# Parallel and Vectorized Matrix Multiplication

**Lucía Afonso Medina**

November 30, 2024

# Contents

**Abstract**

This project investigates the performance optimization of matrix multiplication using various parallel and vectorized approaches. Techniques tested include atomic operations, synchronized blocks, semaphores for thread coordination, vectorized instructions (SIMD), thread pooling with Executor, and Java Streams for functional parallelism. Large matrices were used to evaluate execution time, speedup, parallel efficiency, and resource utilization. By testing with large matrices, we demonstrated significant performance improvements with multi-threading and vectorized approaches. Our results highlight the potential of parallelization and vectorization to enhance computational efficiency for matrix operations in resource-constrained environments.

# 1 Introduction

Matrix multiplication is a fundamental operation in many areas of scientific computing, engineering, and machine learning. However, as the size of matrices increases, the time required to perform these operations can grow significantly, leading to the need for more efficient algorithms. Traditional matrix multiplication algorithms run in a time complexity of $O(n^3)$, making them inefficient for large-scale data.

This project explores various parallel and vectorized approaches to optimize matrix multiplication, taking advantage of multi-threading and data-level parallelism. By implementing atomic operations, synchronized blocks, semaphores, vectorized instructions (e.g., SIMD), thread pooling with the Executor framework, and Java Streams, we evaluate different techniques to improve performance. Each method is tested with large matrices to measure execution time, speedup, parallel efficiency, and resource usage.

Parallelization techniques such as thread pooling and synchronization mechanisms (e.g., semaphores) enable efficient resource utilization by distributing computations across multiple cores. Additionally, vectorization leverages modern processor instructions to perform operations on multiple data elements simultaneously, further reducing execution time. Java Streams offer a functional programming approach to parallelism, simplifying code while maintaining scalability.

The aim of this study is to identify the trade-offs and benefits of each method, analyze their scalability, and understand how they perform under various matrix sizes and computational loads. The findings provide valuable insights into optimizing matrix multiplication for real-world applications, demonstrating the combined potential of parallel and vectorized approaches to tackle computationally intensive tasks.

# 2 Problem Statement

Matrix multiplication is a computationally intensive operation, with the naive $O(n^3)$ algorithm becoming inefficient for large matrices. Modern hardware offers parallel and vectorized capabilities that remain underutilized in basic implementations. This work analyzes the effectiveness of techniques such as parallelism, vectorization, atomic operations, semaphores, and synchronized blocks in optimizing matrix multiplication. By comparing these approaches to the sequential algorithm, the study evaluates improvements in execution time, efficiency, and resource usage, providing insights into best practices for leveraging modern hardware while addressing challenges related to thread safety and synchronization overhead.

# 3 Methodology

This section provides an overview of the various algorithms implemented to optimize matrix multiplication. The implemented approaches are detailed below.

- **Basic Implementation:** A sequential matrix multiplication algorithm ($O(n^3)$) was implemented as a baseline for comparison.
- **Parallel Implementations:**
  - **Parallel Streams:** This method simplifies parallelism by using Java's Stream API. It dynamically distributes tasks among threads via the common ForkJoinPool. The ease of implementation makes it ideal for quick parallelization. However, because all tasks share the same pool, performance can suffer when multiple processes are running simultaneously. It is best suited for applications requiring moderate parallelism without extensive configuration.
  - **ExecutorService:** This approach offers explicit control over thread management by creating a fixed thread pool. Each thread is assigned a row of the result matrix, ensuring efficient workload distribution. Its predictability and configurability make it a robust choice for controlled environments, although thread pool initialization and management add some overhead.
  - **Semaphores:** This method uses semaphores to control the number of threads executing concurrently. It prevents resource exhaustion in multi-threaded environments but introduces synchronization overhead. Each thread computes one row of the resulting matrix while ensuring safe access to shared resources.
  - **Synchronized Blocks:** Threads explicitly compute subsets of the matrix, with results updated in a thread-safe manner using synchronized blocks. While this guarantees data integrity, the overhead of acquiring and releasing locks reduces efficiency, particularly for smaller matrices.
  - **Atomic:** This method uses AtomicInteger to manage row indices in a thread-safe manner. While it avoids explicit locking, its sequential processing of rows limits its ability to fully exploit parallelism, making it more suitable for tasks with smaller matrices or minimal contention.

- **Vectorized Implementation:** The vectorized implementation begins by transforming the input matrices from 2D arrays into 1D arrays to optimize memory access and alignment. Using the Aparapi library, the computation is parallelized with its 'Kernel' class, which distributes the workload across available GPU or CPU threads. Each thread computes a single element of the resulting matrix, leveraging SIMD (Single Instruction, Multiple Data) capabilities to achieve data-level parallelism. Once the computation is complete, the resulting 1D array is reconstructed into a 2D matrix to match the original format.

