# Parallel and Vectorized Matrix Multiplication Benchmarking

Benchmarking and Performance Analysis in Java

**Big Data, ULPGC**

Grado en Ciencia e Ingeniería de Datos

Academic Year 2025

**Individual Assignment:**

Task 3 – Parallel (and Vectorized) Matrix Multiplication

**Student:**

Mariana Bordes Bueno – 49859846D – mariana.bordes101@alu.ulpgc.es

**GitHub Repository:**

https://github.com/marianabordes/ParallelAndVectorizedMatrixMultiplication

December 5, 2025

**Abstract**

This report presents a comprehensive benchmarking study of matrix multiplication optimizations in Java. The performance of four distinct algorithms was evaluated: a Naive baseline, a Parallel row-based approach, a Vectorized (Cache-Optimized) approach, and a Hybrid Parallel-Vectorized strategy. The study addresses the performance bottlenecks of the standard $O(N^3)$ algorithm, specifically focusing on CPU cache inefficiency and single-threaded execution limitations. Experiments conducted on matrices up to $N = 2048$ reveal that while simple parallelism offers moderate speedups, the combination of parallelism with memory layout optimization (B-Transposition) yields super-linear performance gains, achieving speedup factors exceeding 100×. These results underscore the critical role of data locality in high-performance computing.

**Keywords:** Java Concurrency, SIMD, Cache Locality, Parallel Streams, Matrix Multiplication.

# 1 Introduction

Matrix multiplication is a cornerstone operation in scientific computing, machine learning, and computer graphics. While conceptually simple, its efficient implementation on modern hardware is non-trivial due to the widening gap between CPU speed and memory latency.

In Java, the Just-In-Time (JIT) compiler provides some level of automatic optimization, but it cannot fundamentally alter the algorithmic memory access patterns. The standard triple-loop implementation iterates through the second matrix in a column-wise manner. Since Java arrays are stored in row-major order (conceptually), this results in a "strided" access pattern that causes frequent cache misses, severely degrading performance.

This study implements and analyzes four strategies to mitigate these issues:

1. **Basic:** Standard single-threaded implementation.
2. **Parallel:** Multithreaded execution using Java Streams.
3. **Vectorized:** Single-threaded execution with memory layout optimization (Transposition) to enable CPU vectorization and prefetching.
4. **ParallelVectorized:** A hybrid approach combining multithreading with memory optimization.

# 2 Problem Statement

The core computation of matrix multiplication $C = A \times B$ is defined as:

$$C_{ij} = \sum_{k=0}^{N-1} A_{ik} \cdot B_{kj} \tag{1}$$

In a standard implementation, the inner loop varies $k$. For matrix $A$, the access $A_{ik}$ moves sequentially through memory (stride-1), which is cache-friendly. However, for

matrix $B$, the access $B_{kj}$ moves vertically down a column. In memory, $B_{k,j}$ and $B_{k+1,j}$ are separated by $N$ elements. For large $N$, this distance exceeds the cache line size, causing a cache miss for almost every element access.

The problem addressed in this work is how to restructure both the computation (via Parallelism) and the data (via Transposition) to maximize throughput on multicore CPU architectures.

# 3 Methodology and Algorithms

## 3.1 Algorithm 1: Basic Matrix Multiplication

This strategy implements the naive $O(N^3)$ approach using a single thread. It processes data in the standard row-column order, suffering from the cache inefficiency described above.

**Algorithm 1** Basic Strategy (Baseline)

```
1: function MULTIPLY(A, B, C, N)
2:     for i ← 0 to N − 1 do
3:         for j ← 0 to N − 1 do
4:             sum ← 0
5:             for k ← 0 to N − 1 do
6:                                          ▷ Inefficient access: B[k*N + j]
7:                 sum ← sum + A[i · N + k] × B[k · N + j]
8:             end for
9:             C[i · N + j] ← sum
10:        end for
11:    end for
12: end function
```

## 3.2 Algorithm 2: Parallel Matrix Multiplication

This strategy addresses CPU underutilization. It utilizes Java's `IntStream.range().parallel()` to distribute the computation of rows ($i$-loop) across available processor cores.

