Performance Benchmark of Matrix Multiplication

Radosława Żukowska

November 10, 2024

Big Data

Grado en Ciencia e Ingenieria de Datos

Universidad de Las Palmas de Gran Canaria

Abstract

Matrix multiplication is a crucial operation in computer science, underpinning various applications such as machine learning, computer graphics, and scientific computing. Despite the simplicity and ease of implementation of the naive matrix multiplication algorithm, its cubic time complexity $O(N^3)$ makes it impractical for large matrices due to the significant computational cost involved.

This paper investigates several optimization strategies to improve matrix multiplication performance, particularly focusing on Java implementations. Key optimizations explored include Strassen's algorithm, which reduces the number of required multiplications through a divide-and-conquer approach, and loop unrolling, which enhances performance by minimizing loop control overhead. Additionally, the paper addresses optimizations tailored for sparse matrices, leveraging their unique structure to bypass unnecessary operations involving zero elements.

The analysis evaluates the impact of these optimization techniques on execution time and memory usage across different matrix sizes and sparsity levels. Results demonstrate that while Strassen's algorithm initially encounters overhead issues that make it slower for smaller matrices, adjustments that combine the naive algorithm for small cases and Strassen's for larger ones result in substantial performance gains. Loop unrolling proves effective for moderate improvements, while the tailored approach for sparse matrices significantly reduces execution time when the proportion of zero elements is high.

This study concludes that employing a combination of these optimization techniques can greatly enhance the performance and efficiency of matrix multiplication, particularly in applications involving large or sparse matrices. The findings provide valuable insights for selecting the most suitable optimization strategies for specific computational scenarios.

Contents

1 Introduction

Matrix multiplication is a fundamental operation in computer science with a wide range of applications, including machine learning, computer graphics and scientific computing. Given the importance and widespread use of matrix multiplication, optimizing this operation is critical for improving performance across a variety of applications. The naive matrix multiplication algorithm, while simple and easy to implement, is inefficient for large matrices due to its cubic time complexity $O(N^3)$. As the size of the matrices grows, the computational cost increases dramatically, making the naive approach impractical for large-scale problems.

This paper investigates and compares several optimization strategies aimed at improving the performance of matrix multiplication. Specifically, it focuses on algorithmic improvements, such as Strassen's algorithm, which reduces the number of multiplications required[6], and loop unrolling techniques, which reduce overhead in inner loops by performing multiple operations in a single iteration. It particularly focuses on the implementation of the algorithms in Java. While Java is generally slower than languages like C, it offers advantages in terms of ease of implementation and maintainability.[1]

Furthermore, this paper examines the use of sparse matrices, which contain a significant number of zero elements. Sparse matrices can offer substantial performance improvements over dense matrices because they allow for specialized storage and multiplication techniques that bypass unnecessary operations on zero elements. The impact of matrix sparsity on performance will also be explored by testing matrices with varying sparsity levels.

The goal of this research is to provide a comprehensive evaluation of how different optimization techniques impact both execution time and memory usage, especially as matrix sizes increase and sparsity levels vary.

2 Problem statement

The simplest approach to do matrix multiplication is the naive algorithm. as shown in Algorithm 1. However, this method involves three nested loops, resulting in a time complexity of $O(N^3)$. As the size of the matrices increases, the time required to perform the multiplication grows rapidly, making this approach highly inefficient. While small improvements, such as reordering operations to optimize cache usage, can enhance performance to some extent, they do not solve the core inefficiency of the naive method. Therefore, there is a need to explore alternative approaches that optimize matrix multiplication, improving both time and space complexity.

Furthermore, the naive matrix multiplication algorithm becomes especially inefficient when most of the elements in the table are zeros. Even though the product of them with any other number will still be 0, the algorithm goes through everything unnecessarily. Therefore, for the sparse matrices it is better to adjust algorithm so that it only multiplies the elements that are non-zeros.

