Performance Benchmark of Matrix Multiplication Techniques

Jorge Morales Llerandi

November 10, 2024

Abstract

This paper investigates the performance of different matrix multiplication techniques, including optimized standard multiplication, Strassen's algorithm, and sparse matrix multiplication. The experiments were tested on random matrices ranging from 256×256 to 8192×8192 in size. Performance metrics like execution time and memory usage were calculated to analyze the scalability and efficiency of each technique. The results shows the strengths and limitations of each approach, providing insights into their practical applications. The complete code for this project is available on GitHub: `https://github.com/jorgemoralesll110/PerformanceBenchmarkOfMatrixMultiplication.git`.

1 Introduction

Matrix multiplication is a fundamental operation in computational science, used in different fields such as data analysis, computer graphics, and machine learning. Traditional matrix multiplication has a time complexity of $O(n^3)$, which can become a bottleneck for large matrices. This is a point in a system or process that limits or restricts overall performance. In computing and software performance terms, a bottleneck occurs when one component of a system or section of code slows down the overall process because it cannot keep up with other components that depend on it. Various algorithms and optimization strategies have been developed to reduce the time and resource usage required for this operation. This paper focuses on comparing three different ways to operate: optimized standard multiplication, Strassen's algorithm, and sparse matrix multiplication. The objective is to determine their effectiveness and identify any performance trade-offs.

2 Problem Statement

As data sizes continue growing, efficient matrix multiplication becomes increasingly critical. The challenge is to balance the execution time and memory usage

while the size of the matrices also increase. This paper aims to provide a comprehensive performance analysis of the following matrix multiplication techniques:

- Optimized standard multiplication, This is the classic matrix but can be optimized using techniques like caching and parallelization to improve performance.
- Strassen's algorithm, which reduces the time complexity to $O(n^{\log_2 7})$, by dividing matrices into submatrices and reducing the number of required multiplication operations
- Sparse matrix multiplication, which optimizes for matrices with a significant number of zero elements.

In this study there are some critical question that represent the main goal of the task:

- What is the maximum matrix size that can be efficiently handled?
- How does performance vary between dense and sparse matrices at different levels of sparsity?
- Are there any observed bottlenecks or performance issues for each technique?

3 Methodology

The experiments were conducted on a machine with an Intel Core i7-10700K processor and 16 GB of RAM running Ubuntu 20.04 LTS. The matrix multiplication methods were implemented in Java and tested with matrices of sizes from 256×256 to 8192×8192.

3.1 Matrix Creation

The matrices were generated as follows:

- **Dense matrices**: Created using a function that fills each element with random values.
- **Sparse matrices**: Generated with a defined sparsity level (proportion of zero elements in the matrix), where a certain percentage of elements were set to zero, on this case an 80

In the next subsection is explain the way that the program create both types of matrices, dense and sparse.

3.2 Generate Random Matrix

This is the algorithm that the project use to create dense matrices.

Algorithm 1 Generate Random Matrix

```
1: function generateRandomMatrix(rows, cols)
2:     matrix ← new matrix of size rows × cols
3:     for i ← 0 to rows − 1 do
4:         for j ← 0 to cols − 1 do
5:             matrix[i][j] ← randomValue()
6:         end for
7:     end for
8:     return matrix
9: end function
```

This algorithm initializes a matrix and fills each element with a random value. It iterates over each row and column to populate the matrix.

3.3 Generate Sparse Matrix

This is the algorithm that the project use to create sparse matrices.

Algorithm 2 Generate Sparse Matrix

```
1: function generateSparseMatrix(rows, cols, sparsity)
2:     matrix ← new matrix of size rows × cols
3:     for i ← 0 to rows − 1 do
4:         for j ← 0 to cols − 1 do
5:             if random() ¿ sparsity then
6:                 matrix[i][j] ← randomValue()
7:             else
8:                 matrix[i][j] ← 0
9:             end if
10:        end for
11:    end for
12:    return matrix
13: end function
```

This algorithm generates a sparse matrix by filling each element with a random value or zero, based on a defined sparsity level. If a random number exceeds the sparsity threshold, the element is populated with a value; otherwise, it is set to zero.