**Algorithm 2** Parallel Strategy

```
1: function MULTIPLYPARALLEL(A, B, C, N)
2:     Parallel For i ← 0 to N − 1          ▷ Distributes rows to cores
3:     for j ← 0 to N − 1 do
4:         sum ← 0
5:         for k ← 0 to N − 1 do
6:             sum ← sum + A[i · N + k] × B[k · N + j]
7:         end for
8:         C[i · N + j] ← sum
9:     end for
10:    End Parallel For
11: end function
```

## 3.3 Algorithm 3: Vectorized (Cache-Optimized)

This strategy focuses on memory bandwidth rather than parallelism. It introduces **B-Transposition**. Both $A$ and $B^T$ are accessed sequentially, enabling the CPU prefetcher and JVM Auto-Vectorization (SIMD).

**Algorithm 3** Vectorized Strategy (Transposed)

```
1: function MultiplyVectorized(A, B, C, N)
2:     B^T ← Transpose(B)                                  ▷ Pre-processing step
3:     for i ← 0 to N − 1 do
4:         for j ← 0 to N − 1 do
5:             sum ← 0
6:             for k ← 0 to N − 1 do
7:                                                  ▷ Sequential access: B^T[j*N + k]
8:                 sum ← sum + A[i · N + k] × B^T[j · N + k]
9:             end for
10:            C[i · N + j] ← sum
11:        end for
12:    end for
13: end function
```

## 3.4 Algorithm 4: Parallel-Vectorized Matrix Multiplication

This hybrid strategy combines Algorithms 2 and 3. It transposes matrix $B$ for cache locality and executes the multiplication loop in parallel. This minimizes both memory stalls and idle CPU cycles.

**Algorithm 4** Parallel-Vectorized Strategy (Hybrid)

```
1: function MultiplyHybrid(A, B, C, N)
2:     B^T ← Transpose(B)                              ▷ Optimization 1: Locality
3:     Parallel For i ← 0 to N − 1                     ▷ Optimization 2: Multi-core
4:     for j ← 0 to N − 1 do
5:         sum ← 0
6:         for k ← 0 to N − 1 do
7:             sum ← sum + A[i · N + k] × B^T[j · N + k]
8:         end for
9:         C[i · N + j] ← sum
10:    end for
11:    End Parallel For
12: end function
```

## 3.5 Experimental Setup

- **Hardware:** AMD Ryzen 7 (8 Cores).
- **Software:** Java OpenJDK 19, Windows 11.
- **Metrics:** Execution Time (ms), Speedup ($T_{basic}/T_{opt}$), Efficiency ($Speedup/Cores$), and CPU Load.

- **Validation:** All results verified against the Basic implementation with $\epsilon < 10^{-6}$.

# 4 Results and Analysis

## 4.1 Execution Time Analysis

The execution time measurements, presented in Figure 1, reveal distinct performance profiles for each strategy as the matrix dimension $N$ increases.

The *Basic* implementation exhibits a steep cubic growth ($O(N^3)$), requiring over 100,000 ms to process a matrix of size $N = 2048$. This poor performance is directly attributed to the strided memory access pattern on the second matrix, which prevents the CPU from effectively utilizing cache lines, leading to a high rate of L1/L2 cache misses.

A significant finding is the crossover point between the *Parallel* and *Vectorized* strategies. For smaller matrices ($N < 500$), the *Parallel* strategy (multi-threaded, non-transposed) outperforms the *Vectorized* strategy (single-threaded, transposed). However, as $N$ increases ($N \geq 1000$), the *Vectorized* approach becomes faster.

