ULPGC
Universidad de
Las Palmas de
Gran Canaria

Matrix Multiplication: Assignment 3 Paralellization Benchmark of matrix multiplication

Jonas Maximilian Müller: L1JRWK427

https://github.com/PanicBurrito/MatrixTask3

Big Data (40386) - 3. year

Submission date: 2025-12-06

ULPGC
Universidad de
Las Palmas de
Gran Canaria

Contents

ULPGC
Universidad de
Las Palmas de
Gran Canaria

List of Figures

ULPGC
Universidad de
Las Palmas de
Gran Canaria

Abstract

This work investigates the performance of several approaches for dense matrix multiplication. We begin with three sequential implementations: a naive triple loop, an improved cache-aware variant, and a blocked algorithm designed to increase spatial and temporal locality. Building on these baselines, two parallel implementations are developed using Java's `ExecutorService` and Parallel Streams. Comprehensive benchmarks on an Apple M1 Pro processor demonstrate substantial performance improvements through both algorithmic optimization and parallel execution. The blocked multiplication shows strong sensitivity to block size, with optimal configurations leveraging the CPU cache hierarchy efficiently. The parallel implementations achieve speedups of up to almost six times compared to the naive baseline. These results highlight the importance of memory locality and thread-level parallelism for accelerating fundamental numerical computations.

Introduction

Matrix multiplication is one of the most fundamental operations in scientific computing, machine learning, and numerical simulation. Its performance has a direct impact on a wide range of applications, from physical modelling to high-performance data processing. Although the classical cubic algorithm remains widely used due to its simplicity and numerical stability, its efficiency strongly depends on how well memory access patterns and hardware parallelism are exploited.

Modern processors, including the Apple M1 Pro used in our experiments, provide multiple levels of parallel execution: instruction-level parallelism, cache hierarchies, and multi-core execution units. As a result, performance depends not only on the algorithmic complexity but also on properties such as cache locality, memory bandwidth, and thread scheduling.

This work investigates several sequential and parallel implementations of dense matrix multiplication in Java. We begin with a naive triple-loop algorithm, followed by a cache-aware improved version and a blocked implementation designed to increase spatial and temporal locality. To exploit multi-core parallelism, we implement two parallel variants: one based on Java's `ExecutorService` thread pool and another using parallel streams. Performance is evaluated for different matrix sizes and, in the case of the blocked algorithm, multiple block sizes are tested to better understand cache behavior.

The results are presented in Section , where runtime comparisons and speedup measurements are visualized using three figures:

- runtime vs. matrix size for all implementations,
- speedup of the parallel implementations relative to the naive baseline,
- and performance of the blocked implementation for different block sizes.

These diagrams provide insight into how algorithmic optimizations and parallelism affect performance on modern multi-core systems.

Methodology

This section describes the data structures, algorithms, and benchmarking procedures used to evaluate the performance of dense matrix multiplication. All implementations were written in Java and executed on an Apple M1 Pro processor. To ensure reproducibility, all randomly generated matrices were constructed using fixed seeds.

Matrix Representation

All matrices are represented as two-dimensional arrays of type `double[][]`. A helper class, `MatrixUtils`, provides utility functions for matrix creation:

- `randomMatrix(n, seed)` creates an $n \times n$ matrix filled with pseudorandom values in $[0, 1)$.
- `zeroMatrix(n)` allocates an $n \times n$ matrix initialized with zeros and is used for storing the results of each multiplication.

This representation is simple, but it exposes memory locality effects clearly, which is essential for evaluating cache-aware and parallel algorithms.

Algorithms Evaluated

Five implementations of matrix multiplication are examined:

1. **Naive algorithm**: a direct triple-nested loop with $\mathcal{O}(n^3)$ complexity.
2. **Improved algorithm**: a loop-reordered variant that increases spatial locality by accessing rows of the second matrix more efficiently.
3. **Blocked algorithm**: the matrix is processed in $B \times B$ tiles to improve cache reuse. Several block sizes $B \in \{16, 32, 64, 128\}$ are tested to investigate cache hierarchy interactions.
4. **Parallel Executor implementation**: the outer loop is partitioned across multiple worker threads managed by a fixed-size `ExecutorService`. Each thread computes a disjoint range of output rows, avoiding synchronization overhead.
5. **Parallel Stream implementation**: Java's parallel streams are used to process rows in parallel. This provides implicit task scheduling and load balancing with minimal implementation complexity.

