Parallelization of Matrix Multiplication

Wojciech Dróżdż

December 1, 2024

Abstract

This paper focuses on speeding up the matrix multiplication algorithm by splitting the multiplication task into multiple sub-tasks and executing them on independent CPU threads. Compared to a non-multithreaded approach a speedup of around 700% was observed.
The code for this paper can be found in the GitHub repository

1 Introduction

In the previous paper optimization methods for single-threaded matrix multiplication were considered. In this paper the option of parallelization will be considered. Matrix multiplication is an ideal candidate for running on multiple threads, as all of the operations are independent of one another. This means, that theoritically it should be possible to archive infinite speedup with an infinite number of CPU cores.

2 Methodology

2.1 Hardware

All of the tests were performed on a 2023 16 inch MacBook Pro. Relevant specifications:

- Apple M3 Pro - 4 GHz 12-core CPU with 6 performance cores and 6 efficiency cores
- 18GB RAM
- 1024GB of internal storage

In order to preserve consistency all of the tests were performed while the device was plugged into a power source.

2.2 Interpreters and Compilers

- *Java* - Java 17 with JetBrains runtime

2.3 Benchmarking Tools

The *Java Microbenchmark Harness (JHM)*, part of the *OpenJDK* project, was chosen for its microbenchmarking features.

2.4 Benchmarking Methodology

The Benchmarks measure only the time it takes to multiply the matricies, all of the memory is pre-allocated before running the multiplication method. The memory allocation time is not measured. Testing was started with 10 warmup iterations before full testing, allowing for just-in-time compilations and appropriate cache optimizations. This was followed by 10 test iterations during which the shortest, longest, and average execution times were recorded.

3 Experiments

3.1 Baseline runs

To establish baseline results the standard multiplication method and the best method from the last paper - blocked column multiply, were tested.

Benchmark	(size)	Score	r	Units
Task3Main.standardMultiply	1	10		ms/op
Task3Main.standardMultiply	5	10		ms/op
Task3Main.standardMultiply	10	0.001		ms/op
Task3Main.standardMultiply	50	0.068		ms/op
Task3Main.standardMultiply	100	0.585		ms/op
Task3Main.standardMultiply	200	5.177		ms/op
Task3Main.standardMultiply	300	18.821		ms/op
Task3Main.standardMultiply	500	90.156		ms/op
Task3Main.standardMultiply	1000	1254.097		ms/op
Task3Main.standardMultiply	1500	5377.098		ms/op
Task3Main.standardMultiply	2000	15116.272		ms/op
Task3Main.blockedColumnMultiply	1	10		ms/op
Task3Main.blockedColumnMultiply	5	10^{3}		ms/op
Task3Main.blockedColumnMultiply	10	0.001		ms/op
Task3Main.blockedColumnMultiply	50	0.074		ms/op
Task3Main.blockedColumnMultiply	100	0.343		ms/op
Task3Main.blockedColumnMultiply	200	5.530		ms/op
Task3Main.blockedColumnMultiply	300	10.071		ms/op
Task3Main.blockedColumnMultiply	500	78.610		ms/op
Task3Main.blockedColumnMultiply	1000	463.471		ms/op
Task3Main.blockedColumnMultiply	1500	2462.056		ms/op

3.2 Multithreaded Stream Execution

To further optimize matrix multiplication, we implemented a parallelized approach using Java's Stream API. Specifically, the outer loop iterating over matrix rows was parallelized with IntStream.range(0, m).parallel(), allowing concurrent processing of different rows across multiple threads.

Using multithreading, the computational workload is distributed across available CPU cores, significantly improving performance, particularly for larger matrices. Although the overhead of managing threads may diminish gains for smaller matrices, parallel execution results in substantial time reductions for larger problem sizes.

The benchmark results clearly demonstrate the advantages of this approach. For instance, the execution time for a 2000x2000 matrix was reduced from around 15,116 ms using standard multiplication to 2,659 ms with multithreading. This highlights the effectiveness of parallel processing in reducing computational time for matrix multiplication tasks, especially as matrix size increases.

