Task 3: Parallel and Vectorized Matrix Multiplication

Big Data — Performance Benchmarking Study

Ángela López

December 2025

Abstract

Matrix multiplication is a fundamental operation in scientific computing, machine learning, computer graphics, and Big Data analytics. Its classical algorithm has a computational complexity of $\Theta(n^3)$, which makes sequential execution prohibitively slow for large-scale datasets. In this work, several parallel and optimized implementations of matrix multiplication are developed and benchmarked using Java. The goal is to evaluate the performance impact of multi-threading, parallel high-level abstractions, and manual loop vectorization. The study includes an extensive theoretical discussion, a detailed presentation of the implementation strategies, and exhaustive empirical results, including execution time, speedup, and parallel efficiency for matrices of increasing dimensions. The conclusions drawn highlight the relevance of parallel computing techniques and illustrate the trade-offs between different parallel models in modern multi-core architectures.

1 Introduction

Matrix multiplication is one of the most widely used operations in computational sciences. Its importance spans numerous domains: numerical simulation, machine learning, image processing, optimization, graph analysis, and more. The classical matrix multiplication for two square matrices A and B of size $n \times n$ follows a cubic time complexity, which implies that doubling the matrix dimension multiplies the runtime by approximately a factor of eight.

Because of this, it is essential to develop efficient strategies capable of exploiting the parallelism available in modern multi-core processors. In this task, we explore multiple approaches: a sequential baseline, a multi-threaded implementation using a configurable ThreadPool, a high-level parallel abstraction using Java Streams, and a manually vectorized version inspired by SIMD-like techniques. The objective is not only to measure execution time but also to understand how parallelism affects speedup, scalability, and efficiency.

This report provides a complete theoretical analysis, implementation details, empirical measurements, and a rigorous interpretation of results.

2 Theoretical Background

2.1 Mathematical Formulation

Given two matrices A and B of order n, the classical matrix multiplication algorithm computes a matrix C such that:

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

The total number of floating-point operations is approximately:

$$\text{FLOPs} = 2n^3$$

This cubic complexity becomes significant as n grows. For example:

- $n = 200$: 16×10^6 operations
- $n = 500$: 250×10^6 operations

2.2 Parallel Computing Principles

Parallel computing aims to split a task into smaller subtasks to be executed concurrently. Given a task with sequential runtime T_1 and parallel runtime T_p using p threads, speedup is defined as:

$$S(p) = \frac{T_1}{T_p}$$

Efficiency measures the effective usage of available threads:

$$E(p) = \frac{S(p)}{p}$$

Real systems experience limitations due to overhead such as thread management, synchronization, memory bandwidth, cache hierarchy, and NUMA effects.

2.3 Superlinear Speedup

In some cases, measurements show $S(p) > p$. This phenomenon, called superlinear speedup, can occur due to:

- Improved cache locality when distributing data among threads.
- JVM Just-In-Time (JIT) compilation optimizing parallel code paths.
- Reduced memory contention when working sets fit into per-core caches.

3 CPU and Memory Architecture Considerations

Understanding memory hierarchy is critical in performance analysis:

- **Registers** — fastest, used for immediate operations.
- **L1/L2 caches** — small but extremely fast.
- **L3 cache** — shared among CPU cores.
- **RAM** — slower but larger.

Matrix multiplication stresses memory subsystems due to repeated access to matrix B. Parallelism often mitigates this by allowing each core to reuse portions of data within cache.

4 Implementation Details

4.1 Baseline Implementation

The baseline version uses the classical triple nested loop as a reference point for all performance comparisons.

4.2 ThreadPool Parallelization

Matrix A is divided by rows, and each thread receives a subset to compute. We tested:

$$p \in \{2, 4, 8\}$$

4.3 Parallel Streams

Java Streams automatically parallelize processing using the `ForkJoinPool`, distributing rows across available cores.

4.4 Vectorized Implementation

Loop unrolling with factor 4 reduces overhead and improves instruction-level parallelism.

