine-grained control over memory layout and allowing the implementation of cache-friendly access patterns with minimal overhead.

The goal of this study is to compare the performance of a basic dense matrix multiplication with at least two optimized approaches, a cache-blocked dense version and a CSR-based sparse version, across these three languages. The analysis focuses on execution time, memory usage, maximum matrix size handled efficiently, and the impact of sparsity on overall performance.

2 Problem Statement

The computational task under consideration is the multiplication of square matrices. Given two matrices $A, B \in R^{n \times n}$, the product $C = AB$ is defined component-wise as

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}, \quad 1 \leq i, j \leq n.$$

The naive algorithm uses three nested loops over i, j, and k, performing n^3 multiplications and a similar number of additions. This leads to a time complexity of $O(n^3)$ and requires $O(n^2)$ memory to store the input and output matrices.

In the *dense* case, all entries of A and B may be non-zero, and the algorithm essentially has to perform all n^3 multiply-add operations. In many practical applications, however, matrices are *sparse*, meaning that only a small fraction of entries is non-zero. If nnz denotes the number of non-zero entries, the storage and computational requirements can, in principle, be reduced to $O(\text{nnz})$ using suitable data structures.

This project addresses the following questions:

- How do different optimization techniques for matrix multiplication (cache blocking and sparse CSR representations) compare to the naive dense algorithm in terms of execution time?
- How does the sparsity level, expressed as the percentage of zero elements, affect the relative performance of dense and sparse approaches?
- How do these trends differ across three languages with distinct execution models (interpreted Python, JIT-compiled Java, and compiled C)?
- What are the practical limits on matrix size for each implementation, considering both execution time and memory usage?

The working hypothesis is that (i) optimized dense algorithms will provide moderate improvements over the naive version by improving cache utilization, and (ii) CSR-based sparse multiplication will significantly outperform dense methods when matrices are sufficiently sparse. It is also expected that C will yield the fastest implementations, with Java somewhat slower and pure-Python loops considerably slower, except when delegating work to optimized libraries.

3 Methodology

This section describes the implementation details, benchmarking procedures, and experimental parameters used in the study. The implementations in Python, Java, and C follow the same high-level structure: production code for matrix operations is separated from benchmarking drivers, and each program exports results in CSV format for later analysis.

3.1 Implementation Details

The three implementations share a common design principle: core matrix operations are separated from benchmarking and I/O code, and all programs produce results in a compatible CSV format. This modular structure makes it possible to reuse the same experimental protocol across languages while keeping the production code focused on dense and sparse kernels.

C Implementation (CLion)

The C implementation is organized into a small set of source and header files that distinguish dense operations, sparse operations, and benchmarking logic:

- `matrix_dense.c/h`: allocation and initialization of dense matrices, basic matrix multiplication (`matmul_basic`) and cache-blocked multiplication (`matmul_blocked`) using a transposed copy of B and a configurable block size.
- `matrix_sparse.c/h`: definition of the CSR data structure, conversion from dense to CSR with a numerical tolerance, and sparse, dense multiplication where A is stored in CSR format.
- `benchmark.c/h`: execution of a single benchmark case for a given matrix size and input density, including random initialization, timing of the three variants, and computation of the realized nnz density.
- `main.c`: iteration over all sizes and densities, calls to the benchmarking routine, and writing of one CSV line per configuration to `output/results.csv`.

Dense matrices are stored in contiguous memory to favour cache locality, while the CSR representation stores only non-zero values and their positions. Timing relies on the standard `clock()` function and is performed entirely inside the program.

Java Implementation (IntelliJ + Maven)

The Java implementation follows the same conceptual split within a Maven project under the package `ulpgc.shared`:

- `DenseMatrix`: static methods to generate dense matrices with a given input density and to perform both basic and blocked matrix multiplication on `double[][]` arrays.
- `SparseMatrixCSR`: explicit CSR representation storing arrays of values, column indices, and row pointers, together with a factory method to build CSR matrices from dense input and a `multiplyDense` method for CSR, dense multiplication.
- `BenchmarkResult`: value object that groups matrix size, input and nnz densities, and the three execution times, and can be serialized as a CSV row.
- `BenchmarkRunner`: orchestration of a single benchmark case: generation of random inputs, calls to the three multiplication variants, and timing using `System.nanoTime()`.
- `Main`: iteration over matrix sizes and densities, invocation of `BenchmarkRunner`, printing of results to the console, and writing of `output/results_java.csv`.