Algorithm 1 Naive Matrix Multiplication Algorithm

```
0: procedure MULTIPLY($A, B$)
0:   $n \leftarrow$ length of $A$
0:   $C \leftarrow$ new $n \times n$ matrix initialized to 0
0:   for $i \leftarrow 0$ to $n - 1$ do
0:     for $j \leftarrow 0$ to $n - 1$ do
0:       $C[i][j] \leftarrow 0$
0:       for $k \leftarrow 0$ to $n - 1$ do
0:         $C[i][j] \leftarrow C[i][j] + A[i][k] \cdot B[k][j]$
0:       end for
0:     end for
0:   end for
0:   return $C$
0: end procedure=0
```

3 Optimization methods to multiply matrices

3.1 Loop unrolling

Loop unrolling is an optimization technique aimed at reducing the overhead of loop control, which helps to speed up an algorithm. By decreasing the number of iterations and minimizing the checks and updates to the loop variable, loop unrolling can significantly enhance performance. In the context of matrix multiplication, this optimization is applied to the innermost loop of the naive matrix multiplication algorithm.

In this approach, instead of processing one element at a time within each loop iteration, multiple elements are processed simultaneously. Specifically, the loop is "unrolled" by a factor of 4, meaning that instead of performing four separate loop iterations for four elements, we handle all four elements within a single iteration. This reduces the number of loop control checks (such as incrementing the loop index and performing comparisons) from four to one per iteration, which should lead to a reduction in execution time and more efficient use of the CPU.

The step parameter (k) determines how many elements are processed at once, and in this case, it has been chosen to be 4. This choice allows for the processing of four elements per iteration, optimizing the computation by decreasing the total number of iterations and the overhead associated with the loop control logic.[7]

Algorithm 2 Matrix Multiplication with Loop Unrolling

```
0: procedure MULTIPLY($A, B$)
0:   $n \leftarrow$ length of $A$
0:   $C \leftarrow$ new $n \times n$ matrix initialized to 0
0:   for $i \leftarrow 0$ to $n-1$ do
0:     for $j \leftarrow 0$ to $n-1$ do
0:       $C[i][j] \leftarrow 0$
0:       $k \leftarrow 0$
0:       for $k \leftarrow 0$ to $n-4$ step 4 do
0:         $C[i][j] \leftarrow C[i][j] + A[i][k] \cdot B[k][j]$
0:         $C[i][j] \leftarrow C[i][j] + A[i][k+1] \cdot B[k+1][j]$
0:         $C[i][j] \leftarrow C[i][j] + A[i][k+2] \cdot B[k+2][j]$
0:         $C[i][j] \leftarrow C[i][j] + A[i][k+3] \cdot B[k+3][j]$
0:       end for
0:     end for
0:   end for
0:   return $C$
0: end procedure=0
```

3.2 Strassen's algorithm

Strassen's algorithm is an optimization technique used to improve the efficiency of matrix multiplication by reducing the number of multiplication operations required.[6] Unlike the naive algorithm, which has a cubic time complexity (O(N^3)) due to three nested loops, Strassen's algorithm follows a divide-and-conquer approach to lower the overall computational cost.

In this method, each matrix is split into four sub-matrices, which are then recursively multiplied and combined in a way that reduces the number of multiplications needed from eight (as in the naive approach) to seven for each recursive step. This significant reduction lowers the time complexity to approximately O($N^{2.81}$), making it more efficient for large matrices.

Algorithm 4 works by computing intermediate matrices (P1 to P7) using combinations of additions and subtractions, followed by recursive multiplications[5]. These intermediate results are then used to reconstruct the final sub-matrices that form the product matrix. Although Strassen's method offers better performance asymptotically, it introduces some overhead in terms of matrix partitioning and merging, and it is most effective when the matrix size is large or when tailored implementations handle the additional complexity. For best results the matrices' sizes should be a power of 2.