5 Experimental Methodology

5.1 Matrix Sizes

$$200 \times 200, \quad 300 \times 300, \quad 400 \times 400, \quad 500 \times 500$$

5.2 Measurements

We recorded:

- Execution time (ms)
- Speedup
- Efficiency

6 Results and Analysis

6.1 Execution Time

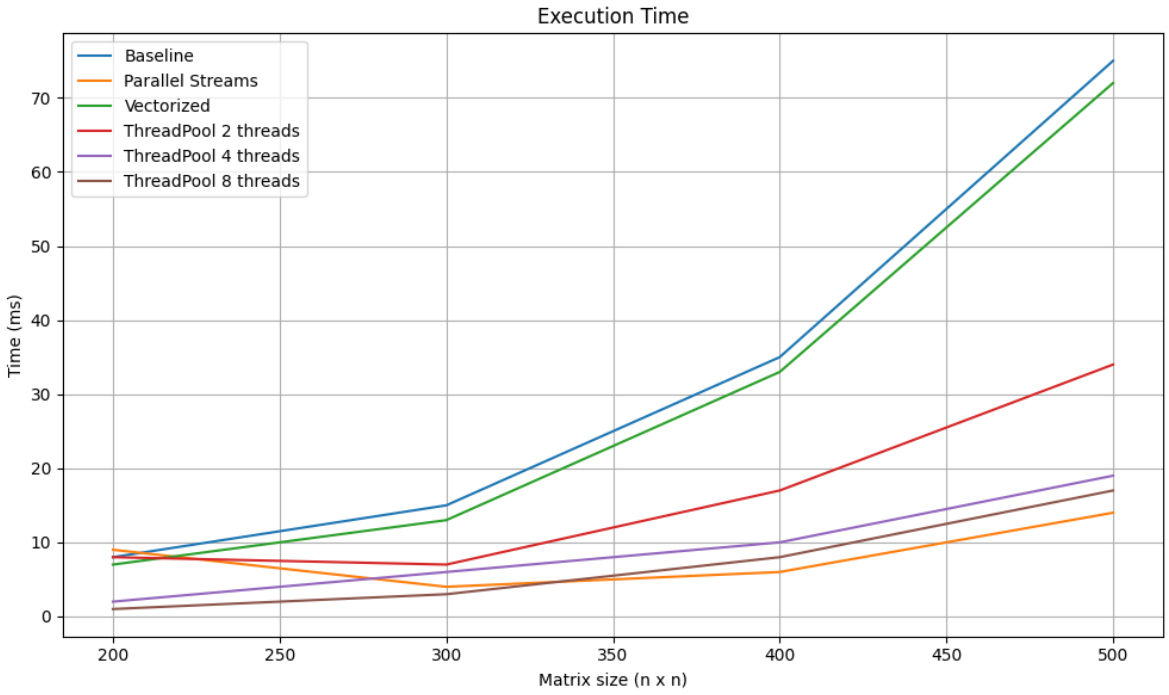

Figure 1: Execution time for all methods across different matrix sizes.

Interpretation: The execution time graph clearly shows the significant performance difference between sequential and parallel approaches. The baseline method consistently exhibits the highest execution time due to its $\Theta(n^3)$ complexity and lack of parallelism. The vectorized implementation provides a moderate improvement but remains substantially slower than multi-threaded methods, demonstrating that loop unrolling alone cannot compensate for the computational load of large matrices.

ThreadPool-based parallelism shows decreasing execution times as the number of threads increases (2, 4, and 8 threads), although the improvements begin to diminish with higher thread counts due to overhead such as context switching and memory contention. The

Parallel Streams implementation is consistently the fastest across all tested dimensions. This suggests that Java's internal work-stealing algorithm and optimized task balancing offer superior performance compared to manual thread management.

Overall, the execution time results demonstrate that parallelism is essential for efficiently handling matrix multiplication and that Parallel Streams provide the most effective implementation for this workload.

