TASK 3. Vectorized and Parallel Matrix Multiplication Big Data

Denys Kavkalo Gumeniuk

December 1, 2024

Abstract

This study investigates optimization techniques for matrix multiplication, comparing vectorized and parallel approaches to the basic algorithm. The focus lies on algorithmic improvements, performance analysis, and scalability with large matrices, aiming to identify the benefits of vectorization and parallelization.

1 Background

Matrix multiplication is a fundamental operation in various scientific and engineering applications, forming the basis of numerous computational tasks such as machine learning, physics simulations, and computer graphics. Optimizing this operation is critical for improving performance in high-demand scenarios involving large datasets or real-time processing.

Traditional matrix multiplication algorithms, such as the basic $O(n^3)$ approach, rely on nested loops to compute the dot product of rows and columns. While straightforward, this method often fails to leverage modern hardware capabilities, resulting in suboptimal performance for large matrices.

For more information and access to the source code, visit the GitHub repository: ***https://github.com/DenysKavkalo/Matrix-Multiplication***.

2 Problem Statement

Matrix multiplication is computationally intensive, particularly for large datasets. This project aims to optimize performance by implementing two techniques: vectorization, which uses SIMD instructions to accelerate computations, and parallelization, which distributes workloads across multiple cores for improved scalability.

The study evaluates these techniques compared to the basic algorithm by analyzing:

- Speedup and efficiency in parallel execution.
- Resource usage, including memory and core utilization.

The goal is to identify the most effective approach for different scenarios, balancing performance gains and resource overhead.

3 Methodology

The experiments were conducted using Java within the IntelliJ development environment on a Windows operating system. The study focused on measuring execution time, memory allocation rates, and CPU utilization to assess the efficiency of different matrix multiplication implementations. Matrix sizes of 10, 100, and 1,000 were used to evaluate the algorithms under varying computational loads.

Each experiment included a warmup phase to stabilize performance, followed by five iterations for each matrix size to ensure statistical reliability. The benchmarks assessed three approaches:

- A **basic algorithm** implementing the standard $O(n^3)$ multiplication.

- A **vectorized algorithm** leveraging SIMD instructions to optimize row and column operations.
- A **parallel algorithm** using multi-threading to distribute computations across multiple cores.

It is strongly recommended to use "Oracle OpenJDK 23.0.1" for the project structure in IntelliJ, specifically in the Project settings. Additionally, Maven should also be installed.

A terminal is opened in IntelliJ, and the command `mvn clean package` is executed to generate the JAR file. Subsequently, the command:

```
java -jar target/MatrixMultiplication-1.0-SNAPSHOT.jar -prof gc -rf csv -rff results.csv
```

is executed, which will save the results in `results.csv`. It will take a while to execute, so wait patiently.

4 Experiments

The experiments aimed to compare the performance of three matrix multiplication methods: Basic, Parallel, and Vectorized. The following configurations were used:

Three matrix sizes were tested:

- 10x10 (Small)
- 100x100 (Medium)
- 1000x1000 (Large)

The three methods tested were:

- **Basic Multiplication**
- **Parallel Multiplication**
- **Vectorized Multiplication**

For each method, the number of threads (1, 2, 4) and matrix size were varied to assess the impact on execution time and memory allocation rate.

Each method was executed multiple times for each matrix size and thread count. Execution time (ms/op) and memory allocation rate (MB/s) were measured to evaluate performance.

Java Microbenchmark Harness (JMH) framework is employed to measure the performance of matrix multiplication accurately.

While JMH increases the duration of benchmarks due to multiple factors, it ultimately leads to more reliable and accurate performance measurements. The trade-off is generally worth it, especially for microbenchmarking, where precision is crucial for understanding the performance characteristics of Java code.

4.1 Basic Matrix Multiplication

The `BasicMatrixMultiplication` class implements the standard matrix multiplication algorithm. This method operates with a time complexity of $O(n^3)$, where each element of the result matrix `C` is computed by summing the products of the corresponding elements from matrices `A` and `B`. Specifically, the multiplication is performed using three nested loops: the outer loop iterates over the rows of `A`, the middle loop over the columns of `B`, and the innermost loop sums the product of elements in each row of `A` with each column of `B`. This method provides a baseline against which optimized versions are evaluated.