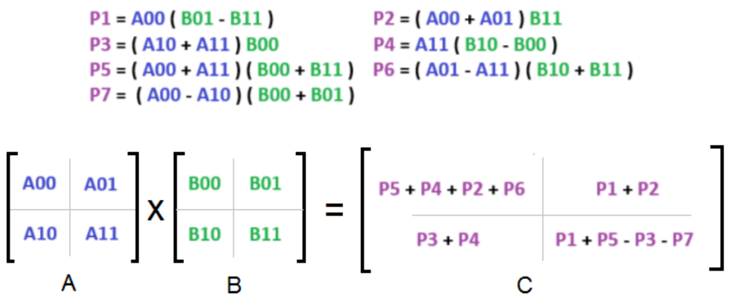

Figure 1: Graphical representation of Strassen's algorithm

Algorithm 3 Strassen's Matrix Multiplication Algorithm

```
0: procedure MULTIPLY($A, B$)
0:   $n \leftarrow$ length of $A$
0:   $C \leftarrow$ new $n \times n$ matrix initialized to 0
0:   if $n == 1$ then
0:     $C[0][0] \leftarrow A[0][0] \cdot B[0][0]$
0:   else
0:     Partition $A$ into submatrices $A_{00}$, $A_{01}$, $A_{10}$, $A_{11}$
0:     Partition $B$ into submatrices $B_{00}$, $B_{01}$, $B_{10}$, $B_{11}$
0:     $P_1 \leftarrow \text{Multiply}(A_{00}, \text{Subtract}(B_{01}, B_{11}))$
0:     $P_2 \leftarrow \text{Multiply}(\text{Add}(A_{00}, A_{01}), B_{11})$
0:     $P_3 \leftarrow \text{Multiply}(\text{Add}(A_{10}, A_{11}), B_{00})$
0:     $P_4 \leftarrow \text{Multiply}(A_{11}, \text{Subtract}(B_{10}, B_{00}))$
0:     $P_5 \leftarrow \text{Multiply}(\text{Add}(A_{00}, A_{11}), \text{Add}(B_{00}, B_{11}))$
0:     $P_6 \leftarrow \text{Multiply}(\text{Subtract}(A_{01}, A_{11}), \text{Add}(B_{10}, B_{11}))$
0:     $P_7 \leftarrow \text{Multiply}(\text{Subtract}(A_{00}, A_{10}), \text{Add}(B_{00}, B_{01}))$
0:     $C_{00} \leftarrow \text{Add}(\text{Subtract}(\text{Add}(P_5, P_4), P_2), P_6)$
0:     $C_{01} \leftarrow \text{Add}(P_1, P_2)$
0:     $C_{10} \leftarrow \text{Add}(P_3, P_4)$
0:     $C_{11} \leftarrow \text{Subtract}(\text{Subtract}(\text{Add}(P_5, P_1), P_3), P_7)$
0:     Merge $C_{00}$, $C_{01}$, $C_{10}$, and $C_{11}$ into $C$
0:   end if
0:   return $C$
0: end procedure=0
```

4 Sparse matrices

Sparse matrices are those in which most of the elements are zero. Storing such matrices in the standard dense format is inefficient in terms of memory usage. A more efficient approach is to store only the positions and values of the non-zero elements, significantly reducing the space required, especially as the matrix becomes sparser.

Given their specialized storage format, the multiplication of sparse matrices differs from that of dense matrices. In sparse matrix multiplication, calculations are only performed at the locations of non-zero elements. This targeted approach allows for the use of specialized algorithms that iterate solely through the relevant non-zero positions, optimizing both time and computational resources.[4].

Algorithm 4 Sparse Matrix Multiplication Algorithm

```
procedure MULTIPLY(other)
  other $\gets$ other.transpose()
  result $\gets$ new SparseMatrix(row * col, row, other.row)
  apos $\gets 0$
  while apos $<$ len do
    r $\gets$ data[apos][0]
    bpos $\gets 0$
    while bpos $<$ other.len do
      c $\gets$ other.data[bpos][0]
      tempa $\gets$ apos
      tempb $\gets$ bpos
      sum $\gets 0$
      while tempa $<$ len and data[tempa][0] $=$ r and tempb $<$ other.len and other.data[tempb][0] $=$ c do
        if data[tempa][1] $<$ other.data[tempb][1] then
          tempa $\gets$ tempa $+ 1$
        else if data[tempa][1] $>$ other.data[tempb][1] then
          tempb $\gets$ tempb $+ 1$
        else
          sum $\gets$ sum $+$ data[tempa][2] * other.data[tempb][2]
          tempa $\gets$ tempa $+ 1$
          tempb $\gets$ tempb $+ 1$
        end if
      end while
      if sum $\neq 0$ then
        result.insert(r, c, sum)
      end if
      while bpos $<$ other.len and other.data[bpos][0] $=$ c do
        bpos $\gets$ bpos $+ 1$
      end while
    end while
    while apos $<$ len and data[apos][0] $=$ r do
      apos $\gets$ apos $+ 1$
    end while
  end while
  return result
end procedure=0
```

5 Methodology

All experiments are conducted on the same machine with parameters shown in Table 1. For each algorithm optimizing and for the naive algorithm the tests will be conducted on the same matrices, all square with sizes from 64x64 to 1024x1024 using Java Microbenchmark Harness (JMH) and collecting data about average time per operation and memory usage with profiler gc and stack. The matrices are filled with random values with the same random seed to ensure comparable results.

