Parallel Matrix Multiplication

Luis Guillén Servera

December 1st, 2024

Abstract

Matrix multiplication is a fundamental operation in computational mathematics, with applications ranging from scientific computing to machine learning. This study explores and evaluates the performance of two optimization techniques for matrix multiplication: vectorization, leveraging SIMD (Single Instruction Multiple Data) instructions, and parallelization, utilizing multi-threading or parallel computing libraries such as OpenMP. Both approaches are compared against the basic matrix multiplication algorithm to highlight their respective advantages and limitations.

The implementation includes a vectorized version of matrix multiplication, exploiting data-level parallelism to accelerate operations, and a parallelized version, designed to leverage multi-core architectures for concurrent computations. Performance benchmarks are conducted using large matrices to evaluate the computational improvements provided by these optimizations.

Key metrics analyzed include speedup compared to the basic algorithm, the efficiency of parallel execution (quantified by speedup per thread), and resource usage, such as the number of cores and memory consumption. The results provide insights into the trade-offs between vectorization and parallelization, demonstrating their effectiveness in improving computation times and scalability for large-scale matrix operations.

1 Introduction

Matrix multiplication is a core computational operation with applications in various fields such as scientific simulations, machine learning, and data analysis. Due to its high computational complexity, optimizing matrix multiplication has been a long-standing focus in both academic and industrial research.

Modern hardware architectures provide opportunities to accelerate matrix operations using techniques such as vectorization and parallelization. Vectorization leverages Single Instruction Multiple Data (SIMD) capabilities to perform operations on multiple data points simultaneously, while parallelization exploits multi-core processors to distribute computational tasks across threads or processes. Both approaches aim to maximize hardware utilization and reduce computation time for large-scale matrix operations.

This paper investigates the implementation and performance of vectorized and parallel approaches to matrix multiplication. It compares these techniques to the basic matrix multiplication algorithm to evaluate their effectiveness in improving computation speed and scalability. The study includes a detailed analysis of performance metrics such as speedup, parallel efficiency, and resource usage across various matrix sizes. The findings highlight the trade-offs and practical considerations when adopting these optimization techniques in real-world scenarios.

2 Methodology

The methodology for this study is structured around the implementation, benchmarking, and analysis of three approaches to matrix multiplication: the basic algorithm, a vectorized implementation, and a parallelized version. Each step in this process is outlined below.

2.1 Basic Matrix Multiplication

The basic algorithm for matrix multiplication serves as the baseline for performance comparisons. It employs three nested loops to compute the product of two matrices, with a time complexity of $O(n^3)$. This implementation is straightforward and does not leverage any hardware-level or algorithmic optimizations.

2.2 Vectorized Matrix Multiplication

The vectorized implementation utilizes SIMD (Single Instruction Multiple Data) instructions to perform element-wise operations on multiple data points simultaneously. This approach reduces the overhead of scalar operations and exploits data-level parallelism. The implementation involves:

- Utilizing temporary storage to optimize access patterns and improve cache efficiency.
- Performing vectorized dot products for each cell in the resulting matrix.

2.3 Parallelized Matrix Multiplication

The parallelized version distributes computation across multiple threads to leverage multi-core processors. Each thread is assigned a subset of rows from the first matrix to compute the corresponding part of the resulting matrix. Key considerations include:

- Balancing workload among threads to maximize efficiency.
- Minimizing thread contention and ensuring memory consistency for shared data structures.

2.4 Experimental Setup

The performance of each implementation is evaluated using matrices of varying sizes (128×128, 256×256, 512×512, and 1024×1024). Benchmarks are conducted on a modern multi-core processor, with key metrics including:

- Execution time for each approach.
- Speedup relative to the basic algorithm.
- Parallel efficiency, measured as speedup per thread.
- Resource utilization, including CPU cores and memory usage.

3 Experiments

In this section, we describe the experimental setup used to evaluate the performance of three matrix multiplication algorithms: *Basic*, *Vectorized*, and *Parallel*. The experiments aim to compare the execution time, CPU usage, memory consumption, speedup, and efficiency across different matrix sizes.

