Individual assigement BIG DATA

Grado en Ciencia e Ingenier´ıa de Datos a

Universidad de Las Palmas de Gran Canaria

Matrix multiplication PART II

Jakub Król

Link to repository: GitHub repository

Contents

Part I

Abstract

This paper explores optimization techniques for matrix multiplication in Java, focusing on methods to handle large matrices efficiently. We compare a traditional dense multiplication approach with two optimized methods: a sparse matrix multiplication optimized by density, and a block-based multiplication that divides the matrix into smaller submatrices. Performance evaluations reveal that the block-based approach consistently outperforms the sparse method, regardless of matrix density, and demonstrates robust scalability for larger matrices. Surprisingly, the classical dense multiplication method, without optimizations, shows competitive results, surpassing the sparse approach in certain cases. Future work will investigate the potential for multithreaded processing to further enhance performance.

Part II

Introduction

Matrix multiplication is a core operation in many computational fields, including scientific computing, data science, computer graphics, and machine learning. The computational complexity of matrix multiplication increases exponentially with the size of the matrices, creating performance bottlenecks in applications that rely on processing large matrices. For instance, a basic implementation of matrix multiplication has a time complexity of $O(n^3)$, where n is the dimension of the matrix. As the matrix size grows, this basic approach becomes computationally expensive, requiring substantial processing power and memory resources.

To address this challenge, the goal of this study is to implement and compare optimized matrix multiplication techniques in Java. This includes exploring algorithmic improvements, such as Strassen's algorithm, and applying optimization techniques like loop unrolling, cache-friendly partitioning of matrices, and sparse matrix implementations. Additionally, the study examines the impact of sparsity—measured by the proportion of zero elements—on the performance of matrix multiplication, leveraging sparse matrix optimizations to avoid unnecessary operations on zero elements.

The primary motivation behind these optimizations is to achieve faster execution times and higher efficiency when handling large-scale matrices. In Java, where memory management and efficient processing are critical, optimized algorithms can significantly reduce execution time and memory usage. This research will implement several optimized approaches, systematically analyze their performance, and assess their scalability across various matrix sizes.

Part III

Problem Statement

The objective of this project is to investigate, implement, and evaluate multiple optimization techniques for matrix multiplication in Java. Given that standard matrix multiplication in Java can be resource-intensive, especially with large matrices, the project focuses on enhancing both the computational efficiency and scalability of the matrix multiplication operation. Specifically, we aim to:

- **Implement two or more optimized matrix multiplication algorithms.** The optimizations will focus on techniques that are computationally feasible in Java, such as algorithmic improvements (e.g., Strassen's algorithm), loop unrolling to reduce overhead, and cache optimization to improve memory access patterns.

- **Explore sparse matrix multiplication.** Sparse matrices, which have a high proportion of zero elements, allow us to bypass operations on these zeros, thus saving computational resources. By implementing sparse matrix multiplication, we can measure how different sparsity levels affect performance.

- **Conduct comparative performance testing.** Each optimized approach will be evaluated and compared with the baseline (non-optimized) approach to determine the performance gains in terms of execution time and resource usage. Key metrics will include execution time, maximum feasible matrix size, and performance differences between dense and sparse matrix multiplications.

To carry out these objectives, we will conduct a series of experiments with increasingly larger matrix sizes (e.g., 32, 64, 128, 256, 512, 1024, and 2048) and varying sparsity levels (e.g., 0.1 and 0.9). Each method's execution time and scalability will be analyzed to identify bottlenecks, limitations, and potential areas for further optimization.