Benchmark	(size)	Score	Units
Task3Main.multithreadedStreamMultiply	1	10	ms/op
Task3Main.multithreadedStreamMultiply	5	0.008	ms/op
Task3Main.multithreadedStreamMultiply	10	0.017	ms/op
Task3Main.multithreadedStreamMultiply	50	0.047	ms/op
Task3Main.multithreadedStreamMultiply	100	0.210	ms/op
Task3Main.multithreadedStreamMultiply	200	0.922	ms/op
Task3Main.multithreadedStreamMultiply	300	2.863	ms/op
Task3Main.multithreadedStreamMultiply	500	18.068	ms/op
Task3Main.multithreadedStreamMultiply	1000	194.598	ms/op
Task3Main.multithreadedStreamMultiply	1500	782.238	ms/op
Task3Main.multithreadedStreamMultiply	2000	2659.189	ms/op

3.3 Parallelization using Executors

In this approach, matrix multiplication was parallelized using Java's ExecutorService, which allows for more explicit control over thread management. Each row of the result matrix was assigned to a separate task, and the ExecutorService distributed these tasks across a pool of threads. This method provides greater flexibility compared to the Stream API, enabling customization of the number of threads and better handling of task execution.

The results show that the use of executors delivers notable performance improvements, especially for larger matrices. For example, multiplying a 2000x2000 matrix took 2,044 ms, compared to 15,116 ms for the standard method. Although slightly slower than the Stream API approach for smaller matrices, Executors offer a more controlled and scalable parallel execution strategy, making them suitable for larger workloads and scenarios where fine-grained thread management is required.

```
Benchmark                                  (size)      Score     Units
Task3Main.multithreadedExecutorMultiply         1      0.038     ms/op
Task3Main.multithreadedExecutorMultiply         5      0.112     ms/op
Task3Main.multithreadedExecutorMultiply        10      0.161     ms/op
Task3Main.multithreadedExecutorMultiply        50      0.358     ms/op
Task3Main.multithreadedExecutorMultiply       100      0.658     ms/op
Task3Main.multithreadedExecutorMultiply       200      2.097     ms/op
Task3Main.multithreadedExecutorMultiply       300      3.676     ms/op
Task3Main.multithreadedExecutorMultiply       500     18.985     ms/op
Task3Main.multithreadedExecutorMultiply      1000    160.919     ms/op
Task3Main.multithreadedExecutorMultiply      1500    783.811     ms/op
Task3Main.multithreadedExecutorMultiply      2000   2044.723     ms/op
```

3.4 Parallelization of the blocked column method

Parallelization of the blocked column method is slightly more complex than other methods. It is important to assign the multiplication tasks to the threads in a way that allows the algorithm to take advantage of the optimized cache access. In our case the array is split into multiple sub-arrays for each thread, which allows us to keep the benefits of the method. Parallelizing this algorithm yielded the best results, with a 500ms execution time for a 2000x2000 array.

Benchmark	(size)	Score	Units
Task3Main.multithreadedBlockedColumnMultiply	1	0.039	ms/op
Task3Main.multithreadedBlockedColumnMultiply	5	0.146	ms/op
Task3Main.multithreadedBlockedColumnMultiply	10	0.317	ms/op
Task3Main.multithreadedBlockedColumnMultiply	50	0.346	ms/op
Task3Main.multithreadedBlockedColumnMultiply	100	0.490	ms/op
Task3Main.multithreadedBlockedColumnMultiply	200	1.060	ms/op
Task3Main.multithreadedBlockedColumnMultiply	300	3.301	ms/op
Task3Main.multithreadedBlockedColumnMultiply	500	14.231	ms/op
Task3Main.multithreadedBlockedColumnMultiply	1000	70.368	ms/op
Task3Main.multithreadedBlockedColumnMultiply	1500	266.928	ms/op
Task3Main.multithreadedBlockedColumnMultiply	2000	546.832	ms/op

This was also the last method, which has been tried. Currently the algorithm is about 3000% faster than the initial version.