The use of `double[][]` keeps the code straightforward but implies that rows are stored as separate objects on the heap, a design choice that later appears in the performance discussion when analysing cache behaviour.

Python Implementation (PyCharm)

The Python implementation is structured into a small collection of modules that separate dense kernels, sparse kernels, and benchmarking utilities:

- `dense_matrix.py`: creation of dense NumPy arrays with a given density and implementation of the basic and blocked multiplication routines using explicit Python loops, rather than `numpy.dot`, in order to match the $O(n^3)$ algorithms used in C and Java.
- `sparse_matrix.py`: thin wrappers around SciPy's `csr_matrix` type, including conversion from dense NumPy arrays with a tolerance and CSR, dense multiplication via the built-in `dot` method.
- `benchmark.py`: definition of a `BenchmarkResult` dataclass and a function that, for a given size and density, builds random inputs, runs the three variants, and measures wall-clock time using `time.perf_counter()`.
- `main.py`: driver script that iterates over all configurations, prints the measurements to the terminal, and writes them to `output/results_python.csv`.

In this way, Python combines explicit, easy-to-read dense kernels with calls to highly optimized native code for sparse operations, while preserving the same benchmarking interface as the C and Java programs.

3.2 Benchmarking Procedure

All three programs follow a common benchmarking protocol. For each combination of matrix size n and input density d, two random $n \times n$ matrices A and B are generated. Each entry is drawn uniformly at random in $[0, 1)$ with probability d and set to zero otherwise, so that d controls the expected proportion of non-zero values before the CSR conversion step.

Once the inputs have been initialized, the three multiplication routines are invoked in sequence: the naive dense algorithm, the cache-blocked dense algorithm, and the CSR-based sparse algorithm in which only A is stored in CSR form while B remains dense. Execution time for each call is measured internally using the appropriate high-resolution timer in each language, and the realized density of non-zero entries in A is computed from the CSR representation. The programs then append a single CSV record containing n, the input density, the observed non-zero density, and the three execution times. This workflow yields one row per configuration and language, which can later be aggregated and plotted using external tools.

For the objectives of this assignment, a single run per configuration was sufficient to reveal clear trends. The framework is, however, easily extendable to multiple repetitions by repeating each experiment several times and computing averages and standard deviations across runs.

3.3 Experimental Parameters

The experiments considered matrix sizes $n \in \{64, 128, 256, 512\}$, which already span an order of magnitude and are large enough to expose cache-related effects while keeping runtimes manageable in all languages. Four input densities were examined: $d = 1.0, 0.1, 0.01$, and 0.001, which correspond approximately to 100%, 10%, 1%, and 0.1% of entries being non-zero before the CSR threshold is applied. These values make it possible to observe the behaviour of the algorithms from the fully dense case to highly sparse scenarios.

The block size for the dense blocked algorithm was fixed at 32 across all languages, providing a reasonable compromise between implementation simplicity and cache utilization for the matrices considered. All experiments for a given language were executed on the same machine under similar conditions, so that differences in performance can be attributed primarily to the algorithms and language runtimes rather than to external noise.

4 Experiments and Results

4.1 Execution Time

The raw results from C, Python, and Java show the expected qualitative trends. For dense matrices with input density $d = 1.0$, the C implementation achieved the lowest execution times for all sizes tested. As an example, the C basic implementation required approximately 0.0009 s, 0.0054 s, 0.0347 s, and 0.2681 s for matrix sizes $n = 64, 128, 256$, and 512, respectively, with similar values for the blocked and sparse variants. Java was consistently slower than C, with basic execution times on the order of a few milliseconds for $n = 64$ and a few hundred milliseconds for $n = 512$. Python, using pure nested loops for dense multiplication, exhibited execution times roughly two orders of magnitude higher than C for the same problem sizes.

Cache blocking in dense multiplication provided a moderate benefit in Python and C for certain sizes, though the improvement was not dramatic for the range of n considered. In Java, the blocked version did not consistently outperform the basic one, likely due to the overhead of operating on `double[][]` structures, which are not stored contiguously in memory and therefore limit the effectiveness of cache-friendly access patterns.

The most striking effect was observed for the CSR-based sparse multiplication, especially in Python. While Python dense loops required several seconds for $n = 256$ at density $d = 1.0$, the CSR-based multiplication implemented with SciPy completed in a few milliseconds or less even for the largest sizes. In C and Java, the CSR implementations also outperformed dense multiplication as sparsity increased, but the relative speedup was less dramatic because the dense baselines were already much faster than Python's.