# 4 Metrics

- **Execution Time:** Measured in milliseconds for all implementations to assess their raw performance.
- **Speedup:** The ratio of the execution time of the sequential algorithm ($T_{\text{sequential}}$) to the parallel/vectorized algorithm ($T_{\text{parallel}}$):

$$\text{Speedup} = \frac{T_{\text{sequential}}}{T_{\text{parallel}}}$$

- **Parallel Efficiency:** Indicates how effectively parallelism is utilized with $n$ threads:

$$\text{Efficiency} = \frac{\text{Speedup}}{n}$$

- **Resource Usage:**
  - **CPU Utilization:** Measured by analyzing how many cores were actively utilized during computation.
  - **Memory Usage:** Monitored to identify any additional overhead introduced by parallelization and vectorization.
- **Scalability:** Evaluated by systematically varying matrix sizes and thread counts. This metric assesses each approach's ability to maintain performance and efficiency as the computational workload and concurrency levels increase, providing insights into their adaptability to larger datasets and more demanding parallel environments.

## 4.1 Experimental Setup

All experiments were conducted on a machine with the following specifications:

- Processor: 13th Gen Intel(R) Core(TM) i5-1340P, 1.90 GHz
- RAM: 16.0 GB
- Operating System: Windows 11 Home, Version 23H2
- Compiler/Interpreter versions:
  - Java: Version 21.0.1
- Number of cores: 16

## 4.2 Experimental Procedure

To ensure reliable and accurate results, each matrix multiplication implementation was executed multiple times for various matrix sizes and concurrency levels. Execution time was measured using the Java Microbenchmark Harness (JMH), a tool specifically designed to provide high-resolution and precise timing for benchmarking in Java. Implementations tested included atomic operations, synchronized blocks, streams, ExecutorService, and vectorized methods. The number of threads was varied systematically to evaluate scalability and parallel efficiency. Additionally, memory usage was monitored to assess the resource overhead introduced by parallel and vectorized techniques. The results were analyzed to determine the trade-offs between performance and resource usage, identify the maximum matrix size each implementation could handle efficiently, and evaluate the scalability and adaptability of each method under increasing computational loads.

# 5 Experiments

The chart 1 illustrates the execution time for multiple matrix computation methods, including Basic, Vectorized, Atomic, Executor with 16 Threads, ParallelStream with 16 Threads, Semaphore with 16 Threads, and Synchronized Blocks with 16 Threads. The X-axis represents the matrix size, while the Y-axis (logarithmic scale) indicates the time required to complete the computation in milliseconds. Each line corresponds to a specific method, allowing a performance comparison as matrix size increases.