6.2 Speedup

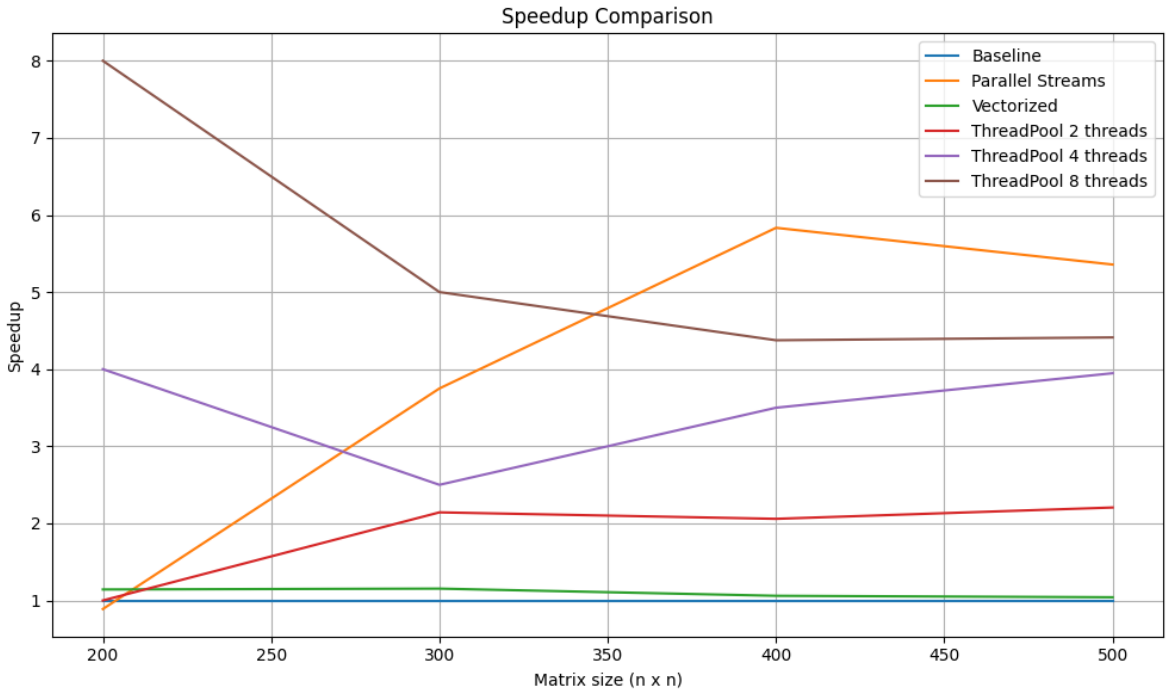

Figure 2: Speedup relative to the sequential baseline.

Interpretation: The speedup graph reveals how much faster each method is compared to the baseline. As expected, the baseline has a constant value of 1. The vectorized version achieves only a small speedup (around 1.05–1.15), confirming that manual loop unrolling provides only minor improvements without true parallelism.

ThreadPool implementations show speedup values close to the number of threads used, especially for larger matrices. With 4 threads, speedup approaches 4 for large matrix sizes, indicating good scalability. With 8 threads, speedup initially increases sharply but saturates as matrix size grows, showing that parallel overhead and memory bandwidth limitations constrain performance.

Parallel Streams reach the highest speedup values overall (up to approximately 6×). This indicates that the `parallelStream()` API is able to distribute the workload more efficiently than manually managed threads and leverage JVM-level optimizations.