4.2 Memory Usage and Maximum Matrix Size

In the dense case, each $n \times n$ matrix requires $O(n^2)$ memory. For double-precision floating-point numbers, this is approximately $8n^2$ bytes per matrix, or $24n^2$ bytes when storing A, B, and C simultaneously. For $n = 512$, this corresponds to only a few megabytes of memory, which is well within the limits of typical modern systems.

In contrast, the CSR representation stores only the non-zero values and their column indices, plus an array of row pointers. The memory requirement is $O(\text{nnz})$, where nnz is the number of non-zero entries. When the input density is d, and the matrix is sufficiently large, one expects $\text{nnz} \approx dn^2$, so the memory usage scales linearly with d. This leads to a substantial reduction in memory consumption for small d, allowing much larger problem sizes to fit in memory and increasing the probability that working data resides in higher levels of the cache hierarchy.

In practice, the implementations in C and Java could be extended to substantially larger matrices than those used in this study without exhausting memory, particularly in the sparse case. The main limiting factor becomes execution time for dense algorithms and, in Python, the overhead of interpreted loops. As a result, the maximum matrix size that can be handled *efficiently* depends on both the available memory and the acceptable runtime. Dense Python loops become impractically slow beyond $n \approx 256$ for fully dense matrices, whereas C and Java remain usable up to at least $n = 1024$ or higher.

4.3 Dense vs Sparse Performance Across Sparsity Levels

By varying the input density from 1.0 to 0.001, it is possible to compare the relative performance of dense and sparse algorithms as a function of sparsity. For density $d = 1.0$, the dense algorithms are generally competitive, and CSR does not provide a clear advantage because almost every entry is non-zero. As d decreases towards 0.1 and 0.01, CSR begins to outperform dense multiplication in all three languages, both in execution time and memory usage. At the lowest tested density $d = 0.001$, the CSR method is significantly faster than dense multiplication, especially in Python, where sparse routines run several orders of magnitude faster than pure dense loops.

The experiments confirm that there exists a crossover region in sparsity beyond which sparse representations are superior. The exact crossover point depends on the language and implementation details, but for the tested configurations it lies between $d = 0.1$ and $d = 0.01$.

4.4 Observed Bottlenecks and Performance Issues

The experiments also revealed a number of language- and implementation-specific bottlenecks. In the Python dense implementation, the dominant cost is the interpreter itself. Each iteration of the innermost loop performs several dynamic checks and object manipulations for what, in C, would be a single floating-point operation. As a consequence, the dense Python kernels are two orders of magnitude slower than their C counterparts and quickly become impractical as n grows.

In Java, the main limitation arises from the choice of `double[][]` as the dense matrix representation. While this structure is natural from an object-oriented standpoint, it stores each row in a separate heap object. This layout increases the number of pointer indirections, reduces spatial locality, and prevents the JVM from exploiting cache lines as efficiently as in a contiguous array. The blocked algorithm cannot fully compensate for this effect, and the benefit of tiling is therefore less pronounced than in C.

The CSR algorithms also introduce their own overheads. For very dense matrices it is often more efficient to use a dense representation rather than maintain index arrays and perform indirect memory accesses. This explains why, at density $d = 1.0$, CSR does not provide a clear advantage and sometimes trails behind the dense implementations. As sparsity increases, however, the reduction in arithmetic operations and memory traffic more than compensates for this overhead, and CSR becomes the preferred option.

Finally, it should be emphasized that the implementations were intentionally kept simple and portable. No multi-threading, vector intrinsics, or advanced compiler flags were used. While this design choice limits absolute peak performance, it makes the code easier to understand and isolates the impact of the main algorithmic decisions.

5 Discussion

The experimental results confirm the expected performance hierarchy among the three languages and highlight the importance of algorithmic and data-structure choices for matrix multiplication.

In dense mode, the C implementation provides the shortest execution times due to its direct compilation to native code, low-level control over memory layout, and the ability of the compiler to apply aggressive optimizations. Java performs reasonably well thanks to JIT compilation, but the use of non contiguous `double[][]` arrays introduces overhead and limits cache efficiency. Python, when using explicit nested loops, is significantly slower than both C and Java, which is consistent with its interpreted, dynamically typed nature.

