Performance Benchmark of Basic Matrix Multiplication Across Programming Languages

Kacper Janiszewski
Wrocław University of Science and Technology

October 23, 2025

Abstract

This report presents a comparative performance analysis of matrix multiplication implemented in three programming languages: Python, Java, and C. Each implementation follows a naive $O(n^3)$ algorithm, and the study evaluates both execution time and memory usage across different dataset sizes. The goal was to identify performance differences resulting from language design, runtime overhead, and compiler optimizations. Professional benchmarking tools were used: `pytest-benchmark` for Python, `JMH` (Java Microbenchmark Harness) for Java, and `perf` for C.

1 Introduction

Matrix multiplication is a fundamental operation used in scientific computing, graphics, and machine learning. Its computational cost makes it a suitable candidate for cross-language performance evaluation. In this experiment, we compared the same basic algorithm implemented in Python, Java, and C to investigate how language execution models and compilation strategies affect real-world performance.

2 Methodology

The benchmarking setup ensured fairness and reproducibility. Each language implementation was divided into production and test code. Benchmarks were executed multiple times, and average execution times were recorded.

Matrix sizes tested: 10, 50, 100, 200, and 400. **Runs per size:** 10.

2.1 Tools and Environment

- **Python:** Version 3.12, benchmarked using `pytest-benchmark`.
- **Java:** Version 17, benchmarked with `JMH`.
- **C:** Compiled with `gcc -O3 -march=native` and profiled using `perf stat`.

Each implementation used the same algorithmic logic and matrix initialization routine. Only native libraries were used to maintain comparability between the languages.

3 Implementation Details

All implementations follow the same computational flow:

1. Random generation of two $n \times n$ matrices.
2. Standard triple-nested loop matrix multiplication.
3. Measurement of execution time and memory consumption.
4. Averaging over multiple runs.

3.1 Python Implementation

The Python benchmark employed `pytest-benchmark` in pedantic mode to control iteration counts precisely. Memory usage was monitored using `tracemalloc`, and all results were saved to `matrix_bench_python.csv`. At higher matrix sizes ($n \geq 200$), Python's performance degraded sharply, with runtime increasing exponentially due to the interpreter overhead and inefficient list-based memory layout.

3.2 Java Implementation

The Java benchmark used the `JMH` framework, which automatically handled warmup and measurement phases, ensuring stable results. The main benchmarking class, `RunBench`, exported results to `matrix_bench_java.csv`. Compilation into a fat JAR was handled by the Maven Assembly Plugin. Java benefited from Just-In-Time (JIT) optimizations, which allowed it to perform substantially faster than Python while maintaining reasonable memory use.

3.3 C Implementation

The C implementation was compiled with high optimization flags (`-O3 -march=native`) and executed using `perf stat`. Results included wall-clock time, CPU cycles, and cache statistics. Output was parsed and saved into `matrix_bench_c.csv`. This implementation exhibited the best raw performance, owing to direct memory access, static typing, and compiler-level optimizations.

4 Results and Analysis

4.1 Execution Time

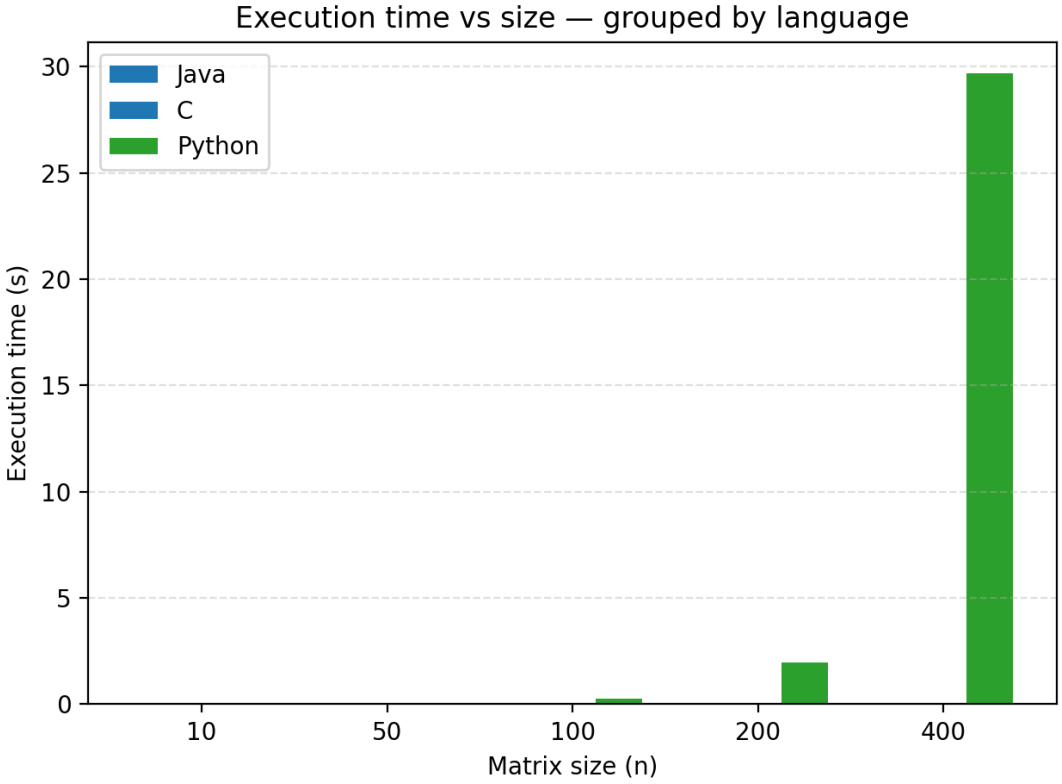

Figure 1: Execution time vs matrix size (grouped by language).