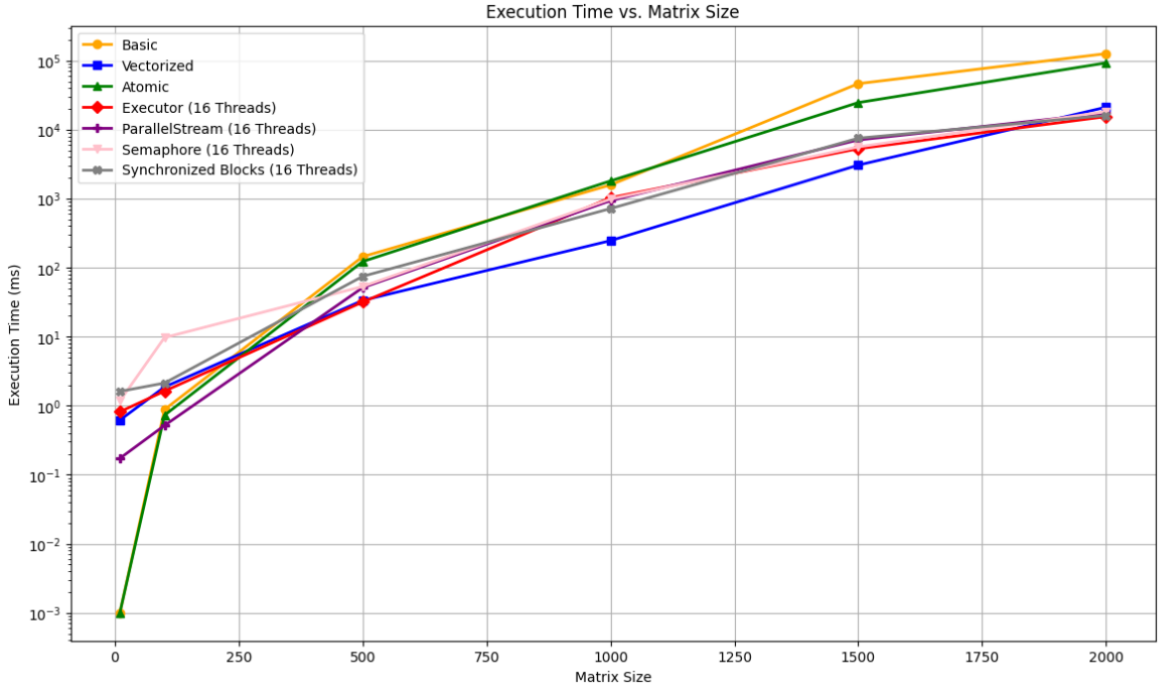


Figure 1: Execution time comparison for different matrix sizes and optimization algorithms.

- **Basic Algorithm:**
  The basic algorithm consistently has the highest execution time as the matrix size increases. This is expected because it lacks any form of parallelism, relying entirely on sequential processing. The growth in execution time is exponential, which aligns with its $O(n^3)$ computational complexity.

- **Vectorized Approach:**
  The vectorized method shows the fastest execution times across all matrix sizes. This highlights the significant impact of hardware-level parallelism (e.g., SIMD instructions) on reducing computational overhead. It scales exceptionally well with increasing matrix sizes, maintaining superior performance even for large matrices.

- **Atomic Operations:**
  For smaller matrices, the execution is fast because of the simplicity of the algorithm and the absence of explicit locking mechanisms. However, as the matrix size increases, the sequential nature of the implementation means that it does not fully utilize the benefits of parallelism. This results in slower performance compared

to other approaches like ExecutorService or Parallel Streams, which distribute the workload more effectively across multiple threads.

- **ExecutorService (16 Threads):**
  ExecutorService performs well across all matrix sizes, offering significant improvements over the basic algorithm. The fixed thread pool allows efficient utilization of resources, maintaining consistent scalability as matrix size increases.

- **Parallel Streams (16 Threads):**
  Parallel Streams show performance comparable to ExecutorService, demonstrating its suitability for functional parallelism with minimal coding complexity. It starts with higher execution times for small matrices but scales better for larger sizes.

- **Semaphore-Based Synchronization:**
  Semaphore-based implementation introduces noticeable overhead for smaller matrices due to the management of thread concurrency. Performance improves as matrix size increases, making it more efficient for larger datasets. However, it remains slower than ExecutorService and Parallel Streams.

- **Synchronized Blocks:**
  Synchronized Blocks show slightly higher execution times compared to Semaphore and Atomic operations, likely due to lock contention. While it scales reasonably well with larger matrices, it is less efficient compared to other parallel approaches.

The plot in Figure: 2 consists of four subplots, each representing the SpeedUp performance of different matrix multiplication algorithms for specific matrix sizes (100, 500, 1500, and 2000). Each subplot displays the relationship between the number of threads (x-axis) and the resulting SpeedUp (y-axis).

