TASK 2. Performance Benchmark of optimized matrix multiplication
Big Data

Denys Kavkalo Gumeniuk

November 11, 2024

Abstract

This study investigates various optimization techniques for matrix multiplication, focusing on algorithmic improvements and performance analysis with both dense and sparse matrices. The basic matrix multiplication algorithm is optimized using Strassen's algorithm, loop unrolling, and cache optimization techniques.

Strassen's algorithm divides the matrix into smaller sub-matrices, recursively computing matrix products, while the unrolled cache-optimized approach utilizes block-based multiplication to improve memory access patterns and reduce cache misses. Additionally, sparse matrix multiplication is explored by introducing different sparsity levels, simulating real-world scenarios where most elements are zero. Benchmarks are conducted on dense matrices with sizes ranging from 10 to 1000, and sparse matrices with varying sparsity levels. The algorithms are evaluated in terms of execution time, memory usage, and scalability, identifying the maximum matrix size each approach can handle efficiently.

A comparative analysis is presented between dense and sparse matrix multiplication, emphasizing the impact of sparsity on computational costs and performance bottlenecks. The study provides insights into how algorithmic optimizations can significantly enhance matrix multiplication performance for both dense and sparse datasets.

1 Background

Matrix multiplication is a key operation across various computing fields, making its optimization essential, particularly in Big Data contexts where performance in speed and memory usage is crucial. This study benchmarks optimizations for dense and sparse matrix multiplication, the latter capitalizing on matrices with numerous zero elements to reduce computation.

Key techniques explored include Strassen's algorithm, which lowers the multiplication complexity from the typical $O(n^3)$ to approximately $O(n^{2.81})$, by recursively partitioning matrices into submatrices. Cache optimization methods, like block-based multiplication, improve memory access and reduce cache misses, while loop unrolling minimizes control overhead to accelerate CPU cycles.

By comparing dense and sparse matrix multiplication approaches, this study reveals practical insights into how algorithmic improvements can enhance the efficiency of matrix multiplication for both dense datasets and sparsity-heavy applications.

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, especially for large datasets. This project focuses on optimizing matrix multiplication through techniques like Strassen's algorithm, loop unrolling, and cache optimization. Strassen's algorithm reduces the typical multiplication complexity, while loop unrolling and cache optimizations improve memory access efficiency.

Additionally, sparse matrix multiplication is explored, which limits computations to non-zero elements, significantly reducing costs for matrices with high sparsity. By implementing and comparing

these optimizations, this study aims to identify approaches that enhance performance across both dense and sparse matrices.

3 Methodology

The experiments will be conducted using Java within the IntelliJ development environment on a Windows operating system, focusing on execution time and memory usage to assess the efficiency of various matrix multiplication implementations. Matrix sizes of 10, 100, and 1000 will be used to evaluate the algorithms under different computational loads.

Each experiment will include a warmup phase with one preliminary iteration lasting two seconds to mitigate initial performance fluctuations. Following the warmup, two iterations of two seconds each will be executed for every matrix size. The benchmarks will evaluate multiple multiplication techniques: a basic dense matrix multiplication, Strassen's algorithm, an optimized approach using loop unrolling and cache management, and sparse matrix multiplication for varying sparsity levels (0.1, 0.5, and 0.8). This setup allows for a comprehensive performance comparison across dense and sparse datasets under consistent conditions.

The data collected from these tests will provide insights into the performance of each algorithm in handling matrices with varying densities and sizes, facilitating a comparative analysis of matrix multiplication techniques and optimizations.

3.1 Java

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/matrix-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

In the following section, the results of the benchmarking tests are presented. Each matrix multiplication algorithm, including basic multiplication, Strassen's algorithm, optimized multiplication with loop unrolling and cache optimization, and sparse matrix multiplication, was evaluated across various matrix sizes and sparsity levels. The performance metrics, specifically execution time and memory allocation rate, provide insights into the efficiency and scalability of each method under different computational conditions.

4.1 Implementation

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.

4.2 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.

4.3 Matrix Generator for Dense and Sparse Matrices

To support testing and benchmarking, the `MatrixGenerator` class creates both dense and sparse matrices. The `generateDenseMatrix` method fills a matrix entirely with random values. In contrast, `generateSparseMatrix` utilizes a sparsity parameter to control the proportion of non-zero elements. By applying sparsity, it generates a `Map`-based representation where only non-zero elements are stored, which allows for reduced computational load in sparse matrix operations.

