COMPARATIVE STUDY OF VECTORIZED AND PARALLEL MATRIX MULTIPLICATION IN JAVA

Raquel Almeida Quesada

Universidad de Las Palmas de Gran Canaria

Grado en Ciencia e Ingeniería de Datos, ULPGC

<raquel.almeida105@alu.ulpgc.es>

GitHub Repository

ABSTRACT

Matrix multiplication is a fundamental operation in scientific computing, machine learning, and numerous applications in engineering. Optimizing this operation is critical for achieving better performance in computationally intensive tasks. This paper presents a comparative study of vectorized and parallel approaches to matrix multiplication implemented in Java. Using vectorization with SIMD instructions and parallel computation via multithreading and OpenMP-like libraries, we assess the performance improvements over the basic matrix multiplication algorithm. Experiments conducted on large matrices reveal substantial speedup and resource efficiency, demonstrating the potential of these optimization techniques. The study offers insights into the trade-offs and benefits of each approach, providing a roadmap for developers aiming to optimize matrix operations in Java-based systems.

1. INTRODUCTION

Matrix multiplication is a cornerstone of computational mathematics, widely used in fields ranging from physics simulations to neural network training. However, the naive implementation of this operation often becomes a bottleneck in performance-critical applications due to its high computational complexity.

Advancements in modern computing architectures, such as Single Instruction Multiple Data (SIMD) capabilities and multicore processors, provide opportunities to accelerate matrix operations. Vectorization leverages SIMD instructions to perform multiple arithmetic operations simultaneously, while parallelization divides the workload among multiple processor cores to improve execution efficiency.

This study focuses on optimizing matrix multiplication in Java, a platform-independent programming language often criticized for its performance in computational tasks. By implementing both vectorized and parallel matrix multiplication, we aim to explore the effectiveness of these optimization techniques. Our goal is to quantify the performance improvements, analyze resource utilization, and identify the scenarios where each approach excels.

2. PROBLEM STATEMENT

Matrix multiplication, despite being a foundational operation in computational mathematics and numerous applications, is inherently computationally expensive with a time complexity of $O(n^3)$ for standard algorithms. This high cost poses significant challenges for performance in domains requiring repeated and large-scale matrix computations, such as machine learning, scientific simulations, and big data analytics.

While modern hardware provides capabilities such as SIMD (Single Instruction Multiple Data) instructions and multicore processors to mitigate computational bottlenecks, leveraging these features requires careful algorithm design and implementation. Additionally, Java, known for its platform independence and widespread usage, is often perceived as suboptimal for high-performance computing tasks due to its abstraction overhead and lack of low-level control.

The problem this study addresses is twofold:

1 How can vectorization and parallelization tech-

niques be effectively applied to matrix multiplication

2 What are the trade-offs, resource utilization patterns, and scalability implications of these techniques compared to the basic implementation?

This research aims to evaluate these questions through a systematic exploration of vectorized and parallelized approaches, providing insights into their practical impact and guiding future optimization efforts in Java-based computational systems.

3. METHODOLOGY

To address the issue of computational inefficiency in matrix multiplication and evaluate the effectiveness of optimization techniques, we implemented and analyzed three different approaches: the basic algorithm, a vectorized version leveraging SIMD instructions, and a parallel implementation utilizing multithreading. This section describes the implementation of these algorithms, the metrics used for performance evaluation, the experimental setup, and the execution process. By systematically exploring these methods, the study aims to provide actionable insights into optimizing matrix multiplication in Java.

3.1. Implementation of Algorithms

This study implemented three distinct matrix multiplication algorithms in Java:

- **Basic Matrix Multiplication:** The naive approach to matrix multiplication follows the standard triple-loop algorithm with a time complexity of $O(n^3)$. This implementation serves as the baseline for performance comparison. Each element of the resulting matrix is computed independently by iterating over the rows of the first matrix and the columns of the second matrix.
- **Vectorized Matrix Multiplication:** This approach utilizes Java's parallel streams to achieve data-level parallelism, processing individual rows of the resulting matrix concurrently. By leveraging the `IntStream.parallel()` method, the computation of each row is distributed across available threads, allowing simultaneous operations on multiple rows. This implementation efficiently exploits modern multicore processors, enhancing performance for large-scale matrix multiplications.
- **Parallel Matrix Multiplication:** This implementation uses a block-based parallel computation strategy. The matrices are divided into smaller sub-blocks, and computations for each block are distributed across multiple threads using Java's `ExecutorService`.