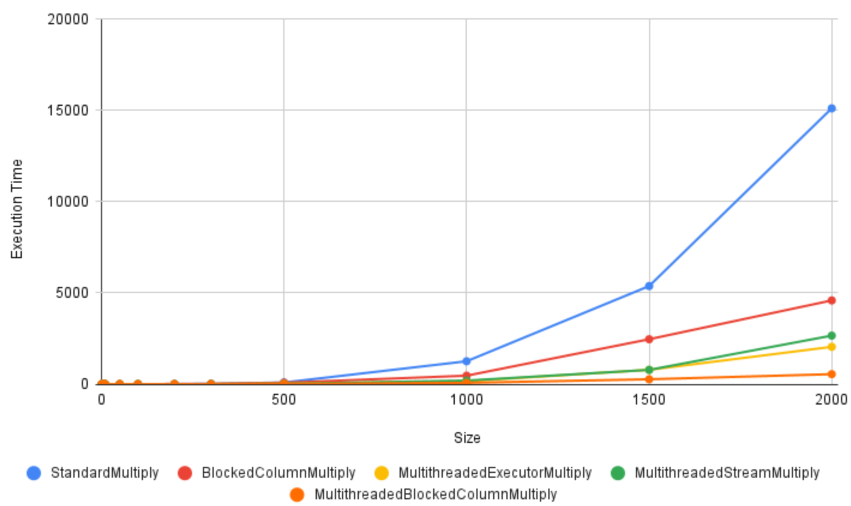

Figure 1: Execution time comparison for the algorithms

3.5 The impact of the number of threads on the execution time

Matrix multiplication is highly parallelizable, making it ideal for optimization through multithreading. Each element in the result matrix is computed independently of others, allowing the workload to be divided across multiple threads with minimal synchronization overhead. As a result, increasing the number of threads can significantly reduce execution time, particularly for large matrices.

However, the performance gains from additional threads are subject to diminishing returns. Initially, adding more threads reduces execution time substantially, but as the number of threads approaches the available hardware cores, overhead from thread management and context switching can offset further improvements. For small matrices, excessive threading may even degrade performance due to the overhead outweighing the computational workload.

In general, we expect faster execution with more threads up to the point where the system's core count is saturated. Beyond that, performance plateaus or may degrade slightly. Our experiments confirm this for the Executor method, showing that optimal results are achieved when the number of threads matches or slightly exceeds the number of available cores, especially for matrix sizes in the range of 1000x1000 and larger. The blocked column method plateaus earlier than the Executor method at 10-11 threads running on 12 cores, which suggests that the memory access is becoming too complex as the number of cores grows and the benefits of multithreading start affecting the benefits of faster cache access.

Task3Main.multithreadedBlockedColumnMultiply	0	1279.263	ms/op
Task3Main.multithreadedBlockedColumnMultiply	1	6652.166	ms/op
Task3Main.multithreadedBlockedColumnMultiply	2	4294.967	ms/op
Task3Main.multithreadedBlockedColumnMultiply	3	3061.842	ms/op
Task3Main.multithreadedBlockedColumnMultiply	4	2348.810	ms/op
Task3Main.multithreadedBlockedColumnMultiply	5	2069.889	ms/op
Task3Main.multithreadedBlockedColumnMultiply	6	1772.093	ms/op
Task3Main.multithreadedBlockedColumnMultiply	7	1740.636	ms/op
Task3Main.multithreadedBlockedColumnMultiply	8	1688.207	ms/op
Task3Main.multithreadedBlockedColumnMultiply	9	1497.367	ms/op
Task3Main.multithreadedBlockedColumnMultiply	10	1530.921	ms/op
Task3Main.multithreadedBlockedColumnMultiply	11	1228.931	ms/op
Task3Main.multithreadedBlockedColumnMultiply	12	1319.109	ms/op
Task3Main.multithreadedBlockedColumnMultiply	13	1344.274	ms/op
Task3Main.multithreadedBlockedColumnMultiply	14	1457.521	ms/op
Task3Main.multithreadedBlockedColumnMultiply	15	1407.189	ms/op
Task3Main.multithreadedBlockedColumnMultiply	16	1600.127	ms/op
Task3Main.multithreadedBlockedColumnMultiply	17	1413.480	ms/op
Task3Main.multithreadedBlockedColumnMultiply	18	1533.018	ms/op
Task3Main.multithreadedBlockedColumnMultiply	19	1405.092	ms/op
Task3Main.multithreadedBlockedColumnMultiply	20	1528.824	ms/op
Task3Main.multithreadedBlockedColumnMultiply	21	1199.571	ms/op
Task3Main.multithreadedBlockedColumnMultiply	22	1346.372	ms/op
Task3Main.multithreadedBlockedColumnMultiply	23	1245.708	ms/op
Task3Main.multithreadedBlockedColumnMultiply	24	1312.817	ms/op
Task3Main.multithreadedExecutorMultiply	0	5435.818	ms/op
Task3Main.multithreadedExecutorMultiply	1	39929.774	ms/op
Task3Main.multithreadedExecutorMultiply	2	20602.421	ms/op
Task3Main.multithreadedExecutorMultiply	3	13807.649	ms/op
Task3Main.multithreadedExecutorMultiply	4	10687.087	ms/op
Task3Main.multithreadedExecutorMultiply	5	8539.603	ms/op
Task3Main.multithreadedExecutorMultiply	6	8724.152	ms/op
Task3Main.multithreadedExecutorMultiply	7	7683.965	ms/op
Task3Main.multithreadedExecutorMultiply	8	7415.529	ms/op
Task3Main.multithreadedExecutorMultiply	9	7230.980	ms/op
Task3Main.multithreadedExecutorMultiply	10	6870.270	ms/op
Task3Main.multithreadedExecutorMultiply	11	6786.384	ms/op
Task3Main.multithreadedExecutorMultiply	12	6417.285	ms/op
Task3Main.multithreadedExecutorMultiply	13	6954.156	ms/op
Task3Main.multithreadedExecutorMultiply	14	6509.560	ms/op
Task3Main.multithreadedExecutorMultiply	15	6887.047	ms/op
Task3Main.multithreadedExecutorMultiply	16	6190.793	ms/op
Task3Main.multithreadedExecutorMultiply	17	6308.233	ms/op
Task3Main.multithreadedExecutorMultiply	18	6509.560	ms/op
Task3Main.multithreadedExecutorMultiply	19	6224.347	ms/op
Task3Main.multithreadedExecutorMultiply	20	6257.902	ms/op
Task3Main.multithreadedExecutorMultiply	21	6383.731	ms/op
Task3Main.multithreadedExecutorMultiply	22	6090.129	ms/op
Task3Main.multithreadedExecutorMultiply	23	6417.285	ms/op
Task3Main.multithreadedExecutorMultiply	24	5771.362	ms/op