System information	**Details**
Operating System Name	Windows 11 Professional
Operating System Version	23H2
Processor	Intel(R) Core(TM) Ultra 7 155H
RAM	32GB
SSD Storage	474 GB
Java version	OpenJDK 21.0.2

Table 1: Test environment parameters

For testing the maximum size each optimized version of the naive algorithm can handle, in terms of memory used are simple runs of the algorithm of some matrix and in terms of reasonable execution time, runs with JMH with only one warm up iteration and measure iteration.
For testing sparse matrices, the matrices are taken from the original Harwell-Boeing collection.[3] For the reason that there are both with double values or just with the positions of the non-zero elements matrices, I decided to uniform the calculation by accepting that every position that is non-zero contains number 1. Matrix multiplication for sparse matrices for testing instead of two random matrices like in the case of testing optimization versions of naive algorithm, uses same matrices from the mentioned above collection and multiplies each matrix by itself as to avoid having to find matrices with the same size and sparsity level.
For checking the impact of the sparsity of the matrix on the performance of the algorithm used are matrices with sizes varying between 180 and 210. In that way we are able to use the matrices from the Harwell-Boeing collection at the same time keeping considerable comparability between execution of the implementation of the algorithm on these matrices.

6 Encountered problems

6.1 Strassen's algorithm

At first the performance of Strassen's algorithm was really poor even compared to the naive approach. The reason was a high overhead of the algorithm for small matrices that were accumulated with each recursion. This overhead does not exist in naive algorithm therefore to improve the efficiency for smaller matrices it is better to use the naive one and for the bigger one use the Strassen's algorithm.[2] After this adjustment the performance has significantly improved.

6.2 Sparse matrices

One challenge I faced was determining the appropriate maximum size for storing the result of sparse matrix multiplication. When reading a matrix from a file, we know in advance how many non-zero elements it contains. However, during multiplication, this number is not known beforehand. Initially, I assumed the result would have at most the same number of non-zero elements as one of its factors, but this turned out to be incorrect. This holds true only for matrices with non-zero elements along the diagonal—where the product remains diagonal—but changes when the non-zero elements are arranged differently. In such cases, the resulting matrix can be denser than the original matrices.
My next idea was to reserve the maximum possible space, which would be the total size squared. I realized that certain sparse matrices with unevenly distributed elements, such as those with non-zero values forming a cross pattern, could result in a fully dense matrix when multiplied by themselves. This prompted me to increase the reserved space to the product of rows and columns. However, when testing this approach on large matrices, I encountered Java heap space errors due to insufficient memory. Despite these matrices being close to diagonal, the high memory requirement led to running out of space on the heap.
Then I developed two version of implementation of the algorithm. One using int[][] in which for the multipliers the size is specified as their actual number of non-zero elements whereas for the product it is maximum size: columns$*$rows

7 Experiments' results

7.1 Optimization techniques

The maximum size that can be handled by each algorithm according to memory was obtained by checking increasingly larger sizes, until finding the maximum with which we did not get an error about exceeding Java heap space.

Algorithm	**Maximum size to handle for memory**	**Maximum size checked**	**Time[ms]**
Naive algorithm	18720	2048	243607.911
Loop unrolling	18720	2500	445015.188
Strassen's algorithm	12810	5000	113655.510

Table 2: Maximum size of a square matrices that the algorithm can handle

From the Table 2 it is evident that Strassen's algorithm performs significantly better than the other two, even with bigger matrices, the time it takes to multiply them is lower than the other methods.

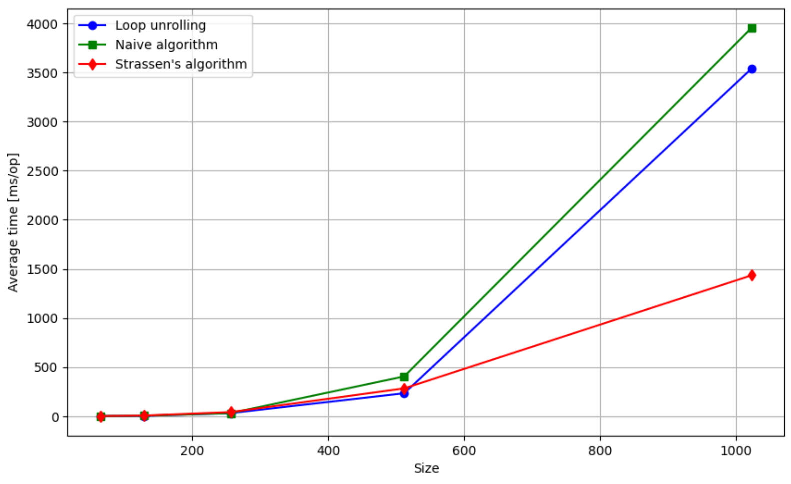

Figure 2: Comparison of average time per operation for different sizes of square matrices for matrix multiplication algorithms