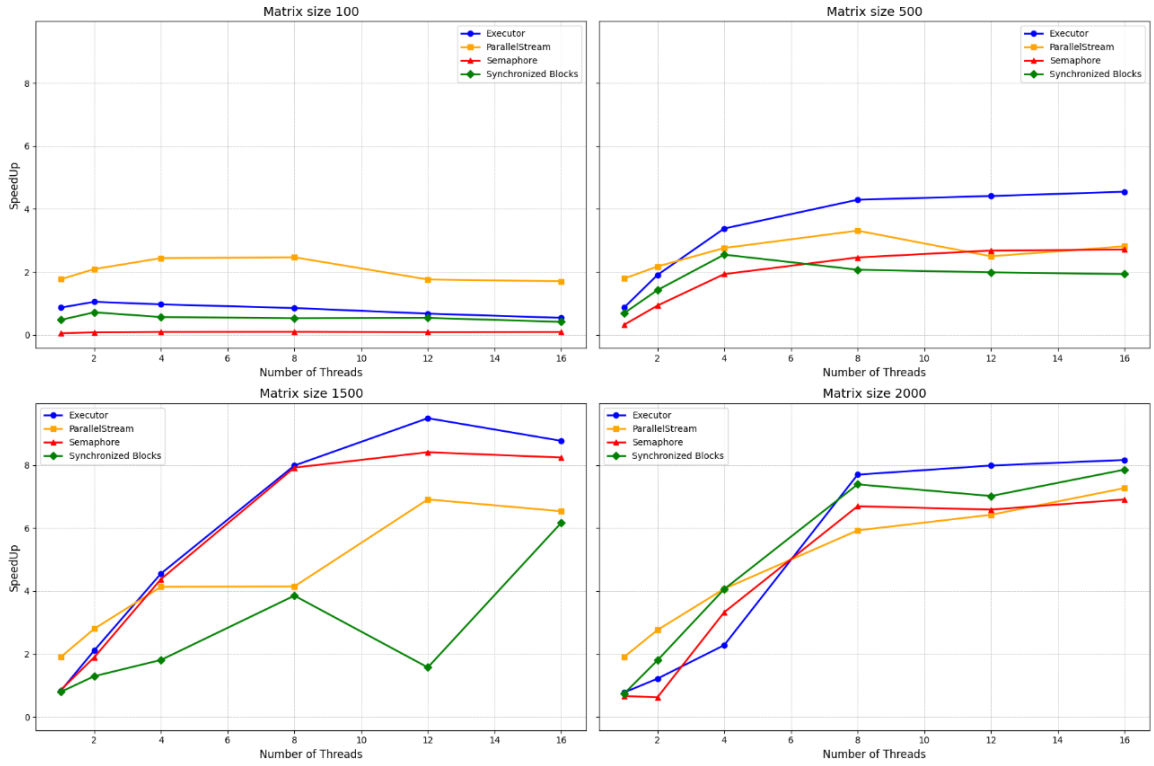


Figure 2: Speedup comparison for different matrix sizes and optimization algorithms.

### Matrix Size 100

- **Overall Observations:** Speedup is minimal across all methods due to the small computational workload.
- **Executor (Blue Line):** Shows slight improvement as the number of threads increases but plateaus quickly. Thread management overhead dominates the computation.
- **ParallelStream (Orange Line):** Performs marginally better than Semaphore and Synchronized Blocks but does not scale effectively due to its higher overhead.
- **Semaphore (Red Line) & Synchronized Blocks (Green Line):** Almost no speedup observed, indicating that synchronization overhead outweighs the benefits of parallelization for such a small matrix.

### Matrix Size 500

- **Overall Observations:** Speedup becomes noticeable, particularly for Executor and ParallelStream.
- **Executor (Blue Line):** Demonstrates better scalability and maintains a consistent increase in speedup with the number of threads.

- **ParallelStream (Orange Line):** Shows competitive performance with Executor but experiences a slight dip at higher thread counts due to overhead.
- **Semaphore (Red Line):** Marginal improvement with more threads but remains limited by the locking mechanism.
- **Synchronized Blocks (Green Line):** Similar to Semaphore, showing poor scalability and minimal speedup due to synchronization overhead.

**Matrix Size 1500**

- **Overall Observations:** Speedup is significant for Executor and ParallelStream, with Executor achieving near-ideal scaling.
- **Executor (Blue Line):** Achieves the highest speedup, peaking near 8x with 16 threads. It demonstrates excellent scalability.
- **ParallelStream (Orange Line):** ParallelStream performs efficiently, reaching its peak performance slightly below ExecutorService with a speedup of approximately 7x to 8x when using 12 threads. While it provides a convenient and compact way to implement parallelism, its reliance on the common ForkJoinPool introduces overhead and limits scalability at higher thread counts. This makes it slightly less efficient compared to ExecutorService in highly concurrent scenarios.
- **Semaphore (Red Line):**Semaphore also performs well but remains slightly behind Executor, especially at higher thread counts. However, its scalability is still limited by locking overhead.
- **Synchronized Blocks (Green Line):** The poorest performer, exhibiting the lowest speedup and scalability due to the high cost of synchronization.

**Matrix Size 2000**

- **Overall Observations:** Speedup continues to increase, with Executor maintaining the highest scalability.
- **Executor (Blue Line):** Achieves the best performance, demonstrating near-linear scaling as the number of threads increases. It reaches a SpeedUp arounds 8 with 16 threads, making it the most efficient approach.
- **ParallelStream (Orange Line):** Performs competitively with Executor, maintaining strong scalability. Although slightly less efficient than Executor, it provides consistent performance gains with increasing threads.
- **Synchronized Blocks (Green Line):** Outperforms Semaphore for this matrix size, achieving a SpeedUp close to 8 with 16 threads. It demonstrates better scalability compared to smaller matrices, as the larger computational workload reduces the impact of synchronization overhead.
- **Semaphore (Red Line):** Shows improvement compared to smaller matrix sizes but falls behind Synchronized Blocks. Its SpeedUp plateaus around 6 with 16 threads, highlighting the limitations imposed by thread management overhead.

