Benchmark of Sparse Matrix Multiplication

Ricardo Cárdenes Pérez
Faculty of Computer Science
University of Las Palmas de Gran Canaria

October 16th, 2023

Abstract— **Matrix multiplication is such a demanding task when the field of application is Big Data. It underlies many data processing techniques and results in the core of many optimization problems. However, it is such a complex operation to execute when the magnitude of the implied matrices becomes significant. Thus, this work aims to implement an efficient way to multiply these data structures, taking into account their density, in order to take advantage of the data itself.**

To achieve this goal, we used sparse matrices and both CRS and CCS representations, implementing the multiplication algorithm over these compressed structures rather than to the original raw matrices. By this, we obtained a Speed-Up of over 530x for matrices of an order of 2048, which increases for larger matrices.

I. INTRODUCTION

Matrix multiplication constitutes a fundamental operation in Data Science. It is widely used in many fields that involve data processing, due to its power to make transformations among data and its ease of implementation. In previous work, Benchmark for Matrix Multiplication [2], we took several approaches to this problem, trying to solve the highly demanding time of these multiplication algorithms by finding the best programming languages to take such a task. However, no code efficiencies were introduced to the standard matrix multiplication algorithm.

It is really hard to find what these efficiencies could be just by looking at the matrix multiplication operation itself, for two matrices A and B

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \tag{1}$$

Its structure highly indicates a linear behavior, and since we have to take the resulting matrix positions (i, j) into account, it is clear that a three-loop algorithm, described in work [2], shall be needed. In fact, there is no optimization we can apply such that we can avoid using any of these loops for the set of all matrices. So, assuming this, all we can do is somehow reduce the elements that are iterated in them.

II. SPARSE MATRICES APPLICATION

Sparse matrices, as their name indicates, are matrices where non-relevant information is omitted. A matrix can be composed of any arrangement of different numbers uniformly distributed, but in real-life applications, this is not often seen. The reality is that the matrices used for Big Data problems have zero value for many entries. And this actually makes sense. If our matrix is used to represent the friendships in a social network [2], a lot of pairs of people won't share any kind of amity. Thus, the entries representing those pairs would store a zero.

It is clear that due to the canceling property of $\mathbf{R}$, these entries will not bring any remarkable value to the resulting matrix, as they will cancel out any other factor. Moreover, it would be enough to use a zero if we ever need those removed entries, so we can be sure that no information will be lost if we omit to store them.

A sparse matrix is no more than an implementation of this idea of discarding zeros, and it can be implemented in several ways. In this article, we try three different approaches, using Coordinates, CRS, and CCS representations. Using these new data structures will allow us to make faster multiplications, for matrices that could not even be saved to heap memory in the first place. In the work Optimization of sparse matrix-vector multiplication on emerging multicore platforms, [1], you can check out how the mentioned sparse matrices are constructed.

III. Experiment

To properly measure the performance of this multiplication, we should first describe the terms of the experiment, so that the reader can make an idea of how accurate the obtained results are.

A. Multiplication Algorithm

There are many possible ways to implement this multiplication. Probably, the easiest and the most appropriate one in terms of software engineering, showing the perfect trade-off between time complexity and readability, is the one based on both CCS and CRS representations shown in Article [1]. Thus, given two matrices $A \in \mathcal{M}_{nxp}$ and $B \in \mathcal{M}_{pxm}$ expressed in CRS and CCS, respectively, we introduce the next multiplication algorithm.

