UNIVERSITY OF LAS PALMAS DE GRAN CANARIA

Parallel and Vectorized Matrix Multiplication

Casimiro Torres, Kimberly

Data Science and Engineering

November 29, 2024

Abstract This project focuses on the analysis and implementation of parallel and vectorized computing techniques applied to matrix multiplication, with the aim of optimizing performance and resource utilization in modern systems. Multiple implementations were developed, including a basic algorithm as a reference, parallel techniques based on executors, streams, and synchronization mechanisms such as atomic variables, synchronized blocks, and semaphores, as well as a vectorized implementation utilizing SIMD (Single Instruction, Multiple Data) instructions. Through a rigorous methodology, key metrics such as speedup, efficiency per thread, CPU consumption, memory usage, cores used, and execution time were evaluated. The results show that configurations such as parallelExecutors with 8 threads offer an optimal balance between performance and efficiency, while the vectorized implementation stands out by maximizing speedup with minimal resource consumption. These conclusions highlight the importance of tailoring optimization techniques to the specific characteristics of hardware and computational workloads, providing a solid framework for applications and future research in high-performance computing.

Keywords: Matrix multiplication, Optimization, Vectorization, Parallelization, Executors, Streams, Synchronization mechanisms, SIMD, Speedup, Efficiency, Threads, CPU usage, Memory usage, Cores used, Execution time, Benchmark

1 Introduction

Matrix multiplication is one of the most widely used mathematical operations in applications that require high computational performance, including scientific simulations, machine learning, image processing, and data analysis. The primary objective of this project is to explore the capabilities of parallel and vectorized computing in the context of matrix multiplication, leveraging modern programming tools and advanced optimization techniques. Through the implementation and evaluation of parallel and vectorized methods, the project aims to improve performance, measure efficiency, and maximize the use of available resources.

The project development focused on implementing a basic version of matrix multiplication as a reference, followed by multiple optimizations that take advantage of modern hardware resources. These optimizations include executor-based parallelization, parallel streams, and advanced synchronization techniques such as synchronized blocks, semaphores, and atomic. Additionally, a vectorized version was developed using SIMD instructions to perform data-level parallel operations, enabling simultaneous processing of multiple elements.

The experimental approach involved generating and testing large-sized matrices, configuring the number of threads, and observing how these parameters affect the scalability of the implemented techniques. The evaluated metrics included the speedup achieved compared to the basic algorithm, the efficiency of parallel execution, and resource usage, particularly execution time, memory consumption, CPU utilization, and cores used. The tests were conducted on a system configured with multiple cores, allowing validation of the impact of the optimizations and highlighting the practical limits of each technique.

To capture and analyze the results, benchmarking tools such as JMH (Java Microbenchmark Harness) were employed, generating detailed data. These results were complemented with automatically generated charts that illustrate the relationship between matrix size, the number of threads, and the performance achieved. This analysis allowed the identification of behavioral patterns and further optimization of the implementations.

In summary, this project provides a detailed and structured evaluation of parallel and vectorized techniques applied to matrix multiplication, highlighting the advantages, limitations, and opportunities for improvement within the context of modern hardware architectures. The practical implementation, combined with rigorous analysis, offers a solid foundation for understanding and applying these techniques in real-world high-performance computational scenarios.

2 Problem Statement

Matrix multiplication is a fundamental operation in scientific computing and the development of advanced technological applications. However, the traditional approach to performing this operation, based on sequential algorithms, presents significant limitations in terms of scalability and efficiency, especially when dealing with large matrices. As modern applications demand processing massive amounts of data in real-time, these limitations become critical bottlenecks that can affect the overall system performance and, consequently, its viability in high-performance environments. This issue is further compounded by the exponential increase in required operations as matrix dimensions grow, leading to higher consumption of time and computational resources.

The transition to multicore hardware architectures and the development of extensions such as SIMD instructions have opened new opportunities to address these challenges. These technologies allow computations to be distributed across multiple processing cores and executed in parallel, theoretically reducing the time required to complete complex tasks such as matrix multiplication. However, efficiently adopting these technologies requires overcoming several practical challenges related to parallel programming, thread synchronization, and shared resource management, as well as optimizing memory usage and minimizing computational overhead.

The main problem addressed by this project is how to implement and optimize matrix multiplication techniques that fully leverage the capabilities of modern hardware, specifically parallelism and vectorization, without compromising the accuracy of the results or causing inefficient resource usage. While well-established approaches exist, this project focuses on a modular implementation that enables the independent analysis of each technique and evaluates its performance under different experimental conditions.

Among the specific issues investigated by this project is the efficiency of parallel execution. While the use of multiple threads can enhance performance, this benefit is not always linear due to challenges such as synchronization overhead, conflicts in accessing shared memory, and unequal workload distribution among threads. A significant challenge is to evaluate how these limitations affect scalability and identify ways to mitigate them.

Another relevant aspect is the impact of the number of threads available. Modern architectures allow the use of multiple processing threads, but fully utilizing these resources requires careful algorithm design. The project evaluates how the number of assigned threads influences the achieved speedup and analyzes the point at which performance saturation occurs.

Efficient synchronization is another key challenge. Parallel methods require synchronization mechanisms to coordinate thread execution and avoid issues such as race conditions or data inconsistencies. This project examines techniques such as synchronized blocks, semaphores, and atomic objects, evaluating their impact on performance and their suitability for different scenarios.

Vectorization and SIMD operations also play a crucial role. The ability to perform multiple simultaneous operations at the data level is a powerful feature of SIMD extensions. However, effectively integrating these instructions into the matrix multiplication algorithm requires adapting the data and execution flow to maximize their impact.

The analysis of resource consumption is another central issue. Parallel and vectorized execution not only affects processing time but also memory usage and other system resources. Determining how to balance these factors to achieve optimal performance is a central challenge of this project.

Finally, the project addresses the comparison with traditional methods. While the basic matrix multiplication approach is well known, establishing a clear comparison between this method and the parallel and vectorized versions is essential to quantify improvements and justify the effort of implementation and optimization.

The problem is framed within the need to design solutions that are not only faster but also scalable and adaptable to different hardware configurations. As modern applications become more demanding, from simulating physical phenomena to executing complex machine learning models, efficient and optimized algorithms for matrix multiplication are a critical necessity. This project, therefore, addresses not only a technical problem but also responds to the growing demand for computational techniques that can support the scalability required in real-world applications.

In conclusion, this project focuses on identifying and overcoming the challenges associated with the implementation of parallel and vectorized techniques for matrix multiplication, maximizing performance, and minimizing the costs associated with resource consumption. Through detailed analysis and comprehensive evaluation, the project aims to provide a solid foundation for future research in high-performance computing, as well as practical tools that can be adapted to real-world, high-impact computational contexts.

3 Methodology

3.1 Development Environment and System Specifications

The project was developed using the IntelliJ IDEA integrated development environment (IDE), a robust and widely used platform for creating Java applications. This environment facilitates effective project management, enabling clear code organization, integration of external libraries, and test execution. Additionally, it offers advanced tools for performance analysis, debugging, and benchmarking—essential aspects for a comprehensive evaluation of the optimization techniques implemented in this work.

The system used for development and testing has the following specifications:

- **Processor:** 13th Gen Intel(R) Core(TM) i7-13700H, 2.40 GHz. This high-performance processor features 14 physical cores and 20 logical cores, allowing for efficient multi-core execution, making it ideal for testing the parallelization and optimization techniques implemented in the project. Furthermore, its modern architecture supports advanced vectorization operations.
- **RAM:** 16 GB. The memory capacity is suitable for handling memory-intensive operations, especially when working with large matrices in high-resource consumption scenarios.
- **Operating System:** Windows 11 Home, 64-bit edition, version 24H2. This operating system provides a stable and compatible environment for the tools and libraries used in the project, facilitating the implementation and evaluation of the optimization techniques.
- **System Architecture:** x64-based, with native support for high-performance operations and efficient execution in multi-core systems.

In addition to the hardware specifications, the development environment leveraged Java's standard toolset for concurrency, optimization, and data analysis, along with additional libraries for managing parallel tasks. These tools were crucial for ensuring an accurate evaluation of the impact of the implemented optimizations.

The chosen configuration proved to be adequate for conducting performance evaluations in matrix multiplication operations of varying sizes and complexity levels. It enabled the measurement of the impact of each optimization technique in terms of efficiency and resource usage. The development environment, combined with the hardware, provided a solid foundation for experimentation, ensuring reproducible and consistent results that reflect the behavior of the implementations in real-world scenarios.

