Benchmarking of matrix multiplication using compressing formats

Carlos Mendoza Eggers

October 29, 2023

Abstract

This study delves into optimizing Java's execution time, employing matrix multiplication as the benchmark and concentrating on reducing operation time. The investigation compares conventional matrix multiplication against one multiplying between two compressed matrix formats: Compressed Row Storage (CRS) and Compressed Column Storage (CCS).

The methodology encompasses comparisons between different matrix multiplication methods using diverse formats and the evaluation of their performance. Symmetric sparse matrices of varying sizes form the basis for the experiments, facilitating the measurement of execution times for each multiplication method.

The findings suggest that the compressed formats showcase significantly improved execution times with bigger number of elements. This study underscores the potential of compressed formats for handling and operating on large sparse matrices efficiently and robustly.

Introduction

In our initial research, our focus was on comparing the performance using vast amounts of data across various programming languages. Following this investigation, we arrived at the conclusion that Java stood out as one of the optimal languages for handling substantial data sets, primarily due to its low execution time.

Nevertheless, even though Java demonstrates inherent speed, it should be feasible to optimize and further reduce its execution time. In this study, we will also use matrix multiplication as the benchmark to explore novel possibilities. These possibilities will be concentrated on diminishing the execution time for this operation.

To achieve this goal, we will employ established matrix compression formats. These formats are designed to reduce the size of extensive sparse matrices, which are matrices containing a large number of zero elements. We will mainly compare between the conventional multiplication and one involving two compressed matrix formats. Further on, we will delve into more detail.

In this paper, we detail our benchmarking methodology, present results, and discuss their different perfomances. Our goal is to contribute valuable insights to explore new options for reducing operative time doing a matrix multiplication.

Methodology

As introduced earlier, the subject under investigation involves comparing the performance of conventional matrix multiplication with one utilizing compressed matrix formats. The formats employed in this experiment include the **Compressed Row Storage (CRS)** and the **Compressed Column Storage (CCS)**. Additionally, other formats, such as the Coordinate format, are used to accomplish a necessary task in this study.

In **Compressed Row Storage (CRS)**, a sparse matrix is represented by three arrays: one for the non-zero values of the matrix, another for the column indices of these values, and a third array indicating the start and end positions of each row in the first two arrays.

Compressed Column Storage (CCS), on the other hand, stores a sparse matrix using a similar approach to **CRS**, but it organizes the data by columns. It uses arrays for non-zero values, row indices, and pointers to the beginning of each column.

In our case, the matrices used for the experiment were **symmetric sparse matrices**.These have an equal number of rows and columns.

The n **values** on which we decided to conduct the tests where concretely these ones:

2,4,8,16,32,64,128,256,512,1024,1536,2048,4096.

As we can observe, there are 14 objects of study.

Having stated this, we proceed to explain the nature of each experiment. For each type of multiplication, the execution time was measured using the

JMH[1] **Java library.** For each n, the code was executed plenty of times.In the end, what we consistently obtained were the mean, minimum, and maximum execution times for each type of multiplication of two matrices with n columns and n rows.

The number of **forks, iterations** and **warmups** were changed significantly as the size n of the matrices increased.

All the information from the different measurements was collected manually in an Excel spreadsheet. This was done to facilitate its manipulation and control to subsequently create the charts.

We will have in consideration the benchmark that was registered in the first task. It should have changed because the conventional multiplication method has been modified to align with the new project implementation.

We will now proceed to explain the various modules and classes that constitute the project. Alongside this explanation, a brief overview will be provided of the purpose of each class and the rationale behind its implementation.

There will be a link at the end of this section where you could access the source code of the project.Nevertheless, a brief visual description will be provided.

Project structure

The project is divided in two different sections, **source code** and **test code.**

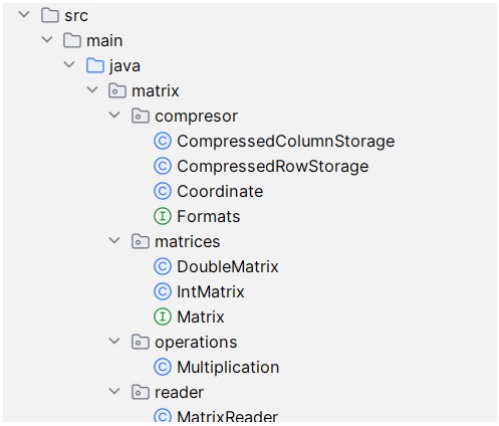

Figure 1: *Source Code of the project.*

Source code. As we can see in 1, within the main module, **matrix**, there are four sub-modules.

- **compressor.** This module contains the various compression formats utilized throughout the experiment. Each class incorporates its respective ArrayLists containing the different formats used by each. The Formats interface is implemented to allow CCS and CRS to share the same methods.
  - **CompressedColumnStorage.**
  - **CompressedRowStorage.**
  - **Coordinate.**
  - **Formats.**
- **matrices.** Here are the two types of matrices utilized, each with its corresponding interface.
  - **DoubleMatrix.**
  - **IntMatrix.**
  - **Matrix.**
- **operations**
  - **Multiplication.**This class contains the two focal methods under study: conventional multiplication and compressed multiplication.
- **reader**
  - **MatrixReader.** This class was created to read the matrix on which the multiplication was to be tested. Later, we will discuss an incident that occurred during the experiment.

