TASK 3. Paralellization Benchmark of matrix multiplication

Samuel Déniz Santana

November 2024

GitHub Repository Link

1 Abstract

Efficient matrix multiplication is critical for many computational fields, including machine learning, scientific simulations, and engineering applications. As matrix sizes grow, the computational and memory demands become increasingly challenging, necessitating optimized approaches. This study investigates three distinct methods for performing dense matrix multiplication: a simple nested-loop implementation, a vectorized approach utilizing Java parallel streams, and a parallel algorithm designed with the ForkJoin framework to exploit multi-threading.

The experimentation involved matrices ranging from 10×10 to 5000×5000, tested on a system with modern hardware and measured using metrics such as execution time, speedup, efficiency, and memory consumption. The Java Microbenchmarking Harness (JMH) framework ensured precise and reproducible results throughout the evaluation process.

Findings indicate that while the basic algorithm is memory-efficient, it is unsuitable for large-scale problems due to its high computational cost. The vectorized algorithm provided moderate performance improvements but required significantly more memory. In contrast, the parallel algorithm exhibited the best scalability, achieving a peak speedup of $224\times$ for the largest tested matrices, with efficient memory utilization.

These results highlight the parallel algorithm as the optimal choice for large-scale computations, providing a foundation for further exploration into distributed solutions for even larger datasets.

2 Introduction

Matrix multiplication is a fundamental operation in numerous scientific, engineering, and machine learning applications. Despite its simplicity, optimizing matrix multiplication is crucial due to its computational cost, especially when dealing with large datasets. Traditional methods employ nested loops to compute the product, which is often inefficient on modern hardware. To address this inefficiency, vectorized and parallelized approaches leverage advancements in hardware capabilities, such as Single Instruction Multiple Data (SIMD) and multi-core processors, to improve performance.

In this context, benchmarking programming techniques for matrix multiplication is an effective way to evaluate the impact of these optimization strategies. This paper focuses on comparing three implementations: (1) a basic nested-loop algorithm, (2) a parallelized algorithm utilizing multi-threading , and (3) a vectorized implementation simulating SIMD-like operations using Java's parallel streams. These implementations are evaluated on computational efficiency and resource utilization.

A review of the literature indicates extensive research in this area. Numerous studies compare parallelization techniques, often using OpenMP, CUDA, or MPI for high-performance computing environments. However, the application of SIMD principles using Java and its comparison with multi-threaded approaches remains underexplored. Furthermore, detailed benchmarking, including resource usage metrics, provides added value to the discussion.

The goal of this paper is to provide a comprehensive analysis of these approaches in terms of speedup, efficiency, and memory usage, particularly for large matrices, offering insights into their practical applicability and limitations.

3 Problem statement

The computational complexity of matrix multiplication, particularly the naive $O(n^3)$ algorithm, poses significant challenges as matrix dimensions increase. Fields like machine learning, scientific simulations, and large-scale data analysis rely heavily on matrix operations, making optimization crucial for practical performance. Addressing these challenges requires innovative approaches that align with the capabilities of modern computing hardware.

Optimizing matrix multiplication involves harnessing parallel and vectorized techniques, each with its unique demands. Parallelization distributes computational tasks across multiple cores, while vectorization exploits SIMD (Single Instruction Multiple Data) capabilities to perform simultaneous calculations on data. Despite their potential, these methods are not universally efficient; their performance depends on effective thread management, optimized memory access, and hardware-specific tuning, particularly in environments like Java.

In this study, we examine and compare three approaches to matrix multiplication—basic, parallelized, and vectorized—within the Java programming environment. By evaluating computational efficiency, resource utilization, and scalability across varying matrix sizes, this research provides actionable insights for practitioners aiming to optimize matrix operations using Java. This work addresses gaps in the existing literature, particularly regarding Java's suitability for high-performance numerical computations.