3.2 Implementation of Optimization Techniques

The project is organized in a modular manner, enhancing code comprehension, maintainability, and extensibility. Each folder and file serves a well-defined purpose, focusing on the implementation of various optimization techniques for matrix multiplication, including basic, parallel, and vectorized methods. Below is a detailed description of the project's structure and the purpose of each component.

3.2.1 Benchmarks

The benchmarks folder plays a crucial role in the project as it groups the components responsible for executing performance tests on the various matrix multiplication implementations. These tests are not only fundamental for evaluating the optimization techniques developed but also for analyzing how these techniques behave under different experimental conditions. The primary purpose of this folder is to provide a platform to measure the impact of optimization techniques in terms of execution time, memory usage, CPU consumption, and other key resources. This enables the generation of detailed metrics that support in-depth and rigorous analysis of the obtained results.

Within this folder, the files BenchmarkRunner and BenchmarkExecutor work together to ensure the accuracy, consistency, and thoroughness of the conducted tests.

The BenchmarkRunner is responsible for executing the benchmarking process using the Java Microbenchmark Harness (JMH) library, which is specifically designed to measure performance in Java applications with high precision. During the tests, the BenchmarkRunner focuses on execution time-related metrics, collecting key data such as average time per operation (Score), margin of error (Error), measurement mode (Mode), number of iterations (Cnt), and units used (Units). These metrics provide a clear and reliable view of the performance of each implementation, ranging from the basic approach to the most advanced parallel and vectorized techniques. Additionally, the use of JMH ensures that the results are consistent by eliminating noise and variations caused by external factors such as garbage collection or operating system management.

On the other hand, the BenchmarkExecutor complements the capabilities of the BenchmarkRunner by focusing on the system resources utilized during the execution of the tests. This component processes the data obtained by the BenchmarkRunner and calculates additional metrics such as memory consumption, CPU usage percentage, efficiency, speedup, and the number of cores used during each experiment. Its modular design allows for configuring and customizing experiments through the config.properties file, which defines key parameters such as matrix sizes, the number of threads for parallel implementations, and the paths where results will be stored. The BenchmarkExecutor also ensures that the results environment is properly organized before starting the tests, creating directories and setting up the necessary structure to store the generated data.

The interaction between BenchmarkRunner and BenchmarkExecutor ensures that performance tests are not only conducted in a structured manner but are also comprehensive and representative of the actual capabilities of the implemented techniques. While the BenchmarkRunner provides a rigorous evaluation of execution times and other direct performance indicators, the BenchmarkExecutor extends the analysis to include the impact on system resources. This allows for a more holistic view of how the various optimizations affect overall system performance.

The outcome of these tests is a set of detailed data structured in formats such as CSV and JSON, facilitating subsequent analysis and visualization in graphs. This data includes critical information such as matrix sizes, the number of threads used, average execution time, memory and CPU usage, and the number of cores occupied. This organization ensures that the results are accessible and reusable, enabling efficient comparative analysis and a clear identification of behavior patterns across the different implementations.

In summary, the benchmarks folder not only constitutes the core of performance evaluation in this project but also establishes a solid and reproducible framework for conducting tests under controlled conditions. The combination of performance analysis with resource usage metrics ensures that the results are comprehensive and that the conclusions drawn from them are reliable. This highlights the importance of this folder in the context of the project, as it provides the necessary tools to validate the implemented optimizations and identify future improvement opportunities.

3.2.2 Charts

The charts folder contains the code files responsible for generating the graphs that visualize the metrics obtained during the performance tests conducted in the project. The generated graphs are essential for interpreting and analyzing the impact of the various optimization techniques implemented, as they present the metrics in a clear and visual manner, facilitating a deeper understanding of the results.

The CoresUsedChart file generates a graph that visualizes the number of cores utilized during parallel executions. This graph provides a detailed view of how implementations scale based on the number of configured threads and the matrix sizes. It allows for analyzing the ability of parallel techniques to leverage the resources available in multicore systems, evaluating their efficiency in task distribution.

The CPUUsageChart file is responsible for generating the CPU usage graph, representing the percentage of processor utilization during the tests. This graph is crucial for identifying how different techniques utilize computational resources, showing variations in CPU usage based on matrix sizes and the number of threads used. Additionally, it helps detect potential bottlenecks or system overloads during the execution of the implementations.

The EfficiencyChart file produces a graph that visualizes efficiency per thread. This metric measures the relationship between the achieved speedup and the number of threads used, highlighting how effectively parallel techniques convert additional resources (such as cores or threads) into significant performance improvements. This analysis is essential for identifying implementations that, despite being parallelized, fail to achieve effective scalability as more resources are allocated.

The ExecutionTimeChart file generates a graph displaying the average execution time for each implemented technique. This graph is fundamental for comparing the speed of the basic implementation with the optimized techniques. By representing results for different matrix sizes, it enables the analysis of how the behavior of each technique varies under different workloads, emphasizing those that offer the greatest reductions in processing time.

The MemoryUsageChart file generates a graph illustrating memory usage during the performance tests. This graph is critical for evaluating the impact of each technique on the system's memory consumption. It is particularly relevant when working with large matrices, as it helps identify implementations that may be more demanding in terms of memory and detect potential issues related to excessive resource consumption.

The SpeedupChart file generates the speedup graph, which shows the relationship between the execution time of the basic implementation and the time of the parallel and vectorized techniques. This graph is key to visualizing the performance improvements achieved through optimizations, demonstrating how each technique leverages parallelism and vectorization to significantly reduce processing time. It helps identify techniques that provide the greatest benefits in terms of speedup and their effectiveness compared to the basic method.

It is important to note that while this folder contains the code for generating the graphs, the resulting images in .png format are stored in a separate folder (output_charts), ensuring a clear separation between the source code and the generated outputs. This enhances project organization and simplifies both the execution of analyses and the direct visualization of final results. The modular and clear structure of this folder allows for the easy integration of new metrics or additional visualizations if necessary, ensuring the project's extensibility and maintainability.

3.2.3 Matrix

The matrix folder forms the core of the project and contains all the implementations developed to address the problem of matrix multiplication, organized modularly according to the applied approaches: basic, parallel, and vectorized. Each implementation focuses on optimizing specific aspects of this computationally intensive operation, allowing for the analysis and comparison of different techniques in terms of performance, scalability, and resource consumption.

The basic implementation, represented by the file BasicMatrixMultiplication, uses the standard sequential algorithm for matrix multiplication with a computational complexity of $O(n^3)$. This approach follows a classic design of three nested loops: the first iterates over the rows of the resulting matrix, the second iterates over the columns, and the third calculates the dot product between the row of the first matrix and the column of the second matrix. While this algorithm is straightforward and easy to understand, it is inefficient for large matrices, as its performance significantly degrades due to its cubic complexity. However, this implementation serves as an essential baseline for comparing the effectiveness of the optimized techniques developed in the project, providing a clear and reliable framework for evaluating improvements in terms of execution time and resource consumption.

In the domain of parallel implementations, the parallel folder groups several approaches that explore the capabilities of modern hardware to distribute calculations across multiple threads or processes. One of these implementations, ParallelMatrixExecutors, uses Java's Executor Services. This approach creates a thread pool that automatically manages task allocation, such as calculating the values for each row of the resulting matrix. By delegating thread management to a service, the complexity of the code is reduced, and resource reuse is optimized, resulting in significant performance improvements for large matrices.

Another technique within the parallel implementations is ParallelMatrixStreams, which employs parallel streams to perform the calculations. This approach allows tasks to be automatically divided among multiple threads using Java's Streams API. While parallel streams significantly simplify the code, their lack of granular control over resources can limit their effectiveness in scenarios that require more specific optimizations.

Within the subgroup of implementations that use synchronization techniques for parallelization, several strategies are designed to ensure data consistency in multithreaded environments. ParallelMatrixThreads directly uses Java threads to distribute calculations among them, offering explicit control over their creation and management. Meanwhile, ParallelMatrixSynchronized introduces synchronized blocks to protect critical regions of the code. This method ensures the integrity of shared data.

In contrast, ParallelMatrixSemaphore uses semaphores to control access to critical sections, enabling more efficient management of shared resources. This approach is particularly useful for limiting the number of threads that simultaneously access an operation, ensuring more precise control over concurrency. Another notable technique is ParallelMatrixAtomic, which uses atomic variables to perform concurrent operations without the need for explicit synchronization. These variables allow atomic operations, such as sums, to be executed directly on the data, avoiding race conditions and improving performance in certain scenarios.

In addition to parallel implementations, the project includes a vectorized implementation named VectorizedMatrixMultiplication. This approach leverages the capabilities of jdk.incubator.vector extensions to perform simultaneous data-level operations using SIMD (Single Instruction, Multiple Data) instructions. The implementation transposes the second matrix before performing calculations to optimize memory access, reducing cache misses and improving overall performance. This approach is particularly effective for large matrices, where vectorized operations can process multiple elements in parallel, maximizing the use of modern hardware.

