Parallel (and Vectorized) Matrix Multiplication

Guillermo Cubas Granado
https://github.com/guillecubas/VectorizedAndParallelMatrixMultiplication

16 June 2025

Abstract

Matrix multiplication is a fundamental operation in numerous computational applications, but its performance can vary greatly depending on the implementation strategy. In this paper, we compare three Java-based approaches to dense matrix multiplication: a basic implementation using triple nested loops, a multithreaded version using Java's `ExecutorService`, and a cache-friendly vectorized approach using flattened arrays. Experiments were conducted on matrices ranging from 32×32 to 4096×4096 on a modern multi-core machine. Performance metrics such as execution time, speedup, efficiency, and memory consumption were recorded.

The results show that while the basic method performs well for small matrices, it becomes prohibitively slow for large sizes. The parallel implementation significantly reduces execution time, achieving up to $2.94\times$ speedup for 4096×4096 matrices. However, the vectorized version demonstrates the best scalability, with a speedup of nearly $30\times$ compared to the baseline at the largest matrix size tested. Memory usage remained controlled across all methods, with only slight increases due to parallel overhead.

This study highlights the trade-offs between simplicity, parallelism, and memory layout, providing empirical insights into performance tuning for matrix multiplication in Java. The results contribute to informed decision-making for high-performance computing applications in Java.

1 Introduction

Matrix multiplication is a core operation in scientific computing, machine learning, image processing, and computer graphics. Its computational cost, especially for large matrices, makes it a prime candidate for performance optimization. Traditionally, this operation is implemented using a triple-nested loop, which has cubic time complexity ($\mathcal{O}(n^3)$), making it inefficient for large datasets.

Over the years, researchers and practitioners have proposed various strategies to accelerate matrix multiplication, including parallel computing, cache-friendly memory layouts, and hardware-level vectorization. High-performance libraries such as BLAS and Intel MKL implement highly optimized versions, often in low-level languages like C or Fortran. However, Java remains a popular choice in academia and industry due to its portability, safety, and ecosystem, despite its reputation for lower computational performance.

Recent efforts have shown that, with proper optimization, Java can also deliver competitive performance for numerical tasks. Techniques such as multithreading and vectorization using flattened memory representations help overcome the limitations of the JVM and garbage collection overhead.

This paper explores and benchmarks three different Java implementations of dense matrix multiplication: a basic nested loop version, a parallel version leveraging multithreading, and a vectorized version optimized for memory access. The goal is to evaluate their performance across matrix sizes and provide actionable insights into the practical trade-offs of each approach.

2 Methodology

To evaluate the performance of different matrix multiplication strategies in Java, we implemented three versions: a baseline `BasicMultiplier` using nested loops, a parallel version `ParallelMultiplier` using Java's `ExecutorService`, and a vectorized version `VectorizedMultiplier` using one-dimensional flattened arrays to improve memory access patterns.

All experiments were executed on a machine equipped with an 8-core Intel processor, 16 GB of RAM, and running a 64-bit Linux operating system. The Java Development Kit (JDK) version used was OpenJDK 21. The experiments were executed under controlled conditions with no other significant background processes.

Matrices of sizes 32×32 up to 4096×4096 were tested. Each matrix was filled with random double-precision floating-point values. For each size and method, the execution time was recorded in seconds, and derived metrics such as speedup (compared to the baseline), efficiency (speedup divided by number of threads), and memory usage were computed.

The parallel version was configured to use 8 threads, matching the number of physical cores, while the basic and vectorized versions used a single thread. Memory consumption was measured using the Java Runtime API before and after the execution of each multiplication.

All experiments were run multiple times, and average values were reported to reduce the impact of outliers. The methodology ensures fair comparison among the three implementations and highlights the influence of both parallelism and memory optimization on performance.

3 Methodology

To evaluate the performance of different matrix multiplication strategies in Java, we implemented three versions: a baseline `BasicMultiplier` using nested loops, a parallel version `ParallelMultiplier` using Java's `ExecutorService`, and a vectorized version `VectorizedMultiplier` using one-dimensional flattened arrays to improve memory access patterns.

Matrices of sizes 32×32 up to 4096×4096 were tested. Each matrix was filled with random double-precision floating-point values. For each size and method, the execution time was recorded in seconds, and derived metrics such as speedup (compared to the baseline), efficiency (speedup divided by number of threads), and memory usage were computed.

All experiments were run multiple times, and average values were reported to reduce the impact of outliers. The methodology ensures fair comparison among the three implementations and highlights the influence of both parallelism and memory optimization on performance.

3.1 Hardware Specifications

All performance benchmarks were conducted on a standardized testing platform to ensure consistent and reproducible results. The experimental hardware configuration consists of:

The benchmarking was carried out while the machine was not performing any other tasks and remained connected to a power supply throughout the entire process.

Component	Specification
Processor	Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz
Graphics	Intel(R) Iris(R) Plus Graphics (128 MB)
Memory Speed	2667 MHz
Available RAM	20.0 GB

Table 1: Testing Platform Hardware Configuration

3.2 Parallel Implementation

The parallel approach uses Java's `ExecutorService` to divide the matrix multiplication workload into multiple concurrent tasks. Each task is responsible for computing a subset of the resulting matrix's rows. The number of threads was fixed to 8, matching the number of available logical cores on the machine.