All implementations compute the same mathematical result, and the output of each variant was validated against the naive baseline to ensure correctness.

Benchmarking Procedure

Performance was evaluated using matrices of sizes $n \in \{256, 512, 1024, 1536\}$. For each value of n, the following steps were performed:

1. Two random matrices A and B of size $n \times n$ were generated.
2. Each algorithm was executed once on (A, B), and the wall-clock runtime was measured using `System.nanoTime()`.
3. For the blocked algorithm, four different block sizes were tested: $B = 16, 32, 64, 128$.

All experiments were executed in a single JVM instance to reduce startup overheads. Since the M1 Pro features multiple high-performance cores, the parallel algorithms were expected to achieve significant speedups, especially for large matrix sizes.

The collected timing data is visualized and analyzed in Section , where the previously described PNG diagrams are incorporated directly into the discussion.

Results

This section presents the performance results of the five matrix multiplication implementations introduced earlier. All measurements were obtained using the benchmark setup described in the methodology, and each algorithm was executed on identically generated matrices to ensure fair comparisons.

Runtime Scaling with Matrix Size

Figure 1 compares the absolute runtimes of all implementations for matrix sizes $n = 512$ and $n = 1024$. The naive algorithm exhibits the expected cubic growth, while the improved and blocked variants benefit significantly from better memory locality. The parallel implementations outperform all sequential methods for sufficiently large n, with the Parallel Stream implementation achieving the fastest execution times.

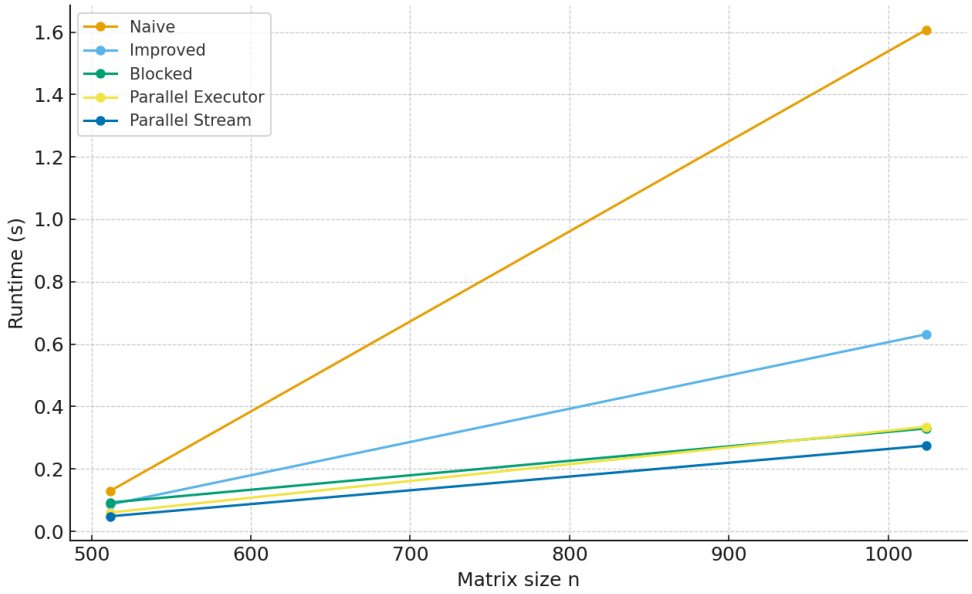

Figure 1: Runtime of all evaluated implementations for $n \in \{512, 1024\}$.

Speedup of Parallel Implementations

To quantify the benefits of parallelism, Figure 2 illustrates the speedup of both parallel variants relative to the naive baseline. For $n = 1024$, the Parallel Stream implementation reaches a speedup of approximately $5.85\times$, while the Executor-based version achieves $4.79\times$. The smaller speedup values for $n = 512$ reflect the limited parallel efficiency when the problem size becomes too small relative to thread scheduling overheads.