Each algorithm was designed to handle matrices of varying sizes, demonstrating performance at small and large scales.

3.2. Performance Measurements

The following metrics were recorded for each algorithm:

- **Execution Time:** Measured in milliseconds to assess the time taken to complete the matrix multiplication.
- **CPU Usage:** Collected before and after execution to determine the CPU utilization percentage.
- **Memory Usage:** Calculated as the difference between total and free memory during execution.
- **Speedup:** Ratio of execution time for the basic algorithm to the optimized ones.
- **Efficiency:** The speedup divided by the number of threads available.

Results were logged in a CSV file for further analysis, ensuring data consistency using `Locale.US` for formatting.

4. EXPERIMENTS

The experiments aim to evaluate the performance of three matrix multiplication algorithms—Basic, Vectorized, and Parallel—by systematically comparing their execution times, resource utilization, and scalability across matrices of varying sizes. The purpose is to quantify the benefits of optimization techniques such as vectorization and parallelization and identify trade-offs between computational speed and resource efficiency.

The results are obtained using controlled experiments on a system with a multicore CPU, ensuring that the hardware capabilities are fully leveraged for parallel computation. Each algorithm was tested on square matrices of dimensions ranging

from 50×50 to 4096×4096, providing a comprehensive understanding of their behavior under different workloads.

4.1. Basic Matrix Multiplication

The basic algorithm follows the naive $O(n^3)$ approach. It computes each element in the resulting matrix $C[i][j]$ by iterating over the corresponding row in matrix A and column in matrix B, summing their products. This straightforward implementation does not utilize any form of optimization.

The algorithm was tested as a baseline to compare against optimized methods. Its execution is single-threaded and sequential, making it resource-light but computationally expensive for large matrices. Execution times were expected to grow cubically with matrix size, highlighting its inefficiency.

```
function MatrixMultiplication(A, B)
    Initialize C as a matrix of size ROWS × COLS with zeros
    for i ← 0 to ROWS − 1 do
        for j ← 0 to COLS − 1 do
            C[i][j] ← 0
            for k ← 0 to COLS − 1 do
                C[i][j] ← C[i][j] + A[i][k] × B[k][j]
            end for
        end for
    end for
    return C
end function
```

Algorithm 1. Basic Matrix Multiplication

4.2. Vectorized Matrix Multiplication

The vectorized implementation distributes the computation of each row in the resulting matrix C across multiple threads. Each thread processes one row of C, leveraging Java's multithreading framework. This approach exploits modern processors' ability to handle multiple simultaneous tasks, achieving a form of data-level parallelism.

To mimic SIMD behavior, threads were assigned rows independently, with the resulting performance dependent on the number of available cores. This algorithm's scalability was evaluated by observing how performance improved as matrix sizes increased and how efficiently it utilized available CPU resources.

```
function VectorizedMatrixMultiplication(A, B)
    rows ← number of rows in A
    cols ← number of columns in B
    commonDim ← number of columns in A
    C ← matrix of size rows × cols initialized to zeros
    Initialize rowCounter as an AtomicInteger with value 0
    Use parallel stream to iterate over rows:
        Parallel For Each row ∈ [0, rows):
            For col ← 0 to cols − 1:
                sum ← 0
                For k ← 0 to commonDim − 1:
                    sum ← sum + A[row][k] × B[k][col]
                C[row][col] ← sum
    return C
end function
```

Algorithm 2. Vectorized Matrix Multiplication

4.3. Parallel Matrix Multiplication

The parallel algorithm employs a block-based approach, where matrices are divided into smaller sub-blocks. Each block is computed in parallel using Java's `ExecutorService`. This strategy not only reduces computation time but also optimizes memory locality by working on smaller chunks of data at a time.