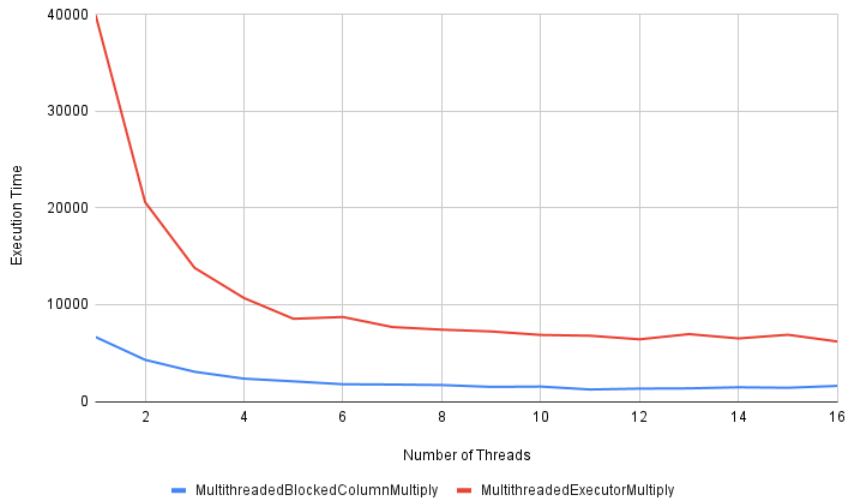

Figure 2: Execution time for different number of threads for both algorithms

4 Conclusion

This paper explored various optimization techniques for matrix multiplication, focusing on multi-threading approaches using Java's Stream API and ExecutorService. The results demonstrate that both methods significantly improve performance over standard matrix multiplication, particularly for larger matrices. Multithreaded Stream execution proved highly effective, reducing computation time by leveraging parallelism across available CPU cores. Meanwhile, the ExecutorService provided more control over thread management, offering comparable performance gains, especially for larger datasets.

The impact of thread count was also examined, revealing that optimal performance is achieved when the number of threads aligns with the system's core count, while excessive threading introduces diminishing returns. Overall, matrix multiplication is well-suited for parallelization, and careful selection of threading techniques can lead to substantial performance improvements. Future work could explore hybrid models or GPU-based approaches to further accelerate matrix multiplication tasks.

5 Future work

In future work, we aim to explore GPU acceleration for matrix multiplication. GPUs are highly optimized for parallel processing due to their numerous cores, making them well-suited for computationally intensive tasks like matrix multiplication, which involves a high degree of independent calculations. Using a GPU can significantly reduce execution times, particularly for large matrices, where CPU-based multithreading faces limitations due to core count and thread management overhead. By leveraging GPU architectures, we expect to achieve even greater performance improvements.