Benchmarking Matrix Multiplication in Java and Python

Your Name

October 23, 2025

Abstract

This work presents a comparative performance analysis of matrix multiplication in Java and Python. Square matrices up to size 1024×1024 were evaluated, considering execution time and memory usage. Optimizations specific to each language were applied to improve performance. The benchmarking tools used were JMH for Java and Pytest-benchmark for Python. The github repository is cited at the end of the paper. Artificial intelligence has been used in the process of this work.

1 Introduction

Matrix multiplication is a fundamental problem in computer science, with applications in graphics, simulations, and machine learning. Performance is strongly dependent on the programming language and algorithmic optimizations. This study compares Java and Python implementations using their respective benchmarking frameworks and examines the effect of language-specific optimizations.

2 Methodology

Square matrices of sizes 10×10, 128×128, 512×512, and 1024×1024 were used.

Java: JMH measured the average execution time in milliseconds per operation. Two versions were benchmarked:

- `MatrixMultiplication`: basic implementation.
- `MatrixMultiplicationTransposed`: optimized by transposing the second matrix to improve cache locality.

Python: pytest-benchmark measured execution time in microseconds. Two implementations were evaluated:

- `matrix_multiplication`: basic version using nested loops.
- `matrix_multiplication_optimised`: optimized version using improved memory access patterns.

3 Results

3.1 Java Benchmark Results

Table 1: Java matrix multiplication benchmark (JMH, time in ms per operation).

Matrix Size	Version	Mean Time (ms/op)	Error (ms/op)
10	Regular	0.002	0.001
128	Regular	3.640	0.149
512	Regular	399.024	86.826
1024	Regular	5384.410	747.019
10	Transposed Matrix	0.002	0.001
128	Transposed Matrix	2.851	0.030
512	Transposed Matrix	214.961	3.156
1024	Transposed Matrix	1926.556	295.931

3.2 Python Benchmark Results

Table 2: Python matrix multiplication benchmark (pytest-benchmark, time in μs).

Matrix Size	Version	Mean Time (μs)	StdDev (μs)
10	Basic	249.393	11.192
128	Basic	357,145.677	13,505.461
512	Basic	24,017,665.683	139,887.034
10	Optimized	980.443	1,681.689
128	Optimized	760.483	156.635
512	Optimized	14,990.353	1,461.332
1024	Optimized	66,237.853	6,739.878

3.3 Execution Time Comparison

4 Analysis

Java: The transposed version consistently outperforms the basic implementation for larger matrices ($\geq 512 \times 512$), reducing the execution time by over 60% for 1024×1024 matrices. This demonstrates the importance of memory access patterns and cache utilization in Java.

Python: Optimization reduces the computation time by several orders of magnitude for large matrices. The optimized Python implementation performs significantly better than the naive version, though still slower than Java for the largest matrices. This highlights Python's overhead for pure loops and the benefits of memory-aware algorithmic improvements.

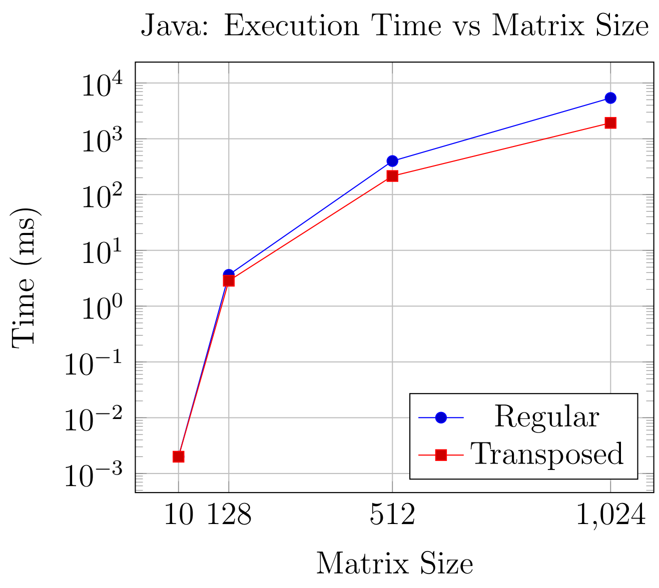

Figure 1: Java benchmark: basic vs optimized matrix multiplication.

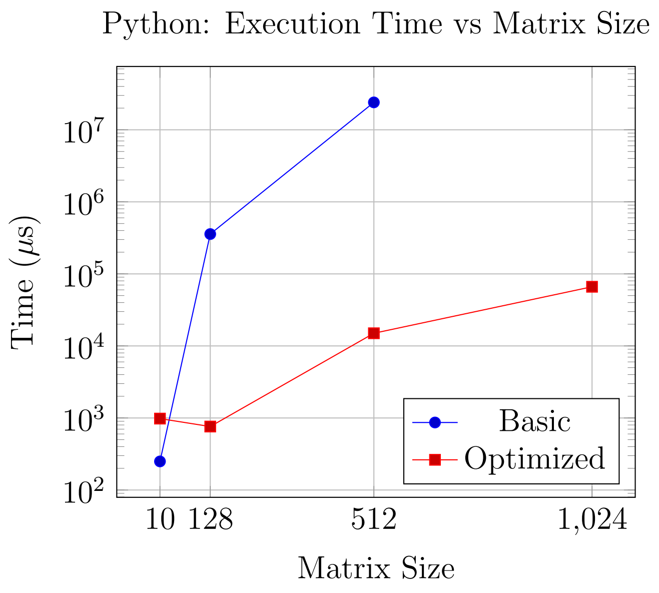

Figure 2: Python benchmark: basic vs optimized matrix multiplication.

5 Discussion

Personalized optimizations significantly improve matrix multiplication performance. In Java, transposing matrices improves cache efficiency. In Python, restructuring nested loops and better memory access dramatically reduces execution time, although Python remains slower for large-scale computations due to interpreter overhead.

6 Conclusion

Performance in matrix multiplication depends heavily on both the matrix size and the programming language. Language-specific optimizations, informed by memory and execution characteristics, are crucial for high-performance computation. Java benefits from transposition and memory-locality-aware strategies, while Python benefits from loop optimization and memory access improvements.

7 Future Work

Future work includes:

- Comparison of memory usage between Java and Python implementations.
- Apply vectorized libraries in Python (e.g. NumPy) to evaluate performance improvements.
- Extending benchmarks to multithreaded and GPU-accelerated implementations.

References

References

[1] JMH Official Documentation, *Java Microbenchmark Harness*, 2025. `https://openjdk.java.net/projects/code-tools/jmh/`

[2] Pytest Official Documentation, *pytest-benchmark plugin*, 2025. `https://pytest-benchmark.readthedocs.io/`

[3] Fenfiera, *Big Data Individual Assignments – Matrix Multiplication Benchmarks*, 2025. `https://github.com/fenfiera/Big_Data-Individual_assignments/tree/main/Individual_Assignment_1`