The MatrixMultiplication file acts as a common interface for all implementations, defining the multiply method that each class must implement. This design provides a uniform structure, facilitating the integration and management of the different techniques within the project. By centralizing the definition of operations, this interface ensures consistency and interoperability among the various implementations, enabling detailed and precise comparisons.

The basic matrix multiplication implementation, with its simplicity and sequential nature, establishes an essential starting point for understanding the improvements introduced by parallel and vectorized techniques. Parallel implementations efficiently distribute calculations, addressing concurrency challenges through synchronization strategies and resource management. On the other hand, the vectorized implementation leverages hardware advancements to process data in parallel, achieving performance levels that significantly surpass traditional approaches. The combination of these techniques within a modular and well-structured framework highlights the versatility and scope of the project, offering a comprehensive view of the possibilities and limitations of the optimizations applied to matrix multiplication.

3.2.4 Utils

The utils folder contains essential utilities for the proper functioning of the project. A key file in this folder is MatrixGenerator, which is responsible for generating test matrices of different sizes with random values. This component is crucial for ensuring that the tests are consistent and reproducible. The matrices are generated using an algorithm that fills them with random numbers within a specific range, allowing for realistic scenarios to be simulated for the multiplication implementations. This file ensures that all implementations can be evaluated under the same experimental conditions, providing a consistent baseline for comparing their performance.

3.2.5 Resources

The resources folder contains the config.properties file, which defines the configuration parameters used during the project's execution. This file allows customization of key aspects such as the matrix sizes to be tested and the number of threads to be used in parallel implementations. The configuration includes default values for matrix sizes such as 64, 128, 512, 1024, and 2048, as well as options to use between 1 and 16 threads for parallel testing. This approach provides flexibility by enabling quick and precise adjustments without the need to modify the source code directly. Additionally, the file specifies the output path where the results will be stored, automatically organizing the generated data.

3.2.6 Results

The results folder contains the files where the results generated during the project's benchmarking tests are stored. The data collected during the tests is saved in two structured formats to facilitate analysis and use in various contexts.

The benchmark_results.csv file stores data in CSV format, organized into columns that include key information such as the number of threads, execution time, CPU usage, memory consumption, and the number of cores used, as well as the system's physical and logical cores. The tabular structure of the CSV format allows for quick data manipulation and facilitates comparison between the different implemented techniques.

On the other hand, the benchmark_results.json file saves the results in JSON format, using a hierarchical structure that is ideal for integration into applications or automated processing. This format is particularly valuable for generating charts and reports, as it enables programmatic and structured access to the data. Additionally, the flexibility of JSON makes it an ideal option for scenarios where the data needs to be reused in external systems or tools.

Both formats ensure that the results are easily accessible and reusable for further analysis. Whether through a visual approach, such as manually inspecting the data, or programmatic processing, the results stored in these files enable a precise and detailed evaluation of the performance of the matrix multiplication techniques implemented in the project.

3.2.7 Output_charts

The charts generated by the code in the charts folder and stored as images in the output_charts folder provide a detailed visual representation of the key metrics obtained during the performance tests. Among these charts is one that displays the number of cores used during parallel executions, enabling an analysis of the scalability of the implementations based on hardware resources. Another chart illustrates the percentage of CPU usage, allowing for an evaluation of the techniques' efficiency and the identification of potential system bottlenecks.

Additionally, a chart detailing relative efficiency per thread highlights how optimized techniques utilize additional resources to improve performance. A separate chart showcases the average execution time for each technique, making it clear which implementations achieve the greatest reductions in processing time. Another chart provides insights into memory consumption during the tests, which is crucial for identifying potential inefficiencies in resource usage.

Finally, a chart illustrates the speedup achieved by parallel and vectorized techniques compared to the basic implementation, emphasizing the performance improvements attained through optimizations. These visualizations, stored in PNG format, ensure that the results are easily accessible and can be directly used in reports, presentations, or subsequent analyses without needing to re-execute the code that generated them.

4 Experiments

In the context of this project, experiments play a fundamental role in evaluating the effectiveness of the various techniques implemented for matrix multiplication. The primary objective of the experiments is to quantitatively analyze how parallel and vectorized techniques, compared to the basic approach, impact key metrics such as execution time, CPU usage, per-thread efficiency, memory usage, and scalability. These metrics not only validate the optimizations developed but also provide valuable insights into the practical limitations and improvement opportunities of each technique.

The experiments cover a wide range of matrix sizes, from small matrices, where differences between techniques may be less pronounced, to large matrices, which impose a significant load on the system, maximizing the impact of optimizations. This approach allows for the evaluation of implementation performance across different scenarios and helps understand how each technique scales with increasing computational complexity.

Moreover, the experiments go beyond measuring execution time, focusing also on how the techniques utilize hardware resources such as processing cores and memory. This is particularly relevant in the context of modern multicore systems, where the ability of a technique to efficiently distribute tasks and leverage available hardware can significantly influence performance.

To facilitate the analysis and interpretation of the results, the data collected during the tests is visualized through charts that highlight key patterns and trends. These visualizations not only provide a more intuitive understanding of the results but also enable the quick identification of each technique's strengths and weaknesses. The results obtained and the observations derived from them are analyzed in detail below, starting with the analysis of execution time as a function of matrix size.

4.1 Execution Time vs. Matrix Size

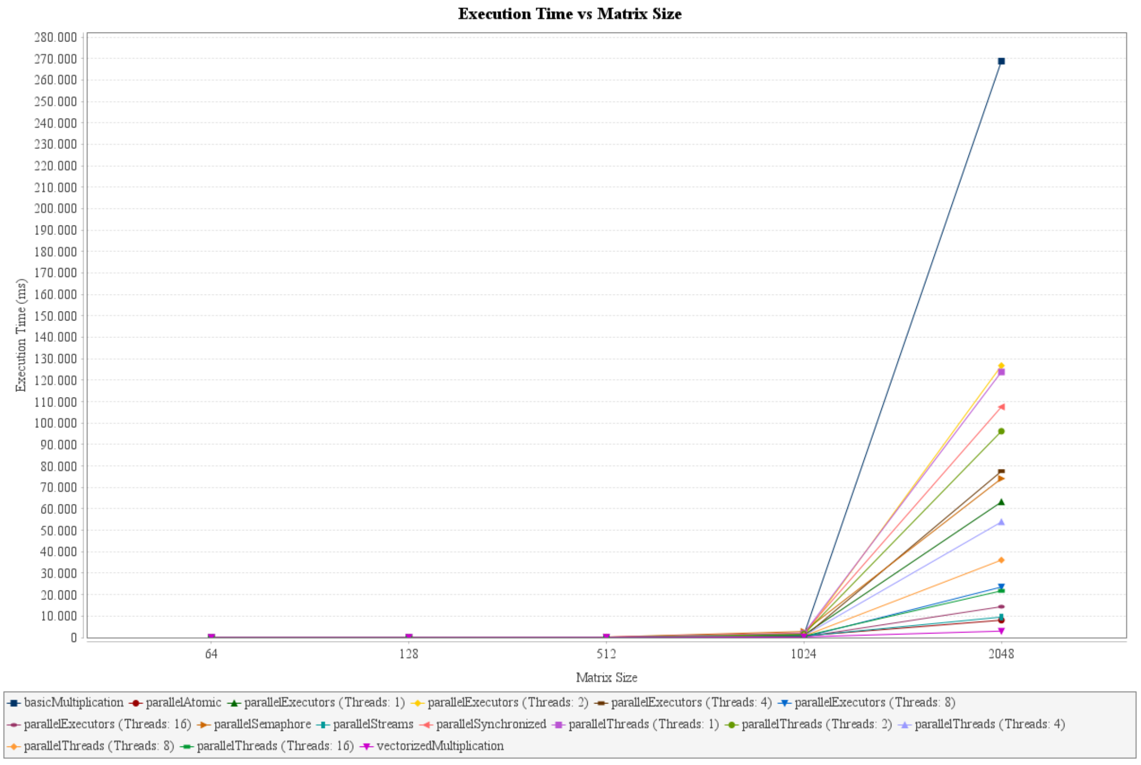

Figure 1: Execution Time vs. Matrix Size

The graph 1 provides a detailed view of the average execution time for various matrix multiplication implementations as a function of the evaluated matrix sizes. This analysis is critical for understanding how different optimized, parallel, and vectorized techniques perform when handling increasing computational loads, highlighting the strengths and limitations of each approach in practical scenarios.

