Parallelization Benchmark of Matrix Multiplication

Radosława Żukowska

December 1, 2024

Big Data

Grado en Ciencia e Ingenieria de Datos
Universidad de Las Palmas de Gran Canaria

Abstract

Matrix multiplication is a fundamental operation in computer science, with applications spanning machine learning, computer graphics, and scientific computing. However, the naive matrix multiplication algorithm, with its cubic time complexity $O(N^3)$, becomes computationally expensive for large matrices. This paper explores optimization strategies to enhance matrix multiplication performance, focusing on parallelization using Java. Two parallelization methods are examined: thread-based implementations and parallel streams, both applied to the naive algorithm.

Experiments were conducted on matrices up to $10,000 \times 10,000$, comparing execution times and resource utilization for parallel and sequential implementations. Results indicate that parallel execution significantly accelerates computation for large matrices, with parallel streams slightly outperforming threads due to more efficient resource management. For smaller matrices, however, the overhead of thread management diminishes the benefits of parallelization. The study also analyzes the impact of varying thread counts, finding optimal performance with thread numbers matching the hardware's core count.

Additional evaluations reveal differences in garbage collector behavior, with parallel streams showing lower allocation times for small matrices but threads maintaining consistent performance as matrix sizes grow. These findings underscore the importance of aligning parallelization strategies with hardware capabilities to achieve maximum efficiency.

This study concludes that parallelization is a highly effective strategy for accelerating matrix multiplication, particularly for large-scale problems.

Contents

1 Introduction

Matrix multiplication is a fundamental operation in computer science with a wide range of applications, including machine learning, computer graphics, and scientific computing. Given its importance and widespread use, optimizing this operation is critical for improving performance across various domains. The naive matrix multiplication algorithm, while simple and easy to implement, exhibits inefficiency for large matrices due to its cubic time complexity, $O(N^3)$. As the size of the matrices increases, the computational cost grows dramatically, resulting in poor performance for large-scale problems.

Improving the performance of matrix multiplication involves multiple considerations. These range from selecting the appropriate platform and programming language to optimizing the algorithm itself and leveraging available hardware. In previous works, the first two approaches were analyzed. This paper focuses on enhancing performance through the parallelization of the algorithm.

Typically, programs are executed sequentially, completing one task after another. However, some computational tasks, such as matrix multiplication, involve independent operations that can be performed simultaneously to accelerate execution. This can be achieved through parallelization, which involves decomposing a problem into sub-problems that can be processed concurrently using multiple cores or processors.

There are various methods to achieve parallelization, such as employing threads or parallel streams, each configurable with different parameters. This paper aims to explore the performance improvements offered by parallelizing matrix multiplication, comparing its speedup against the standard sequential approach.

2 Parallelization with Java Streams

The matrix multiplication algorithm was implemented using Java's parallel streams to leverage parallelism for computational efficiency. This approach utilizes the `IntStream` class from the `java.util.stream` package to distribute tasks across multiple threads.

a. **Stream-Based Parallelism:** The `IntStream.range(0, size)` method generates a stream of integers representing the row indices of the result matrix. By invoking the `parallel()` method on this stream, the computations for individual rows are executed in parallel.

b. **Row-Wise Computation:** For each row index, a lambda expression calculates all elements in that row by iterating over the corresponding rows and columns of the input matrices. The nested loops within the stream's operation handle the element-wise multiplication and summation for matrix multiplication.

c. **Automatic Thread Management:** Parallel streams utilize the `ForkJoinPool` framework, which

automatically manages thread allocation based on the system's available processors. This simplifies thread management compared to manually handling an executor service.

d. **Thread-Safe Execution:** Each thread computes its assigned row independently, ensuring no shared mutable state is accessed concurrently. The result matrix is pre-allocated, and individual elements are updated within the thread's scope, avoiding race conditions.

This approach provides a high-level abstraction for parallel computation, simplifying the implementation while achieving significant performance gains for large matrices compared to sequential processing.