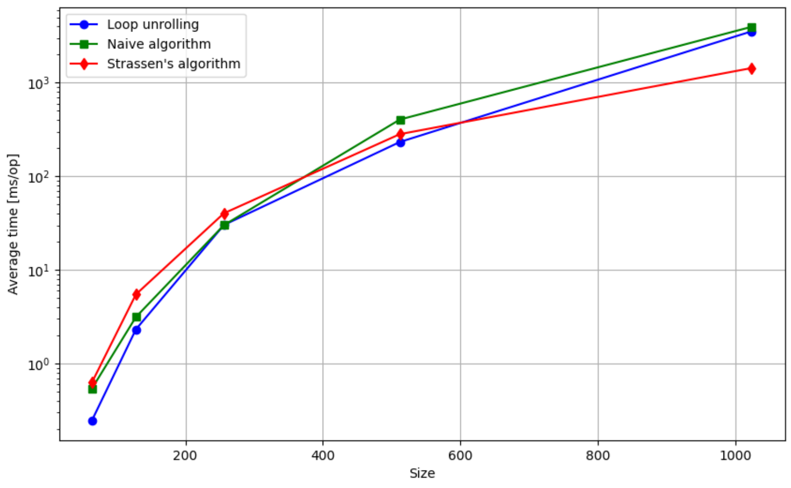

Figure 3: Comparison of average time per operation for different sizes of square matrices for matrix multiplication algorithms on logarithmic scale of y-axis

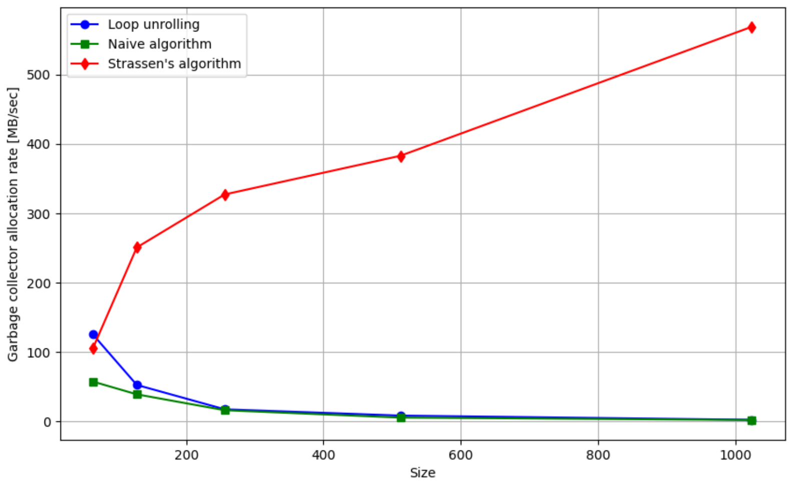

Figure 4: Comparison of garbage collector allocation rate for different sizes of square matrices for matrix multiplication algorithms

The plots demonstrate that Strassen's algorithm outperforms the others in terms of execution time, particularly as the matrix size increases, where its efficiency becomes more pronounced. For smaller matrices, the naive algorithm and loop unrolling deliver better performance, as shown more clearly in the plot with a logarithmic time axis in Figure 3. Among these, loop unrolling slightly surpasses the basic naive algorithm. Although Strassen's algorithm defaults to the naive approach for matrices smaller than 127x127 due to overhead, leading to initially slower multiplication compared to the naive algorithm and loop unrolling, its advantages become increasingly evident as matrix sizes grow.

7.2 Sparse matrices

First experiment for sparse matrices measured different metrics for increasing size of the matrices for two implementation of sparse matrix multiplication algorithm. First to store the data about the matrix used int[][] where the size of the maximum size of the matrix had to be decided on the initialization, whereas the second one stored it using java.util.ArrayList which provided more flexibility in terms of space.