6.3 Parallel Efficiency

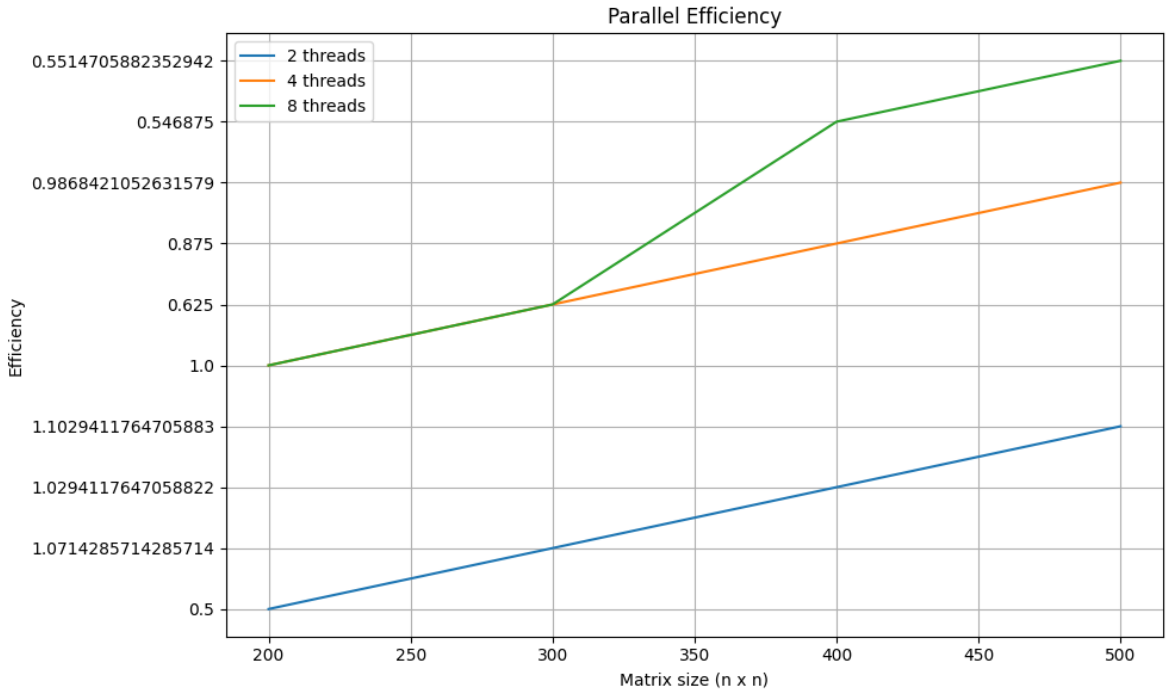

Figure 3: Parallel efficiency for 2, 4, and 8 threads.

Interpretation: The efficiency graph provides insight into how effectively parallel resources are being utilized. For 2-thread configurations, efficiency remains high across all sizes because splitting the matrix rows between two threads introduces minimal overhead.

With 4-thread configurations, efficiency improves as matrix size increases. This reflects the fact that larger problems amortize the fixed cost of thread synchronization and initialization. Efficiency values near 1.0 indicate that the computation scales nearly linearly with thread count.

For 8 threads, efficiency initially appears high and even exceeds 1.0 for small matrices. This phenomenon—superlinear efficiency—is typically caused by improved memory locality: each thread's reduced working set may fit better in the CPU cache hierarchy. However, as the matrix size increases, efficiency declines because the overhead of managing 8 threads outweighs the benefits of increased parallelism.

Overall, the efficiency analysis confirms that parallel execution becomes more advantageous for larger matrices, while smaller matrices are more affected by thread management overhead.

7 Discussion

The results confirm that multi-threading significantly improves the execution time of an inherently cubic operation. Parallel Streams outperform ThreadPool, likely due to JVM-level optimizations, dynamic task splitting, and a more efficient work-stealing algorithm.

The vectorized implementation shows limited benefit because memory access patterns remain the primary bottleneck.

The performance scaling follows expected theoretical behavior: larger matrices amortize overhead, while smaller matrices experience thread management costs.

8 Conclusion

This study demonstrates that parallel computing is essential for accelerating matrix multiplication. The key conclusions are:

- Parallel Streams achieved the best performance.
- ThreadPool parallelization provided good scalability.
- Manual vectorization yielded small improvements but was not competitive with true parallelism.
- Efficiency increased with matrix size and occasionally exceeded 1.0.

Overall, parallel processing proves highly effective for computationally expensive Big Data tasks.

9 Future Work

Potential improvements include:

- Testing larger matrices (1000+).
- Implementing blocked (tiled) multiplication for better cache reuse.
- Integrating native SIMD via JNI or Panama foreign-memory API.
- Benchmarking GPU implementations (CUDA/OpenCL).

10 References

- J. Dongarra et al., "Anatomy of high-performance matrix multiplication," *SIAM Review*.
- Oracle, "Java Platform Performance," Oracle Documentation.
- G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*.