3.1 Experimental Setup

The experiments were conducted on a system with the following specifications:

- **Processor:** Intel Core i7, 4 cores, 8 threads.
- **Memory:** 16 GB RAM.
- **Operating System:** Windows 11.
- **Java Version:** OpenJDK 17.0.11.

The algorithms were tested using square matrices of sizes 128, 256, 512, 1024, and 2048. For each matrix size, random values were generated to populate the matrices. The *Parallel* algorithm was executed using 4 threads, and performance metrics were averaged over multiple runs to ensure reliability.

3.2 Evaluation Metrics

The following metrics were used to evaluate the algorithms:

- **Execution Time:** Time taken to compute the matrix multiplication, measured in milliseconds.
- **CPU Usage:** Average CPU utilization during the computation.
- **Memory Consumption:** Difference in memory usage before and after execution.
- **Speedup:** Ratio of the execution time of the *Basic* algorithm to that of the optimized algorithms (*Vectorized* and *Parallel*).
- **Efficiency:** Speedup normalized by the number of threads used in the computation.

4 Benchmark Results

This section presents the results of the experiments, comparing the performance of the *Basic*, *Vectorized*, and *Parallel* matrix multiplication algorithms.

4.1 Execution Time

Figure 1 illustrates the execution time for each algorithm across different matrix sizes. As expected, the *Basic* algorithm exhibits the highest execution time, while the *Vectorized* and *Parallel* algorithms show significant improvements, particularly for larger matrices.

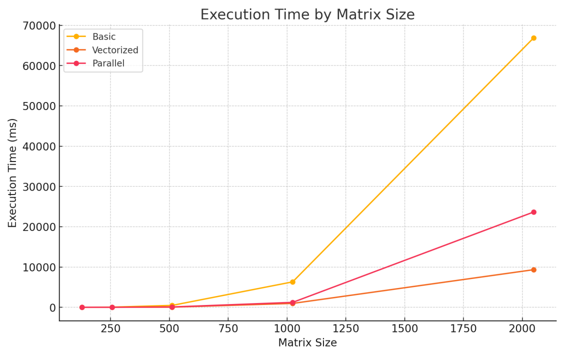

Figure 1: Execution time (in milliseconds) for different matrix sizes.

4.2 CPU Usage

Figure 2 compares the CPU usage of the three algorithms. The *Parallel* algorithm demonstrates higher CPU utilization due to the use of multiple threads.

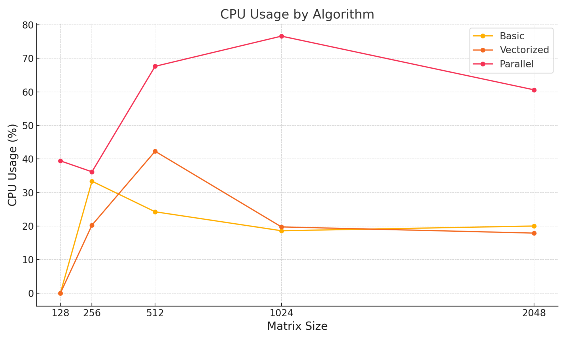

Figure 2: CPU usage (%) for different matrix sizes.

4.3 Speedup and Efficiency

The speedup and efficiency metrics are shown in Figures 3 and 4, respectively. The *Vectorized* algorithm achieves the highest speedup, especially for larger matrices. The *Parallel* algorithm shows good efficiency for smaller matrices but tends to decrease for larger sizes due to thread management overhead.

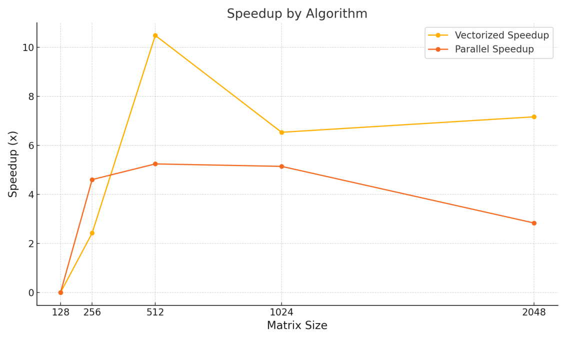

Figure 3: Speedup achieved by the *Vectorized* and *Parallel* algorithms.