The plot in Figure 3 consists of four subplots, each illustrating the **Efficiency** of different matrix multiplication parallelization methods across specific matrix sizes (100, 500, 1500, and 2000). Each subplot depicts the relationship between the number of threads (x-axis) and the resulting **Efficiency** (y-axis), where Efficiency measures the utilization of computational resources relative to the number of threads.

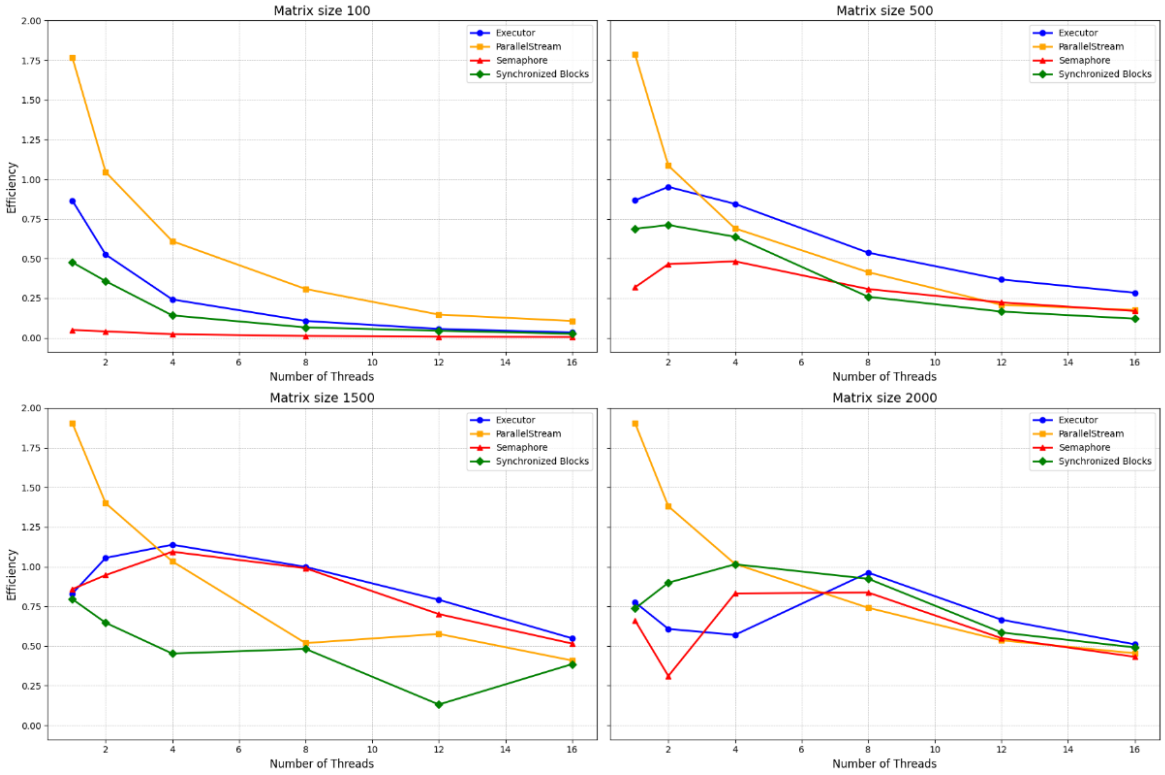


Figure 3: Efficiency comparison for different matrix sizes and optimization algorithms.

### Matrix Size 100

- **General Observation:** Efficiency is very low across all methods due to the small computational workload, which makes the overhead of thread management and synchronization dominate.
- **Executor (Blue Line):**
  - Shows gradual efficiency decline as the number of threads increases, indicating that the thread overhead outweighs computational gains.
- **ParallelStream (Orange Line):**
  - Begins with the highest efficiency among all methods but declines sharply with increasing thread count. The rapid drop indicates that the overhead of managing the parallel streams outweighs the benefits for such a small workload.

- **Semaphore (Red Line) and Synchronized Blocks (Green Line):**
  - Maintain consistently low efficiency, suggesting that synchronization overhead significantly affects small matrix sizes.