On the horizontal axis, matrix sizes are displayed, ranging from 64x64 to 2048x2048, representing an exponential increase in the amount of data to be processed. The vertical axis illustrates execution time in milliseconds (ms), a key metric for evaluating performance. Each line on the graph corresponds to a specific implementation, enabling direct performance comparisons under the same experimental conditions.

The basic algorithm exhibits the worst performance among all implementations. As a sequential technique, it fails to leverage the parallel resources of modern hardware, resulting in extremely high execution times as matrix size grows. For 2048x2048 matrices, the execution time approaches 270,000 ms, clearly demonstrating this method's lack of scalability. While unsuitable for applications requiring high computational loads, its simplicity makes it an ideal baseline for comparing the performance of optimized techniques.

The Parallel Executor with 2 threads follows as the next worst-performing implementation. While it introduces some level of parallelization, the use of only two threads is insufficient to effectively manage large matrices. This results in limited improvement over the basic algorithm, particularly for intermediate and larger sizes, where additional parallelization could have a more significant impact.

The Parallel Threads with 1 thread also delivers limited performance. By utilizing a single thread for computations, its behavior closely mirrors that of the basic algorithm, with only a slight improvement due to more efficient resource usage. However, its scalability remains minimal, restricting its effectiveness in high-computational-load environments.

The Parallel Synchronized achieves marginally better performance, though the overhead introduced by synchronization mechanisms negatively impacts its execution time. While it ensures data consistency in multithreaded environments, this implementation cannot fully leverage the available resources, offering modest improvements compared to earlier implementations.

The Parallel Threads with 2 threads shows progressive improvement over the Synchronized implementation, but its performance remains constrained due to the limited number of threads employed. Although it can distribute computations across two threads, the scalability of this implementation is insufficient for handling large matrices effectively.

The Parallel Executor with 4 threads marks a notable improvement over its 2-thread counterpart. By distributing tasks across four threads, this implementation begins to better utilize the system's parallel resources, though it still falls short of the more advanced techniques in terms of overall performance.

The Parallel Semaphore balances parallelization and synchronization reasonably well. Its ability to control concurrent access to shared resources ensures a more controlled execution, but it introduces additional overhead that impacts performance for larger matrices. This positions it as an intermediate option, effective in certain scenarios but not optimal for intensive computational loads.

The Parallel Executor with 1 thread exhibits surprisingly efficient performance for a single-threaded technique. This suggests that its design effectively utilizes available resources. However, like other implementations with fewer threads, its scalability is limited, particularly for large matrices.

The Parallel Threads with 4 threads demonstrates considerable improvement over its 2-thread version. Increasing the thread count allows this implementation to handle larger matrices more effectively, though it still cannot match the performance of techniques with higher levels of parallelization or more advanced approaches.

The Parallel Threads with 8 threads emerges as one of the better-performing implementations in this category. The addition of more threads enables more efficient distribution of computations, significantly reducing execution times for larger matrices. This makes it a strong option for systems with multicore capabilities.

The Parallel Executor with 8 threads further enhances execution times compared to its 4-thread counterpart. Its ability to manage a larger number of parallel tasks positions it as one of the most competitive implementations in terms of execution time, especially for large matrices.

The Parallel Threads with 16 threads maximizes resource utilization, achieving much lower execution times than less-parallelized implementations. However, for extremely large matrices, its performance seems to stabilize, suggesting potential limitations in thread management or resource saturation.

The Parallel Executor with 16 threads surpasses the Threads with 16 threads in execution time, establishing itself as one of the most effective parallel techniques. Its ability to handle large matrices efficiently places it among the best options on this graph, though it still faces challenges related to resource saturation at extreme sizes.

The Parallel Streams delivers highly competitive performance, standing out for its simplicity and effectiveness in handling parallel tasks using the Streams API. This implementation achieves notably low execution times, especially for large matrices, where its ability to leverage the system's parallel resources becomes more apparent.

The Parallel Atomic outperforms even the Parallel Streams in terms of efficiency. Its design handles concurrent operations effectively through atomic variables, eliminating synchronization issues and improving performance in high computational load scenarios.

Finally, the Vectorized Multiplication is the most efficient implementation of all. By leveraging SIMD (Single Instruction, Multiple Data) instructions, this technique processes multiple elements simultaneously at the hardware level, achieving significantly lower execution times than any other implementation. Its exceptional performance, particularly for large matrices, establishes it as the most advanced and optimized technique in this comparison.

In conclusion, the graph clearly highlights the advantages of optimized, parallel, and vectorized techniques over the basic approach. While parallel techniques significantly reduce execution times by distributing tasks across multiple threads, the Vectorized Multiplication takes optimization to the next level by directly utilizing advanced hardware capabilities. This analysis underscores the importance of selecting the appropriate technique based on matrix size and available resources, with the Vectorized Multiplication standing out as the most efficient choice for intensive computational loads.

4.2 Memory Usage vs. Matrix Size

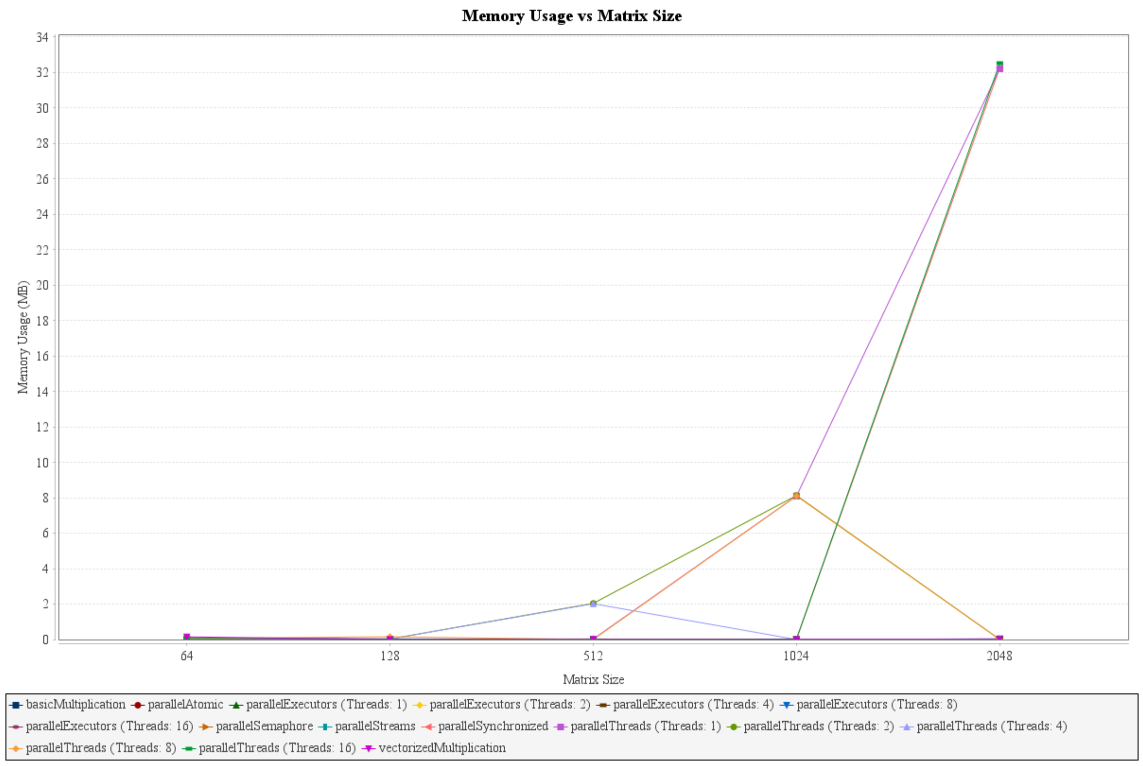

Figure 2: Memory Usage vs. Matrix Size

The graph 2 provides a comprehensive view of how different matrix multiplication implementations manage memory consumption based on the size of the matrices. This analysis is essential for evaluating the efficiency of these techniques not only in terms of execution time but also regarding their impact on hardware resources, a key aspect in real-world applications where memory can be a limited resource.

The horizontal axis of the graph presents matrix sizes, ranging from 64x64 to 2048x2048, showing exponential growth in the amount of data that needs to be processed. On the other hand, the vertical axis illustrates memory consumption in megabytes (MB), a fundamental metric for evaluating the sustainability of each technique when handling matrices of various sizes.

For small matrices, such as 64x64, memory consumption is low across all implementations. This reflects that, under these initial conditions, most techniques efficiently handle computational loads without demanding significant amounts of memory. This is particularly evident in implementations such as basicMultiplication and parallelAtomic, which demonstrate very low memory usage. These implementations are ideal for applications with small-sized matrices as they balance simplicity and resource efficiency.