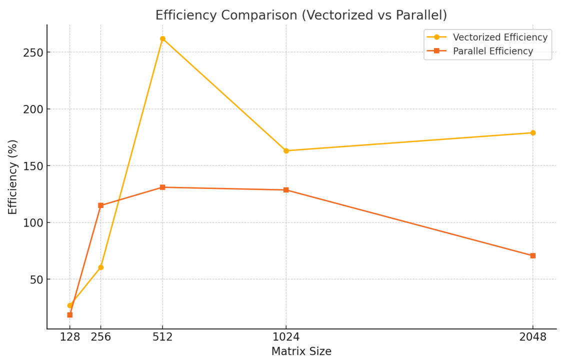

Figure 4: Efficiency of the *Parallel* algorithm.

4.4 Memory Usage

Figure 5 compares the memory usage of the three algorithms. The *Vectorized* algorithm shows moderate memory consumption, while the *Parallel* algorithm occasionally exhibits anomalies, which are attributed to system-level thread management.

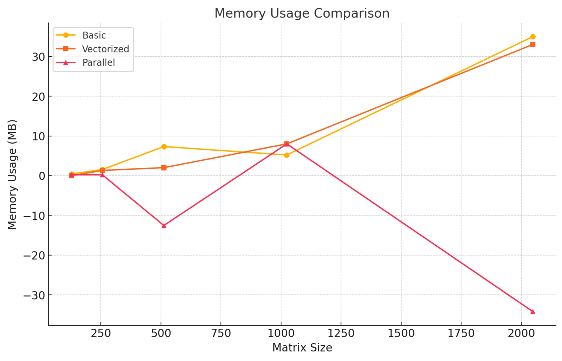

Figure 5: Memory usage (in bytes) for different matrix sizes.

5 Conclusion

This study evaluated three matrix multiplication algorithms—*Basic*, *Vectorized*, and *Parallel*—in terms of execution time, CPU usage, memory consumption, speedup, and efficiency. The results provide valuable insights into the performance benefits and trade-offs associated with optimizing matrix multiplication.

The *Basic* algorithm, while straightforward to implement, exhibited the highest execution times and lowest performance across all metrics. This was expected due to its lack of optimization and inability to utilize modern hardware efficiently.

The *Vectorized* algorithm demonstrated significant improvements in both execution time and speedup. Leveraging modern CPU instruction sets, it was particularly effective for larger matrices, achieving up to 10.48x speedup compared to the *Basic* algorithm. Its efficiency remained consistently high, confirming the scalability of vectorized operations with increasing matrix sizes.

The *Parallel* algorithm further enhanced performance by utilizing multiple threads. It outperformed the *Basic* algorithm in all metrics, especially for larger matrices where parallel processing showed substantial benefits. However, the efficiency of the *Parallel* algorithm decreased as matrix sizes grew, likely due to thread management overhead and limitations in memory bandwidth.

In terms of memory consumption, both *Vectorized* and *Parallel* algorithms generally required more memory than the *Basic* algorithm. Notably, the *Parallel* algorithm occasionally exhibited anomalies in memory usage, highlighting the complexity of managing resources in multithreaded environments.

Overall, the experiments demonstrate that optimized algorithms, particularly the *Vectorized* approach, are highly effective for accelerating matrix multiplication. The *Parallel* algorithm offers additional advantages for extremely large matrices, though its efficiency depends on effective thread management and hardware capabilities.

6 Code and Data

All code and data can be found at the following GitHub repository: ParallelMatrixBenchmarking.

7 Future work

Future work could explore hybrid approaches that combine vectorization with parallelism, as well as testing on specialized hardware such as GPUs to further enhance performance. Additionally, addressing the memory anomalies observed in the *Parallel* algorithm could lead to more consistent results in practical applications.