4.2 Memory Usage

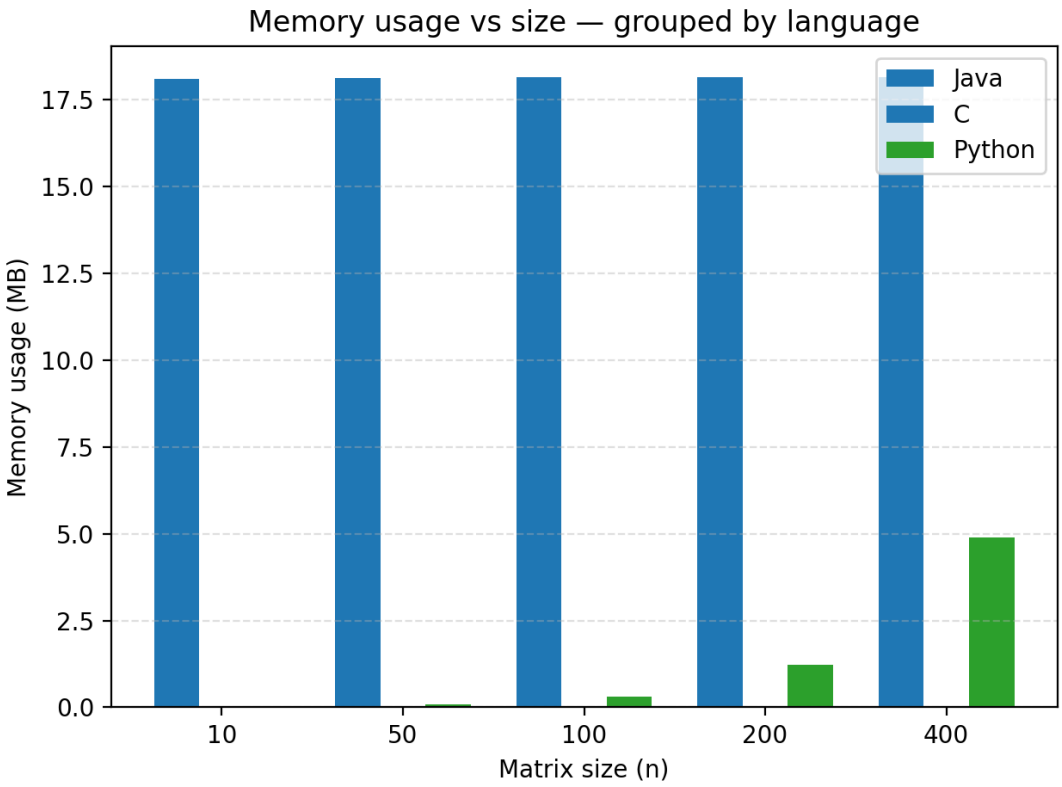

Figure 2: Memory usage vs matrix size (grouped by language).

As expected, C achieved the lowest execution time across all tested matrix sizes. Java demonstrated moderate performance, balancing speed and resource management due to JIT compilation and garbage collection. Python's performance, however, decreased drastically for larger matrices. At size $n = 400$, the Python implementation required several seconds to complete, while Java and C completed in fractions of a second. This behavior highlights the cost of dynamic typing and lack of low-level memory optimization in Python.

5 Profiling and Discussion

Profiling with `perf` confirmed that the C implementation achieved the highest instruction throughput and cache locality. Java performed efficiently thanks to its JIT compiler and automatic optimizations during runtime. Python, while user-friendly and expressive, suffered from high interpreter overhead and frequent memory allocation events. For larger matrices, it became impractical to run additional tests without optimized numerical libraries such as NumPy.

6 Conclusion

This benchmark study illustrates the trade-offs between development productivity and execution performance:

- **C:** Fastest execution, most efficient memory use, ideal for performance-critical applications.
- **Java:** Good balance between speed, safety, and portability.
- **Python:** Easiest to write and test, but the slowest in computational workloads.

For small matrices (up to 100), all languages performed comparably. However, as problem size increased, the efficiency gap widened drastically, with C outperforming Python by several orders of magnitude.

7 Future Work

Future experiments could involve optimized algorithms such as Strassen's or block matrix multiplication. Additionally, GPU acceleration (CUDA, OpenCL) and parallelization (OpenMP) could further demonstrate language-specific strengths in high-performance computing.

8 Repository

All code, benchmark data, and plots are available on GitHub: `https://github.com/Janisz11/Individual_Assignment`