4.4 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.

4.5 Sparse Matrix Multiplication

In real-world applications, matrices often contain many zero elements, motivating the use of sparse matrix representations. The `SparseMatrixMultiplication` class implements an approach that efficiently handles sparse matrices represented as nested `Map` structures. Here, matrix elements are only multiplied if both corresponding elements are non-zero, reducing the number of operations significantly. This optimization leverages sparsity to minimize computation time and memory usage, providing a specialized solution for scenarios involving sparse data.

4.6 Strassen's Matrix Multiplication

Finally, the `StrassenMatrixMultiplication` class implements Strassen's algorithm, an advanced recursive technique that reduces the time complexity of matrix multiplication from $O(n^3)$ to approximately $O(n^{2.81})$. Strassen's algorithm divides the input matrices into submatrices, computing products and sums of these smaller matrices recursively. Padding is applied to ensure input matrices are of size $2^n \times 2^n$, a requirement for the recursive division. The matrix is then trimmed after multiplication to produce the final result. By reducing the number of multiplicative operations, Strassen's algorithm demonstrates efficiency improvements, particularly for larger matrices.

4.7 Benchmarking Setup

Benchmarking of these implementations is conducted using the Java Microbenchmark Harness (JMH), with configurations set to ensure reliable performance analysis. The matrix sizes tested include 10×10, 100×100, and 1000×1000, with each experiment run at varying sparsity levels to simulate diverse real-world conditions. The benchmarking code includes warmup and measurement phases to obtain stable metrics for execution time and memory usage, focusing on average performance across multiple iterations.

5 Results

Table 1: Basic Multiplication

Matrix Size	Execution Time (ms/op)	Memory Allocation Rate (MB/s)
10x10	0.001294	599.94
100x100	0.770	81.24
1000x1000	2671.51	2.76

Table 2: Optimized Multiplication

Matrix Size	Execution Time (ms/op)	Memory Allocation Rate (MB/s)
10x10	0.001145	676.88
100x100	0.714	87.73
1000x1000	1830.67	3.68

Table 3: Sparse Multiplication (Sparsity = 0.1)

Matrix Size	Execution Time (ms/op)	Memory Allocation Rate (MB/s)
10x10	0.026	1173.55
100x100	25.99	1149.20
1000x1000	92882.79	1090.07

Table 4: Sparse Multiplication (Sparsity = 0.5)

Matrix Size	Execution Time (ms/op)	Memory Allocation Rate (MB/s)
10x10	0.008	1195.20
100x100	10.02	930.79
1000x1000	30005.01	1031.51

Table 5: Sparse Multiplication (Sparsity = 0.8)

Matrix Size	Execution Time (ms/op)	Memory Allocation Rate (MB/s)
10x10	0.002	1392.15
100x100	2.26	759.02
1000x1000	3993.65	1193.19

Table 6: Strassen Multiplication

Matrix Size	Execution Time (ms/op)	Memory Allocation Rate (MB/s)
10x10	0.809	722.91
100x100	274.11	754.33
1000x1000	186469.83	471.76

5.1 Comparison and Observations

This section presents the results obtained from benchmarking various matrix multiplication techniques: basic multiplication, optimized multiplication, sparse multiplication at different sparsity levels, and Strassen's multiplication. The following parameters were measured:

- **Execution Time**: The time taken for each operation (in milliseconds per operation).
- **Memory Allocation Rate**: The rate of memory allocation (in MB per second).
- **Maximum Matrix Size Handled**: The largest matrix size each multiplication technique was able to handle efficiently.
- **Performance Comparison**: A comparison of the performance between dense and sparse matrices at different sparsity levels.
- **Bottlenecks or Performance Issues**: Identification of performance bottlenecks or issues observed during the tests.

5.2 Execution Time and Memory Allocation Rate

From the results, it is evident that different matrix multiplication techniques perform differently when tested with larger matrices. Both the basic and optimized multiplication methods handled matrix sizes up to 1000×1000 efficiently, although execution time increases significantly with larger matrices. For

sparse matrices, however, the execution time for a 1000×1000 matrix grows substantially, particularly at higher sparsity levels. Strassen's algorithm improves multiplication time for large matrices, but still faces a performance bottleneck for the 1000×1000 matrix size, where execution time rises drastically.