Algorithm 1: Matrix Multiplication

```
Data: Matrices $A \in \mathcal{M}_{n,p}(\mathbb{R})$ and $B \in \mathcal{M}_{p,m}$
Result: Matrix $C \in \mathcal{M}_{n,m}(\mathbb{R})$
1  $C \leftarrow (0)_{nxm}$
2  for $i$ from 1 to $n$ do
3      for $j$ from 1 to $m$ do
4          $ii \leftarrow a.rowPtr[i]$
5          $iEnd \leftarrow a.rowPtr[i+1]$
6          $jj \leftarrow b.colPtr[j]$
7          $jEnd \leftarrow b.colPtr[j+1]$
8          $s \leftarrow 0$
9          while $i < 10$ do
10             $aa \leftarrow a.columns[ii]$
11             $bb \leftarrow b.rows[jj]$
12             if $aa == bb$ then
13                 $s \leftarrow$
                     $s + a.values[ii] * b.values[jj]$
14                 $ii \leftarrow ii + 1$
15                 $jj \leftarrow jj + 1$
16             else if $aa < bb$ then
17                 $ii \leftarrow ii + 1$
18             else
19                 $jj \leftarrow jj + 1$
20         if $s \neq 0$ then
21             $c_{ij} \leftarrow s$
```

By the end of this algorithm, the matrix C would have been successfully computed, having omitted all calculations for the null entries of A and B.

As the reader might guess, the more zeros the matrices A and B have, the shorter the algorithm takes. For fully matrices, this algorithm gets even slower than the standard one, but solves the problem for those times when there is no enough memory to store the matrix during the execution.

B. Implementation

In order to implement this algorithm, we first defined the model of the project, which includes all the kinds of matrices used. The diagram in Figure 1 shows its architecture.

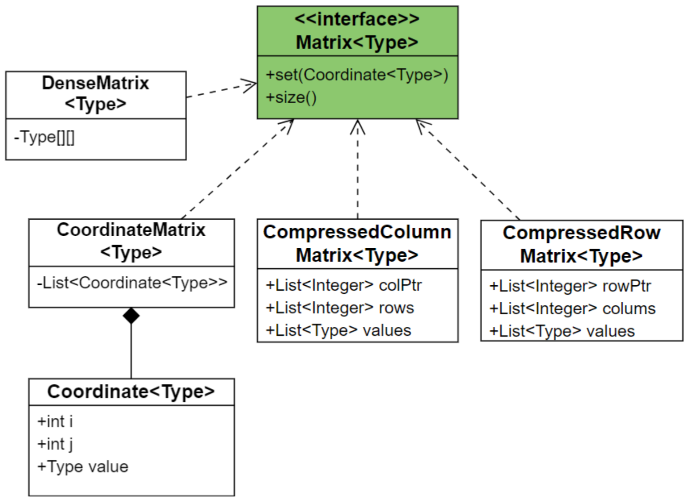
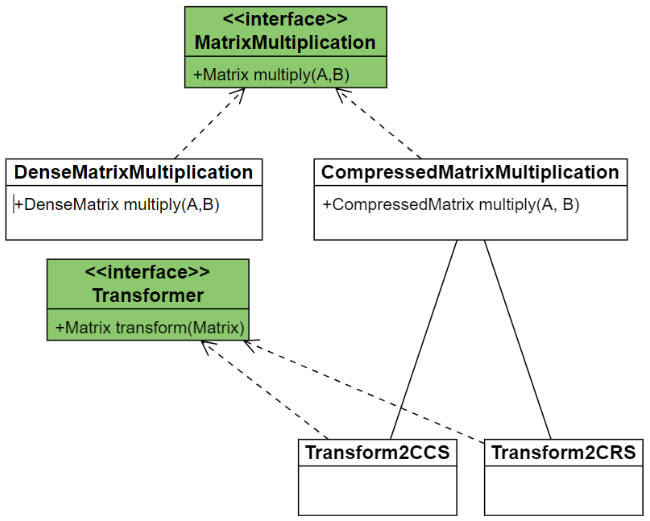


Fig. 1. Model Class Diagram

As you can see, there are three different kinds of Sparse Matrix Implementations, but just two of them are compatible with multiplication in algorithm 1, due to the way it was constructed.