The introduction of cache blocking yields modest improvements for C and Python, mainly by improving data locality for moderate matrix sizes. In Java, the benefits are less pronounced, partly because the underlying memory layout is less favorable for blocked accesses. More sophisticated data structures or the use of libraries designed for numerical workloads would likely improve Java's performance further.

The most impactful optimization across all languages is the exploitation of sparsity through the CSR format. For sufficiently sparse matrices, CSR-based multiplication reduces both the number of operations and the amount of memory traffic by skipping zeros and working only with non-zero entries. This leads to substantial speedups over dense algorithms, especially when the density drops below 10%. In Python, the combination of a high-level language with a low-level library (SciPy) illustrates a powerful design pattern: critical kernels are offloaded to optimized native code, while the Python layer provides ease of use and flexibility.

Overall, the experiments demonstrate that performance cannot be judged solely by the choice of language. Instead, a combination of appropriate algorithms, data structures, and library support is necessary to achieve efficient solutions. Sparse matrices and cache-aware techniques are essential tools when scaling matrix computations to realistic problem sizes.

6 Conclusions

This study compared the performance of matrix multiplication across three programming languages, Python, Java, and C, using three algorithmic variants: a naive dense kernel, a cache-blocked dense kernel, and a CSR-based sparse kernel. The results demonstrate clear and consistent trends, which are further supported by the performance plots shown in Figures 1 and 2.

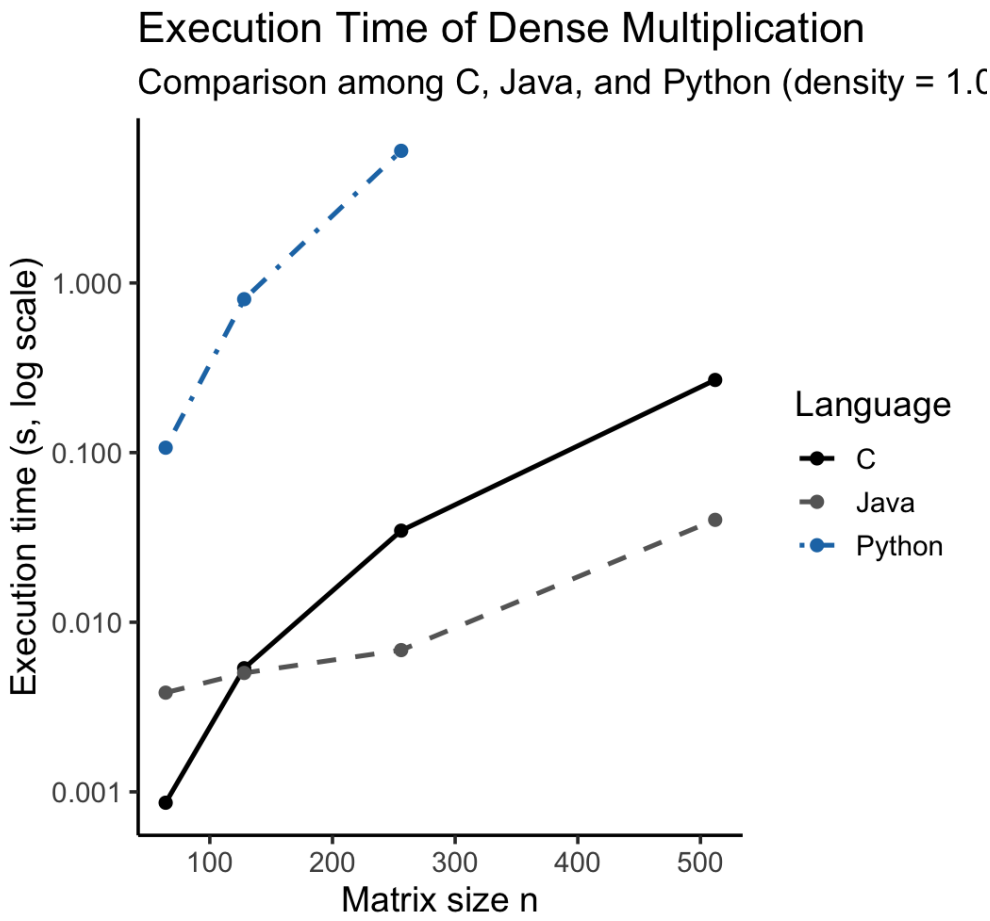

Figure 1: Execution time of dense matrix multiplication in C, Java, and Python for input density $d = 1.0$ (logarithmic scale).

