Third Task: Benchmark of Matrix Multiplication Formats

Alejandro Jesús González Santana

October 2023

Contents

Github Repository: `https://github.com/Alejeglez/ThirdBenchmarkingTaskMatrixMultiplication`

Abstract

One of the most challenging aspects to consider when initiating a software project is resource management. While our primary focus is often on achieving desired outcomes, selecting the right development environment can be pivotal in maximizing efficiency while minimizing resource utilization.

We have previously advocated for this idea in our past work, where we compared various programming languages in the context of matrix multiplication. In this instance, our focus shifts to how we represent matrices, particularly emphasizing the concept of density in Java. Density represents the percentage of non-zero elements within our matrix. We aim to investigate whether we can effectively perform matrix multiplication using a compressed format and assess the accuracy of the results.

The outcomes of this experiment have led to conclusions about the ability to handle large matrices while conserving space if we are careful about how we define our data structures. Additionally, we explore strategies to reduce computation time by employing alternative representations. Moreover, we will gain insights into the selection of Java data types that offer enhanced performance.

---

1 Introduction

Software engineering aims to simplify tasks and maximize productivity by searching efficient solutions. To assess the performance of a program resources usage, benchmarking serves as a crucial tool for gauging resource utilization.

Matrix multiplication proves to be a practical standard for evaluating how well a programming language manages resources during execution. Yet, from our prior experience, we know that these operations can become computationally intensive, resulting in a steep rise in complexity. This increased complexity often translates into longer processing times.

An effective strategy for conserving both space and time is the use of sparse matrices. This approach is well-documented in the research community, with approximately 321,695 studies dedicated to it, as reported by the scientific paper search engine Core[1].

Various formats exist for representing sparse matrices, including List of List (LIL) or Dictionary of Keys (DOK). For this research, however, we will focus on the Coordinate List format. This format involves storing lists of tuples in the form (row, column, value) to indicate the coordinates of non-zero values in the matrix. We will also explore the Compressed Row Storage (CRS) and Compressed Column Storage (CCS) methods.

[1]Core website: `https://core.ac.uk/search?q=sparse+matrix&page=1`

The principal objective of this research is to validate the results of these operations and compare the time differences with dense matrix multiplication. Through this, we aim to demonstrate that we can achieve similar outcomes while optimizing both space and time.

2 Problem Statement

A matrix serves as a robust data structure, fundamental in database systems, capable of storing diverse data types. However, obtaining data is not always straightforward, and matrices may include null or missing values. In the realm of data analysis, the presence of such data can be vexing and space-inefficient.

To face this challenge, matrices offer space-saving techniques, operating on the premise that absent elements can be interpreted as zeros. While a dense matrix retains all its non-zero elements, we'll explore the Coordinate List (COO) format to simplify this issue. Although COO helps conserve space, further optimization can be achieved by leveraging Compressed Row Storage (CRS) and Compressed Column Storage (CCS).

Both CRS and CCS formats consist of three primary arrays: a "values" array, housing non-zero elements sorted by either rows or columns, a "columns(for CRS) or rows(for CCS)" array indicating the column or row location of each element, and a "pointers" array, where each element points to the starting position in the "values" and "columns or rows" arrays for a given row(CRS) or column(CCS).

Our approach involves defining tests using JUnit[2] to validate the performance of matrix multiplications between compressed matrices. We'll also create classes to represent these matrices and enable serialization and deserialization from .mtx files into COO format. Our objective is to demonstrate that these compressed matrices perform equivalently to dense matrices while efficiently utilizing resources. This not only enhances data structures capabilities but also optimizes resource utilization.

3 Methodology

Machine specs:

One of the key aspects in bechmarking is the hardware used and the operating system. In this field, the specification of the machine which ran the multiplications is:

- Model: Lenovo Ideapad 330.
- CPU : Intel Core i7-8750H (6 cores, 2,2 GHz - 4,1 GHz, 9 MB).

[2]JUnit reference https://junit.org/junit5/

- GPU : NVIDIA GeForce GTX 1050 4 GB.
- RAM : 16 GB 2400 MHz DDR4.
- HDD: 1 TB.
- SSD: 256 GB.
- OS: Microsoft Windows 10 Pro 64 bits.