As matrix size increases, such as in 512x512 matrices, more noticeable differences among implementations begin to emerge. At this stage, implementations such as parallelThreads (2 threads) with a memory consumption of 2.04 MB and parallelThreads (4 threads) with 2.02 MB stand out. These techniques, while using more memory than the most basic implementations, offer a good balance between memory usage and parallelization capacity, making them appropriate for intermediate-sized matrices. This behavior reflects their ability to scale while maintaining controlled memory consumption.

For larger matrices, such as 1024x1024, memory consumption increases, as expected for implementations designed to handle greater volumes of data. parallelThreads (2 threads) stands out with 8.13 MB, along with parallelThreads (1 thread) and parallelThreads (8 threads), both with 8.10 MB. These implementations achieve a balance between moderate resource usage and efficient computational load management, standing out for their ability to handle more complex calculations without a significant memory impact. While their consumption is higher than for smaller sizes, these techniques are functional and practical for scenarios where balancing performance and resource usage is crucial.

For larger-scale matrices, such as 2048x2048, differences in memory consumption become more pronounced. Here, three implementations stand out with the highest memory consumption: parallelThreads (16 threads) with 32.53 MB, parallelSynchronized with 32.27 MB, and parallelThreads (1 thread) with 32.21 MB. These techniques are highly effective at distributing computational loads across multi-threaded systems, allowing for faster processing of intensive tasks. However, the significant increase in memory consumption may limit their applicability in environments with hardware constraints, making them more suitable for robust systems capable of supporting higher memory demands.

On the other hand, implementations such as vectorizedMultiplication maintain low memory consumption across all matrix sizes, making them particularly attractive for environments where memory is limited, without significantly compromising performance. Similarly, basicMultiplication demonstrates notable efficiency in memory management, reinforcing its utility in applications with lower computational demands.

In conclusion, the graph demonstrates how matrix multiplication implementations vary significantly in terms of memory consumption depending on matrix size and specific design. For small matrices, such as 64x64, most techniques efficiently manage resources, with implementations like basicMultiplication and parallelAtomic standing out for their low memory consumption. These implementations are ideal for applications where hardware is limited or computational demands are minimal.

For larger-scale matrices, such as 2048x2048, memory consumption peaks in implementations like parallelThreads (16 threads), parallelSynchronized, and parallelThreads (1 thread). These techniques exhibit a clear ability to handle intensive computations and leverage the parallelism of modern hardware. However, their memory costs position them as more appropriate for robust systems where memory is not a critical constraint.

When compared to the graph 1, it is evident that implementations with high memory consumption, such as parallelThreads and parallelSynchronized, tend to achieve better execution times for large matrices due to their focus on maximizing parallelism. Conversely, implementations like vectorizedMultiplication, which maintain low memory usage, also excel in performance, making them an ideal option for systems with both time and memory constraints. This contrast highlights the importance of choosing a technique based on the necessary balance between available resources and required performance.

4.3 CPU Usage vs. Matrix Size

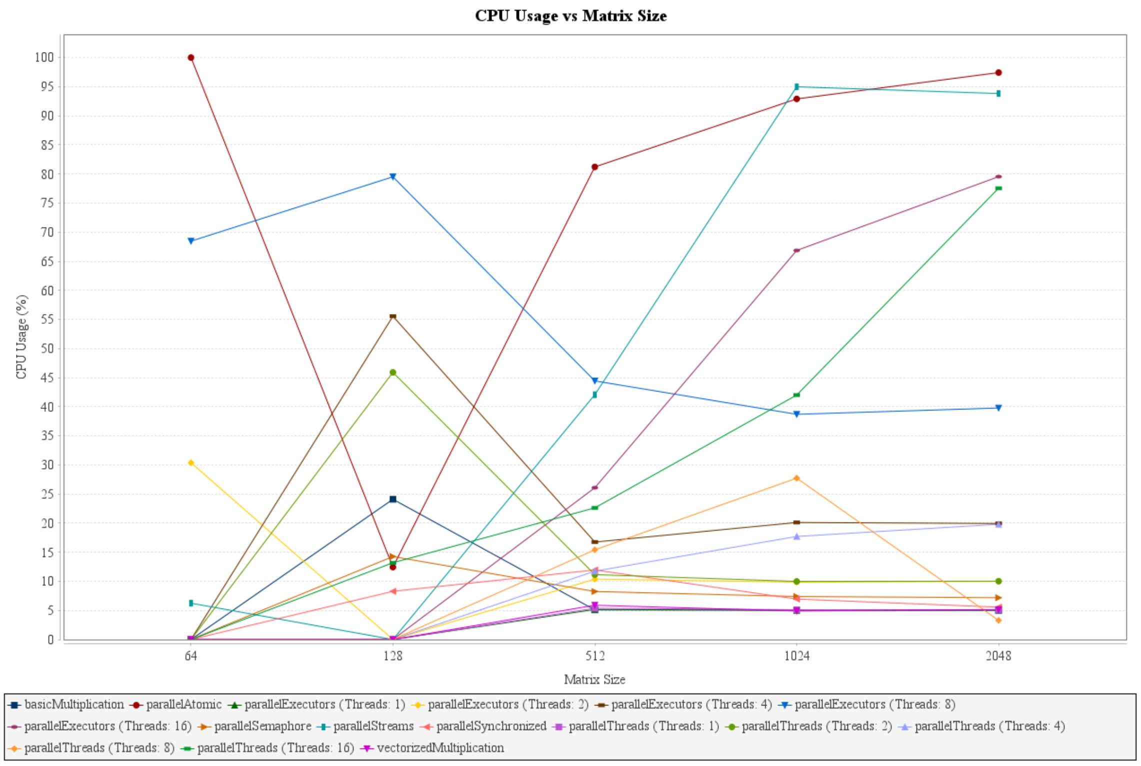

Figure 3: CPU Usage vs. Matrix Size

The graph 3 provides a clear and detailed perspective on how different matrix multiplication implementations utilize processor resources as the matrix size increases. This analysis is essential to understand how these techniques handle computational loads and scale across systems with varying hardware configurations. On the horizontal axis, matrix sizes are represented, ranging from 64x64 to 2048x2048, while the vertical axis shows the CPU usage percentage, a key indicator of processing intensity.

The most prominent behavior is observed in the parallelAtomic implementation, which consistently exhibits the highest CPU usage, reaching 100% even for smaller matrices, such as 64x64. This indicates a highly intensive processing approach, likely due to the inherent overhead of managing atomic operations in parallel contexts. While this technique maximizes hardware utilization, it also reflects a less optimized approach in scenarios where the computational load does not require such high processing intensity.

Next is parallelStreams, which remains among the implementations with the highest CPU usage for intermediate and large sizes, such as 1024x1024 and 2048x2048. Its design, based on automatic parallelization through streams, allows intensive CPU usage but also introduces some inefficiency when handling smaller matrices, where the cost of initiating parallel streams may outweigh the benefits.

The parallelExecutors implementation with 16 threads and parallelThreads with 16 threads also stand out for their high CPU usage on large matrices like 2048x2048. The former employs a model based on explicit task distribution management, while the latter relies on the creation and management of multiple threads. In both cases, the elevated CPU usage reflects their capacity to effectively divide tasks, although managing a high number of threads introduces significant overhead that may not be ideal for systems with limited resources.

Another notable implementation is parallelExecutors with 8 threads, which also maintains high consumption for large matrices but in a more moderate manner compared to its 16-thread versions. This implementation represents a good balance between resource usage and scalability, performing well on intermediate matrices such as 512x512 and 1024x1024, where it efficiently leverages available resources.

Subsequently, parallelExecutors with 4 threads and parallelThreads with 4 threads exhibit similar behavior, with CPU usage progressively increasing alongside matrix size. These implementations are effective for moderate scales, but their performance begins to plateau on large matrices due to a reduced capacity to distribute the workload compared to techniques employing a higher thread count.

ParallelThreads with 2 threads and parallelSemaphore exhibit moderate CPU usage, standing out for their ability to handle intermediate matrices with controlled consumption. Although not the fastest or most scalable, these techniques offer a reasonable balance between resource usage and computational load, especially in scenarios where available hardware is constrained.

Finally, the remaining implementations, such as vectorizedMultiplication, basicMultiplication, parallelSynchronized, and simpler versions of parallelExecutors and parallelThreads, consistently show the lowest CPU usage across all matrix sizes. These techniques are ideal for applications with light computational loads or where resource conservation is prioritized over execution speed.

