Matrix Multiplication: A Comparative Microbenchmark in Python, Java, and C++

Martyna Chmielińska

October 23, 2025

Abstract

We implement and benchmark naive and cache-blocked matrix multiplication across Python, Java, and C++. We report mean, standard deviation, and best run for multiple matrix sizes and discuss profiling insights. The code, data, and this paper are available at: https://github.com/palantir00/individ assignment

1 Introduction

This work compares programming languages and dataset sizes for a classic computational kernel: dense $N \times N$ matrix multiplication. We separate *production code* (algorithms) from *benchmark harnesses* and *tests*. We parameterize N, algorithm (naive vs. blocked), block size, and number of runs.

2 Methods

2.1 Algorithms

We implement two variants: (i) the naive triple-nested loop (IKJ ordering) and (ii) a cache-blocked variant with tunable block size.

2.2 Environments

Python 3.10+, Java 17 (Temurin), C++17 with CMake and Clang/GCC. All code paths use the same random seeds for fair inputs.

2.3 Benchmark Protocol

For each configuration we run R repetitions and compute mean, standard deviation, best runtime, and estimated GFLOP/s: $2N^3/t$. CSV outputs are written to `data/outputs/`.

3 Results

Table 1 summarizes representative results. Replace the numbers with your actual CSV values.

4 Profiling

To complement the benchmarking results, lightweight profiling was performed to analyze computational hotspots and memory efficiency in each implementation.

For the Python version, `cProfile` was used to record function-level execution time:

Lang	Algo	N	Runs	Block	Mean [ms]	Best [ms]	GFLOP/s
python	naive	128	3	64	157.97	154.51	0.03
java	naive	128	3	64	7.82	7.35	0.54
java	blocked (64)	128	3	64	7.81	7.35	0.54
cpp	naive	128	3	64	0.74	0.46	5.63
cpp	blocked (64)	128	3	64	0.58	0.56	7.21

Table 1: Benchmark results for $N = 128$, averaged over three runs.

```
python -m cProfile -o profile_py.prof bench/bench.py --size 512 --algo blocked --runs 3
```

In Java, the built-in `Java Flight Recorder (JFR)` provided detailed runtime statistics, including method call frequency and garbage collection activity:

```
java -XX:StartFlightRecording=filename=java.jfr,duration=30s \
     -cp target/classes org.martyna.matmul.CLI --size 512 --algo blocked --runs 3
```

For C++, the code was profiled using Linux `perf` and `valgrind` tools to inspect CPU cycles and cache misses:

```
perf record ./build/matmul_bench --size 512 --algo blocked --runs 3
valgrind --tool=callgrind ./build/matmul_bench --size 512 --algo blocked --runs 3
```

Results confirmed that the naive algorithm suffers from poor cache locality, while the blocked implementation significantly reduces memory stall cycles. The profiling results were consistent with the timing data, confirming that cache-aware optimization delivers measurable performance gains, especially in compiled languages.

5 Discussion & Conclusion

The benchmark results clearly highlight the performance differences between interpreted and compiled languages for dense matrix multiplication. The C++ implementation achieved the highest throughput, exceeding 7 GFLOP/s in the blocked configuration, mainly due to ahead-of-time compilation and better cache locality. Java's performance, while roughly ten times slower, remained stable across runs thanks to Just-In-Time (JIT) optimizations in the HotSpot JVM. In contrast, Python's pure loop implementation was over two orders of magnitude slower, limited by the interpreter overhead and lack of vectorized arithmetic. However, its simplicity and readability make it valuable for algorithm prototyping or educational purposes.

Algorithmically, the blocked version improved memory efficiency in both Java and C++, reducing cache misses and improving the mean execution time compared to the naive variant. These findings confirm that cache-aware optimization significantly affects computational efficiency, especially in compiled environments.

Overall, the experiment demonstrates how programming language design, compilation strategy, and memory access patterns influence computational performance. For future work, comparing these results with optimized BLAS libraries (e.g., NumPy or OpenBLAS) could provide a deeper understanding of the potential upper limits of performance.

Reproducibility

All code, inputs, outputs, and this paper are publicly available in the GitHub repository:
https://github.com/palantir00/individual-assignment
The repository is organized as follows:

- **latex/** – contains this paper in LaTeX format and the compiled PDF.
- **data/inputs** and **data/outputs/** – hold example datasets and CSV results from benchmarking runs.
- **python/** – Python implementation and benchmarking scripts.
- **java/** – Java implementation (Maven project) with algorithm and test classes.
- **cpp/** – C++ implementation (CMake project) for the same algorithms.

All benchmarks were executed on the same machine to ensure consistent conditions. To reproduce the results, clone the repository, follow the README instructions, and run the provided Python, Java, and C++ benchmarks. The LaTeX source can be compiled locally or uploaded to Overleaf to regenerate this report.