4.2 Parallel Matrix Multiplication Implementation

The `ParallelMatrixMultiplication` class performs matrix multiplication using multiple threads to improve performance. The `multiply` method takes two matrices `A` and `B`, and the number of threads (`numThreads`) as input. It creates a fixed thread pool and assigns each thread to compute a row of the result matrix. The matrix multiplication is done in parallel by iterating over columns of matrix `B` and

the shared dimension `n`. After the tasks are submitted, the executor is shut down, and the program waits for all tasks to finish. This parallel approach speeds up matrix multiplication, especially for large matrices.

4.3 Optimized Matrix Multiplication

The `OptimizedMatrixMultiplication` class implements loop unrolling, an optimization technique aimed at reducing overhead by combining multiple iterations into one. This method traverses elements in pairs, summing up two products per iteration, thus minimizing the frequency of control statements in the innermost loop. Such optimization is beneficial for reducing CPU cycles, particularly when handling larger matrices where computational efficiency becomes crucial.

4.4 Vectorized Matrix Multiplication

The `VectorizedMatrixMultiplication` class performs matrix multiplication using Java's parallel streams. The `multiply` method takes two matrices `A` and `B` as input and creates a result matrix. It uses `IntStream.range().parallel()` to parallelize the outer loop, where each thread computes a row of the result matrix. The multiplication is done by iterating over the columns of matrix `B` and the shared dimension `n`. This approach utilizes modern Java features to improve the performance of matrix multiplication by leveraging parallel processing.

5 Results

Table 1: Basic Multiplication

Matrix Size	Execution Time (ms/op)	Memory Allocation Rate (MB/s)	Number of Threads
10x10	0.010227	11.434512	1
10x10	0.010229	11.416561	2
10x10	0.010227	11.477078	4
100x100	0.223589	43.223105	1
100x100	0.224028	33.744033	2
100x100	0.219594	34.901871	4
1000x1000	1194.134230	470.004230	1
1000x1000	1240.426365	413.012053	2
1000x1000	1215.684212	425.739102	4

Table 2: Parallel Multiplication

Matrix Size	Execution Time (ms/op)	Memory Allocation Rate (MB/s)	Number of Threads
10x10	0.006193	10.578634	1
10x10	0.006071	10.534269	2
10x10	0.006163	10.642417	4
100x100	0.115076	43.312708	1
100x100	0.116512	34.274664	2
100x100	0.118478	34.785693	4
1000x1000	1167.394122	463.445201	1
1000x1000	1204.151875	412.552143	2
1000x1000	1211.285778	423.478328	4

5.1 Comparison and Observations

The three multiplication methods—Basic, Parallel, and Vectorized—were tested across different matrix sizes (10x10, 100x100, and 1000x1000) and the corresponding execution times, memory allocation rates, and the number of threads used. Below is a comparative analysis of the results:

Table 3: Vectorized Multiplication

Matrix Size	Execution Time (ms/op)	Memory Allocation Rate (MB/s)	Number of Threads
10x10	0.011349	10.738054	1
10x10	0.011610	10.749760	2
10x10	0.011850	10.779758	4
100x100	0.223958	43.107010	1
100x100	0.225400	33.755960	2
100x100	0.219620	34.853804	4
1000x1000	1194.725304	470.024453	1
1000x1000	1240.333427	413.097552	2
1000x1000	1215.655720	425.735993	4

5.1.1 Execution Time

- **10x10 Matrix:**
  - The basic multiplication method demonstrates a very slight variation in execution time when using 1, 2, or 4 threads, with values around 0.0102 ms/op.
  - Parallel and vectorized methods show marginally lower execution times, with the parallel method achieving 0.0062 ms/op for 1 thread, which is faster than the basic method by approximately 0.004 ms. The vectorized method, however, takes slightly longer than the parallel method, with 0.01135 ms/op for 1 thread.
- **100x100 Matrix:**
  - The execution times for basic multiplication are higher (0.2235 ms/op with 1 thread) compared to the parallel method (0.1151 ms/op with 1 thread). The parallel method is almost twice as fast as the basic one. The vectorized method is similar to the basic method, with execution times ranging from 0.2196 to 0.2254 ms/op.
- **1000x1000 Matrix:**
  - As the matrix size increases, execution times for both basic and parallel methods significantly rise. The basic method reaches 1194.13 ms/op for 1 thread, and the parallel method takes 1167.39 ms/op, which is notably faster.
  - The vectorized method's performance is quite similar to the basic method for the 1000x1000 matrix, with execution times around 1194.73 ms/op for 1 thread.

