ULPGC · UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

BD

Sparse Matrix: big problems require compressed solutions

Victoria Torres Rodríguez[1] [1]*victoria.torres101@alu.ulpgc.es*

Abstract

Programming optimization is one of the most important steps in delivering a quality software product. Starting from a simple program that multiplies standard matrices to a product that takes into account the type of matrix (sparse or dense) and different multiplication methods is an extensive process. In this study we wanted to focus on improving the execution time of the product of square matrices, following a TDD methodology and SOLID and polymorphic design patterns. Said experimentation has been executed on different types of arrangements and methods, taking into account the time elapsed in its execution in seconds. For all these reasons, it has been concluded that the optimal way to approach a matrix problem is to analyze its environment and characteristics. If we are dealing with small matrices, a transpose multiplication can be an effective solution, averaging 1.3 seconds per operation. However, if the problem is addressed to large arrays, it is unfeasible due to the scarce memory resources to perform operations with dense arrays, being able to use methods based on their compression, which maintains an execution time of 2 hours per operation on average.

Key words: *Execution time, Benchmarking, Dense Matrix, Sparse Matrix, Java, TDD, Matrix Multiplication*

1 Steps into a matrix multiplication optimization

To assess the optimization process of a software product, benchmarking can be one of the most complete primary solutions. A problem such as the product of two matrices may seem simple at first glance, because after all, it is a composition operation carried out between two matrices (in our problem). However, the problem expands once we take into account the algorithms that generate these matrices, those that multiply them, and the type of matrices. At first glance it may seem like a simple question, without any relevance. Instead, Matrix multiplication is among the most fundamental and computationally intensive operations in machine learning. Consequently, there has been significant work on the efficient approximation of matrix multipliers. For this reason, new methods and algorithms are currently being investigated that allow these execution times to be reduced almost to a minimum ("Multiplying Matrixes Without Multiplying", Davis Blalock, John Guttag). In this way, this study wanted to demonstrate the importance of a good programming methodology, and the search for new methods according to the type of matrix in question can make a difference in a relevant way. Thus, the importance of the project lies in its flexibility and ability to adapt according to the problem to be solved, due to its large number of methods.

2 Problem statement

The matrix problem may seem complex. To define the typology of a matrix we must take into account the term "density". This refers to the quantity or content of an array. Depending on the density of zeros in a matrix, these can be classified as sparse, in which values equal to zero are dominant, or dense ("dense"), in which there are few records equal to zero. The sparse matrices maintain a multiplication algorithm different from the dense ones, since by maintaining a high number of 0 in their content, it is possible to ignore certain operations, since the product of a number by 0 is always 0. On the other hand, these can be compressed in a more "manageable" format, becoming compressed by rows or by columns, a process carried out in this study. The development of this project has taken into account the properties of matrix multiplication and above all, taking into account that we only deal with square matrices, that is, those that have the same number of rows as columns (n x n).

3 Method

3.1 labor structure

3.2 Analysis and design

Prior to implementation, a class diagram has been created in StarUML to represent the fundamental objects of the system that will be part of the solution and the relationships between the elements of the system, to see what components the system needs and how they influence each other. For this purpose, SOLID methodology and polymorphism classes have been taken into account. The implementation and testing is done in the Java programming language (version 11). This product has been developed as a group, in which the construction of the classes present has been divided into work subgroups, through routine meetings. In this case, we have focused on developing the classes associated with matrix multiplications, although a general idea of the work is maintained.

The rest of the multiplication method is based on the same execution algorithm, three nested loops that go through, depending on the type of algorithm, columns or rows to multiply their values and build the new matrix. The only change is that in RowMultiplication we alter the insertion of elements, because instead of traversing the matrix in a traditional way, we traverse it by rows. In turn, the transpose part of this idea, but when multiplying, we transpose matrix B. These methods are ideal for dense type matrices.