With that clear, if we were to multiply two compressed matrices A and B, we have to make sure that they are expressed in CRS and CCS formats respectively, as explained in subsection A. Otherwise, we should apply the corresponding transformation on the matrix, so that we can apply algorithm 1. We tested the multiplication algorithm by checking if the associative property holds for different triples of matrices, as you can see in the GitHub repository, [3], of the project, in the test package.

The transformations between different kinds of matrices are executed by some objects called transformers, which can be observed in the source code, and allow transformations between whatever two classes of matrix. This is the reason why, by using the developed library, algorithm 1 could be used for any pair of matrices, as far as you apply the appropriate transformations on them, so they are expressed in representations CRS and CCS respectively.



<<interface>> MatrixMultiplication
+Matrix multiply(A,B)
DenseMatrixMultiplication
+DenseMatrix multiply(A,B)
CompressedMatrixMultiplication
+CompressedMatrix multiply(A, B)
<<interface>> Transformer
+Matrix transform(Matrix)
Transform2CCS
Transform2CRS

Fig. 2. Matrix Multiplication Class Diagram

This project was developed following SOLID principles, with particular emphasis on interface segmentation and Liskov principles, so we can extrapolate the source code for future works.

C. *Hardware*

For this task, we used a PC of brand MSI, a Katana model, with 16GB of RAM memory, solid state disk, and processor Intel Core i7 (16 cores). All windows will be closed while executing, except those needed to run the matrix multiplication code. The PC will be connected to power every time, as it is known that charging the PC turns to make the computer faster.

PC Model Katana GF66 11UE-486XES (MSI)
GPU NVIDIA® GeForce RTX™3050 Ti

IV. Multiplication Performance

To show the out-performance of this compressed multiplication algorithm for large matrices, we introduce the time measures obtained for the multiplication of several order matrices using both mentioned techniques. We ran the experiment for matrices with orders equal to some powers of two, several orders of magnitude bigger than in previous works, so that we are able to compare the performance of both implementations.

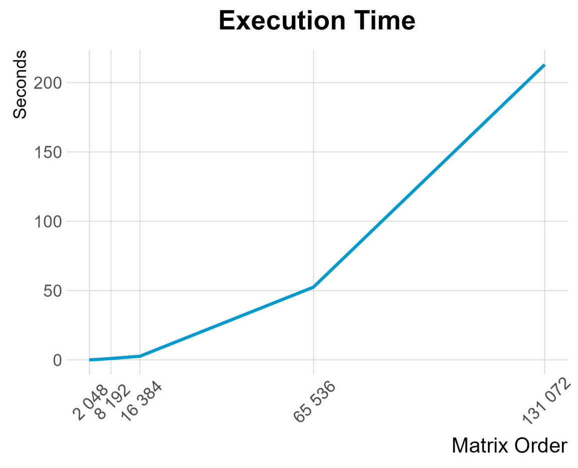


Fig. 3. Sparse multiplication execution time

As we can see in Figure 3, by omitting all coordinates with a value equal to zero, we obtain a significant decrease in time execution. Naturally, this claim is true just for matrices that exceed a certain threshold of null entries. Otherwise, this algorithm will take even longer than the standard one. However, that is often the case in real matrix problems. Comparing this with the results obtained in Benchmark for Matrix Multiplication, [2], shown in Figure 4, we realize this great improvement.