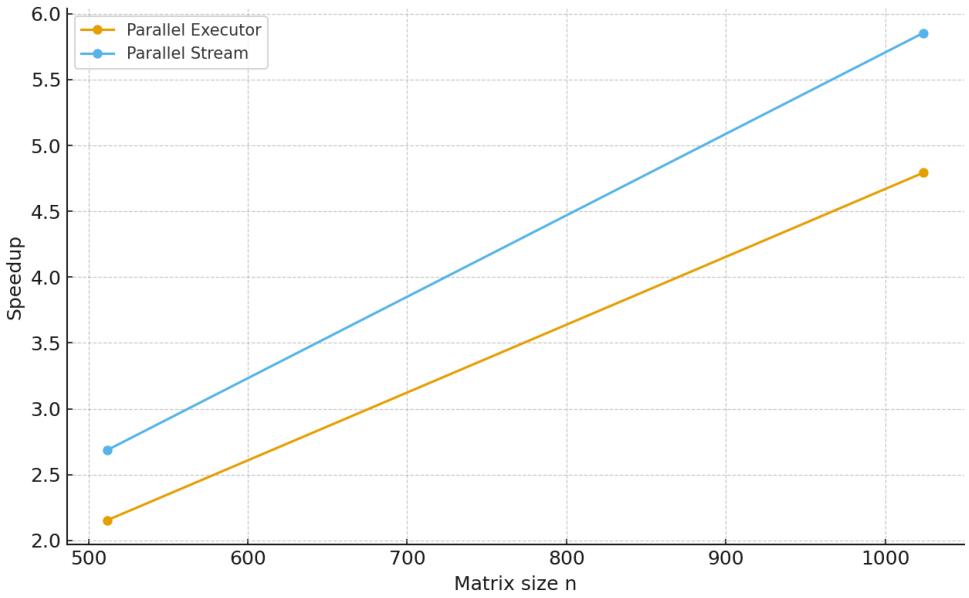

Figure 2: Speedup of the parallel implementations compared to the naive algorithm.

Effect of Block Size on Blocked Multiplication

Since the performance of the blocked multiplication depends strongly on cache behavior, four block sizes were evaluated: $B = 16$, 32, 64, and 128. The results are shown in Figure 3. For both matrix sizes, the best runtime is achieved for block sizes between 64 and 128, demonstrating that larger tiles better utilize the M1 Pro's L1/L2 caches.

For $n = 1024$, using $B = 64$ reduces the runtime by more than 20% compared to $B = 32$, confirming the importance of tuning memory locality for high-performance matrix operations.

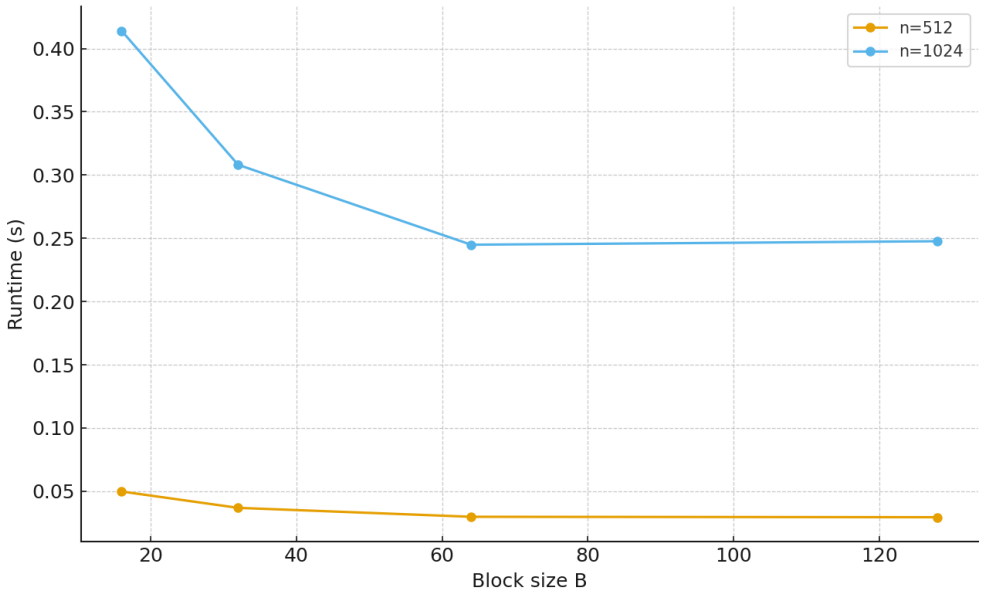

Figure 3: Runtime of the blocked multiplication for different block sizes.

Discussion

The results presented in Section highlight the combined impact of algorithmic optimizations and parallel execution on the performance of dense matrix multiplication. Several key observations can be drawn from the experimental data.