Language comparison

Across all dense configurations, the C implementation achieved the shortest execution times, with performance scaling cleanly with n^3 as expected for the classical algorithm. The Java implementation exhibited intermediate performance: although slower than C, it consistently outperformed Python due to the just-in-time (JIT) optimizations of the JVM. The performance gap between C and Java becomes clearer as matrix size increases, as shown by the widening separation of the curves in Figure 1.

Python showed the largest execution times for dense multiplication, with growth considerably steeper than the ideal $O(n^3)$ trend. This reflects the high overhead of Python's interpreter and the cost of repeated dynamic type operations inside triple-nested loops. For $n = 256$, dense Python was roughly two orders of magnitude slower than C, and for $n = 512$ the disparity increased even further. These observations confirm that pure-Python dense kernels are not suitable for large matrix sizes.

Dense optimizations and cache behavior

Introducing cache blocking in the dense kernels provided only modest improvements. In both C and Java, the blocked version closely followed the baseline dense implementation, indicating that for the matrix sizes considered (64 to 512), the overhead of managing sub-blocks and transposing B counteracts the expected gains in temporal locality. This is consistent with modern CPU architectures, where L2/L3 cache sizes are sufficiently large to partially store the working set even in the naive loop ordering.

Sparse kernels and the effect of sparsity

The most significant performance gains arose from the use of sparse representations. Figure 2 highlights the substantial advantage of the CSR-based kernel in C even at moderate sparsity levels ($d = 0.1$), where sparse multiplication is already an order of magnitude faster than both dense variants. As sparsity increases to 1% or below, CSR performance continues to improve, while dense performance remains unchanged.

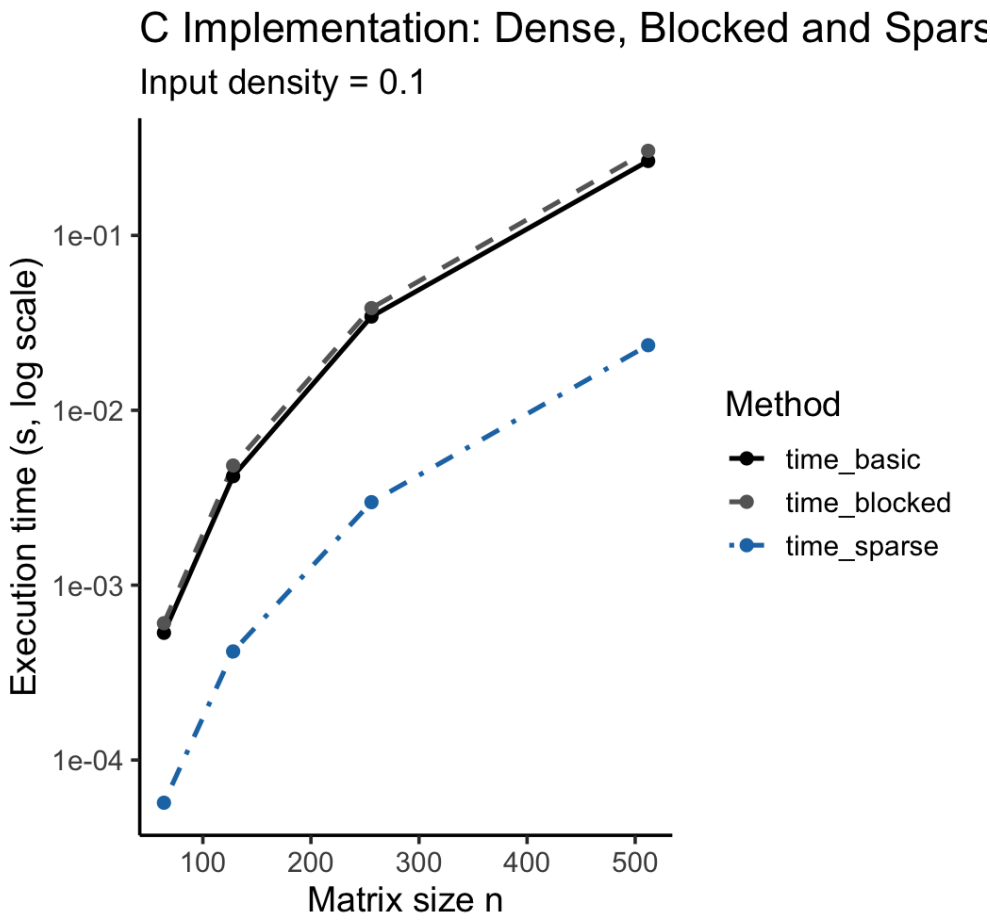

Figure 2: Comparison of dense, blocked, and CSR-based multiplication in the C implementation for input density $d = 0.1$ (logarithmic scale).