The programs were executed on an HDD disk, so the results may differ compared to running them on an SSD disk.

3.1 Matrix Formats Overview

Dense Matrix

Dense matrices are the typical representation using 2D arrays where all values in the matrix are stored, regardless of whether they are zeros or non-zero. This results in a fixed size for the elements that need to be stored. For example, in a matrix with 1024 rows and 1024 columns, you would have to store 1,048,576 values.

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & 2 & 0 \end{pmatrix}$$

Example of Dense Matrix

The class of Dense Matrix in our Java program will look like this implemented for the long data type:

```java
        public class DenseMatrixLong implements Matrix<Long> {


    private final long[][] values;

    public DenseMatrixLong(long[][] values) {
        this.values = values;
    }
    @Override
    public int size() {
        return values.length;
    }

    @Override
    public Long get(int i, int j) {
        return values[i][j];
    }

}
```

Listing 1: Dense Matrix Long Class

Coordinate List Matrix (COO)

The Coordinate List matrix will serve as a pivotal type that enables us to transition from a dense matrix to a Compressed Row or Column Storage (CRS or CCS) format. In this format, a coordinate describes the position of a non-zero element in the matrix using a tuple (row, column, value). Once we have a list of such coordinates, we possess all the essential components to perform matrix operations. However, as we've already mentioned, further optimizations can be achieved by implementing CRS or CCS. To illustrate this, consider the following example of a Coordinate List Matrix:

```
    {
        (1, 1, 1),
        (1, 2, 2),
        (1, 3, 3),
        (2, 2, 2)
    }
```

Listing 2: Coordinate List Matrix (COO)

In Java the class of Coordinate List Matrix uses as support a Coordinate record, which will represent an instance of the location of an element. Its components will be i(row), j(column) and value. We have created a public abstract class to group all the sparse matrix formats.

```
public class CoordinateMatrixLong extends SparseMatrixLong {
    private final int size;



    private final List<CoordinateLong> coordinates;

    public CoordinateMatrixLong(int size, List<CoordinateLong>
    coordinates) {
        this.size = size;
        this.coordinates = coordinates;
    }

    public List<CoordinateLong> getCoordinates() {
        return coordinates;
    }
    @Override
    public int size() {
        return size;
    }

    @Override
    public Long get(int i, int j) {
        return coordinates.stream()
                .filter(c->c.i() == i & c.j() == j)
                .findFirst()
                .map(CoordinateLong::value)
                .orElse(0L);
}
```

Listing 3: Coordinate Matrix Long Class

We will also discuss certain methods that can be useful in creating CRS and CCS matrix formats. In order to incorporate these matrices, it is essential to pre-sort the elements of our matrix based on their appearance in rows or columns. To ensure this, we have included two methods in our class

```java
public List<CoordinateLong> sortCoordinatesByColumn() {
    coordinates.sort(new Comparator<CoordinateLong>() {
        @Override
        public int compare(CoordinateLong c1, CoordinateLong c2
) {
            int result = Integer.compare(c1.j(), c2.j());
            if (result == 0) {
                result = Integer.compare(c1.i(), c2.i());
            }
            return result;
        }
    });
    return coordinates;
}

public List<CoordinateLong> sortCoordinatesByRow() {
    coordinates.sort(new Comparator<CoordinateLong>() {
        @Override
        public int compare(CoordinateLong c1, CoordinateLong c2
) {
            int result = Integer.compare(c1.i(), c2.i());
            if (result == 0) {
                result = Integer.compare(c1.j(), c2.j());
            }
            return result;
        }
    });
    return coordinates;
```

Listing 4: Coordinate Matrix sorting methods

This matrix will be the default format for the matrix we will use in our experiment. The website where we can found example matrices is Suit Sparse Matrix Collection[3], where we can see the matrix are applied to certain topics. This topics go from economics to graphs or robotics.