Overall, the graph reveals a clear hierarchy in CPU consumption, with parallelAtomic at the top, followed by parallelStreams, the higher-thread versions of parallelExecutors and parallelThreads, and finally, the simpler and less intensive techniques. Comparing this graph with the graph 2, it becomes evident that implementations with high CPU consumption also tend to demand more memory, reflecting an intensive resource usage approach to maximize performance. On the other hand, techniques like vectorizedMultiplication achieve an exceptional balance, maintaining low CPU and memory consumption, making them highly efficient options in scenarios with hardware constraints. This analysis underscores the importance of selecting the appropriate implementation based on the available resources and the specific system requirements.

4.4 Speedup vs. Matrix Size

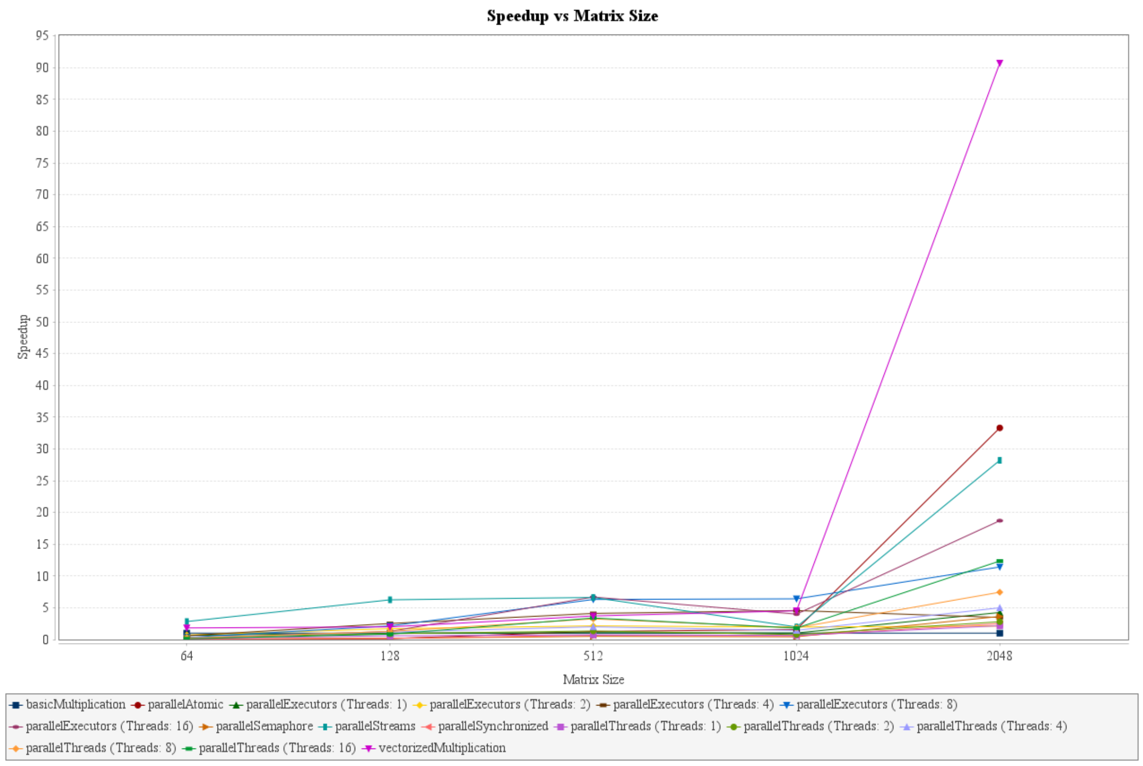

Figure 4: Speedup vs. Matrix Size

The speedup is a key metric used to evaluate the performance improvement of an optimized algorithm compared to a baseline implementation, in this case, the basic matrix multiplication algorithm. It is calculated as the ratio between the execution time of the basic algorithm and the execution time of the optimized implementation, using the following equation:

$$\text{Speedup} = \frac{\text{Execution Time of the Basic Algorithm}}{\text{Execution Time of the Optimized Algorithm}}$$

A speedup greater than 1 indicates that the optimized implementation is faster than the basic algorithm, while a speedup equal to 1 shows no improvement in execution time.

The graph 4 is essential for understanding how parallel and vectorized techniques maximize performance compared to the basic algorithm. This analysis is a critical part of the project as it quantifies the benefits of parallelization and vectorization in handling large-scale matrices. The x-axis shows the matrix sizes, ranging from 64x64 to 2048x2048, and the y-axis represents the speedup, illustrating the acceleration achieved by each implementation.

The basic algorithm, as a reference, is shown with a constant speedup of 1. This reinforces its role as a starting point for evaluating optimizations but also highlights its inability to scale with large matrix sizes due to its sequential approach. This limitation underscores the need for advanced techniques that better utilize hardware resources.

The implementation that stands out the most is vectorizedMultiplication, achieving an exceptional speedup, especially for 2048x2048 matrices. This is due to its use of SIMD (Single

Instruction, Multiple Data) instructions, which process multiple elements simultaneously at the hardware level. Additionally, compared to the CPU and memory usage graphs, this technique maintains low resource consumption, making it a highly efficient and scalable solution for high computational loads.

ParallelAtomic ranks second in speedup, excelling particularly with large matrices. It uses atomic variables to handle concurrent operations efficiently, eliminating synchronization issues. However, its efficiency decreases for smaller matrices due to the overhead associated with managing these operations.

ParallelStreams provides a notable balance between ease of implementation and performance. Its ability to handle intermediate and large matrices makes it a solid option, although the overhead of initializing and managing parallel streams may limit its efficiency for smaller matrices. Nevertheless, this technique is ideal for environments with hardware optimized for concurrent operations.

Parallel implementations with a large number of threads, such as parallelExecutors and parallelThreads with 16 threads, achieve significant speedups for large matrices by effectively dividing the computational load. However, the introduction of a high number of threads can generate overhead, limiting their efficiency for small matrices and systems with constrained hardware.

Implementations like parallelExecutors with 8 threads and parallelExecutors with 2 threads deliver more moderate but consistent speedups, proving useful in systems with lower multicore capacity or applications with intermediate computational requirements. These techniques offer good scalability without excessive resource consumption.

Finally, implementations such as parallelSemaphore, parallelSynchronized, and the lower-thread versions of parallelExecutors and parallelThreads show the lowest speedups. While less effective for large matrices, these techniques may be suitable for applications where simplicity and stability are more important than maximum performance.

In conclusion, this graph highlights how vectorized and parallel techniques can maximize performance by significantly reducing execution times. vectorizedMultiplication stands out as the most efficient technique, offering high speedup with minimal CPU and memory consumption, making it the ideal solution for handling large matrices in systems with hardware constraints. Compared to the CPU and memory usage graphs, techniques with high speedup tend to consume more resources, except for vectorizedMultiplication, which achieves an exceptional balance between performance and efficiency. This analysis emphasizes the importance of selecting the right technique based on system needs and available resources, prioritizing implementations that optimize both performance and resource utilization.