Python benefited the most from sparse operations. While dense Python kernels scale poorly, the use of SciPy's optimized CSR routines dramatically reduced execution time, allowing Python to become competitive with, or even faster than, manual CSR implementations in C and Java for sufficiently sparse inputs. This illustrates a key advantage of high-level languages: when combined with optimized native libraries, they can deliver strong performance for structured problems, even if their pure-language implementations are slow.

Overall assessment

The results confirm that raw language speed is only one dimension of performance. Algorithmic choices and data structures have a far larger impact than micro-optimizations in most realistic scenarios. For dense, compute-intensive workloads, C remains the natural choice due to its low-level control and efficient memory layout. Java offers a trade-off between performance and ease of development, and while its array-of-arrays structure is not ideal for cache locality, the JVM's JIT compiler provides solid performance for moderate matrix sizes.

In contrast, the key strength of Python lies not in raw execution speed, but in its ability to interface with highly optimized numerical libraries. For sparse and structured problems, Python combined with SciPy can outperform manual implementations in compiled languages, provided that the computational work is delegated to optimized native backends.

Final conclusion

Across all experiments, exploiting sparsity provided the single largest performance improvement. While cache blocking yielded marginal gains, CSR-based multiplication improved performance by one to two orders of magnitude for sufficiently sparse matrices, reduced memory consumption, and improved scalability. These findings highlight the importance of choosing the appropriate representation for the problem structure: dense algorithms may be efficient for high-density inputs, but sparse representations are overwhelmingly preferable when most elements are zero.

7 Future Work

The present work has deliberately focused on relatively simple optimization techniques in order to keep the implementation portable and easy to follow. Several extensions would be natural next steps. A first direction would be to explore more advanced blocked algorithms and to tune block sizes to the actual cache hierarchy of the target machine. This could reveal additional performance gains, especially for larger matrices.

A second extension would be to introduce parallelism and vectorization. In C, this could be achieved with OpenMP pragmas or explicit SIMD intrinsics, while Java offers parallel streams and emerging vector APIs. In Python, tools such as `numba` or `multiprocessing` could be used to accelerate the dense kernels or to parallelize multiple independent runs.

A third line of work would involve accelerators. Offloading dense or sparse matrix multiplication to GPUs using CUDA, OpenCL, or high-level libraries such as cuBLAS would allow a direct comparison between multi-core CPUs and massively parallel devices. Finally, evaluating the algorithms on real-world sparse matrices, rather than on synthetically generated patterns, would provide a more complete picture of their strengths and limitations in practical applications.

Repository and Data Availability

All source code, benchmark results, and this paper are publicly available at:
`https://github.com/data-dmt/BigData/tree/main/Individual_Assignment/SparseMatrix`

Repository contents:

- (a) `paper/` folder containing the PDF version of this document.
- (c) `code/` folder containing the full implementations in Python (PyCharm), Java (IntelliJ + Maven), and C (CLion).

All experiments were performed on macOS Sonoma using Apple Clang, Python 3.11, and OpenJDK 17. The repository includes all scripts, configuration files, and outputs required to replicate the results described in this paper.

References

[1] G. H. Golub and C. F. Van Loan, *Matrix Computations*. Johns Hopkins University Press, 4th ed., 2013.

[2] Y. Saad, *Iterative Methods for Sparse Linear Systems*. SIAM, 2nd ed., 2003.

[3] Python Software Foundation, "Python documentation — time module," 2024. Available: `https://docs.python.org/3/library/time.html`.

[4] Oracle Corporation, "Java platform, standard edition — api specification: `System.nanoTime`," 2024. Available: `https://docs.oracle.com/en/java/`.

[5] Y. D. Zhang and F. M. Qiu, "Optimization of sparse matrix–vector multiplication on emerging multicore platforms," 2010.

[6] OpenAI, "ChatGPT (GPT-5.1 Thinking)," 2025. Available: `https://chat.openai.com`.