A block size of 50×50 was chosen as a trade-off between task granularity and overhead from thread management. The algorithm was expected to perform exceptionally well for larger matrices due to efficient workload distribution among threads. The impact of block size on performance was noted for future optimizations.

```
function ParallelMatrixMultiplication(A, B, blockSize)
    rows ← number of rows in A
    cols ← number of columns in B
    C ← matrix of size rows × cols initialized to zeros
    Initialize an ExecutorService with available processors
    for i ← 0 to rows − 1 step blockSize do
        for j ← 0 to cols − 1 step blockSize do
            rowStart ← i, colStart ← j
            Submit task to executor:
            for row ← rowStart to min(rowStart + blockSize, rows) − 1 do
                for col ← colStart to min(colStart + blockSize, cols) − 1 do
                    sum ← 0
                    for k ← 0 to commonDim − 1 do
                        sum ← sum + A[row][k] × B[k][col]
                    end for
                    C[row][col] ← sum
                end for
            end for
        end for
    end for
    Shutdown executor and wait for termination
    return C
end function
```

Algorithm 3. Parallel Matrix Multiplication

Results

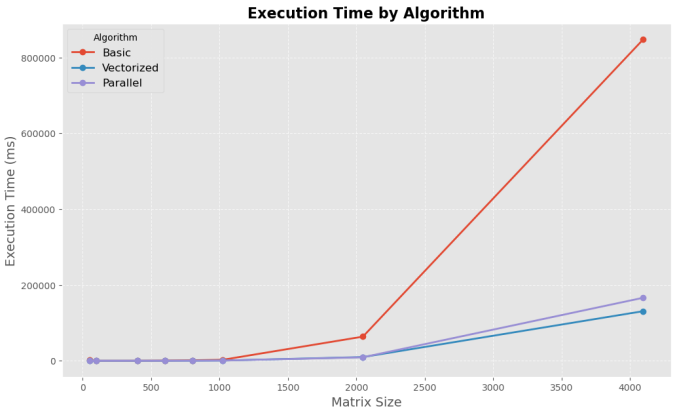

Figure 1. Execution Time Comparison

The Figure 1 compares the execution times of the three algorithms for matrix multiplication across varying matrix sizes. The x-axis represents the matrix size, while the y-axis indicates the execution time in milliseconds.

The execution time for the Basic algorithm increases exponentially as the matrix size grows. For smaller matrices (e.g. up to size 1500x1500), the algorithm shows relatively low execution times, comparable to the optimized algorithms. However, as the matrix size increases, its execution time significantly diverges, reaching over 800,000 ms, far exceeding the other approaches.

The Vectorized algorithm performs consistently better than the Basic algorithm, showing a much slower growth in execution time as matrix size increases. It demonstrates significant performance improvement for larger matrices, remaining nearly linear in comparison to the Basic algorithm. The gap between the Vectorized and Parallel algorithms is minimal, indicating that vectorization effectively accelerates matrix multiplication.

The Parallel algorithm achieves the best performance overall, particularly for larger matrix sizes. Its execution time grows more gradually than both the Basic and Vectorized algorithms, maintaining the smallest increase as the matrix size scales. The Parallel algorithm consistently outperforms the Basic algorithm and slightly edges out the Vectorized algorithm for large matrices.

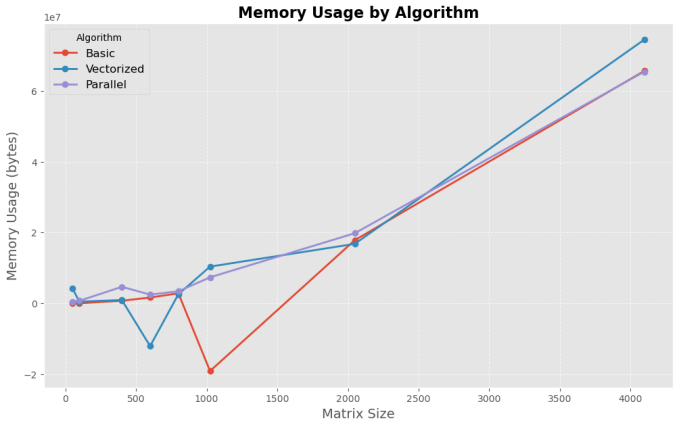

Figure 2. Memory Usage Comparison

Figure 2 shows the memory usage for each algorithm.

The Basic algorithm exhibits a consistent increase in memory usage as the matrix size grows. Its memory usage remains closely aligned with that of the Parallel algorithm for most matrix sizes. At smaller matrix sizes, the memory usage is minimal and relatively stable.

The Vectorized algorithm demonstrates slightly higher memory usage compared to the other two algorithms, particularly for larger matrices. This increased memory usage can be attributed to the overhead introduced by Java's parallel streams and additional threading management. For smaller matrices, its memory usage fluctuates but remains close to the Parallel algorithm.