Part IV

Methodology

This study follows a structured methodology to implement and analyze different optimized matrix multiplication algorithms in Java. The following steps outline the approach:

1. **Algorithm Development and Implementation:**
    - **Standard Matrix Multiplication.** Implement a baseline matrix multiplication method in Java as a reference point. This implementation will use the straightforward $O(n^3)$ approach without optimizations.
    - **Optimization Techniques.** Develop two or more optimized matrix multiplication algorithms, including:
        - *Algorithmic Optimization (Strassen's Algorithm):* Strassen's algorithm reduces the computational complexity from $O(n^3)$ to approximately $O(n^{2.81})$, providing significant speedup for larger matrices. This technique will be implemented and compared with the baseline.
        - *Cache Optimization with Block Matrix Multiplication:* To enhance cache usage, the matrix is divided into smaller sub-blocks, improving the locality of memory access. This approach leverages Java's memory management by minimizing cache misses and enhancing execution speed.

    - *Loop Unrolling:* This technique reduces loop overhead by decreasing the number of iterations. It will be applied to the matrix multiplication loop to improve runtime efficiency.

2. **Sparse Matrix Implementation and Optimization:**
    - **Sparse Matrix Representation.** Implement sparse matrices using efficient data structures, such as lists or hashmaps, to store only non-zero elements. This reduces memory requirements and processing time.
    - **Performance with Varying Sparsity Levels.** Test the sparse matrix multiplication with different sparsity levels (e.g., 10% non-zero elements and 90% non-zero elements). This will help identify the relationship between sparsity and computational savings in Java.

3. **Testing and Performance Evaluation:**
    - **Experimental Setup:** Test all implementations with matrix sizes of 32, 64, 128, 256, 512, 1024, and 2048 to observe how each approach scales. Record execution times for each implementation, noting the maximum matrix size each can handle efficiently within the constraints of Java's memory and processing limitations.
    - **Metrics for Evaluation:**
        - *Execution Time:* Measure and compare the time taken for each approach to complete matrix multiplication.
        - *Maximum Feasible Matrix Size:* Identify the largest matrix size each method can handle without exceeding reasonable memory and time limits.
        - *Performance Comparison between Dense and Sparse Matrices:* Evaluate the computational efficiency of dense versus sparse matrix multiplication, with a focus on how varying sparsity levels impact performance.
    - **Analysis of Bottlenecks and Issues:** Note any observed bottlenecks, such as memory constraints or processing delays, and analyze the causes. This may include limitations due to Java's garbage collection or the need for further optimization of loop structures and memory allocation.

Each result will be thoroughly analyzed and documented, with insights into the advantages and disadvantages of each optimization method in Java. Special attention will be given to identifying how specific optimizations influence performance as matrix sizes grow, providing a comprehensive understanding of efficient matrix multiplication techniques in Java.

Part V

Experiments

The experiments evaluate the performance of three optimized matrix multiplication implementations in Java, comparing each to a traditional dense multiplication approach. Each experiment measures execution time, memory usage, and scalability across varying matrix sizes and densities. The experiments focus on three optimization strategies:

- **Experiment 1: Sparse Matrix Optimization based on Density.** This approach optimizes matrix multiplication by multiplying only non-zero elements when density levels are low.
- **Experiment 2: Block Matrix Multiplication for Cache Efficiency.** This method divides matrices into smaller blocks to maximize cache usage, which is effective for large matrices.
- **Experiment 3: Pre-loaded Sparse Matrix from File.** This experiment uses a large, pre-defined sparse matrix, directly measuring performance without matrix generation.

Each optimized approach is compared with traditional dense matrix multiplication using element-by-element calculations.

1 Experiment: Sparse Matrix Optimization based on Density

The first experiment uses the `MatrixSparse` class, which switches between dense and sparse multiplication based on density. Densities of 0.1, 0.2, 0.5, and 0.9 are tested to analyze performance variations with increasing sparsity.

Results: Lower densities (e.g., 0.1, 0.2) yield faster performance in the sparse multiplication due to reduced operations on zero elements. As density increases (e.g., 0.5 and 0.9), the sparse method loses efficiency compared to dense multiplication, which has less overhead from checking for non-zero values.

2 Experiment: Block Matrix Multiplication for Cache Efficiency

The block multiplication strategy in `MatrixMultiplication` divides matrices into blocks, optimizing cache usage for large matrices.