As a general description of the project idea, everything revolves around the Matrix interface, in the center, which represents different attributes of an immutable matrix, this interface is implemented by DenseMatrix, SparseMatrixCOO, SparseMatrixCCS (compressed by columns), SparseMatrixCRS (Compressed by rows). On the other hand, we have the builder interface, with which it is possible to place the values of a future matrix created with .toMatrix(). Finally, in the diagram of Figure 1, it can be seen that we have an interface in charge of implementing the multiply() method, where different matrix multiplication algorithms will be executed. In addition, in said diagram an interface can be seen, Deserializer with a deserialize method that returns an Matrix, and its different implementations focused on receiving a file with the extension .mtx, which represent arrays in large COO format to be converted to array formats sparse compressed. These methods, in turn, will resort to those of the COOfromMTXReader interface, in charge of reading methods for this type of file.

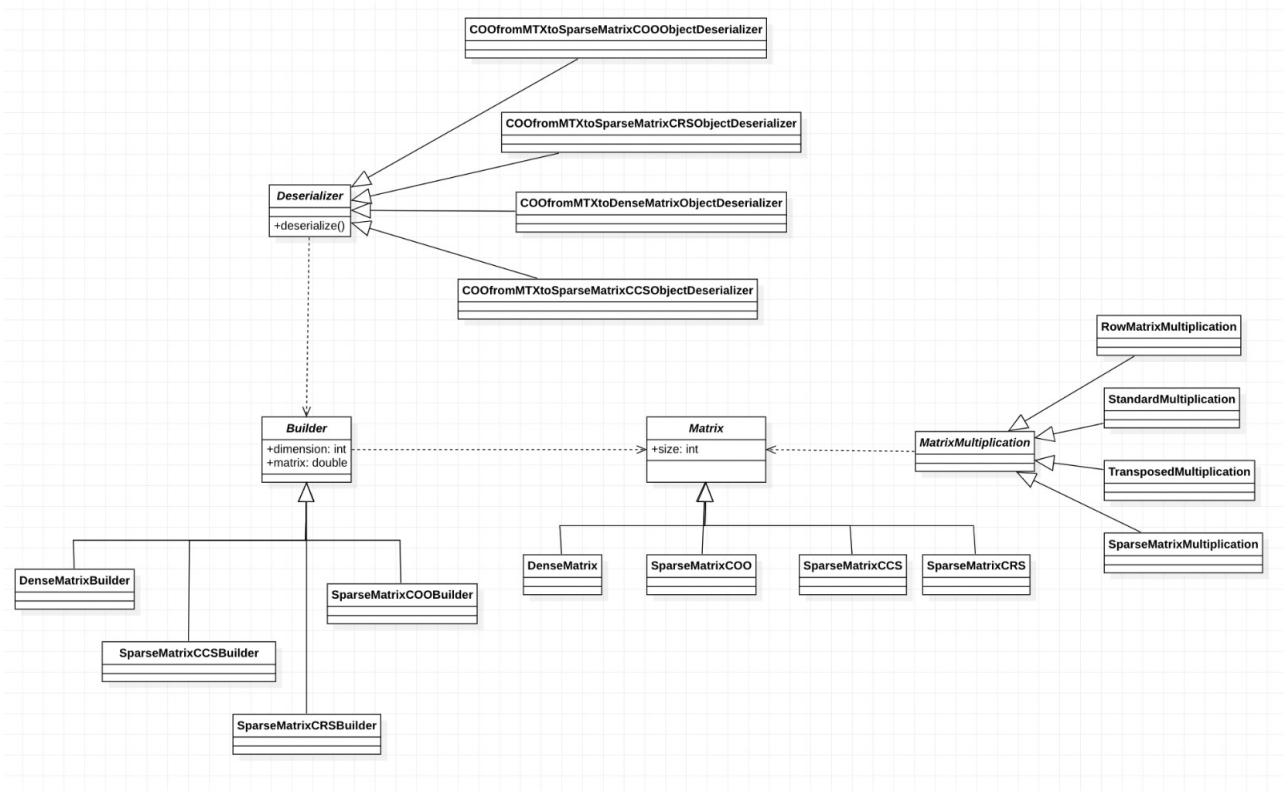

Figure 1: Class diagram for Matrix Multiplication implementation

3.3 Test environment

Once the representation of the system in the UML class diagram is done, we start the implementation. The implementation is done in the Java programming language (version 11), an interpreted language widely used by companies around the world to build web applications, analyze data, automate operations and create reliable

and scalable business applications. The tests will be run in the IntelliJ integrated development environment (IDE), specifically for the Java programming language, developed by the company JetBrains. * Note that the tests will be run on an Legion 5 15IMH05, with a Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz 2.59 GHz and 16 GB of RAM, so the execution program under other conditions will vary.