**Matrix Size 500**

- **General Observation:** Efficiency increases slightly for larger matrix sizes, but synchronization methods (Semaphore and Synchronized Blocks) still underperform.
- **Executor (Blue Line):**
  - Maintains the best efficiency across most thread counts. Efficiency declines steadily as the thread count increases, but the drop is more gradual compared to smaller matrix sizes. Demonstrates good scalability with a relatively consistent performance across all threads.
- **ParallelStream (Orange Line):**
  - Starts with higher efficiency than other methods but experiences a sharp decline after 2 threads. Becomes comparable to Executor for thread counts greater than 8, indicating overhead issues at higher parallelism levels.
- **Semaphore (Red Line):**
  - Shows noticeable improvement in efficiency compared to smaller matrix sizes but remains below Executor and ParallelStream. Peaks around 4 threads and declines steadily, highlighting limitations caused by locking overhead.
- **Synchronized Blocks (Green Line):**
  - Efficiency is slightly better than Semaphore but remains low overall. Exhibits poor scalability as the number of threads increases, with minimal improvements beyond 4 threads.

**Matrix Size 1500**

- **General Observation:** Efficiency improves significantly compared to smaller matrix sizes, demonstrating that the larger computational workload effectively offsets thread management and synchronization overhead. Peak efficiency is observed at lower thread counts (4–8 threads) for most methods, after which efficiency gradually declines.
- **Executor (Blue Line):**
  - Consistently achieves the highest efficiency among all methods, peaking around 4–6 threads. Gradual decline in efficiency with more threads, but it maintains superior scalability compared to other methods.
- **ParallelStream (Orange Line):**
  - Starts with the highest efficiency at 1 thread, but its performance dips as threads increase, stabilizing below Executor at higher thread counts.

- **Semaphore (Red Line):**
  - Starts with low efficiency but shows noticeable improvement at higher thread counts, with efficiency peaking around 4 threads.
- **Synchronized Blocks (Green Line):**
  - Lowest efficiency among all methods, showing poor scalability as thread counts increase.While slightly better at 4-8 threads, it quickly drops below 0.5 efficiency, indicating high synchronization overhead.

**Matrix Size 2000**

- **General Observation:** Efficiency is highest among all tested matrix sizes, indicating that larger matrices leverage thread parallelism better.
- **Executor (Blue Line):**
  - Shows the best performance, maintaining consistent efficiency with increasing thread counts.Peaks around 4–8 threads, demonstrating strong scalability for the larger workload.
- **ParallelStream (Orange Line):**
  - Begins with the highest efficiency at 1 thread but experiences a sharp decline as thread count increases.Stabilizes around the same efficiency as Executor at higher thread counts, though slightly less consistent.
- **Semaphore (Red Line):**
  - Demonstrates significant improvement compared to smaller matrices but plateaus at higher thread counts.
- **Synchronized Blocks (Green Line):**
  - Surprisingly competitive for this matrix size, performing close to or matching Semaphore and even ParallelStream at certain thread counts. Scalability improves due to the larger computational workload, which reduces the impact of synchronization overhead.

In Figure: 4 we plot the speedup of both the Vectorized and Atomic algorithms relative to the Basic algorithm. The purpose of this analysis is to evaluate the performance improvement achieved by these optimization techniques and understand their scalability as the matrix size increases.