The Parallel algorithm shows a memory usage pattern that closely follows the Basic algorithm at all matrix sizes. Although slightly higher in memory consumption for small matrices, it converges with the Basic algorithm as matrix size grows.

For smaller matrices such as 500x500 to 2000x2000, there are noticeable fluctuations in memory usage, likely due to the impact of memory allocation and garbage collection in Java.

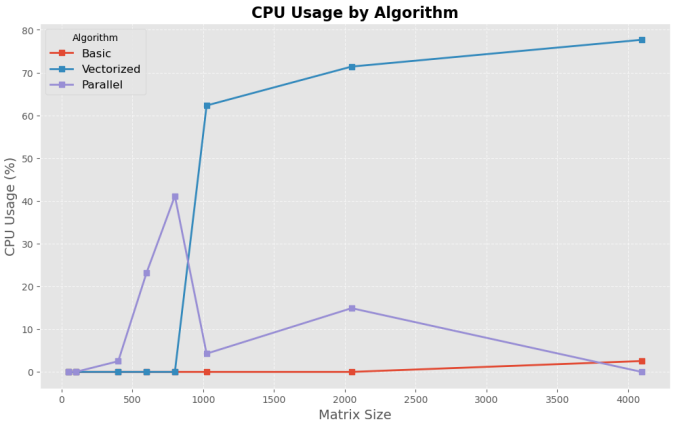

Figure 3. CPU Usage Comparison

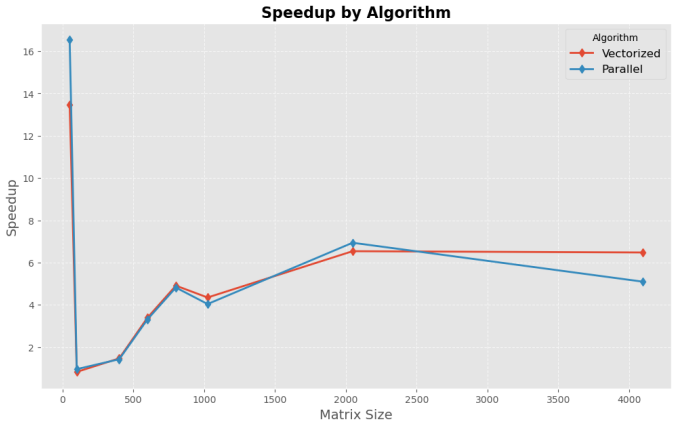

Figure 4. Speedup Comparison

As seen in Figure 3, the Basic algorithm consistently demonstrates minimal CPU usage, remaining near zero throughout all matrix sizes. This low CPU utilization is expected, as the Basic algorithm is single-threaded and does not exploit multicore processing capabilities. The flat trend line indicates a lack of scalability in terms of CPU resource utilization.

The Vectorized algorithm displays a sharp increase in CPU usage for smaller matrix sizes, peaking near 75% for matrices around 1000x1000. As the matrix size increases, CPU usage stabilizes at a high level, indicating efficient parallel processing and consistent utilization of multicore resources. The trend confirms that the algorithm effectively scales with matrix size and fully leverages CPU capabilities.

The Parallel algorithm exhibits a unique pattern, with CPU usage spiking significantly for smaller matrix sizes, surpassing 50%. However, as the matrix size grows, CPU usage decreases, eventually converging towards the levels of the Basic algorithm. This decline suggests that the overhead associated with managing multiple threads outweighs the computational benefits for larger matrices.

In Figure 4 it is shown that for very small matrices, the Parallel algorithm achieves an extremely high speedup, peaking above 16x. This can be attributed to the Basic algorithm's inefficiency with even small workloads, while the Parallel algorithm fully exploits threading. The Vectorized algorithm also shows notable speedup for smaller matrices but slightly less than the Parallel algorithm.

Both Vectorized and Parallel algorithms exhibit relatively stable speedup in this range, averaging around 2x to 5x compared to the Basic implementation. The speedup increases as the matrix size grows, indicating that these algorithms handle medium workloads more efficiently.

The Vectorized algorithm achieves consistently high speedup, leveling off at approximately 4x for large matrices. The Parallel algorithm initially matches the Vectorized algorithm but begins to decline slightly as matrix size exceeds 3000x3000. This decline could be due to thread management overhead and diminishing returns from parallelism at very large scales.