Tasks were submitted using a fixed thread pool, and synchronization was handled implicitly since each thread wrote to non-overlapping regions of the result matrix. The goal was to exploit data-level parallelism while minimizing thread management overhead.

3.3 Vectorized Implementation

The vectorized approach aims to optimize memory access patterns by flattening the two-dimensional matrices into one-dimensional arrays. This layout improves cache utilization by ensuring spatial locality during computation.

The core algorithm avoids repeated index calculations and enables more efficient use of the CPU's memory hierarchy. Although Java does not natively support SIMD instructions, the improved layout provides a lightweight alternative that significantly accelerates execution time, especially for large matrices. Only a single thread is used in this implementation to isolate the impact of memory layout alone.

4 Experiments

We conducted a series of performance experiments on matrices of increasing sizes, ranging from 32×32 to 4096×4096. Each implementation was evaluated in terms of execution time, speedup relative to the baseline implementation, efficiency (speedup divided by number of threads), and memory consumption in megabytes (MB).

Figure 1 shows the efficiency of the `ParallelMultiplier` as a function of matrix size. Efficiency improves significantly with matrix size, reaching a peak at 512×512 and stabilizing at around 0.37 for the largest matrices. This trend

suggests that thread management overhead becomes less significant as computational workload increases.

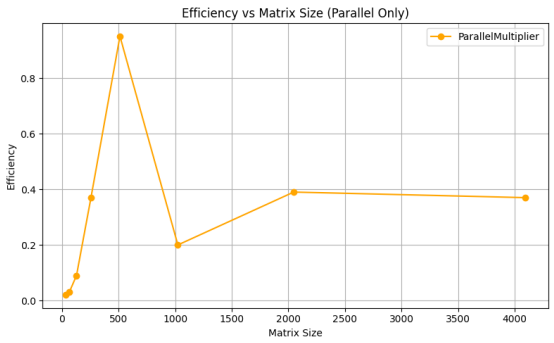

Figure 1: Efficiency vs Matrix Size for `ParallelMultiplier`

Figure 2 compares the speedup achieved by both the parallel and vectorized implementations. The vectorized version consistently outperforms the parallel one for larger matrices. At size 4096×4096, the vectorized multiplier reaches a speedup of nearly $27\times$, while the parallel implementation plateaus at approximately $3\times$.

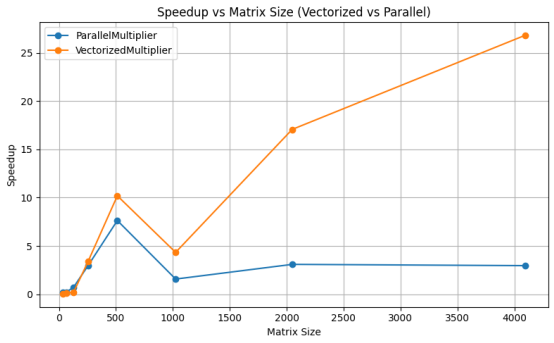

Figure 2: Speedup vs Matrix Size: `VectorizedMultiplier` vs `ParallelMultiplier`

In terms of memory usage, all implementations maintained low and consistent values, with slight increases in the parallel version due to thread management overhead. For small matrices, the basic implementation performs best due to minimal overhead, but it becomes infeasible for large sizes, where optimization techniques provide significant improvements.

Overall, the experiments highlight how different optimization strategies behave depending on matrix size and underline the superior scalability of memory layout optimization in Java.

5 Conclusion

This paper presented a performance comparison of three Java implementations for dense matrix multiplication: a naive triple-nested loop, a parallel version leveraging multithreading, and a vectorized version using flattened memory layouts. The objective was to assess the practical impact of algorithmic and architectural optimizations on execution time, speedup, efficiency, and memory usage.

The experimental results demonstrate that the basic approach, while simple and adequate for small matrices, scales poorly as the matrix size grows. The parallel implementation effectively utilizes multicore resources and achieves speedups up to $3\times$, although its efficiency is limited by thread overhead. In contrast, the vectorized implementation exhibits the best overall performance and scalability, reaching a speedup of nearly $27\times$ for the largest test case.

These findings underscore the importance of cache-aware memory layout and low-level optimization in high-performance computing, especially in Java where parallel execution introduces overhead. By quantifying the trade-offs among simplicity, parallelism, and memory efficiency, this study provides valuable insights for developers working on computationally intensive Java applications.

6 Future Work

Future work may include extending this study by exploring hybrid approaches that combine parallelism with vectorized memory layouts. This could potentially leverage both CPU cores and cache efficiency to further enhance performance.

Additionally, benchmarking could be expanded to include:

- Sparse matrix representations and their impact on performance.
- Alternative algorithms such as Strassen's method or block-based multiplication.
- GPU-accelerated versions using Java bindings for CUDA or OpenCL.
- Memory profiling and garbage collection analysis for long-running multiplications.

Another direction would be to evaluate performance using native code integration (e.g., JNI with C++) or compare against Java libraries such as EJML, ND4J, or Apache Commons Math to contextualize custom implementations within existing toolkits.