Compressed Row Storage

The Compressed Row Storage is composed by three arrays. The first one will represent the non-zero values sorted by row and then by column. The second array will represent the column for each one of the elements in the values array. The first element of column will be the column of the first element in value and going on. The last array is a row pointer which keeps track of the starting index in the values array for each row. Each element in the row pointers array points to the beginning of a row in the values array.

```java
{
```

[3]Suit Sparse Matrix Collection: https://sparse.tamu.edu/

```
        values=[1, 2, 3, 2],
        columns=[0, 1, 2, 1],
        rowPointers=[0, 3, 4] (4 could not be included, it is the
    number of elements in values)
    }
```

Listing 5: Compressed Row Storage(CRS) Example

The class CRS Matrix in Java will be the following one, like the Coordinate List matrix it extends SparseMatrix class:

```java
public class CRSMatrixLong extends SparseMatrixLong {

    public List<Integer> rowPointers;
    public List<Integer> columns;
    public List<Long> values;

    public CRSMatrixLong(List<Integer> rowPointers, List<Integer>
    columns, List<Long> values) {
        this.rowPointers = rowPointers;
        this.columns = columns;
        this.values = values;
    }

    @Override
    public int size() {
        // Calculate the maximum column index
        int maxColumnIndex = columns.isEmpty() ? -1 : Collections.
    max(columns);

        // Get the size of the rowPointers list
        int rowPointersSize = rowPointers.size();

        // Return the maximum value between maxColumnIndex and
    rowPointersSize
        return rowPointersSize;
    }
}
```

Listing 6: CRS Matrix Class

Compressed Column Storage

The Compressed Column Storage is composed by three arrays. The first one will represent the non-zero values sorted by column and then by row. The second array will represent the row for each one of the elements in the values array. The first element of row will be the row of the first element in value and going on. The last array is a column pointer which keeps track of the starting index in the values array for each column. Each element in the column pointer array points to the beginning of a row in the values array. An example for the current matrix will be:

```
    {
        values=[1, 2, 2, 3],
        columns=[0, 0, 1, 0],
        rowPointers=[0, 1, 3, 4] (4 could not be included, it is
    the number of elements in values)
    }
```

Listing 7: Compressed Column Storage(CCS) Example

It appears that CCS is very similar to CRS, so the Java code for CCS will be nearly identical.

3.2 Sparse Matrix Multiplication in Compressed format

The sparse matrix multiplication we are about to perform is based on compressed formats. To execute this type of multiplication, we need the Compressed Row Storage (CRS) representation of the matrix, as well as the Compressed Column Storage (CCS). The result of this multiplication will yield a Coordinate List Matrix.

The algorithm begins by initializing an empty Coordinate List to store our results. To determine its size, we choose the maximum size between our CRS and CCS. This size corresponds to the highest column or row index.

We iterate through the indexes of the arrays in our sparse matrices, first by CRS and then by CCS. Within each inner iteration, we obtain the current row and column pointers as well as the following ones. Additionally, we initialize a variable 's' to accumulate our sums.

While the current row or column pointer is less than the value of the following row or column pointer, we consider that we are in the same row or column. If this condition is met, we retrieve the corresponding column and row from the rows or columns arrays using the current pointers as indices.

When we have both the column and row values, we check if they match. If they do, we accumulate the multiplication of the values in the respective values arrays using the current pointer as the index. After this, we increment both the row and column pointers. If the row value is less than the column value, we increment the row pointer; conversely, if the column value is smaller, we increment the column pointer.

Upon exiting the while loop, we store the value of 's' only if it is not equal to zero. After completing all iterations, we obtain a matrix from the builder object. We'll explain the Builder class in more detail later.

```
    public CoordinateMatrixLong multiplyCompressed(CRSMatrixLong
    crsMatrixLong, CCSMatrixLong ccsMatrixLong){
        int maxSize = Math.max(crsMatrixLong.size(), ccsMatrixLong.
    size());
        CoordinateMatrixBuilderLong builderLong = new
    CoordinateMatrixBuilderLong(maxSize, new ArrayList<>());
        for (int i = 0; i < crsMatrixLong.size() - 1; i++){
            for (int j = 0; j < ccsMatrixLong.size() - 1; j++){
                int actualRowPointer = crsMatrixLong.rowPointers.
    get(i);
```

```java
                int followingRowPointer = crsMatrixLong.rowPointers.get(i+1);
                int actualColPointer = ccsMatrixLong.columnPointers.get(j);
                int followingColPointer = ccsMatrixLong.columnPointers.get(j+1);
                long s = 0;
                while (actualRowPointer < followingRowPointer && actualColPointer < followingColPointer){
                    int aa = crsMatrixLong.columns.get(actualRowPointer);
                    int bb = ccsMatrixLong.rows.get(actualColPointer);
                    if (aa == bb){
                        s += crsMatrixLong.values.get(actualRowPointer) * ccsMatrixLong.values.get(actualColPointer);
                        actualRowPointer++;
                        actualColPointer++;
                    }
                    else if (aa < bb) actualRowPointer++;
                    else actualColPointer++;
                }
                if (s != 0) builderLong.set(i, j, s);
            }
        }

        return (CoordinateMatrixLong) builderLong.get();
    }
```