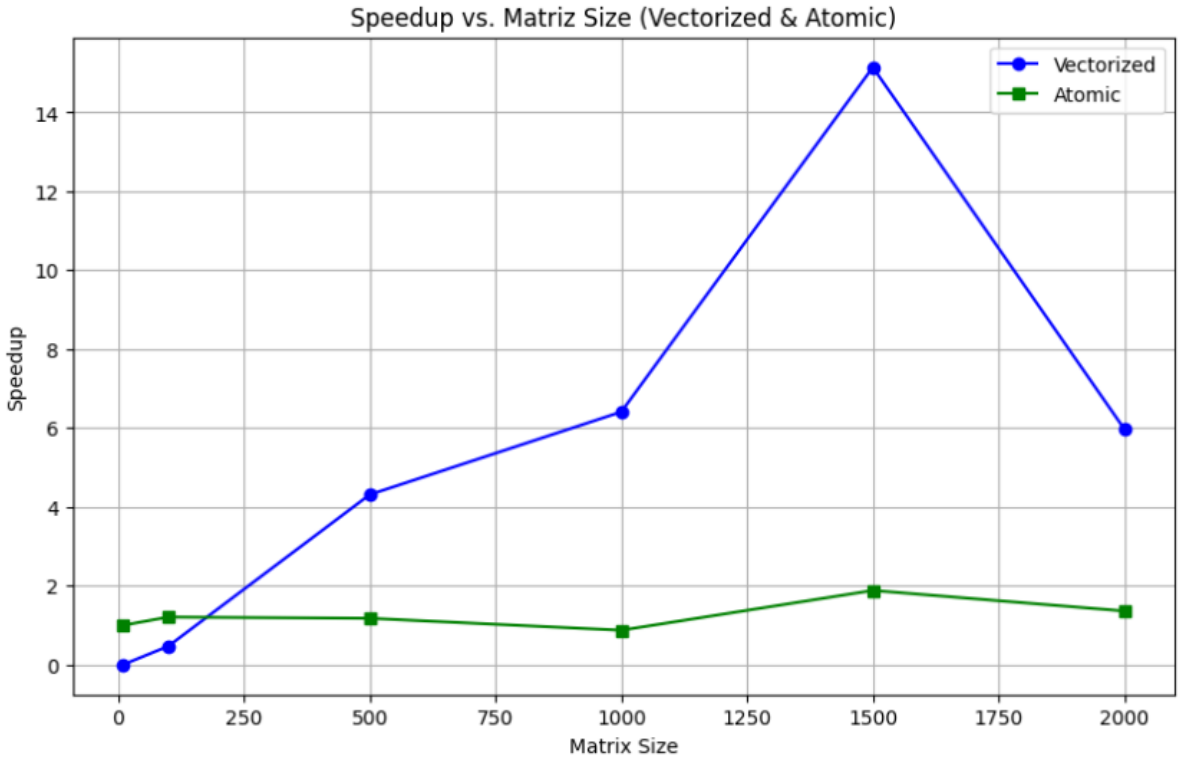


Figure 4: Speedup comparison for vectorized and atomic algorithms.

The vectorized approach demonstrates a substantial performance improvement over the basic algorithm, particularly for larger matrix sizes. The speedup increases steadily as the matrix size grows, reaching a peak at around 1500, where it surpasses 14x. This highlights the efficiency of vectorized operations, which take advantage of hardware-level optimizations like performing multiple calculations simultaneously within a single instruction (SIMD). However, after the peak, the speedup slightly decreases, possibly due to hardware limitations or memory bandwidth bottlenecks.

In contrast, the atomic approach provides a modest and consistent speedup of approximately 1.5–2x across all matrix sizes. While it improves over the basic algorithm, the speedup remains relatively flat. This suggests that the overhead of atomic operations, such as ensuring memory consistency or synchronization, limits their overall performance. As a result, the atomic approach does not scale as effectively as the vectorized approach.

From a scalability perspective, the vectorized algorithm excels, achieving substantial improvements as matrix size increases. On the other hand, the atomic approach shows limited scalability, maintaining a similar speedup regardless of the matrix size.

The 'Memory Usage vs. Matrix Size' graph in Figure 5 highlights the memory consumption patterns of the different algorithmic approaches as the matrix size increases.