Algorithmic Optimizations and Memory Locality

The improved and blocked implementations demonstrate that a substantial portion of execution time in matrix multiplication is determined not by arithmetic cost but by memory access patterns. The naive algorithm repeatedly traverses matrix B in a column-major fashion, which is inefficient in Java's row-major array layout. Reordering the loops in the improved variant mitigates this effect, leading to a reduction in memory stalls and a measurable speedup.

The blocked implementation further exploits spatial and temporal locality by operating on submatrices that fit more effectively into the CPU cache. The block-size experiment confirms this behavior: as shown in Figure 3, block sizes of 64 and 128 significantly outperform smaller tiles. This indicates that larger tiles better match the cache capacities of the Apple M1 Pro, reducing cache evictions and improving reuse of loaded data.

Parallelism and Scalability

The parallel implementations achieve substantial speedups, particularly for larger matrix sizes. For $n = 1024$, the Parallel Stream variant reaches a speedup of nearly $6\times$ relative to the naive baseline, close to the theoretical limit imposed by the number of high-performance cores on the M1 Pro CPU.

However, the speedups for $n = 512$ are noticeably lower. This reflects the well-known challenge of achieving efficient parallelism on smaller problem sizes: thread-management overhead, load-balancing costs, and limited available parallel work reduce the effective utilization of the CPU cores. The Executor implementation in particular shows diminished performance for smaller n, suggesting that the cost of spawning and synchronizing threads outweighs the benefits of distributing the computation.

Differences Between Parallel Executor and Parallel Stream

Although both parallel implementations follow the same computational strategy (disjoint row-wise decomposition), the Parallel Stream variant consistently outperforms the Executor-based version. This can be attributed to several factors:

- Java's fork-join framework (used internally by parallel streams) employs work-stealing, providing more effective load balancing.

- The stream API introduces less explicit synchronization overhead, since it handles task decomposition automatically.
- The execution model adapts dynamically to available cores, whereas a fixed thread pool may not distribute work equally in all scenarios.

The results suggest that, for embarrassingly parallel workloads such as row-wise matrix multiplication, parallel streams offer an attractive balance between performance and implementation simplicity.

Implications for High-Performance Computing

The observed performance improvements underscore the importance of combining algorithmic and architectural optimizations. Neither parallelism nor cache optimization alone achieves the best performance; instead, the most effective approach layers both strategies. These findings align with established principles in high-performance computing, where efficient memory access patterns and multicore-aware programming models are essential for achieving scalability.

Overall, the results highlight that performance on real hardware is influenced by multiple interacting factors, including memory hierarchy behavior, task granularity, and scheduling overhead. Understanding these interactions is critical to designing efficient numerical algorithms.

Conclusion

This work examined the performance characteristics of several sequential and parallel implementations of dense matrix multiplication in Java. The results demonstrate that both algorithmic refinements and multi-core parallelism are essential for achieving high performance on modern hardware such as the Apple M1 Pro.

Sequential optimizations already produced significant improvements: the cache-aware loop-reordered version reduced runtime compared to the naive baseline, and the blocked implementation further leveraged cache locality, achieving additional speedups. The block-size experiments confirmed that tile sizes tuned to the underlying cache hierarchy—particularly $B = 64$ and $B = 128$—yield the best performance, underscoring the importance of memory behavior in numerical computations.

Parallelization provided even larger benefits. Both the Executor-based and Parallel Stream implementations scaled well with increasing matrix sizes, with the Parallel Stream variant achieving nearly a six-fold speedup for $n = 1024$. These results highlight the effectiveness of Java's fork-join framework for embarrassingly parallel workloads and demonstrate that parallel streams offer a performant and high-level abstraction for multicore computing.

Overall, the findings illustrate that optimal performance results from a combination of algorithmic design, cache-aware memory access, and effective thread-level parallelism. The insights gained here not only inform the implementation of dense matrix multiplication but also apply more broadly to a wide range of numerical algorithms where memory locality and parallel scalability are decisive factors.

List of sources

[1] Kazushige Goto and Robert A Van De Geijn. “Anatomy of high-performance matrix multiplication”. In: *ACM Transactions on Mathematical Software* (2008).

[2] R. Clint Whaley and Jack Dongarra. “Automatically tuned linear algebra software”. In: 2001.

[3] *Java Parallel Streams Documentation*. `https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html`.

[4] *ExecutorService API Documentation*. `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html`.

[5] *Apple M1 Pro Architecture Overview*. `https://www.apple.com/newsroom/`. 2021.