The testing of the proper functioning of the classes has been carried out with different use cases, in the multiplication functions, checking mathematically (through the multiplication of a matrix by a random vector, multiplying it by A and B). All this has been executed under JMH, a Java harness for building, running, and analyzing nano/micro/milli/macro benchmarks written in Java and other languages targeting the JVM. All of them have been developed under the same circumstances and the same type of computer, described previously. It should be noted that much of the initial testing of matrices with the .mtx extension has been carried out with matrices provided from the paper "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms".

3.4 Methodology

The labour structure proposed in this study has been in pairs. The coworker in question has been Alba Martín Lorenzo, maintaining the same code structure and sharing her architecture.

Once the way of working has been proposed, we will deal with the multiplication methodology. Among the different types of multiplication methods that could exist, 4 have been defined, according to the different characteristics of the exposed matrices.

In the first place, there is the ideal one for sparse-type matrices that have been compressed (by row and column). This algorithm saves a large amount of execution time, since it avoids multiplying elements by 0, avoiding a large number of multiplications between matrix elements. This is done through the creation of a vector that represents each matrix through the previously exposed algorithm. It should be noted that there is another class that is responsible for doing with this same type of matrix, multiplication, in which it is first checked in the loop prior to the operations, if any of the elements to be multiplied is 0.

The rest of the multiplication method is based on the same execution algorithm, three nested loops that go through, depending on the type of algorithm, columns or rows to multiply their values and build the new matrix. The only change is that in RowMultiplication we alter the insertion of elements, because instead of traversing the matrix in a traditional way, we traverse it by rows. In turn, the transpose part of this idea, but when multiplying, we transpose matrix B. These methods are ideal for dense type matrices.

4 Experiments

The tests that were executed and the results that were obtained in said analysis will be demonstrated. To evaluate matrix multiplication implementation, it has to be must measured different aspects of performance in order to establish which one is the best.

These experiments have been developed under JMH, a Java harness for building, running, and analyzing nano/micro/milli/macro benchmarks written in Java and other languages targeting the JVM. All of them have been developed under the same circumstances and the same type of computer, described previously.

4.1 Experiment 1

First, we wanted to evaluate the performance of randomly created 1138 x 1138 dense matrix multiplications. For this, random arrays of said dimension were created. Furthermore, all multiplication methods have been tested over 6 warm-up iterations and 6 time measurement iterations. From all this it was obtained that the best method in dense matrices is the transposed method, reaching 1.331 seconds per operation. On the other hand, it is striking that the rest of the algorithms maintain notable differences, with an average of 6 and a half seconds per operation, tripling the execution time of the best method. Therefore, the transpose multiplication method has been selected as ideal for multiplying dense matrices, as can be seen in Table 1. The error that the "Row" form of matrix multiplication maintains is notable, since it seems to be the most varied variant. All in all, a very undesirable feature.

It should be noted that this high dimensionality has been taken, since in cases of 100 x 100 matrixes, the differences in execution were 0.01 seconds per operation between the different methods. Therefore, in cases of small matrices, it seems that the chosen algorithm is not a decisive step.

Type of matrix multiplication	Mode	Iterations	Execution Time (seconds)	Error (seconds)	Units (s/op)
rowMultiplication	avgt	12	6.362	0.394	s/op
standardMatrixMultiplication	avgt	12	6.455	0.736	s/op
transposedMultiplication	avgt	12	1.331	0.244	s/op

Table 1: Benchmarking of dense matrix multiplications

4.2 Experiment 2

Second, we wanted to check the sparse matrix multiplication methods, both basic and in CCS and CRS format. It should be noted that several matrices in .mtx format have been used to test this implementation, taking as a reference one of the same size as in the previous study. Therefore, they have been deserialized to CCS and CRS format respectively to perform the operations.