4 Methodology

This section describes the experimental approach used to evaluate and compare the performance of different optimization techniques for dense matrix multiplication. The experiments were carried out using the IntelliJ IDEA development environment, with Java as the programming language, to leverage its capabilities for multi-threading and vectorized computations. Key performance indicators, including execution time, speedup, efficiency, and memory usage, were measured to assess the effectiveness of each implementation. The experimental design ensures reproducibility through a standardized setup, which includes hardware specifications, benchmarking tools, and controlled parameters for testing.

4.1 Experimental Setup

The experiments will be executed on a system with the following specifications:

- **Hardware**:
  - **RAM**: 16 GB
  - **Processor**: 12th Gen Intel® Core™ i7-12700H @ 2.30 GHz
  - **Operating System**: Windows 11
- **Software**:
  - **Development Environment**: IntelliJ IDEA
  - **Programming Language**: Java
  - **Benchmarking Framework**: JMH (Java Microbenchmarking Harness)

JMH will be used to run the benchmarks and measure both execution time and memory usage for each matrix optimization technique.

4.2 Benchmark Configuration

To measure the performance of the optimizations, the benchmarks will be configured with the following parameters:

- **Benchmark Mode**:
  - `@BenchmarkMode(Mode.AverageTime)`: This will measure the average execution time of each operation.
  - **Output Time Unit**: `@OutputTimeUnit(TimeUnit.MILLISECONDS)` will output the time in milliseconds.
- **Warmup**:
  - `@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)` will perform 5 warmup iterations of 1 second each to allow the JVM to optimize the code before measurements are taken.

- **Measurement**:
  - `@Measurement(iterations = 10, time = 1, timeUnit = TimeUnit.SECONDS)` will perform 10 measurement iterations, each lasting 1 second.
- **Forking**:
  - `@Fork(1)` will ensure that each experiment runs in a separate JVM process to isolate the results and prevent interference from previous runs.

4.3 Matrix Sizes and Implementations

The benchmarks tested the performance of three matrix multiplication implementations: the **basic**, **vectorized**, and **parallel** algorithms. To evaluate scalability, matrices of varying sizes were used, ranging from 10×10 to 5000×5000. These sizes were selected to represent small, medium, and large-scale computational workloads.

The following steps were used for each experiment:

- Randomly generate two input matrices filled with values between 0 and 10 for each size.
- Run all three implementations using identical input matrices to ensure fair comparisons.
- Measure execution time, memory usage, and resource utilization for each algorithm.
- Record and analyze speedup and efficiency for the vectorized and parallel algorithms relative to the basic implementation.

4.4 Evaluation Metrics

The following metrics were used to evaluate the algorithms:

- **Execution Time**: Measured the time taken to compute the matrix multiplication for each algorithm.
- **Speedup**: Calculated as the ratio of the basic algorithm's execution time to that of the optimized algorithms.

$$\text{Speedup} = \frac{\text{Execution Time (Basic)}}{\text{Execution Time (Optimized)}}$$

- **Efficiency**: Measured the speedup per thread for the parallel algorithm.

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of Threads}}$$

- **Memory Usage**: Tracked memory consumption during execution to evaluate resource utilization.

This methodology ensures a comprehensive assessment of the algorithms' computational performance and resource efficiency, providing reproducible results that can guide future optimizations and comparisons.

5 Experiments

We conducted a series of experiments to evaluate the performance of three matrix multiplication algorithms—basic, vectorized, and parallel—across a range of matrix sizes, from small (10×10) to extremely large (5000×5000). Each algorithm was executed under identical conditions on a system with a 12th Gen Intel(R) Core(TM) i7-12700H CPU, utilizing its 14 physical cores and 20 logical threads for parallel processing. The metrics measured include execution time, speedup, efficiency, and memory usage. The goal of these experiments was to assess the scalability and resource efficiency of each approach, particularly as the matrix size increased. This section introduces the results obtained, analyzes the performance trends, and evaluates the resource usage.