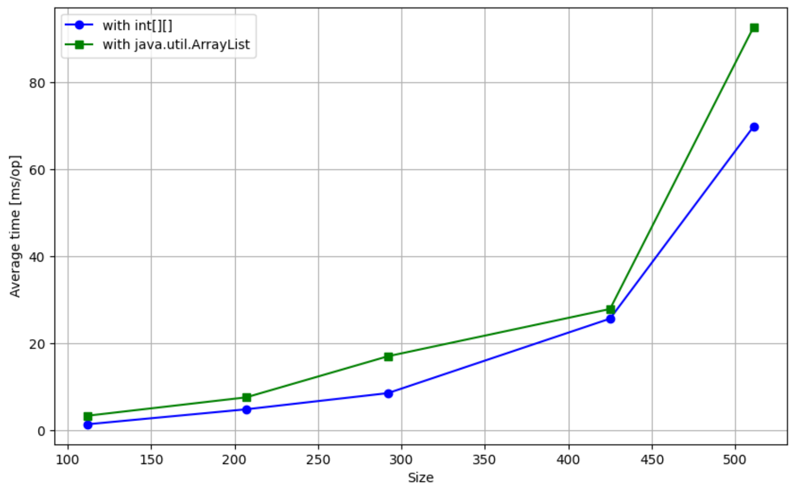

Figure 5: Comparison of average time per operation for different sizes of square matrices for two implementations of sparse matrix multiplication algorithm

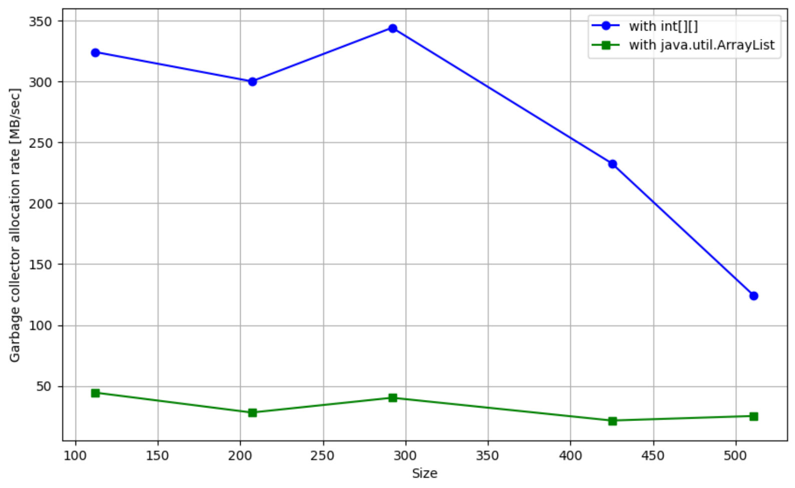

Figure 6: Comparison of garbage collector allocation rate for different sizes of square matrices for two implementations of sparse matrix multiplication algorithm

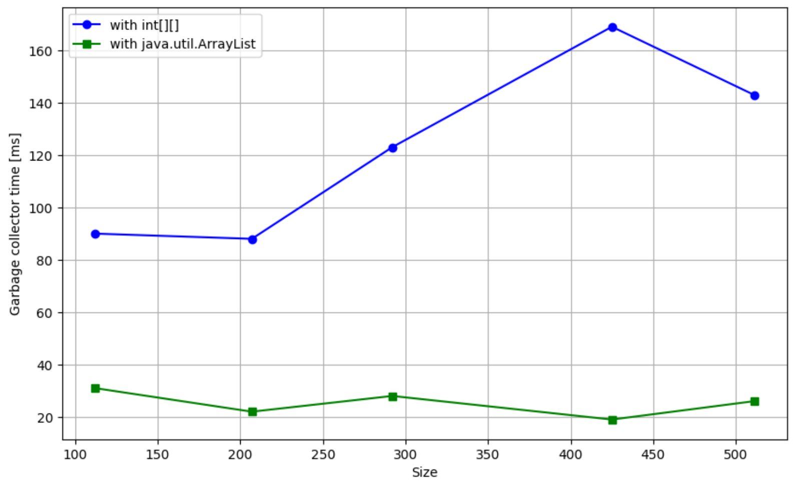

Figure 7: Comparison of garbage collector time for different sizes of square matrices for two implementations of sparse matrix multiplication algorithm

As the plots show using int[][] is more efficient in terms of both execution time and memory. Therefore for the following experiments this implementation is used. In these experiments tested is the impact of sparsity on the sparse matrix multiplication. Sparsity refers to the percentage of 0s in the matrix. The Table 3 shows the precise results and the following plots present the metrics visually.