3 Parallelization with Java Threads

The matrix multiplication algorithm was implemented using Java's multithreading capabilities to optimize computational efficiency. The approach leverages the `ExecutorService` from the `java.util.concurrent` package to manage threads, enabling parallel execution of row-wise calculations for large matrices.

a. **Thread Pool Management:** A fixed thread pool was created using the `Executors.newFixedThreadPool()` method. The number of threads can be customized, with a default of 16 threads if not specified. This approach ensures controlled concurrency and avoids the overhead of creating and destroying threads for each task.

b. **Parallel Computation:** For each row in the result matrix, a task is submitted to the executor service. Each task computes all elements in its assigned row by iterating over the corresponding rows and columns of the input matrices. This design ensures independent computation of rows, enabling effective parallelization.

c. **Synchronization and Completion:** After submitting all tasks, the executor service is gracefully shut down using `shutdown()`, and the program waits for all tasks to complete within a specified timeout period using `awaitTermination()`. If interrupted, an exception is handled to ensure robust error management.

d. **Thread-Safe Execution:** By delegating each row computation to separate tasks, race conditions are inherently avoided since no shared mutable data is modified across threads. This results in a thread-safe implementation.

This parallelized method effectively distributes the computational load across multiple threads, significantly reducing execution time for large matrices when compared to sequential processing.

4 Methodology

All experiments are conducted on the same machine with parameters shown in Table 1. They are done on the same matrices for each implementation using Java Microbenchmark Harness (JMH) and collecting data about average time per operation and memory usage with profiler gc and stack. The matrices are filled with random values with the same random seed to ensure comparable results.

System information	Details
Operating System Name	Windows 11 Professional
Operating System Version	23H2
Processor	Intel(R) Core(TM) Ultra 7 155H
Number of CPU cores	16
RAM	32GB
SSD Storage	474 GB
Java version	OpenJDK 21.0.2

Table 1: Test environment parameters

The experiments explores different parameters of the implementations with focus on execution time and memory usage while testing both parallel implementation and the naive one on increasing sizes of matrices.

5 Experiments' results

5.1 Comparison with naive method

First experiment were to test the performance of the parallelized implementation with the naive standard one for matrix multiplication. They were done on matrices with size up to 512 x 512.

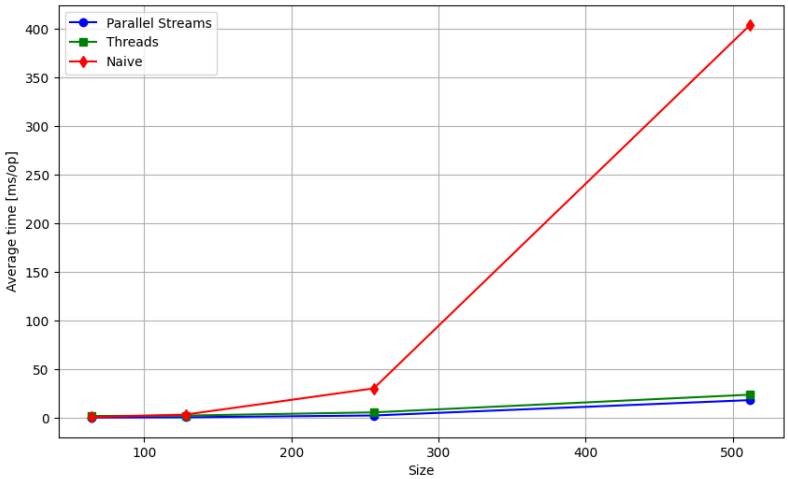

Figure 1: Execution time for different matrix multiplication methods over size of the matrices

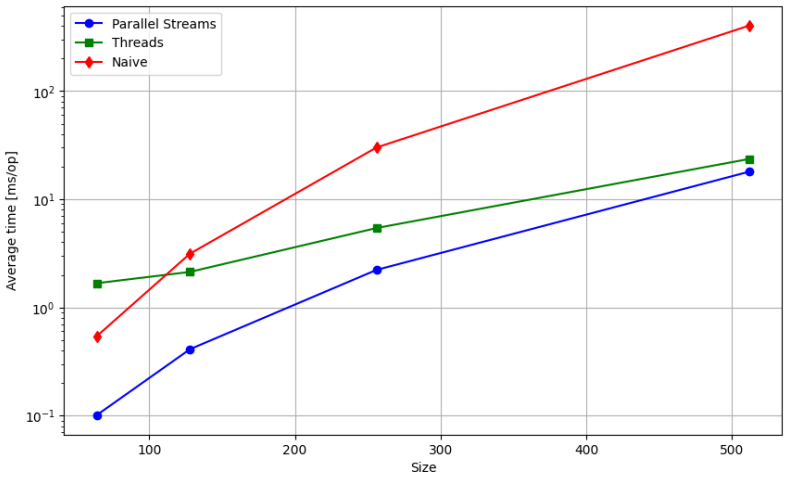

Figure 2: Execution time for different matrix multiplication methods over size of the matrices in logarithmic scale on time axis

