Comparative Analysis of Matrix Multiplication in Sparse Format: Matrix Density Effects

SUSANA SUÁREZ MENDOZA[1]

[1] *University of Las Palmas de Gran Canaria, Data Science and Engineering Degree*
[1] *Las Palmas, Canarias, 35001, Spain*

Compiled October 28, 2023

In the context of this study, we face the challenge of evaluating and comparing the performance of matrix multiplications in two main formats: sparse matrices and dense matrices. Matrix multiplication is a fundamental operation in many computational applications, from numerical simulation to machine learning. However, the performance of this operation can vary significantly depending on the matrix format and data density.

In this paper, we perform extensive experiments to evaluate the performance of matrix operations. We varied the size of the matrices (N) and the densities of the sparse matrices, and measured the execution time on a specific hardware configuration. One of the most salient findings is the influence of the choice between dense and sparse matrices on computational efficiency in various applications. Our work provides valuable information for algorithm optimisation and matrix format selection in real-world applications.

In summary, our study sheds light on the advantages and disadvantages of matrix formats as a function of data and hardware characteristics, which is crucial for future developments in algorithms and computational applications.

Keywords: Sparse Matrix, CCS, COO, CRS, Multiplication

1. INTRODUCTION

Matrix multiplication is a fundamental operation in scientific computing and plays an essential role in a variety of fields, from optimisation and solving linear systems to artificial intelligence and machine learning. Efficiency in matrix multiplication is a constant goal in research and software development, as it can have a significant impact on the performance of applications and algorithms that rely on this operation. In this context, this article focuses on matrix multiplication in two main formats: sparse matrices and dense matrices. Sparse matrices are those in which most of the elements are zero, while dense matrices contain a large number of non-zero elements. The choice between these two formats can have an important impact on the performance of matrix operations.

Matrix multiplication in sparse format has been an area of constant interest in computational and mathematical research due to its importance in a wide variety of applications. Several authors have explored this issue from different perspectives, focusing on the efficiency and optimisation of matrix multiplication operations in sparse formats.

My contribution focuses on analysing and comparing the performance of sparse and dense matrix multiplication, taking matrix density into account. In particular, this study brings significant value to the field by providing a quantitative and comparative assessment, backed by rigorous experiments, that will help researchers and practitioners make informed decisions about which matrix format to use based on the specifics of their applications and algorithms. In addition, our results can have direct implications for optimising algorithms and improving performance in a variety of fields of computer and data science.

2. PROBLEM STATEMENT

To understand the context of this study, it is essential to consider the most common sparse array storage formats: COO (Coordinate Format), CRS (Compressed Row Storage) and CCS (Compressed Column Storage). These formats are vital for efficiently managing matrices with a high proportion of null elements in various computational applications.

- **COO (Coordinate Format):** In COO, non-null elements are stored in tuples containing their location by row, column and value. Although intuitive and easy to construct, it may not be efficient for large-scale operations, such as matrix multiplication, due to the lack of structure.

- **CRS (Compressed Row Storage):** In CRS, the matrix is stored in three separate arrays for non-null values, location columns and a row start index. This format focuses on compressing rows and is efficient for operations such as matrix-vector multiplication, especially with long, sparse rows.

- **CCS (Compressed Column Storage):** In CCS, the matrix is stored in a similar way to CRS, but focuses on compressing the columns. It is especially efficient for matrix multiplication operations when the columns are long and sparse.

These sparse array formats, which include COO, CRS and CCS, are used to reduce memory consumption and improve performance in arrays with many null elements. In this study, we evaluate how matrix multiplication performance in these formats is affected by matrix density. This will provide essential information to understand when and why one format may be more suitable than another depending on data characteristics and specific operations.

3. METHODOLOGY

To evaluate the performance of matrix multiplication, a project based on the java programming language has been implemented. Three different tests have been carried out which consist of measuring the number of times Java can perform an operation in one millisecond, as well as the time in milliseconds it takes to perform the various tests. On the one hand, the matrices used for some tests have been randomly generated with dimensions of power-of-2 increments from 64 to 1024 and on the other hand, matrices have been selected from the "SuiteSparse Matrix Collection" page.

All the code has been separated into production and tests, thus achieving code cleanliness and also, the execution of Benchmark tests to measure performance.