Size	Number of nonzero elements	Sparsity [%]	Time [ms/op]
193x193	3493	90.62	10.083±2.557
183x183	1069	96.81	6.079±6.128
183x183	1000	97.01	11.445±24.252
185x185	975	97.15	3.591±0.216
199x199	701	98.22	3.845±6.405
207x207	572	98.67	2.886±0.945

Table 3: Comparison of time in seconds per operation for varying sparsity of the matrix

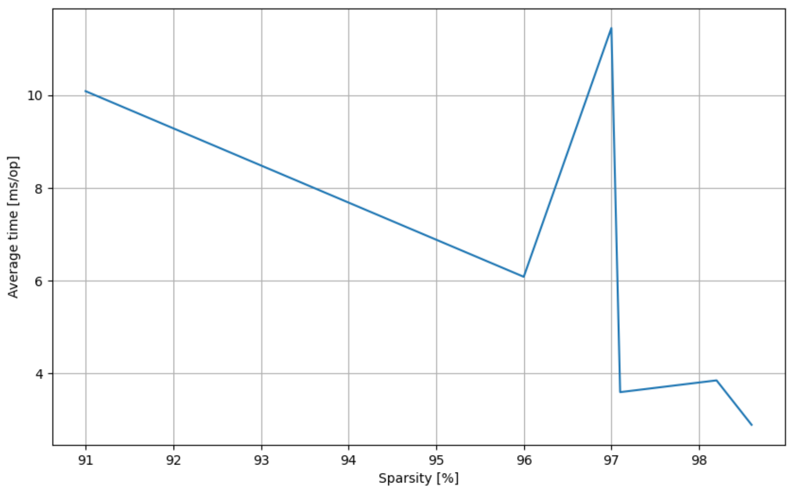

Figure 8: Average time per operation for different sparsity of the matrix in sparse matrix multiplication

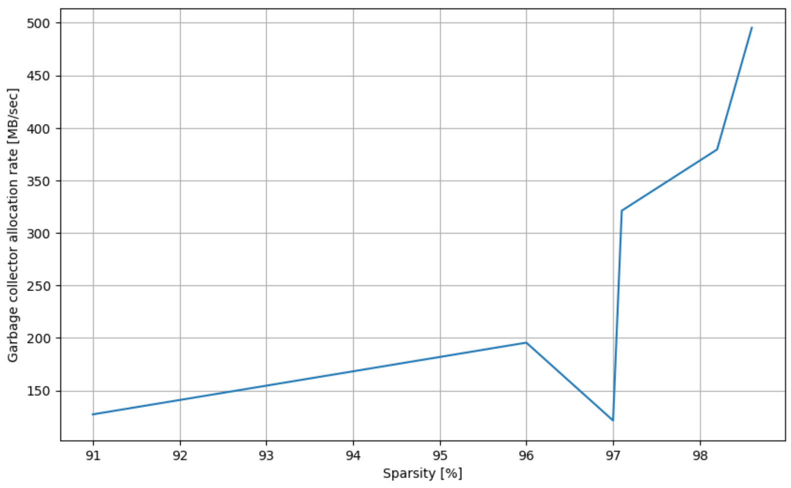

Figure 9: Garbage collector allocation rate for different sparsity of the matrix in sparse matrix multiplication

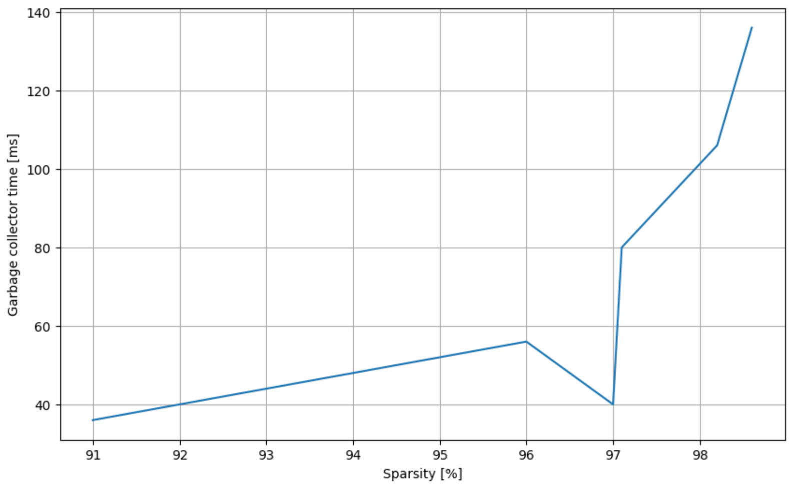

Figure 10: Garbage collector time for different sparsity of the matrix in sparse matrix multiplication