Execution Time

The execution time of each algorithm was measured across a range of matrix sizes to evaluate their performance under different workloads. Table 1 presents the execution times for the basic, vectorized, and parallel algorithms, while Figure 1 visually illustrates the trends. These results highlight how each method scales with increasing matrix sizes and reveal the advantages and limitations of the optimizations applied in the vectorized and parallel implementations.

For smaller matrices ($n \leq 100$), the basic algorithm performed comparably to, or slightly better than, the optimized versions. This is primarily due to the minimal overhead in its straightforward nested-loop implementation. However, as matrix sizes increased ($n \geq 500$), the performance gap widened significantly. The vectorized and parallel algorithms both outperformed the basic approach, with the parallel algorithm demonstrating the most substantial gains for large matrices.

Matrix Size (n)	**Vectorized Algorithm**	**Basic Algorithm**	**Parallel Algorithm**
10	0.0038	0.0034	0.0082
50	0.0042	0.0037	0.0086
100	0.0046	0.0051	0.0137
200	0.0056	0.0103	0.0237
500	0.0146	0.2096	0.0245
1000	0.0950	9.7976	0.0678
2000	0.7981	57.3437	0.5150
5000	17.5360	1737.7803	7.7436

Table 1: Matrix Multiplication Performance of Vectorized, Basic, and Parallel Algorithms for Different Matrix Sizes (s)

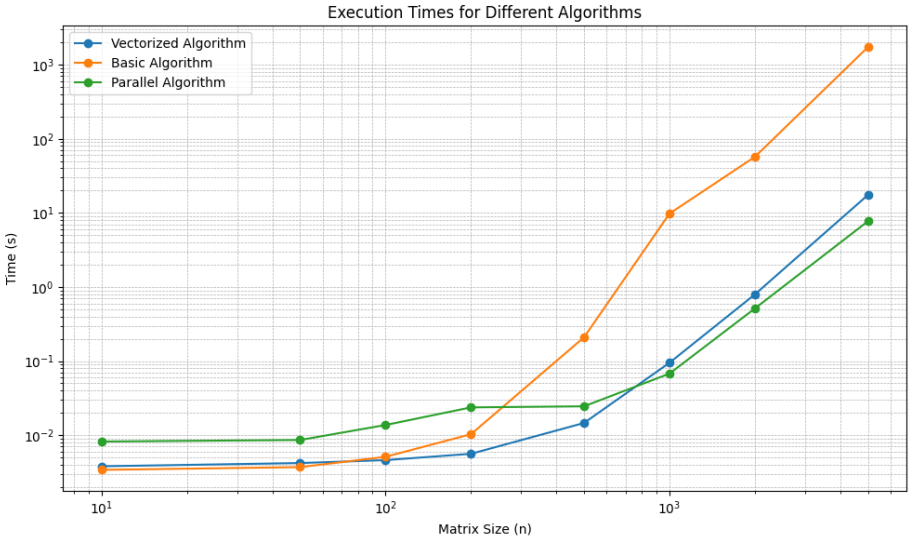

Figure 1: Execution Time of Different Matrix Multiplication Algorithms

Speedup and Efficiency

Performance improvements achieved by the vectorized and parallel algorithms were quantified using **speedup** and **efficiency** metrics. Speedup measures the relative performance gain compared to the basic algorithm, while efficiency evaluates the effective utilization of computational resources, defined as speedup per thread in the parallel implementation. These metrics are critical for understanding the scalability and practicality of each approach.

The formulas used to compute these metrics are as follows:

- **Speedup** compared to the basic algorithm:

$$\text{Speedup} = \frac{\text{Execution Time (Basic)}}{\text{Execution Time (Optimized)}}$$

- **Efficiency of Parallel Execution**:

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of Threads}}$$

Table 2 summarizes the speedup and efficiency results for the vectorized and parallel algorithms across various matrix sizes.

The **vectorized algorithm** provided modest speedup for small-to-medium matrix sizes, demonstrating its ability to optimize computations up to a certain threshold. However, as the matrix size increased, its performance gains diminished due to the overhead caused by memory flattening and matrix transposition.