As the plots present parallel execution significantly improves the execution time for matrix multiplication. This becomes especially evident as the size of the matrices increases. Although for smaller matrices as 64x64 naive algorithm is faster than executing with threads, with larger matrices the naive algorithm has significantly longer execution time. For smaller matrices operating on threads gives worse results because of the overhead of the management of the threads.

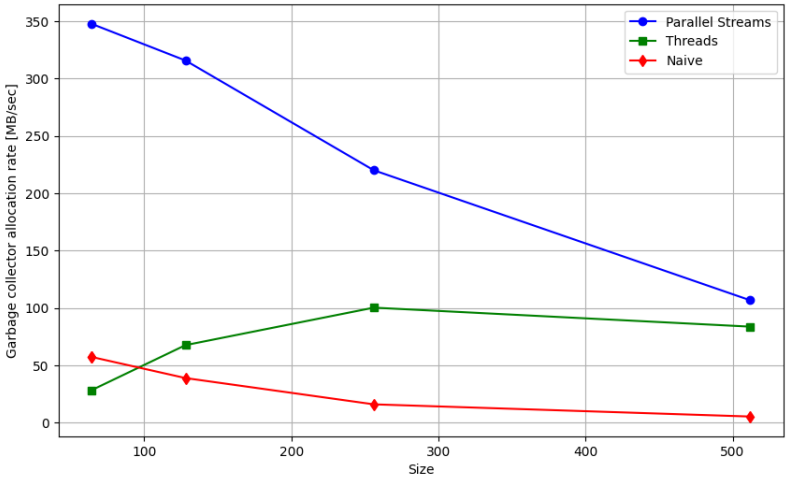

Figure 3: Garbage collector allocation rate for parallel and not parallel implementations over different matrices' sizes

Garbage collector allocation time is best for parallel streams however as the size of the matrices increases it drops linearly, for implementation with threads it isn't as good for small ones, but for bigger ones it seems to stay steady at around 100 Mb/sec. Naive has the lowest garbage collector allocation time.

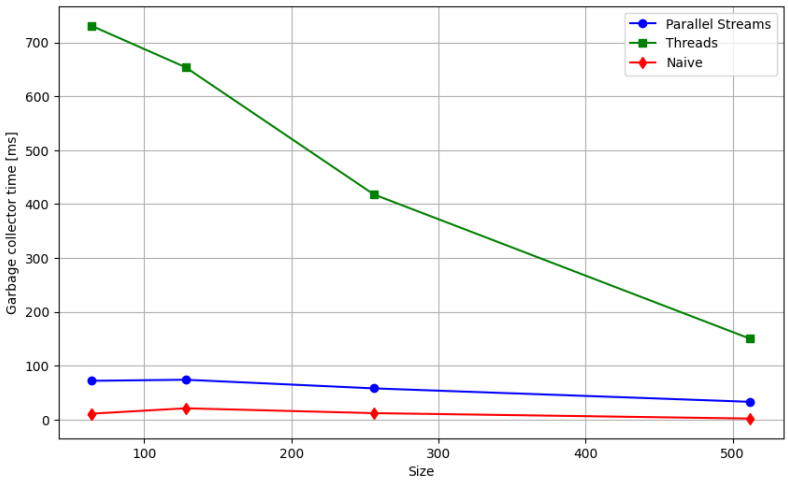

Figure 4: Garbage collector time for parallel (parallel streams and threads) and not parallel (naive) matrix multiplication for different sizes of matrices

5.2 Comparison between streams and threads

Implementations of matrix multiplication with threads and parallel streams were further tested on larger matrices. The results are shown on the plot below.