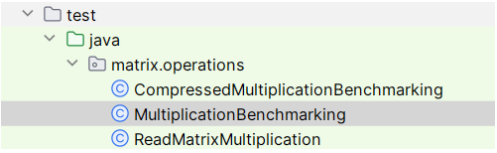

Figure 2: *Test Code of the project.*

Test code. As we can see in 2, there are three different Java classes for benchmark inside the **matrix.operations** module.

- **CompressedMultiplicationBenchmarking**
- **MultiplicationBenchmarking**
- **ReadMatrixMultiplication**

Links

You can review the code of the project at the following link: `https://github.com/carlillous/matrixBenchmarking`.

Experiments

One aspect I can assert is that the implementation of the normal matrix multiplication in this delivery appears to be slightly more efficient than the previous one. This is due to the change made in the execution of the operation, along with the input as parameters

[1] Github

of the **Matrix** class. Although initially, it might seem that the creation of new instances as an input could have a negative impact, it turned out to be quite the opposite. This is largely due to the change that we can observe in 3 and 4 inside the loops of the calculus.

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```

Figure 3: *First implementation loop.*

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        double aux = 0;
        for (int k = 0; k < n; k++) {
            aux += a.getValue(i,k) * b.getValue(k,j);
        }
        c.setValue(i,j,aux);
    }
}
```

Figure 4: *Second implementation loop.*

We cannot begin this section without addressing the issue encountered when attempting to read the matrix of 500 thousand elements x 500 thousand elements. While converting the matrix into the **CRS** and **CCS** formats, an **OutOfMemory exception** occurred, indicating that the **heap was full**. I tried to resolve this issue but was unable to find a solution. I believe the implementation of my classes is not bad, although they could likely be more optimized. Due to this, the heap error is occurring.

Despite these challenges, I believe that the **Reader, Coordinates**, and all related components necessary for the process **are functioning correctly**. However, they are not optimized for such large matrices.

As observed in 5,6,7, it appears that the initial performance difference between both is quite marginal, while as the number of elements increases, we can ascertain that one seems to be significantly faster than the other.

Conclusion

In conclusion, throughout the experimentation and benchmarking of both methods—conventional matrix multiplication and the compressed formats—we have observed a clear and more efficient method.

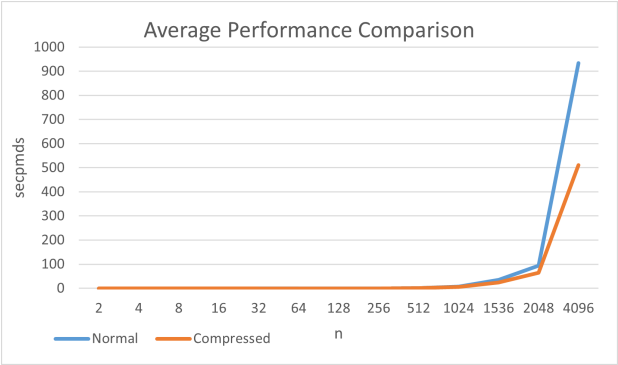

Figure 5: *Performance comparison overall.*

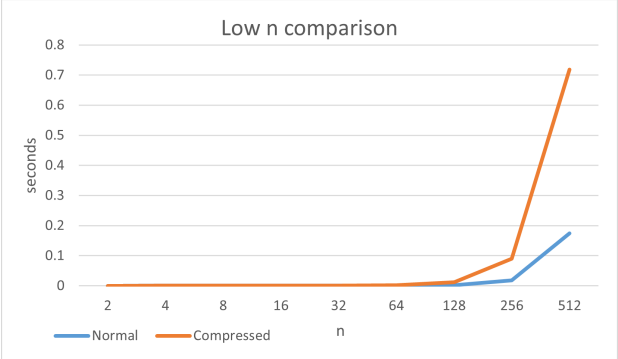

Figure 6: *Performance from 0 to 512 number of elements.*

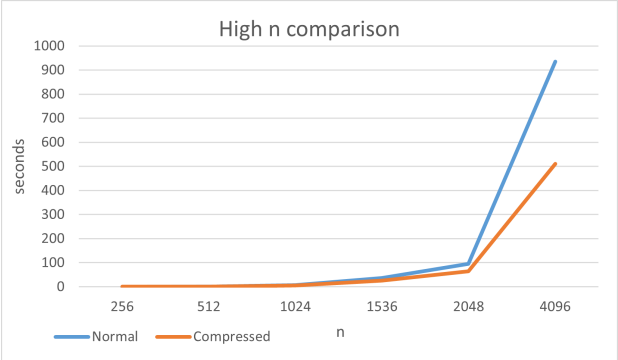

Figure 7: *Performance from 256 to 4096 number of elements*

With a small number of elements, the execution time difference between sparse matrices was more or less the same, although the compressed format was slower. However, starting at n=1024 elements, we can observe how the **compressed format multiplication becomes faster than the traditional approach**. Our theory is confirmed with the following two n values, where one multiplication outperforms the other.

Therefore, we can conclude that **compressed formats** appear to be a very promising option for storing and working with large sparse matrices. Not only they are efficient in terms of storage, but they also demonstrate robustness in operations performed.