Matrix Size (n)	Speedup (Vectorized)	Speedup (Parallel)	Efficiency (Parallel)
10	0.89	0.41	0.02
50	0.88	0.43	0.02
100	1.11	0.37	0.02
200	1.84	0.43	0.02
500	14.36	8.56	0.43
1000	103.13	144.48	7.22
2000	71.89	111.36	5.57
5000	99.07	224.41	11.22

Table 2: Speedup and Efficiency for Vectorized and Parallel Algorithms.

The **parallel algorithm**, on the other hand, achieved remarkable speedup for larger matrices. For instance, it attained a peak speedup of 224.41x for $n = 5000$. Its efficiency improved significantly with larger matrix sizes, reflecting better utilization of the available 20 logical threads as computational workload increased. This indicates that the parallel algorithm scales well with matrix size, making it highly effective for large-scale computations.

Memory Usage

Memory usage is a critical factor in the performance and scalability of matrix multiplication algorithms, particularly for large datasets where memory resources are often a limiting constraint. Table 3 provides a detailed comparison of memory consumption across the basic, vectorized, and parallel algorithms for various matrix sizes. The results reveal key trade-offs in how each approach utilizes memory, directly impacting their scalability and applicability.

Matrix Size	Vectorized Algorithm	Basic Algorithm	Parallel Algorithm
10x10	0.24	0.16	0.19
50x50	1.10	0.16	0.20
100x100	1.26	0.16	0.55
200x200	2.66	0.47	0.97
500x500	13.06	1.95	4.67
1000x1000	54.08	7.80	16.63
2000x2000	195.88	31.50	77.55
5000x5000	1038.69	193.77	424.48

Table 3: Memory Usage for Different Matrix Multiplication Algorithms (MB)

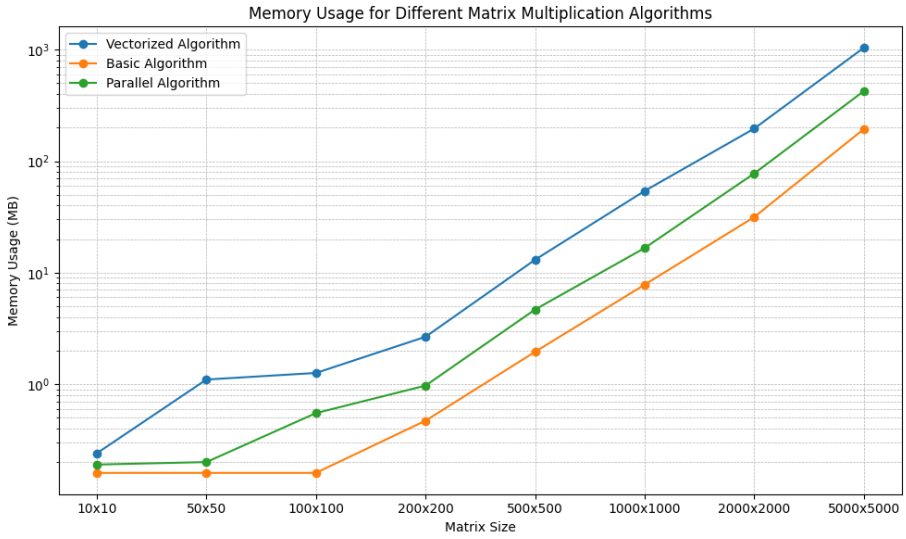

Figure 2: Memory Usage for Different Matrix Multiplication Algorithms

The **basic algorithm** demonstrated the lowest memory usage across all matrix sizes. Its simplicity lies in directly iterating over the input matrices without requiring additional memory for intermediate transformations or data structures. This low memory footprint makes it a viable option for scenarios with strict memory limitations, such as embedded systems or environments with constrained hardware. However, while it is memory-efficient, its computational inefficiency often offsets this advantage for large matrices.