The Vectorized algorithm shows a more stable speedup pattern across all matrix sizes, maintaining its performance advantage over the Basic algorithm consistently. The Parallel algorithm demonstrates impressive speedup for smaller matrices but loses its edge slightly as matrix size increases.

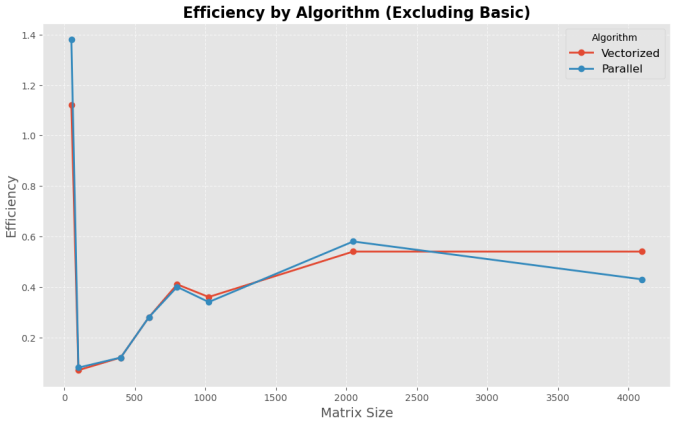

Figure 5. Efficiency Comparison

As seen in Figure 5, for very small matrices, both the Vectorized and Parallel algorithms demonstrate efficiencies exceeding 1.0, which suggests an anomaly likely caused by minimal workload overhead, resulting in artificially inflated performance metrics. These values stabilize quickly as the matrix size increases.

Both algorithms show a steady improvement in efficiency as the matrix size grows, peaking around 2000x2000. This indicates that the algorithms become more effective at distributing the workload across available threads for medium-sized matrices. The Vectorized algorithm consistently matches or slightly outperforms the Parallel algorithm in efficiency for these sizes.

The Vectorized algorithm maintains a relatively stable efficiency for larger matrices, plateauing at around 0.5 (50% of theoretical maximum efficiency). The Parallel algorithm shows a slight decline in efficiency as the matrix size grows, dropping below 0.5 for the largest matrices. This suggests that thread management overhead and diminishing returns from parallelism reduce its effectiveness at large scales.

Both algorithms exhibit sublinear efficiency (below 1.0) for most matrix sizes, indicating that while they scale well with increased workload, neither fully utilizes the available computational resources. The Vectorized algorithm is more consistent in maintaining efficiency, particularly for larger matrices, whereas the Parallel algorithm's efficiency declines more noticeably.

5. CONCLUSIONS

This study demonstrates that the choice of the best matrix multiplication algorithm depends on the workload and the performance metric of interest:

Small matrices (less than 1000x1000):

The **Parallel algorithm** is the best choice for small matrices, achieving the highest speedup (peaking above 16x) and efficient CPU utilization. Its ability to exploit multithreading for smaller workloads makes it ideal for quick computations with smaller data sizes.

The **Vectorized algorithm**, while effective, is slightly less efficient for small matrices due to overhead from parallel streams, but it still performs significantly better than the Basic algorithm.

Medium-Sized Matrices (1000x1000 to 2000x2000):

For medium workloads, both the **Vectorized** and **Parallel** algorithms perform well, offering substantial speedup and consistent efficiency. The Vectorized algorithm edges out the Parallel approach slightly, thanks to its more stable resource utilization and ability to scale consistently.

Large Matrices (bigger than 2000x2000):

The **Vectorized** algorithm is the best option for large matrices due to its stable execution time, consistent speedup (around 4x), and better efficiency as matrix size increases. It avoids the thread management overhead observed in the Parallel algorithm, making it more predictable and robust for large-scale computations. The **Parallel** algorithm struggles with efficiency for larger matrices, as the cost of managing threads reduces its performance advantage.

6. FUTURE WORK

This study opens several avenues for future exploration:

- **Thread Management Optimization:** Improving task scheduling and dynamic thread allocation in the Parallel algorithm could enhance efficiency for larger matrices.
- **Energy Efficiency:** Analyzing energy consumption alongside execution time and memory usage would provide a more holistic view of algorithm performance.
- **Sparse Matrices:** Investigating optimizations for sparse matrices, common in real-world applications, could expand the applicability of these algorithms.