5.3 Maximum Matrix Size Handled

From the results, it is evident that the matrix multiplication techniques performed differently when tested with larger matrices. The basic and optimized multiplication methods handled matrix sizes up to 1000×1000 efficiently, with execution time increasing significantly as the matrix size grows. For sparse matrices, the execution time for a 1000×1000 matrix becomes substantially higher, particularly at higher sparsity levels. Strassen's algorithm, while improving the multiplication time for large matrices, still faces a performance bottleneck for the 1000×1000 matrix size, where the execution time rises drastically.

5.4 Performance Comparison Between Dense and Sparse Matrices

A notable observation from the results is the significant difference in performance between dense and sparse matrices, especially when varying sparsity levels are considered. Dense matrix multiplication (as seen with the basic and optimized methods) shows a gradual increase in execution time as matrix size grows, with memory allocation remaining relatively stable across different sizes. For sparse matrices, however, the performance is heavily influenced by the sparsity parameter.

5.5 Bottlenecks and Performance Issues

Despite the optimizations, several performance bottlenecks were observed during the benchmarking process:

- The **StrassenMatrixMultiplication** algorithm showed significant performance degradation with larger matrices (1000×1000), where execution time escalated to extreme levels. This is likely due to the recursive nature of the algorithm and the overhead of matrix padding and trimming.
- **Sparse matrix multiplication** also encountered performance issues for matrices of larger sizes with high sparsity levels. As sparsity increases, the number of non-zero entries decreases, but the complexity of managing sparse data structures in memory grows, making efficient memory allocation more challenging.
- Memory allocation for large matrices was also a limiting factor, particularly for sparse matrices, where the memory rate increased with larger matrix sizes and higher sparsity levels.

These observations highlight the complexity of optimizing matrix multiplication, and suggest that further improvements could be made by combining multiple techniques. Hybrid approaches for sparse matrix multiplication or additional enhancements to Strassen's algorithm may be necessary for large-scale computations.

6 Conclusion

In this work, various matrix multiplication techniques were explored and compared, including basic, optimized, sparse, and Strassen's multiplication algorithms. The performance of each technique was evaluated based on execution time, memory usage, and matrix size handling, while also considering the impact of sparsity on sparse matrix multiplication.

The results demonstrate that:

- The basic matrix multiplication algorithm, while simple, becomes increasingly inefficient as the matrix size grows, particularly for larger matrices such as 1000×1000. Both execution time and memory allocation rate increase significantly with larger matrices.

- Optimized multiplication methods, including loop unrolling, offer improved performance, reducing execution time and improving memory allocation rates for larger matrices. This optimization proves especially effective for matrices of size 1000×1000.
- Sparse matrix multiplication exhibits substantial performance gains when applied to matrices with high sparsity. As sparsity increases, both execution time and memory allocation rate decrease, making sparse matrix multiplication a highly efficient technique for large, sparse matrices.
- Strassen's algorithm provides improvements in execution time for dense matrices, particularly at smaller sizes (10×10 and 100×100), but struggles with larger matrix sizes due to its recursive nature and increased computational overhead.

Additionally, it was observed that the sparsity of the matrices plays a crucial role in the performance of sparse matrix multiplication. Higher sparsity levels result in significant reductions in execution time, although some trade-offs in memory allocation were noted, especially for matrices with sparsity levels of 0.8.

Although all techniques were able to handle 1000×1000 matrices, performance bottlenecks were observed as the matrix size and sparsity increased. Strassen's algorithm, in particular, showed notable performance degradation for larger matrices, indicating that further optimization or hybrid techniques may be needed for large-scale computations.

In conclusion, the study highlights the importance of selecting the appropriate matrix multiplication technique based on matrix size, sparsity, and the specific problem at hand. While basic methods are suitable for smaller matrices, advanced techniques such as optimized and sparse matrix multiplication, or Strassen's algorithm, should be considered for larger and more complex computations. Future research could focus on developing hybrid algorithms that combine the strengths of each technique to further improve performance.