The **vectorized algorithm**, in contrast, consumed the most memory due to the overhead associated with matrix flattening and transposition operations. Flattening the matrix into a one-dimensional array for vectorized computations significantly increases memory usage, especially for larger matrices. Although this approach improves computational performance for medium-sized matrices, its excessive memory demands limit its scalability. For example, at $n = 5000$, the vectorized algorithm required over 1 GB of memory, highlighting a critical drawback in memory-constrained environments.

The **parallel algorithm** strikes a balance between performance and memory usage. By utilizing multi-threading, it avoids the need for extensive data restructuring while still achieving significant performance improvements. Its memory requirements were consistently higher than the basic algorithm but substantially lower than the vectorized implementation. This makes the parallel algorithm more scalable for large matrices while maintaining a reasonable memory footprint. For instance, at $n = 5000$, it required approximately 424 MB of memory, less than half of the vectorized approach's usage.

6 Conclusion

The growing complexity of scientific and engineering applications demands efficient solutions for matrix multiplication, a cornerstone operation in many computational tasks. This study focused on analyzing the performance of three distinct implementations of matrix multiplication: a basic algorithm, a vectorized approach using Java parallel streams, and a parallel implementation utilizing the ForkJoin framework. Each method was evaluated based on execution time, speedup, efficiency, and memory usage, providing a comprehensive comparison across a wide range of matrix sizes.

The results highlighted the simplicity and reliability of the **basic algorithm**, which is easy to implement and requires minimal memory. However, its computational cost increases drastically for larger matrices, making it unsuitable for high-performance scenarios. While it serves well for small-scale computations, its poor scalability and inefficiency for large matrices limit its broader applicability.

The **vectorized algorithm** demonstrated notable improvements in execution time for small to medium matrix sizes, leveraging the power of parallel streams for moderate speedup. However, its advantages diminished as the matrix size increased, largely due to the overhead associated with matrix flattening and transposition. Additionally, this approach had the highest memory usage among the three algorithms, which restricts its utility for large-scale problems where memory resources are limited.

In contrast, the **parallel algorithm** emerged as the most effective solution for large-scale matrix multiplication. By distributing computations across multiple threads, it achieved significant speedup and improved efficiency as the matrix size increased. For instance, the parallel algorithm achieved a speedup of over 224 for a matrix size of 5000×5000, making it highly suitable for computationally intensive tasks. However, the parallel algorithm's complexity and thread management overhead reduced its efficiency for smaller matrices, where the performance gains were less pronounced.

In summary, the choice of the algorithm depends on the specific use case and constraints. For small matrices or scenarios with limited memory, the basic or vectorized algorithms may suffice. However, for large datasets requiring high computational performance, the parallel algorithm is the preferred choice due to its scalability and resource efficiency.

7 Future work

The current study focuses on optimizing matrix multiplication within a single-machine environment using basic, parallel, and vectorized approaches. While this provides valuable insights into performance improvements achievable with multi-threading and SIMD-like techniques, future work could extend this exploration to distributed systems. Distributed matrix multiplication frameworks, such as Apache Spark or Hadoop, enable scaling computations across multiple machines, making them suitable for handling matrices too large to fit in the memory of a single system. Investigating these frameworks would provide a deeper understanding of how distributed architectures influence scalability, data partitioning strategies, and fault tolerance in matrix operations.

Building on the techniques explored in this work, the next logical step is to study how distributed computing frameworks handle matrix multiplication for extremely large datasets. This would involve analyzing key challenges such as network communication overhead, data shuffling, and synchronization across nodes. By comparing the distributed approach with the basic, parallel, and vectorized methods presented here, it will be possible to evaluate how scalability is achieved as matrix sizes increase and how efficiently distributed systems can manage resource utilization across multiple nodes.

Lastly, there are opportunities to refine the benchmarking methodology itself. Future studies could incorporate a broader range of performance metrics, such as energy efficiency, detailed network latency analysis, and strategies for minimizing data transfer overhead. By expanding the scope of experimentation and focusing on distributed systems, future work can provide a more comprehensive framework for selecting and optimizing matrix multiplication strategies for large-scale, distributed computational environments.