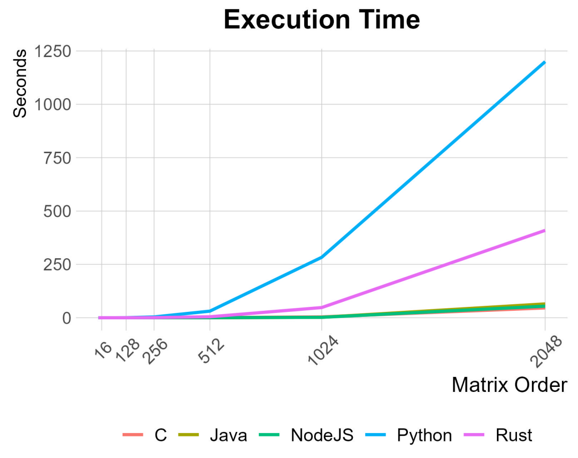


Fig. 4. Execution time for standard matrix multiplication

Now, due to the algorithmic nature of matrix multiplication and the fact that large-order matrices are commonly the ones involved in Big Data problems, it makes sense to study this execution time once the Java compiler has already warmed up. This is, once the Java application has reached the optimum compiled code performance. To do so, we used the JMH library, which stands for Java Micro-benchmark Harness. By this, we obtain some stats about the executed operations per millisecond and vice-versa, once the code has been compiled to its optimum. The obtained results are given in Figure 5.

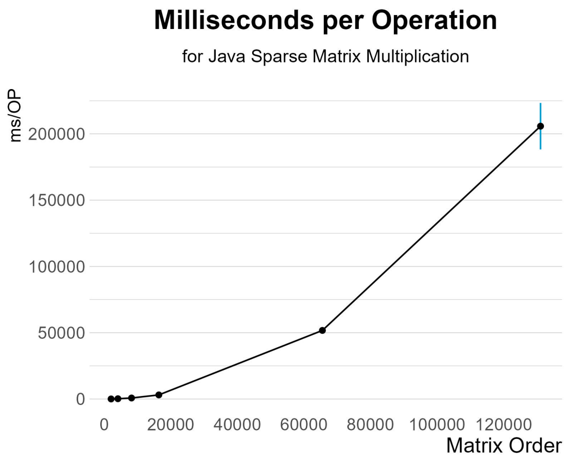


Fig. 5. Warmed-Up multiplication execution time

Comparing these measures with those given in Figure 3, there is a slight improvement on average during this phase of the program execution, and some independent tests were even completed in under 190 seconds for the last measured order. Now, for matrices of such big orders, that require higher execution times, these results will turn more accurate, due to the amount of time the compiler has to efficiently optimize the code. Just as an illustrative example, it took an hour and a quarter to multiply a matrix of order $5.25 \cdot 10^5$ by itself, something that would have been impossible with the previous implementation.

V. CONCLUSIONS

As seen in section 4, Sparse Matrix Multiplication supposes an enormous advance in time efficiency than Standard Multiplication. If we execute both algorithms for the same matrix of an order of 2048, [4], we reach a Speed-up of over $\frac{StandardTime}{SparseTime} = \frac{24696ms}{46.592} \approx 530x$. That is, 530 sparse multiplications could be done meanwhile the standard one is being executed. In other words, sparse multiplication is 530 times faster.

It is clear then that whenever the field of application is Big Data and matrix multiplication is the discussion topic, it should come to mind sparse matrix multiplication. In fact, it does not only reduce the amusing demand of time the multiplication takes but solves the problem caused by the need to locate the entire matrix in heap memory. Thus, this is the better approach to take this task in the long term.

In the rapidly evolving landscape of data processing, where efficiency and scalability are of paramount importance, the superiority of Sparse Matrix Multiplication is unmistakable. As we delve deeper into the digital age, the significance of efficient matrix multiplication becomes increasingly pronounced. Sparse Matrix Multiplication not only improves performance but also offers a more sustainable, forward-looking solution for the ever-growing demands of modern data science. It is a testament to the power of optimization in algorithms and the crucial role it plays in shaping the future of data processing.

FUTURE WORK

For future works, we would like to implement parallelism to the code, to make matrix multiplication even more efficient.

REFERENCES

[1] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, Reno, NV, USA, 2007, pp. 1-12, doi: 10.1145/1362622.1362674.
[2] Cárdenes Pérez, Ricardo J. (2023) Benchmark for Matrix Multiplication for Different Programing Languages
[3] Cárdenes Pérez, Ricardo J. (2023) Source Code https://github.com/ricardocardn/SparseMatrixMultiplication
[4] Speed-up matrix (2048x2048) https://sparse.tamu.edu/Bai/dw1024