4. EXPERIMENTS

A. Hardware and IDE

The experiments were conducted using a computer system featuring an Intel Core i7 processor, 15.7 GB of available RAM, and a 64-bit Windows operating system. This choice of hardware provided a suitable platform to evaluate the performance of the different matrix multiplications. In addition, the algorithms were run in the IntelliJ integrated development environment using Maven's Benchmark dependency for proper performance testing.

B. Source Code

The code can be divided into the different interfaces and modules in order to generalize and decouple the different functions:

- **Matrix interface**: contains all the matrix models that we will be working with in the project: dense matrix, coordinate matrix, Compressed Row matrix and Compressed Column matrix. Figure 1 shows the attributes and dependencies of that module.

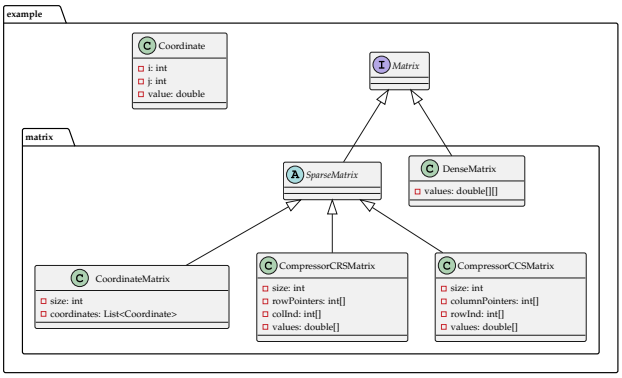

Fig. 1. Matrix Module Class Diagram

- **MatrixBuilder interface**: It contains the different operations to create each type of array from the coordinate format since it would be the center of our program. We can observe the different classes that build their format around the coordinate format, the MatrixMarketReader class that reads the files downloaded from the SuiteSparse Matrix Collections page and the CoordinateMatrixGenerator class that generates a random array with the dimension and density that we pass to it.

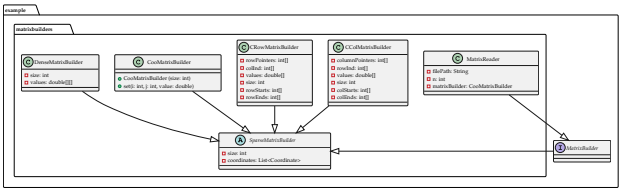

Fig. 2. Matrix Builder Module Class Diagram

- **MatrixMultiplication interface**: It contains the performance of the two main multiplications performed for the test: dense multiplication and sparse multiplication.

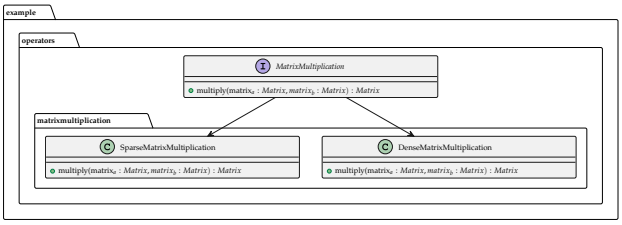

Fig. 3. Matrix Multiplication Module Class Diagram

- **Checker module**: an extra module whose function is to visualize the different formats using the Viewer class, write the results of the multiplication in two files and check that the multiplication methods work correctly using the following associative property of arrays::

  $(A \times B) \times C = A \times (B \times C)$

- **TestSuite module**: contains the class that performs the time measurement of the different array files in the SuiteSparse Collections and the class that performs the same measurement but for a single array.

- **MatrixOperations class**: initializes matrix multiplication based on the received object.

- **MatrixTransformations class**: Implement the different bidirectional transformations of the project.

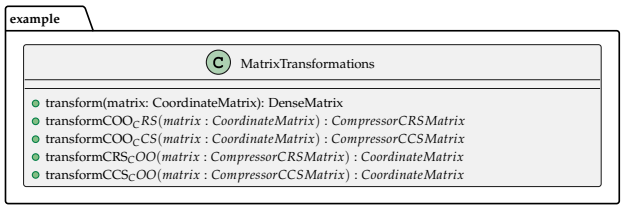

Fig. 4. Matrix Transformations Class Diagram

Finally, the core of the project is based on the following bidirectional transformations in order to improve performance since, for example, going from dense matrix to matrix in Compressed Row format is less costly if there is an intermediate step where a temporary matrix is created in Coordinates format. The complete code is in the references.