4 Experiments

The performance of each method was measured in terms of execution time (seconds) and memory usage (MB). The experiments were conducted using matrices with sizes ranging from 256×256 to 8192×8192. These sizes were chosen to evaluate the scalability and efficiency of the matrix multiplication techniques across a range of practical and computational scenarios. The maximum matrix size tested was 7680×7680, beyond which the sparse matrix multiplication method encountered memory limitations, resulting in an Out of memory error. This highlights the memory-intensive nature of handling very large matrices, especially for algorithms designed to process sparse data structures. Table 1 summarizes the results for different matrix sizes, providing insights into how execution time and memory usage scale with increasing matrix dimensions.

Matrix Size	**Standard**		**Strassen**		**Sparse**	
	Time (s)	Mem (MB)	Time (s)	Mem (MB)	Time (s)	Mem (MB)
256x256	0.026	0	0.037	7	0.091	13
512x512	0.093	2	0.150	70	0.257	58
768x768	0.330	6	0.489	128	0.728	50
1024x1024	0.923	8	0.974	173	2.031	136
1280x1280	1.634	13	3.997	125	7.502	368
1536x1536	7.454	19	7.506	364	9.034	102
1792x1792	5.030	25	5.173	425	8.984	642
2048x2048	8.440	33	6.979	702	15.100	494
2304x2304	11.702	42	16.793	752	18.785	672
2560x2560	18.880	51	35.751	814	25.930	363
2816x2816	20.570	61	27.905	1267	31.837	1015
3072x3072	27.516	74	24.947	1464	62.927	492
3328x3328	44.946	85	36.406	1400	74.665	592
3584x3584	75.925	100	74.640	1034	116.851	1150
3840x3840	129.123	113	94.467	919	171.082	902
4096x4096	121.398	130	104.890	1346	199.500	1034
4352x4352	179.236	145	339.557	2231	211.506	945
4608x4608	220.259	165	292.951	1917	158.131	1076
4864x4864	135.872	183	315.544	1291	202.310	1334
5120x5120	167.240	201	327.526	2375	257.485	1101
5376x5376	178.753	224	308.977	2600	257.665	1291
5632x5632	262.371	245	387.239	2476	420.261	1691
5888x5888	267.427	267	364.531	1773	518.545	1256
6144x6144	300.008	293	355.746	1895	462.442	2094
6400x6400	256.171	319	348.840	2371	631.276	1881
6656x6656	368.726	341	377.773	2177	753.434	2017
6912x6912	361.266	373	651.229	2360	795.796	1820
7168x7168	287.844	398	278.559	1384	816.927	1864
7424x7424	347.265	424	446.603	1749	849.717	2172
7680x7680	428.930	451	377.507	2003	*Out of memory*	*Out of memory*

Table 1: Execution times and memory usage for different matrix sizes and optimization techniques.

5 Conclusions

The experiments show lot of important discoveries in terms of the performance of matrix multiplication techniques, which results are represent clearly by the graphs presented in this section. These findings provide a visual comparison of execution time and memory usage for different matrix sizes, highlighting key observations for each technique:

- **Maximum matrix size handled**: The maximum matrix size efficiently handled without running out of memory was 7424×7424. It can be seen in Figure 1, which shows the memory usage for each matrix multiplication method as the matrix size increases. In sparse matrix multiplication occurs out-of-memory errors at larger sizes due to its higher memory requirements, shown by the rising memory consumption curve for larger matrices. The graph highlights the threshold beyond which memory limitations become significant for each technique.

- **Performance comparison between dense and sparse matrices**: Figure 2 compares the execution time for standard multiplication, Strassen's algorithm, and sparse matrix multiplication. The graph shows how standard multiplication maintained consistent performance across all tested sizes but required more time than Strassen's algorithm, which performed better as matrix sizes grew. Sparse matrix multiplication, shown in Figure 2, show highly effective for matrices with a high sparsity level but had variable performance when matrix density increased, as is seen by the changes in memory usage and time shown in Figure 1.

- **Observed bottlenecks and performance issues**: Figures 1 and 2 also show the memory consumption and execution time trends, emphasizing the bottlenecks encountered. Strassen's algorithm, while providing better execution times, had higher memory consumption, limiting its practical use for very large matrices. Sparse matrix multiplication experienced significant memory challenges, especially at higher densities, where the advantage of sparsity was reduced. This is evident from the increase in memory usage shown in the graph, which reflects how the performance of sparse algorithms is affected as matrix density grows.