4.5 Efficiency vs. Matrix Size

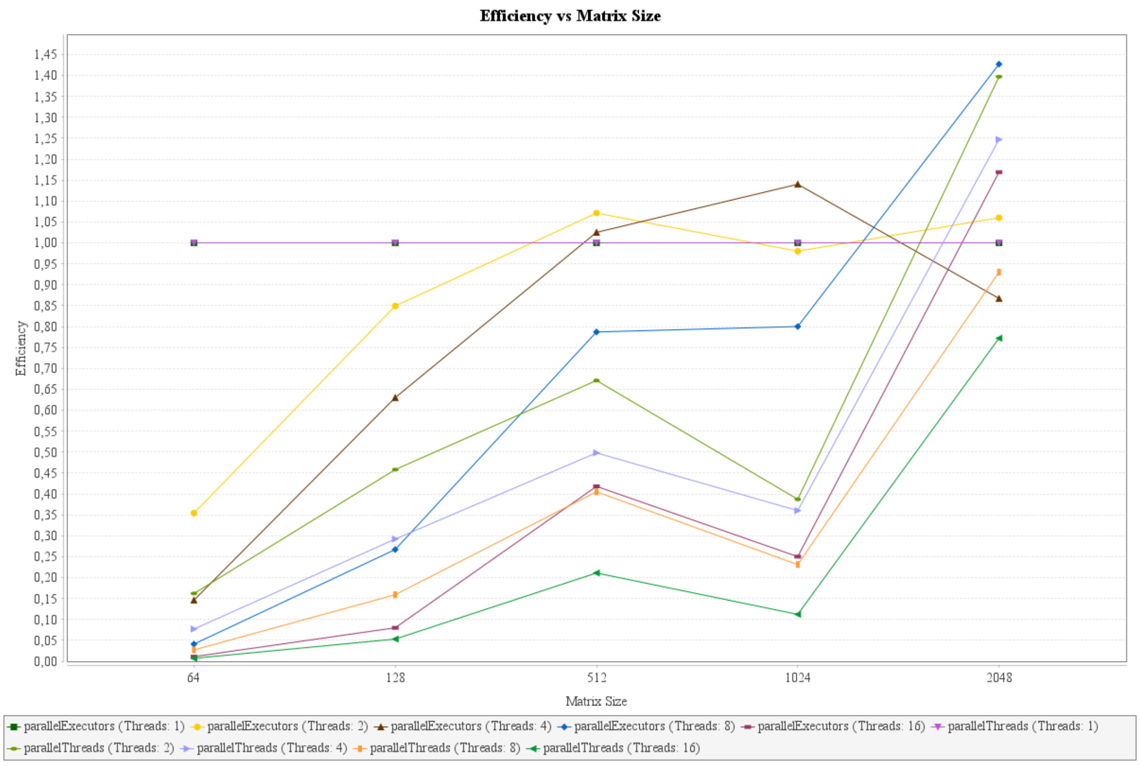

Figure 5: Efficiency vs. Matrix Size

Parallel execution efficiency is a fundamental metric for evaluating how the increase in the number of threads affects the performance of parallel matrix multiplication implementations. It is defined as the ratio between the achieved speedup and the number of threads used, allowing us to measure how effectively the additional resources provided by parallelization are being utilized. Its calculation can be expressed with the following equation:

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

An efficiency value close to 1 indicates that the threads are being used optimally, while values below 1 reflect a loss of performance due to factors such as management overhead or inefficiencies in task division.

The graph 5 specifically analyzes parallel implementations based on Executors and Threads, which allow adjusting the number of threads as a parameter, excluding the basic, vectorized, and other parallel implementations such as ParallelMatrixStreams, ParallelMatrixAtomic, ParallelMatrixSemaphore, and ParallelMatrixSynchronized. The main reason for this exclusion is that these implementations do not allow the number of threads to vary, always operating with a single thread or under a fixed resource management model. As a result, their efficiency would always be equal to 1, since the speedup achieved would be proportional to the number of threads, which does not provide relevant or differentiating information in this context.

On the horizontal axis of the graph, the matrix size is represented, including values such as 64, 128, 512, 1024, and 2048, while the vertical axis shows the efficiency values, expressed as a ratio ranging from 0 to 1.5. This approach is appropriate since these implementations directly depend on the number of threads, and analyzing efficiency is particularly important to understand their behavior and the impact of scalability.

The implementation with the highest efficiency is parallelExecutors with 8 threads, standing out particularly in large matrices such as 2048x2048. This demonstrates that the balance between the number of threads and the workload assigned to each allows for effective use of the available resources. This result also reflects the ability of this implementation to distribute the computational load without generating significant overhead.

In second place is parallelThreads with 2 threads, which shows remarkable efficiency in intermediate and large matrices. This behavior suggests that with a small number of threads, the task division is less costly, and parallel operations are performed more effectively. However, in smaller matrices, its performance is more limited because the cost of starting the threads does not compensate for the benefits.

ParallelThreads with 4 threads follows on the list, showing consistent efficiency in medium-sized matrices. This result evidences that a moderate increase in the number of threads improves the distribution of the computational load without significant efficiency loss.

ParallelExecutors with 16 threads is another implementation that stands out, particularly in large matrices. Although its efficiency is lower than in configurations with fewer threads, it still manages to leverage resources in multicore systems. However, managing a higher number of threads introduces overhead, reducing efficiency in smaller matrices.

ParallelExecutors with 2 threads shows inferior performance, ranking below parallelExecutors with 16 threads. Although it achieves some improvement in small and intermediate matrices, the reduced number of threads limits its ability to scale effectively for larger matrices.

The implementations parallelThreads with 1 thread and parallelExecutors with 1 thread maintain efficiencies equal to 1. This is expected, as there is no additional workload division among threads; the calculation is performed sequentially, reflecting full but limited utilization of the available resources.

On the other hand, implementations such as parallelThreads with 8 threads and parallelExecutors with 4 threads have lower efficiencies. This is because, in these configurations, the number of threads becomes too high in relation to the computational load, especially in small and intermediate matrices, leading to increased overhead without a proportional benefit in speedup.

Finally, parallelThreads with 16 threads shows the lowest efficiency overall, especially in small matrices. This occurs because the high number of threads introduces significant overhead, which greatly reduces its effectiveness. However, in large matrices, this implementation partially recovers, indicating that the size of the computational load is a key factor for its performance.

In conclusion, this graph shows that efficiency in parallel implementations is directly influenced by the balance between the number of threads and the computational load. Configurations such as parallelExecutors with 8 threads and parallelThreads with 2 or 4 threads stand out for their ability to maintain high efficiency across a wide range of matrix sizes. Comparing with graphs 4, it is observed that configurations with high speedup, such as those with 16 threads, may not be the most efficient due to the associated overhead. On the other hand, implementations such as parallelExecutors with 8 threads achieve an excellent balance between performance and efficiency, positioning themselves as the most suitable option for systems with hardware constraints or controlled scalability needs.

4.6 Cores Used vs. Threads Used

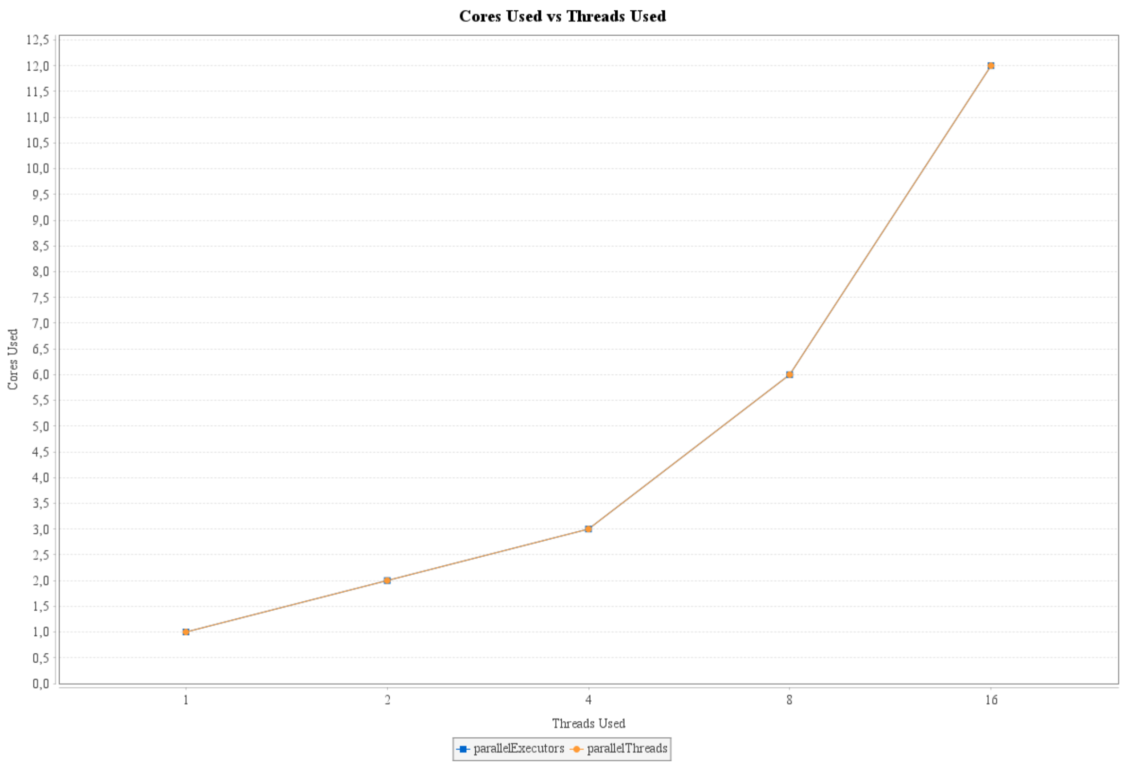

Figure 6: Cores Used vs. Threads Used

The graph 6 provides a clear representation of how parallel implementations (parallelExecutors and parallelThreads) distribute and utilize system cores based on the number of threads assigned. This analysis is crucial for evaluating how these techniques leverage the available hardware resources in the context of matrix multiplication, aligning with the project's requirements, which include measuring resource usage such as CPU cores.

On the horizontal axis of the graph, the threads used are represented, with values of 1, 2, 4, 8, and 16. On the vertical axis, the number of cores utilized is shown, which varies proportionally with the threads. This relationship allows for observing the degree of parallelism achieved by the implementations and how they distribute the computational load.