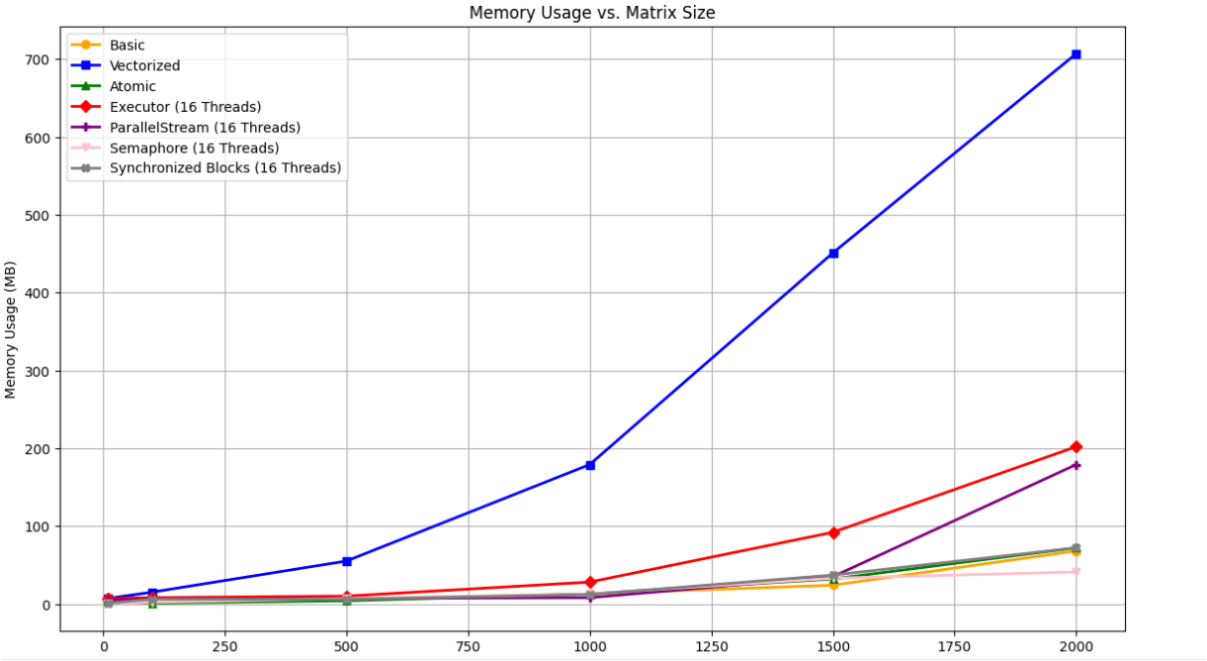


Figure 5: Memory usage comparison for different matrix sizes and optimization algorithms..

- **Basic Algorithm:** The memory usage of the basic algorithm (orange line) remains relatively low and steady across all tested matrix sizes, demonstrating minimal memory overhead.
- **Vectorized Implementation:** The vectorized approach (blue line) exhibits a steep increase in memory usage as matrix size grows. It surpasses all other approaches, indicating significant memory demands for this implementation.
- **Atomic Approach:** The atomic implementation (green line) has very minimal memory usage, suggesting an efficient memory model. It consistently uses less memory than most other methods.
- **Executor Service (16 Threads):** The executor approach (red line) demonstrates moderate memory usage, increasing gradually with matrix size. It balances parallel performance with reasonable memory consumption.
- **Parallel Stream (16 Threads):** The parallel stream implementation (purple line) shows controlled memory growth, but its memory consumption is slightly higher compared to the semaphore approach at larger matrix sizes.
- **Semaphore (16 Threads):** The semaphore approach (pink line) maintains a relatively low and consistent memory usage pattern, scaling linearly with matrix size.
- **Synchronized Blocks (16 Threads):** The synchronized blocks implementation (grey line) follows a similar trajectory to the semaphore, maintaining efficient memory usage even as the matrix size grows.

# 6 Conclusions

To concluede, the results demonstrate clear differences in performance, scalability, and memory efficiency across the tested algorithms, highlighting the importance of selecting the appropriate method based on workload and resource requirements.

Among the methods tested, **ExecutorService** consistently exhibits the highest efficiency and scalability, peaking at thread counts between 4 and 8. Its robust performance across various matrix sizes makes it the most reliable approach for parallelizing matrix multiplication. While **ParallelStream** offers competitive performance, particularly for larger matrices, it suffers from higher overhead at smaller thread counts, leading to a decline in efficiency as the number of threads increases. Despite this limitation, it remains a viable alternative for scalable computations, especially when implementation simplicity is a priority.

Synchronization-based methods, such as **Semaphore** and **Synchronized Blocks**, demonstrate the lowest efficiency due to significant synchronization overhead. While **Semaphore** shows slight improvements at larger matrix sizes and higher thread counts, it consistently underperforms compared to ExecutorService and ParallelStream. **Synchronized Blocks**, although slightly more competitive for the largest matrix size (2000), suffer from poor scalability as thread counts increase, reflecting the high cost of synchronization operations.

The **vectorized approach** achieves the fastest execution times, particularly for larger matrices, by leveraging hardware-level parallelism (e.g., SIMD instructions). However, this speed advantage comes at the expense of significantly higher memory usage, making it less suitable for memory-constrained environments. In contrast, the **basic algorithm**, while memory-efficient, exhibits the highest execution times and the poorest scalability due to its sequential nature and $O(n^3)$ complexity.

Memory usage reflects key trade-offs between performance and resource efficiency. The vectorized approach, despite its superior speed, incurs the highest memory demands. Synchronization-heavy methods (Semaphore and Synchronized Blocks) and **atomic operations** consume much less memory but lag in execution time and scalability. Multi-threaded methods like **ExecutorService** and **ParallelStream** strike an effective balance between memory usage and performance, making them ideal for high-performance matrix computations requiring both speed and resource efficiency.

# 7 Future Work

Future work in this area could involve optimizing the distributed matrix multiplication approach by exploring more advanced distributed computing frameworks. Further research could focus on improving the efficiency of network communication by reducing data transfer overhead and enhancing load balancing across nodes to ensure optimal resource utilization. Additionally, investigating the impact of distributed memory architectures, such as GPU clusters or cloud-based solutions, could offer promising directions for handling even larger matrices. Finally, a deeper analysis of scalability across varying matrix sizes and network configurations could help identify bottlenecks and provide insights into how distributed systems can be further optimized for matrix operations.

The source code for this project is available at the following GitHub repository: *Parallel-and-Vectorized-Matrix-Multiplication.git*