Results: This approach improves performance for large matrix sizes (512 and above) by maintaining data locality within the cache, significantly reducing execution time. Changing density does not significantly impact performance in this method, so only the extremes (0.1 and 0.9) were tested here.

3 Experiment: Pre-loaded Sparse Matrix from File

In the third experiment, the `GivenM` class loads a pre-defined sparse matrix from a file, directly testing large sparse matrices.

Results: Pre-loading sparse matrices minimizes data load time and reduces memory usage. Sparse multiplication achieves substantial speed improvements, especially at lower densities, as fewer elements are processed. This approach demonstrates the highest efficiency in handling large, real-world sparse matrices compared to traditional multiplication.

4 Comparison with Traditional Dense Matrix Multiplication

Each method is compared with a standard dense matrix multiplication implementation to assess execution time and memory usage:

- **Sparse Matrix Optimization:** Delivers faster performance at lower densities by bypassing zero elements.

- **Block Matrix Multiplication:** Significantly enhances performance for large dense matrices, especially at sizes of 512 and above.
- **Pre-loaded Sparse Matrix:** Optimized for real-world sparse matrices, achieving efficient multiplication with minimal memory usage.

Part VI

Conclusions

Based on the experimental data, several key insights regarding the performance of different matrix multiplication optimizations can be drawn:

1. Block Multiplication Outperforms Sparse Multiplication: Across all tested densities and matrix sizes, the block matrix multiplication approach consistently shows better performance compared to sparse multiplication. The block method leverages cache optimization and avoids the overhead associated with checking for non-zero elements in sparse matrices, which contributes to its efficiency.

2. Density Impact on Sparse vs. Block Multiplication: For sparse multiplication, performance significantly decreases as density increases. Higher density means fewer zero elements to skip, reducing the effectiveness of the sparse optimization. On the other hand, the block method shows stable performance across all densities (0.1, 0.2, 0.5, 0.9), as it primarily relies on cache efficiency rather than density-related optimizations.

3. Comparison to Traditional Dense Multiplication: The traditional dense matrix multiplication implementation yields a time of approximately 94512.2 ms for large matrices, which is faster than the sparse optimization in most cases. This indicates that for higher densities, where the matrix is nearly full, the sparse approach becomes inefficient and can even be slower than the non-optimized traditional method.

4. Pre-loaded Matrix from File: In cases where a matrix is pre-loaded, the block multiplication method also demonstrates superior performance for large matrices. This approach effectively handles pre-defined large matrices with high density without compromising on speed.

4.1 Performance Summary Table

The table below summarizes the results for matrix sizes and densities tested during the experiments.

Size	**Time [ms/op]**						
	Density						**Given matrix**
	Sparse				**Block**		1000.182
32	0.024	0.036	0.026	0.027	0.028	0.028	
64	0.219	0.422	0.229	0.238	0.238	0.239	
128	1.665	3.176	2.002	2.03	1.978	2.6	
256	14.701	26.647	18.071	18.254	16.267	16.355	
512	119.804	250.779	173.472	222.813	142.7	139.313	
1024	1192.731	2853.029	6194.853	6670.788	1195.495	1197.284	
2048	16606.03	26689.225	107537.622	106145.74	9687.757	9751.603	

Table 1: Performance comparison based on matrix size and density

4.2 Performance Graphs

The following graphs illustrate the performance differences between sparse and block multiplication at varying densities. These visualizations help to further highlight the impact of density on sparse multiplication and the consistency of the block method:

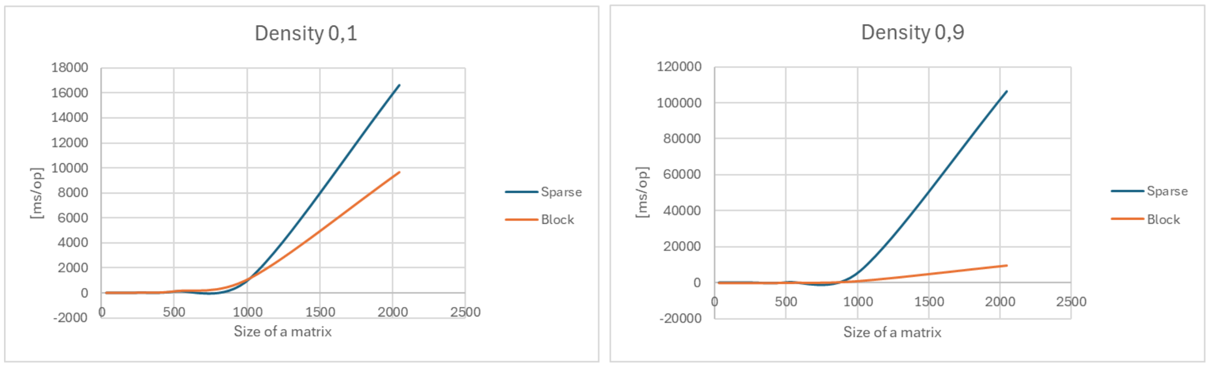

Figure 1: Performance of Sparse and Block Multiplication at Density 0.1 (left) and Density 0.9 (right)

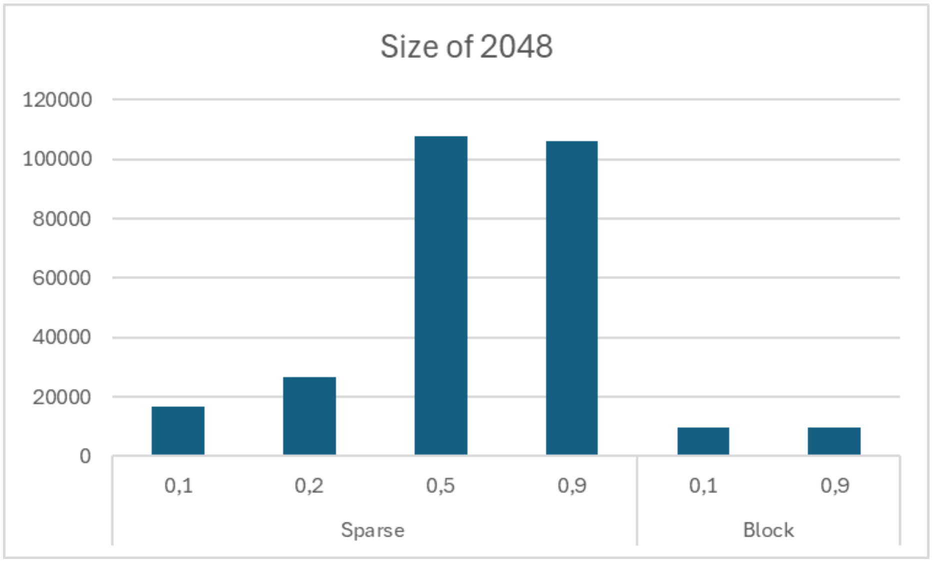

Figure 2: Performance Comparison for Matrix Size 2048 at Different Densities

Final Remarks: The block multiplication method proves to be the most efficient across different densities and matrix sizes, while the sparse optimization is only beneficial for low-density matrices. In situations with high density, a traditional or block-based approach is recommended over sparse multiplication due to reduced overhead and faster performance.

Part VII

Future Work

In future stages of this research, we plan to implement and analyze the impact of multithreading on matrix multiplication performance. By leveraging parallel processing, particularly on large matrices, we expect to see further improvements in computation speed, especially for the block multiplication method, which could benefit significantly from concurrent processing of independent matrix blocks.

Additionally, we aim to explore the effectiveness of multithreading in conjunction with different matrix densities. This approach will allow us to examine whether multithreaded sparse multiplication can close the performance gap with block multiplication, particularly at intermediate densities where the current single-threaded sparse implementation shows reduced efficiency.

Implementing multithreading will help us determine the scalability of both methods in a multi-core environment, providing insights into optimal configurations for high-performance matrix operations. These findings will potentially extend the applications of optimized matrix multiplication methods in large-scale data processing and scientific computing tasks.