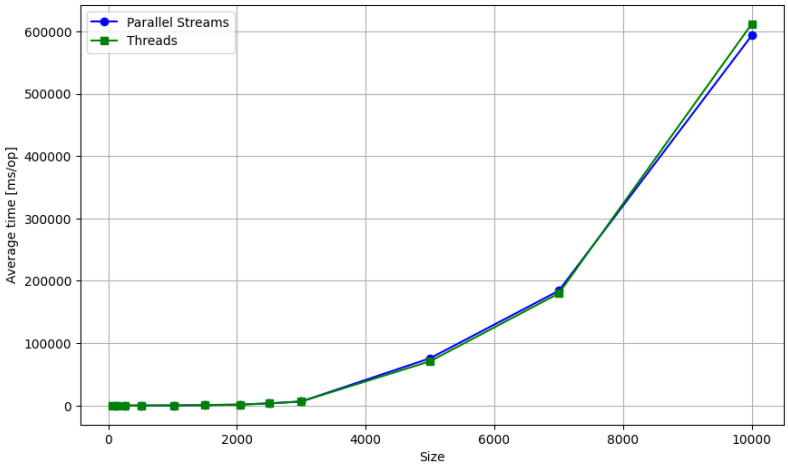

Figure 5: Execution time for matrix multiplication with parallel streams and threads for increasing sizes of the matrices

The sizes of the matrices were up to 10,000 x 10,000 and even for such large matrices both implementations performed very well with time. Multiplication of such large matrices took them around 10 minutes. Although the difference is very minor, parallel streams performed slightly better than threads.

Threads	Parallel Streams	State
89.8%	83.4%	RUNNABLE
10.0%	12.5%	TIMED_WAITING
0.2%	4.0%	WAITING

Table 2: Stack states and their percentages for both implementations

The table presents stack states and their percentages for two implementations of matrix multiplication: traditional threads and parallel streams.

- **RUNNABLE State:** The threads implementation has a higher runnable state percentage (89.8%) compared to parallel streams (83.4%), indicating better CPU utilization.
- **TIMED_WAITING State:** Parallel streams exhibit a higher percentage of threads in the timed waiting state (12.5%) than threads (10.0%), suggesting additional overhead.
- **WAITING State:** The waiting state is more pronounced in parallel streams (4.0%) compared to threads (0.2%), indicating more synchronization or blocking operations.

Overall, the results suggest that the thread-based approach maintains greater efficiency in runnable states, while parallel streams incur overhead affecting performance.

5.3 Different number of threads

For the implementation of matrix multiplication with threads tested were executions with different number of threads starting from 1 thread to 24 threads also with increasing sizes of matrices.

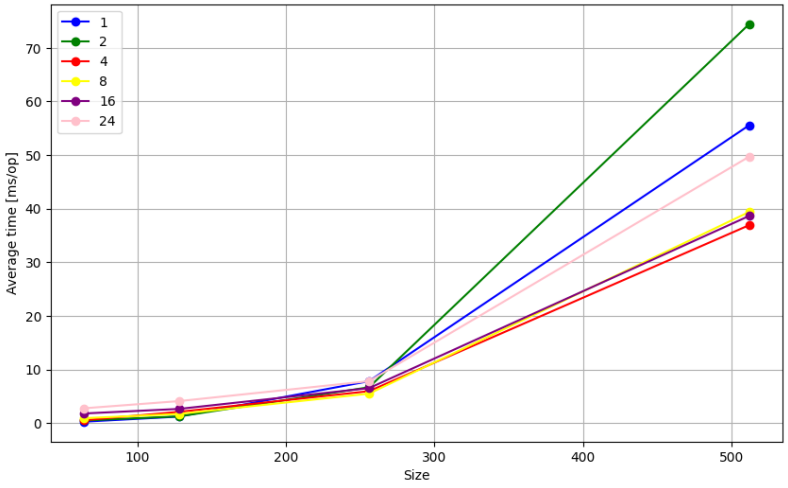

Figure 6: Execution time for different number of threads with increasing size of the matrices