The time obtained was 0.194 seconds per operation, as can be seen in Table 2. In this case, the method that multiplies sparse-type compressed matrices has been chosen, since it improves its performance by 90 percentage, compared to the matrix multiplication method. dense transpositions. At the moment, it is clear enough that compressed formats improve both in execution time and in memory.

Type of matrix multiplication	Mode	Iterations	Execution Time (seconds)	Error (seconds)	Units (s/op)
SparseMatrixMultiplication	avgt	12	0.194	0.223	s/op

Table 2: Benchmarking of sparse matrix multiplication

4.3 Experiment 3

Finally, it has been decided to operate with larger size matrices, such as 500,00 x 500,000. These matrices come in default .mtx format, and their respective multiplications were performed both in dense and sparse form. It is not surprising that the dense matrices for this type of problem are slow, what is more, they give errors in their execution since the amount of memory they occupy greatly exceeds that of average computers. Therefore, its execution was unfeasible, giving memory errors. However, if we use the compressed formats we can see that although the execution is not excessively fast, it is efficient. The results obtained were

For all these reasons, it can be verified that in the case of maintaining high-dimensionality matrices, one of the possible methods to implement could be the sparse matrices and their multiplication algorithms.

Type of matrix multiplication	Mode	Iterations	Execution Time (seconds)	Error (seconds)	Units (s/op)
BigSparseMatrixMultiplication	avgt	2	6,453,125	127.6	s/op

Table 3: Benchmarking of Big sparse matrix multiplication

5 Conclusion

The operations between matrices constitute one of the bases of the functionality of many computer science processes. Maintaining its optimal development can allow a software system to be maintained over time for a long time. As we have seen throughout this study, there are infinite ways to represent a matrix: Sparse, Sparse CCS, Sparse CRS, Dense... In turn, each format allows certain computational advantages over others, such as the incredible ease of conversion to COO and less memory usage on large arrays, something that could not be done with dense arrays as it takes up a lot of memory when processing them, leading to errors.

Based on the benchmarking described above, it seems that an optimal way to perform matrix multiplications is transpose. Well, if it is about small matrices, whose dimensions do not exceed 100,000 columns and rows, they could be effective methods. However, as it was shown in the experimentation, when we deal with a large matrix, we can see that the execution of multiplications with dense matrices is not a good option, since it occupies an excess of memory space, making it impossible to even generate them. . Therefore, we wanted to know that the best method depends on the problem being addressed. If we take small matrices, a transpose multiplication with dense matrices can be an effective solution, reaching an average of 1.3 seconds per operation. However, without showing a limited memory in our execution, this method is not correct, since we have to deal with compressed arrays, although these execution times are not far from the average 2 hours of execution.

6 Future Work

Many different adaptations, tests and experiments. they have been left to the future due to lack of time (ie experiments with real data are often very time consuming and require equal days to complete a single run). For all these reasons, we would like to name some aspects that would be taken into account for a better implementation of this study. First of all, the lack of matrix variability is notable, since this software product only handles square and double-type matrices. What's wrong with it? The lack of compatibility with future big data projects. Matrix operations are a foundation in algebra and many processes depend on this system. Because of this, we narrowed down the search for large-scale solutions. For this reason, the idea of creating an interface that, through the use of generics, allows us to edit and make compatible any matrix format, of type integer, double... In addition, modifying certain aspects in the algorithms (such as checking that the columns of matrix A are the same as the rows of B), we can deal with multiplications with matrices of different dimensions, other than square ones.

References

[1] *Clean Code* Robert C. Martin. 2019.

[2] *Multiplying Matrices Without Multiplying* David Blablock, John Guttag. 2021.

[3] *Sparse matrix storage format* Fethulah Smailbegovic, Georgi Nedeltchev Gaydadjiev. 2005.

[4] *Optimization of sparse matrix-vector multiplication on emerging multicore platforms* Samuel Williams†, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, James Demmel. 2007.