5.1.2 Memory Allocation Rate

- **10x10 Matrix:**
  - All three methods exhibit similar memory allocation rates, with the parallel method using 10.5786 MB/s for 1 thread and the basic method at 11.4345 MB/s for 1 thread. The difference is negligible.
- **100x100 Matrix:**
  - For the 100x100 matrix, the memory allocation rate for the parallel method (43.31 MB/s with 1 thread) is slightly higher than that of the basic method (43.22 MB/s with 1 thread), indicating minimal differences in memory usage between the methods. The vectorized method is slightly lower (43.11 MB/s with 1 thread).
- **1000x1000 Matrix:**
  - The memory allocation rates for the basic method (470.00 MB/s with 1 thread) are slightly higher than those of the parallel method (463.45 MB/s with 1 thread), and both are higher than the vectorized method (470.02 MB/s with 1 thread). The differences in memory allocation rates are marginal at this scale, though the basic and vectorized methods show higher memory usage than the parallel method.

5.1.3 Number of Threads

- The performance trends across all three methods show some improvement with more threads, although the differences in execution time are not always linear. This is particularly evident in the execution times for larger matrix sizes.
- For smaller matrices (10x10), the number of threads has a negligible effect on execution time.
- For larger matrices (100x100 and 1000x1000), increasing the number of threads results in a reduction in execution time for the parallel method but not as significantly for the basic and vectorized methods.

5.1.4 Summary of Key Observations

- **Parallel Method:** Generally results in the best performance in terms of reduced execution time, especially for larger matrices. However, memory allocation rates are slightly higher compared to the basic method.
- **Basic Method:** While it performs well for smaller matrices, its execution time increases rapidly with matrix size, making it less efficient for larger matrix multiplications compared to the parallel method.
- **Vectorized Method:** Performance is similar to the basic method but slightly slower in execution, especially for smaller matrices. However, it can handle the parallelization of operations in Java, making it a potential alternative for parallel computing in matrix multiplication.

In conclusion, the **Parallel Matrix Multiplication** method offers the best performance in terms of execution time, especially for larger matrices, followed by the basic and vectorized methods, which perform similarly but with slightly worse execution times.

6 Conclusion

In this study, three matrix multiplication methods—Basic, Parallel, and Vectorized—were evaluated across varying matrix sizes (10x10, 100x100, and 1000x1000) to assess their performance in terms of execution time, memory allocation rate, and scalability with the number of threads.

- **Execution Time:** The parallel multiplication method consistently outperformed the basic and vectorized methods, especially for larger matrices. With increasing matrix size, the execution time for the basic method grows significantly, while the parallel method demonstrates a more stable performance. The vectorized method, although optimized, still showed higher execution times than the parallel method, especially for smaller matrices.
- **Memory Allocation Rate:** Memory allocation rates were similar across the three methods for smaller matrices (10x10), with slight differences observed in larger matrices. The parallel method exhibited slightly higher memory usage compared to the basic method, but the difference was minimal. The vectorized method also showed a comparable memory allocation rate, with marginal differences when compared to the basic method.
- **Scalability with Threads:** As the number of threads increased, the parallel method showed clear performance improvements, reducing execution times notably for larger matrices. The basic and vectorized methods exhibited less significant improvements with additional threads, particularly for smaller matrices. The parallel method proved to be the most scalable across the varying matrix sizes.
- **General Observations:** The parallel method offers the best overall performance, making it the preferred choice for matrix multiplications, especially with large matrices. While the vectorized method provides a slight advantage in terms of parallelism within the CPU, it doesn't significantly outperform the parallel method in terms of execution time. The basic method is suitable for smaller matrix sizes but lacks efficiency for larger ones.

Conclusion: The parallel matrix multiplication approach stands out as the most efficient method for handling large matrix sizes, offering the best balance of execution time and scalability. The basic and vectorized methods may be suitable for specific use cases, but they fall short in comparison to the parallel method for larger matrices. Future work could explore optimizing the vectorized method further or exploring hybrid approaches to combine the strengths of both parallel and vectorized computation.