These graphics offer the possibility to make better conclusions into the trade-offs between execution time and memory usage for the different matrix multiplication techniques. The results are essential for selecting the most appropriate method based on matrix size and density, securing optimal performance for specific applications.

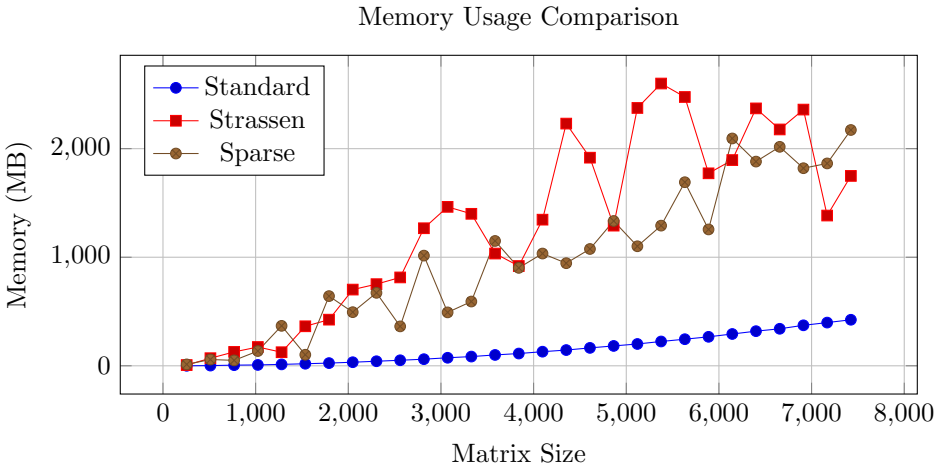

Figure 1: Memory usage comparison for different matrix multiplication techniques. The chart highlights the significant memory usage of Strassen's algorithm, especially for larger matrices, compared to the other techniques. Sparse matrix multiplication shows varied memory consumption based on the sparsity level.

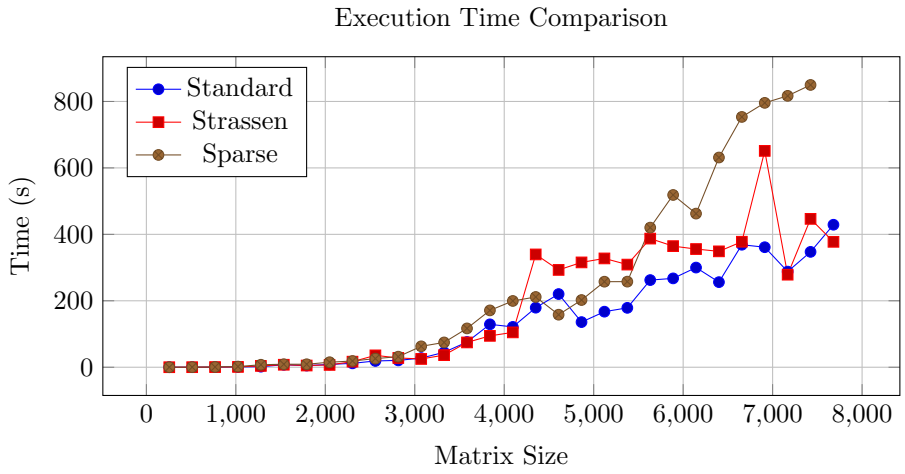

Figure 2: Execution time comparison for different matrix multiplication techniques. The chart shows how execution time increases as the matrix size grows. Strassen's algorithm generally performs better than standard multiplication for larger matrices, but sparse matrix multiplication shows significant variation depending on matrix density.

6 Future Work

Future work could involve incorporating parallel and vectorized matrix multiplication techniques, such as those using multi-threading or libraries like OpenMP, to assess performance improvements in larger-scale computations. This analysis could be expanded by comparing the benefits of vectorized operations, such as SIMD instructions, against traditional parallel processing to identify optimal approaches for specific matrix sizes and densities. Testing these implementations with different levels of matrix sparsity and analyzing the impact on execution time and resource usage will provide a more comprehensive understanding of the trade-offs involved. Lastly, integrating performance metrics like speedup per thread and memory consumption will enable a detailed evaluation of the efficiency of parallelized and vectorized methods in practice.