These implementations were selected because they allow direct adjustment of the number of threads as a parameter, unlike other parallel implementations such as ParallelMatrixStreams, ParallelMatrixAtomic, ParallelMatrixSemaphore, and ParallelMatrixSynchronized, which operate with a single thread. These latter implementations are therefore not relevant to this analysis, as their core usage remains constant and does not provide information about the relationship between threads and cores.

Both implementations, parallelExecutors and parallelThreads, exhibit identical behavior in terms of core usage, with an ascending line proportional to the number of threads configured. This suggests that both techniques efficiently utilize the available hardware, scaling the number of cores used according to the number of threads assigned.

With 1 thread, only one core is used, which is consistent with a sequential execution model where no parallelism is present. This behavior is predictable, as the load is not distributed across multiple cores. With 2 threads, the graph shows the use of 2 cores, indicating that each thread is assigned to a separate physical core, achieving a moderate increase in the level of parallelism and improving performance for small and intermediate-sized matrices. When increasing to 4 threads, the core usage rises to 3, demonstrating a more effective distribution of the computational load. This result reflects a significant improvement in performance for intermediate-sized matrices, although not all available system resources are fully utilized.

With 8 threads, 6 cores are used, representing considerable scalability and more complete utilization of the multicore system. This level of parallelism results in an adequate distribution of the computational load, especially in large matrices, where performance improves noticeably. Finally, when increasing to 16 threads, 12 cores are used, approaching the physical resource limit of the system, which has 14 physical cores and 20 logical cores. In this configuration, while a high level of parallelism is achieved, there is also an increase in the overhead associated with thread synchronization and management, which can limit performance compared to more balanced configurations.

The results show that both parallelExecutors and parallelThreads maintain similar behavior regarding core allocation based on the number of threads. This demonstrates that both techniques can scale efficiently, utilizing the available resources up to configurations with 16 threads. However, the use of 12 cores for 16 threads indicates that although a high degree of parallelism is achieved, the system does not fully distribute the threads across unique cores, likely due to resource contention or the nature of the tasks being executed.

Comparing this graph with those of efficiency and speedup, it is evident that increasing the number of threads and cores does not always result in a proportional increase in performance. Configurations with 16 threads, while achieving high core usage, often show a decrease in efficiency due to synchronization overhead. On the other hand, configurations with 8 threads seem to offer an optimal balance, effectively utilizing 6 cores without incurring significant management costs. In conclusion, this graph highlights the importance of adjusting the number of threads based on the hardware characteristics and the specific requirements of the task, with intermediate configurations, such as 8 threads, being the most suitable for achieving an optimal balance between resource usage and performance in multicore systems.

5 Conclusion

In this project, parallel and vectorized computing techniques were analyzed and applied to matrix multiplication with the aim of evaluating their ability to leverage modern hardware resources and overcome the limitations of traditional sequential approaches. The results obtained and the analysis conducted allowed us to draw detailed and robust conclusions, providing a clear understanding of the advantages, disadvantages, and optimal applications of each implemented technique.

The implementation of the basic algorithm served as a benchmark to evaluate the performance of the optimized techniques. While its simplicity makes it a reliable standard for comparison, its inability to scale and utilize resources such as multiple cores or threads significantly limits its effectiveness. This algorithm exhibited extremely high execution times, especially for large matrices, reinforcing the need for more advanced approaches.

Parallel implementations, based on explicit threads (parallelThreads) and executor services (parallelExecutors), proved to be a significant improvement over the basic approach. These techniques enable the computational load to be divided among multiple threads, effectively reducing execution time. However, it was observed that the performance of these techniques does not always scale linearly with the number of threads. For instance, configurations with a very high number of threads, such as 16, introduced synchronization and management overhead that reduced efficiency for small and intermediate-sized matrices. On the other hand, configurations with 8 threads demonstrated an optimal balance between efficiency, resource consumption, and execution time, emerging as the most balanced option for multicore systems.

Efficiency, defined as the ratio between speedup and the number of threads used, highlighted how increasing parallelism can lead to performance losses due to management overhead and inefficient task distribution. Configurations such as parallelExecutors with 8 threads and parallelThreads with 2 or 4 threads maintained high efficiency levels, making them ideal options for applications requiring a combination of speed and efficient resource utilization. In contrast, implementations with 16 threads, while achieving high speedup, showed a significant drop in efficiency due to additional synchronization costs.

In terms of CPU and memory consumption, highly parallel implementations like parallelAtomic and parallelStreams were observed to intensively utilize these resources to maximize performance. While this can be advantageous in robust systems with high-capacity hardware, it may limit their applicability in resource-constrained environments. In contrast, the vectorized technique (vectorizedMultiplication) stood out as an exceptional solution, maintaining low CPU and memory consumption while achieving the highest speedup among all evaluated techniques. This is attributed to the use of SIMD (Single Instruction, Multiple Data) instructions, which allow multiple elements to be processed simultaneously at the hardware level.

The analysis of core usage based on the configured threads showed that parallel implementations scaled well up to configurations of 16 threads, utilizing up to 12 cores in systems with 14 physical cores and 20 logical cores. However, increasing the number of threads did not always translate into optimal resource usage, especially when the computational load was insufficient to justify the overhead associated with managing multiple threads. This behavior underscores the importance of selecting the number of threads based on the specific characteristics of the hardware and the size of the computational load.

Comparing the different metrics, it can be concluded that no single technique is ideal for all situations. Parallel implementations with a moderate number of threads (such as parallelExecutors with 8 threads) are suitable for applications requiring a good balance between speed and efficiency. On the other hand, the vectorized technique stands out as the most efficient option for systems aiming to maximize performance while minimizing resource consumption. This versatility makes it the preferred choice for environments with high computational loads and hardware constraints.

In conclusion, this project has demonstrated how parallel and vectorized computing techniques can significantly transform the performance of matrix multiplication. From the basic implementation to the most advanced optimizations, each technique evaluated exhibits strengths and limitations that must be carefully considered when selecting the most appropriate one for a specific system or application. Intermediate configurations, such as parallelExecutors with 8 threads, and the vectorized implementation stand out as optimal solutions for balancing performance, efficiency, and resource consumption. This analysis provides a solid foundation for future research and applications in high-performance computing scenarios.

6 Future Work

Future work in the field of matrix multiplication should focus on exploring distributed approaches to address the inherent limitations of basic, optimized, and parallel implementations. The primary motivation behind this effort is the need to handle matrices of such large dimensions that they exceed the memory capacity of a single system. As applications in Big Data and machine learning increasingly demand real-time or near-real-time data processing, scalability and efficiency become essential.

A key direction would be to implement matrix multiplication using distributed computing frameworks such as Apache Spark or MPI (Message Passing Interface). These frameworks enable matrices to be divided into blocks and distributed across multiple nodes in a cluster, with each node independently processing a portion of the matrix. This approach not only allows for handling extremely large matrices but also significantly improves system scalability by adding more nodes to the cluster as needed. However, this approach introduces new challenges, such as managing communication between nodes and data synchronization, which must be carefully addressed to avoid bottlenecks.

Another fundamental aspect would be to analyze the impact of data transfer times between nodes on the overall performance of the system. In a distributed environment, data must be moved between different machines over a network, which can introduce significant overhead. Future research should focus on minimizing this transfer time by optimizing data partitioning algorithms and compressing information before transmission. Additionally, the use of advanced load-balancing techniques could ensure that cluster nodes are evenly utilized, preventing some nodes from becoming bottlenecks while others remain underutilized.

In terms of metrics, it would be essential to evaluate how system performance changes as the size of the matrices increases. This would include measuring total execution time, memory utilization per node, CPU usage, and overall system efficiency. It would also be valuable to analyze how the cluster's hardware architecture affects the results. For instance, clusters with high-speed networks and nodes equipped with modern processors that support vectorization could deliver significantly better performance.

Furthermore, integrating additional optimization techniques, such as vectorization in distributed environments, represents an exciting opportunity for future work. While vectorized approaches have proven effective in local environments, their implementation in distributed systems might require significant adaptations to manage data partitioning and synchronization across nodes.

In summary, the focus on distributed matrix multiplication promises to overcome the scalability limits imposed by current methods, providing robust solutions for applications handling massive datasets. However, achieving this goal requires a combination of advanced techniques, efficient resource management, and a deep understanding of the inherent challenges of distributed computing. This future work will not only strengthen the theoretical and practical foundation of matrix multiplication but also lay the groundwork for tackling broader problems in the realm of high-performance computing.

GitHub Link

All the code implemented for this project can be found on my GitHub repository at the following link: GitHub Repository.