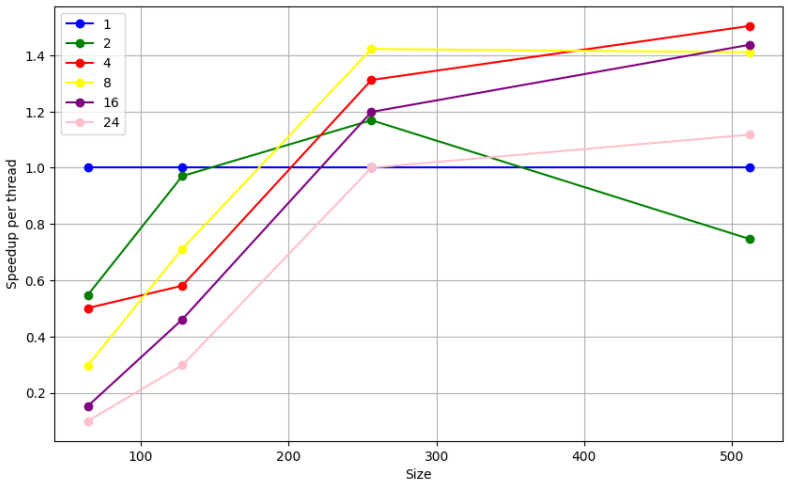

Figure 7: Speedup per thread for different number of threads with increasing size of the matrices

For smaller matrices the difference in performance is not as evident, but for larger ones it becomes more visible. Best average time of execution were achieved for running algorithm with 4 to 16 threads, while the worse with 2 and 1 threads, also 24 was only slightly better than the one with 2 threads which shows that more threads not always will go in pair with better performance. This maybe also related to the fact that the processor which executed the experiments has 16 cores so it is most efficient to divide the task between them, and dividing them further into smaller chunks will not improve performance as the processor cannot run more tasks at the same time.

6 Conclusion

This study explored the parallelization of the naive matrix multiplication algorithm using threads and parallel streams, comparing their performance to the sequential implementation. The results demonstrate the significant benefits of parallel execution, particularly for larger matrices. Parallel implementations achieved considerable speedups as matrix sizes increased, with the overhead of thread management making parallelization less effective for smaller matrices (e.g., 64×64). For larger matrices, however, the parallelized methods consistently outperformed the sequential approach.

Parallel streams exhibited slightly better performance than threads for large-scale matrices, likely due to their efficient utilization of the `ForkJoinPool` framework and lower garbage collector overhead. In contrast, the thread-based implementation provided greater flexibility, allowing the number of threads to be explicitly controlled. Experiments with varying thread counts revealed optimal performance with 4 to 16 threads, aligning with the number of processor cores available on the testing hardware. Increasing the thread count beyond the number of cores yielded diminishing returns, highlighting the importance of matching thread configurations to hardware capabilities.

Further insights into garbage collector allocation times showed that parallel streams performed best for

smaller matrices but experienced a linear decrease in allocation rates for larger sizes. Thread-based implementations maintained more consistent garbage collection performance as matrix sizes increased, while the naive approach had the lowest allocation times overall but at the cost of significantly higher execution times.

Overall, this work demonstrates the potential of parallelization to enhance matrix multiplication performance, with both threads and parallel streams offering robust solutions for large-scale computations. Future work will build upon these findings by applying parallelization to more advanced matrix multiplication algorithms, such as Strassen's, and exploring distributed execution across multiple nodes to handle even larger datasets and achieve further performance improvements.

7 Future Work

Future work could focus on extending the parallelization techniques to other matrix multiplication algorithms beyond the naive standard approach, such as Strassen's algorithm and other divide-and-conquer methods. These algorithms offer different computational patterns and may present unique opportunities for parallel optimization. Additionally, it would be worthy to explore distributed execution as a means to further accelerate matrix multiplication, leveraging multiple machines or nodes to divide the workload. This approach could significantly improve performance for extremely large matrices, where single-machine parallelization reaches its limits. Such advancements would provide a broader understanding of optimizing matrix multiplication across various computational paradigms.

8 Appendices

GitHub repository

https://github.com/radoslawazukowska/ulpgc-big-data-ind

Here is whole code used in experiments and in obtaining the results together with the results and plots. Everything for this paper can be found in 'Assignment 3' directory.