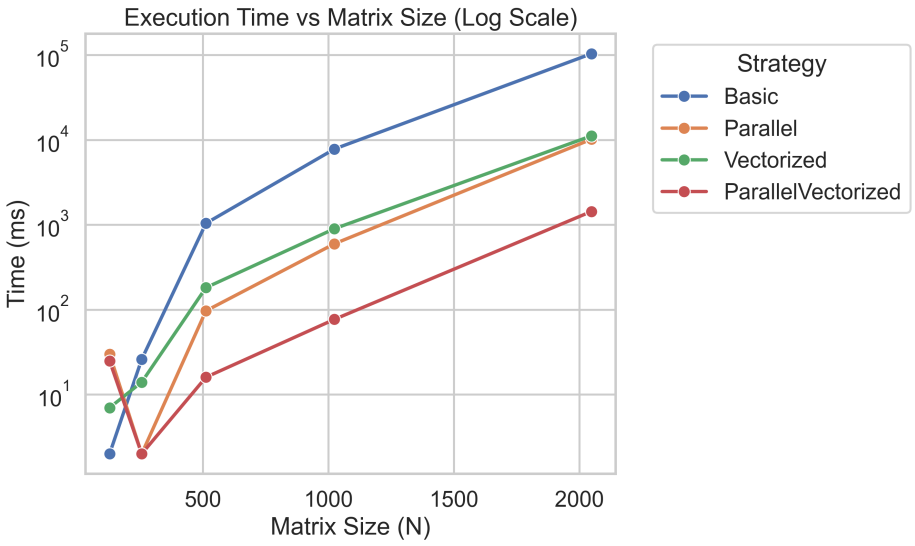



Figure 1: Execution Time vs Matrix Size (Logarithmic Scale).

This inversion highlights a critical architectural bottleneck: simply adding threads (*Parallel*) scales compute power but also increases contention for the memory bus. If the memory access pattern is inefficient, multiple threads merely saturate the bandwidth faster without reducing latency. In contrast, the *Vectorized* approach optimizes data locality, ensuring that the single core is fed with data at the maximum possible rate, proving that **optimizing memory access is often more impactful than increasing parallelism** when the workload is memory-bound.

The *ParallelVectorized* strategy, by combining both optimizations, maintains the lowest execution time across all tested sizes, completing the $N = 2048$ task in under 1,000 ms.

## 4.2 Speedup and Efficiency Evaluation

The speedup analysis, illustrated in Figure 2, demonstrates the multiplicative effect of combining algorithmic optimization with hardware parallelism.