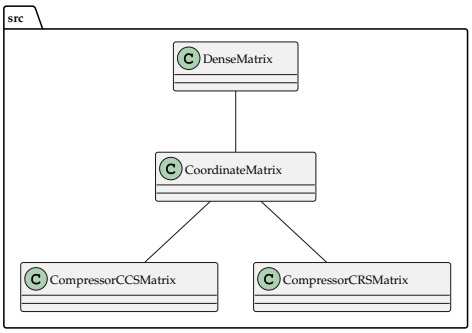

Fig. 5. Matrix Transformations Diagram

C. Tests and results

This section tries to implement and implement the different tests on the various situations. A total of four tests have been carried out, of which three have been measured.

C.1. Multiplication done right

When having different multiplications of such large matrices, it must be taken into account that the method implemented to perform the multiplication is correct. This is done by means of the Checker class which contains the implementation of the associative property of three matrices and its comparison between both results. Finally, in the various executions, a true result has been obtained, which means that the method works correctly.

C.2. Multiplications of matrices with different dimensions and densities

On the one hand, it is interesting to note that depending on the size and density of the corresponding matrix, different yields are achieved. Throughput is set as measured by the Benchmark dependency which is interpreted as operations in milliseconds. Therefore, the more trades you make, the better the performance. All this is manifested in the following figure, where this measure obtained from the multiplication of randomly generated matrices in Sparse format is represented. A high performance is clearly observed.

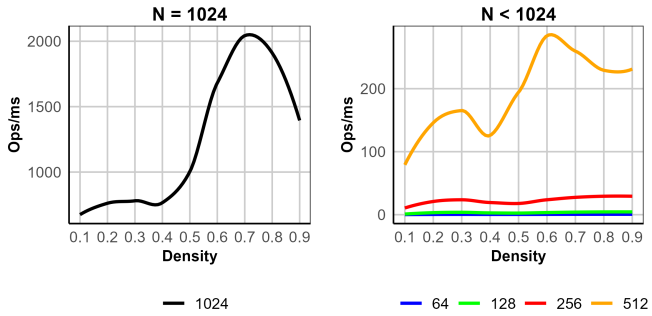

Fig. 6. Dense multiplication vs Sparse Multiplication

On the other hand, it is necessary to compare this performance according to how the multiplication is carried out: by means of dense or compressed matrices according to the different densities. It is noteworthy that the performance of the multiplication of dense matrices is much lower than that of the other type of multiplication.

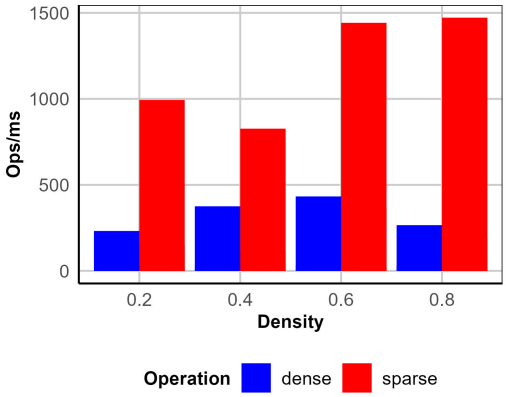

Fig. 7. Dense multiplication vs Sparse Multiplication

In addition, you only have to look at the distance between the two lines to see that despite increasing the density of the various matrices, the multiplication with the highest performance is the sparse.