The results generally show a clear trend, with higher sparsity leading to better performance, as expected. Matrices with a greater number of zero elements result in faster computation times and more efficient memory usage across the algorithms. There is one notable outlier with the 183x183 matrix, which has a sparsity of 97.01%. However, this anomaly appears to be an isolated case, as the results for matrices with slightly lower and higher sparsities follow the expected pattern of improved performance with increasing sparsity.

8 Conclusion

This study addressed the optimization challenges associated with matrix multiplication algorithms, with a particular focus on Strassen's algorithm, loop unrolling, and sparse matrix handling. Initially, Strassen's algorithm faced performance issues due to high overhead from recursive calls. However, by adjusting the approach to use the naive algorithm for smaller matrices and Strassen's algorithm for larger ones, significant performance improvements were achieved. Furthermore, optimizing the naive algorithm through loop unrolling enhanced its efficiency, though Strassen's algorithm ultimately demonstrated the greatest improvements in performance, especially for larger matrices. In the case of sparse matrix multiplication, the proposed solution proved effective by eliminating unnecessary calculations for zero elements, leading to a more efficient use of resources. The impact of sparsity was evident, as matrices with a higher percentage of zero elements resulted in reduced execution times, underscoring the importance of sparsity in optimizing matrix multiplication. Overall, the findings suggest that combining these optimization techniques can significantly improve the performance and efficiency of matrix multiplication, particularly for large and sparse matrices.

9 Future work

For further exploration, it would be valuable to explore additional optimization techniques to further enhance the performance of matrix multiplication algorithms. While improvements have already been made, there is still potential to better utilize memory and minimize overheads. Specifically, focusing on optimizing memory usage could lead to more efficient handling of large matrices, reducing the risk of memory overflows and improving overall computational efficiency. Moreover, incorporating parallelism into the algorithms could yield significant performance gains, particularly for algorithms like Strassen's, which already benefits from reduced multiplicative complexity. By leveraging parallel computing, whether through multi-threading or distributed computing techniques, Strassen's algorithm could be further optimized to handle larger matrices more effectively. These advancements could greatly enhance the scalability and performance of matrix multiplication, particularly for high-dimensional or sparse matrices in real-world applications.

10 Appendices

GitHub repository

https://github.com/radoslawazukowska/ulpgc-big-data-ind

Here is whole code used in experiments and in obtaining the results together with the results and plots. Everything for this paper can be found in 'Assignment 2' directory.

References

[1] G. Ballard et al. "Improving the numerical stability of fast matrix multiplication". In: *SIAM Journal on Matrix Analysis and Applications* 37.4 (2016), pp. 1382–1418. DOI: 10.1137/15m1032168.

[2] Domingo Gimenez Canovas. "Revisiting Strassen's Matrix Multiplication for Multicore Systems". In: *Annals of Multicore and GPU Programming: AMGP* 4.1 (2017), pp. 1–8. URL: https://documat.unirioja.es/descarga/articulo/6426989.pdf.

[3] Iain Duff, Roger Grimes, and John Lewis. "Sparse Matrix Problems". In: *ACM Trans. on Mathematical Software* 14.1 (1989), pp. 1–14. URL: https://sparse.tamu.edu/hb.

[4] GeeksForGeeks. *Operations on Sparse Matrices*. Aug. 2022. URL: https://www.geeksforgeeks.org/operations-sparse-matrices/.

[5] Javatpoint. *Strassen's Matrix Multiplication*. URL: https://www.javatpoint.com/strassens-matrix-multiplication.

[6] T. Kouya. "Accelerated multiple precision matrix multiplication using strassen's algorithm and winograd's variant". In: *JSIAM Letters* 6.0 (2014), pp. 81–84. DOI: 10.14495/jsiaml.6.81.

[7] Goran Velkoski, Marjan Gusev, and Sasko Ristov. "The performance impact analysis of loop unrolling". In: *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 2014, pp. 307–312. DOI: 10.1109/MIPRO.2014.6859582.