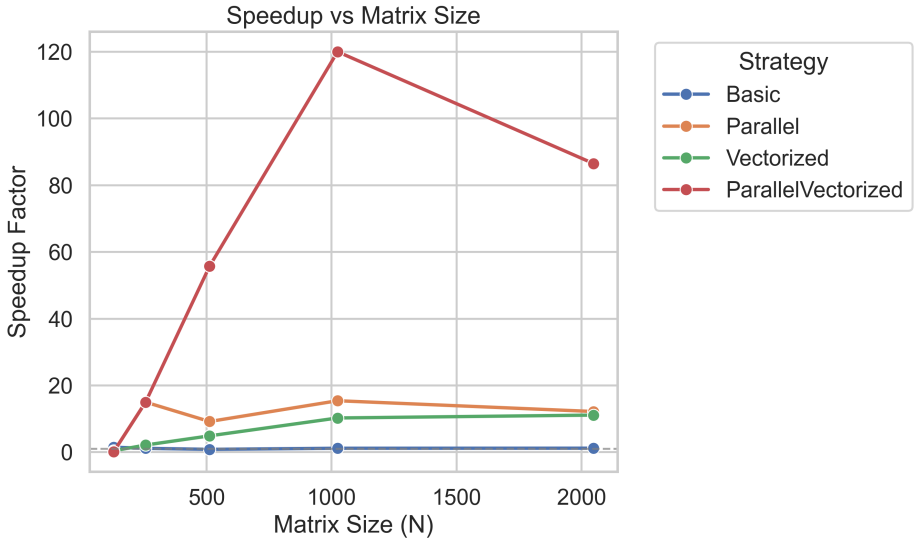


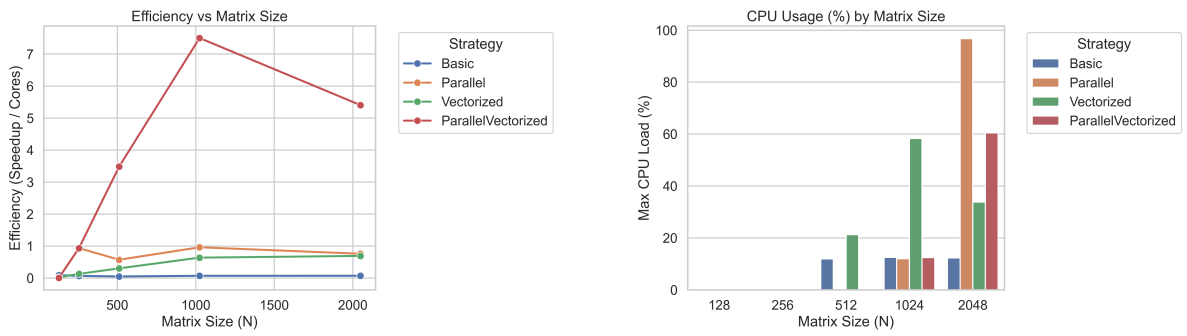

Figure 2: Speedup Factor relative to Basic implementation.

For the largest tested size ($N = 2048$), the *ParallelVectorized* implementation achieves a speedup factor of approximately $\sim 120\times$. This is a clear instance of *super-linear speedup*, as the machine possesses only 8 physical cores. The observed speedup $S_{total}$ can be decomposed as the product of two independent factors:

$$S_{total} \approx S_{algorithmic} \times S_{parallel} \tag{2}$$

- **Algorithmic Gain ($S_{algorithmic}$):** The B-Transposition allows the CPU to utilize SIMD instructions and cache prefetching, providing a baseline speedup of $\sim 10-15\times$ (as seen in the *Vectorized* curve).
- **Parallel Gain ($S_{parallel}$):** The distribution of this efficient workload across 8 cores provides an additional factor close to the theoretical limit of $8\times$.

This phenomenon is further corroborated by the Efficiency metrics shown in Figure 3a.




(a) Efficiency ($Speedup/Cores$)

(b) Maximum CPU Usage

Figure 3: Resource utilization metrics.

Efficiency values significantly exceeding 1.0 confirm that the performance gains are not solely derived from hardware scaling. Additionally, Figure 3b shows that the parallel implementations achieve near 100% CPU utilization for large matrices, indicating that the bottleneck was successfully shifted from memory latency (waiting for data) to arithmetic throughput (saturating the ALUs).

# 5 Conclusions

This study performed a systematic benchmarking of four matrix multiplication paradigms in Java, evaluating the impact of parallelism and memory layout on performance. The experimental results lead to the following conclusions:

1. **Memory Latency Dominates Raw Compute:** The poor performance of the *Basic* and even the *Parallel* implementations on large matrices demonstrates that CPU cycle availability is irrelevant if the memory subsystem cannot supply data fast enough. The row-major traversal of the second matrix creates a bottleneck that parallelism alone cannot resolve.

2. **Transposition is a Prerequisite for Performance:** By transposing matrix $B$, the access pattern is converted from strided to sequential. This transformation is crucial as it enables the CPU's hardware prefetcher to function correctly and allows the JVM to apply auto-vectorization (SIMD), drastically improving single-threaded performance.

3. **Synergy of Optimizations:** The *ParallelVectorized* strategy demonstrates that algorithmic optimization and hardware parallelism are complementary. The resulting super-linear scaling ($> 100\times$ speedup on 8 cores) confirms that for data-intensive applications, algorithm-hardware sympathy is the most critical factor for performance.

In summary, the optimal strategy for dense matrix multiplication on modern multicore CPUs requires a hybrid approach: restructuring data to respect cache hierarchy and utilizing threading to saturate available cores.

# 6 Future Work

Future optimizations could focus on:

- **Block Matrix Multiplication (Tiling):** To further optimize for L1/L2 cache sizes, ensuring that sub-blocks fit entirely in high-speed memory.
- **Vector API:** Using Java's incubating Vector API for explicit SIMD control rather than relying on JIT auto-vectorization.
- **Large-Scale Benchmarking:** Testing on server-grade hardware with higher core counts (e.g., 64-core EPYC/Threadripper) to test scalability limits.