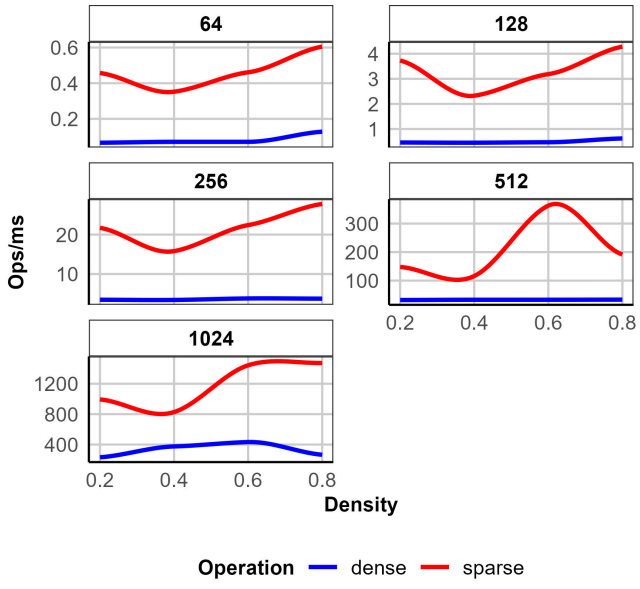

Fig. 8. Dense multiplication vs Sparse Multiplication with dimensions

C.3. Multiplications of matrices with different dimensions and densities from SparseMatrix Multiplication Collection

Once the performance for the small arrays has been checked, the execution time it takes to multiply arrays of the SuiteSparse Matrix Collections page included in the references has been measured, reaching a matrix of 90,000 by itself. These matrices have densities less than 0.01 since these matrices with these dimensions are not feasible to handle in dense format.

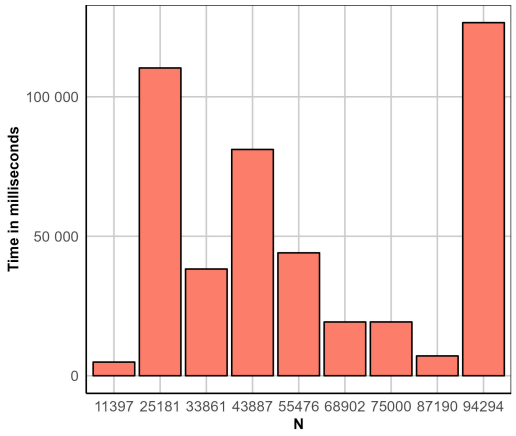

Fig. 9. Time it takes to perform sparse multiplication with much larger dimensions

Thanks to the discovery of the sparse format, this type of multiplication can be done in just seconds, which could not be imagined if you were working with dense matrices.

C.4. Multiplication with 525,825 rows and columns

Finally, a multiplication of a matrix of dimension 525.825 x 525.825 with a total of 2.100.225 non-zero elements has been performed. The density of this matrix is therefore low, namely 0,000007. This type of matrix, despite being of such a low density, would not be possible to multiply in dense format.

The time it took to multiply was 4.819.888 milliseconds, which is approximately one and twenty hours. Quite a remarkable time for a large-scale matrix.

5. CONCLUSIONS

Based on previous data and results, we can conclude that, in terms of pure performance, sparse multiplication is much better. Furthermore, it has been observed that the increase in density of the arrays does not affect the performance of such multiplication for the simple fact that the path of one-dimensional arrays in compressed formats are much faster than the path of two-dimensional arrays.

However, a drawback of this type of multiplication is that it must be taken into account that matrices are not easily understandable in such formats and that in order to work with this type of multiplication, the corresponding transformations of each format must be programmed.

Nevertheless, if I were faced with the decision to select a high-dimensional matrix multiplication format, I would definitely select that format as the program would be much more efficient in terms of memory usage and computation time.

6. FUTURE WORK

In a future project, while a variety of classes and modules have been developed to deal with dense and sparse matrices, there may be room for performance improvements by exploring additional optimisation techniques, such as parallelisation or the use of hardware acceleration (e.g. GPU) to further speed up matrix operations.

In addition, other advanced sparse matrix formats exist, such as diagonal matrices, symmetric matrices, etc. Exploring their implementation and their impact on performance could be an interesting direction.

It is important to keep in mind that technology is advancing rapidly so it is quite likely that a future study will not necessarily yield the same results as the present one, which underlines the importance of continuous evaluation and review in projects of this nature.

7. REFERENCES

1. Java code
2. Rajat/rajat17 matrix
3. Rajat/rajat28 matrix
4. GHSindef/ncvxqp3 matrix
5. Hamm/bcircuit matrix
6. GHSindef/copter2 matrix
7. IPSO/OPF10000 matrix
8. Zhao/Zhao1 matrix
9. DIMACS10/ri2010 matrix
10. vanHeukelum/cage10 matrix
11. Williams/mc2depi matrix