Listing 8: Compressed Matrix Multiplication

3.3 Project Structure

For this experiment, we have designed a flexible structure, considering the possibility of adding more implementations or classes in the future. The schema of our source code (src) folder is as follows:

```
java
 ├─Matrix (folder with the matrix format implementations seen before)
 │  └─Subfolders for each matrix format where the classes for each
 │    data type will go
 ├─Matrix Builder (folder with the tools to build matrix and create
 │  a matrix object using the classes in Matrix folder)
 │  └─Subfolders for each matrix format where the classes for each
 │    data type will go
 ├─Matrix Operations (folder containing classes with the matrices
 │  multiplication functions, and functions to generate vector and
 │  matrices)
 ├─Matrix Serializer (folder containing classes to read and write
 │  matrices in .mtx file format)
 └─Matrix Transformations (folder containing classes to change the
    format of a matrix)
```

```
|
|
├── Matrix (Interface that establish the bases of a matrix for the
│   subsequent implementations)
├── Matrix Builder (Interface that establish the bases of a matrix
│   builder for the subsequent implementations)
└── Matrix Serializer (Interface that establish the bases of a matrix
    serializer for the subsequent implementations)
```

The folder structure was a crucial aspect of our project, primarily in the way we implemented interfaces. Rather than creating a separate interface for each data type, we opted for more general implementations organized into subfolders for each matrix type. This approach prevents the creation of massive folders for each data type and keeps all the implementations together, facilitating specific changes when needed.

In our review, we will focus on the interfaces, Matrix Transformations, and Matrix Operations folders. Other folders, which mostly involve implementing interfaces, will not be examined in depth, as we will only analyze specific functions within them

3.4 Matrix Interface

The matrix interface defines the methods a Matrix object should contain. We have created a common implementation for any primitive data type. The methods all matrix should be able to run is the size(), which will return us the size of the matrix and the method get, which based in a coordinate should return us the value of an element. The code will be the following one, where T represent a primitive data type. In our case, in the implementations we use Long.

```java
public interface Matrix<T> {
    int size();
    T get(int i, int j);
}
```

Listing 9: Matrix Interface

3.5 Matrix Builder Interface

The Matrix Builder interface shares a similar philosophy with the Matrix interface. A Matrix Builder object is responsible for populating a matrix with specified values and their corresponding coordinates. For all implementations, except the Coordinate Matrix Builder, they strictly adhere to the methods outlined in the interface.

The primary method, set(), requires a list of coordinates and utilizes them to populate the matrix with values.

The secondary method, get(), serves the purpose of returning a Matrix object using the fields and parameters within the builder. This occurs once all the necessary set() operations have been performed to construct the matrix. The code for these operations is as follows:

```java
public interface MatrixBuilder<T> {
    void set(int i, int j, T value);

    Matrix<T> get();
}
```

Listing 10: Matrix Builder Interface

The Coordinate Matrix builder has also its own set method to add each created coordinate in a list of Coordinates, which will be used for the creation of a Coordinate List Matrix.

3.6 Matrix Serializer Interface

The purpose of a matrix serializer object is to facilitate reading matrices from .mtx files (deserialization) and writing matrices to an mtx file (serialization). The foundational matrix format employed for both processes is the Coordinate List matrix. Consequently, a Matrix Serializer object should implement two essential methods.

Firstly, the serialize method takes the name of a file located in the resources folder of our project as a parameter and returns a Matrix object. This method reads the content of the specified file and translates it into a Matrix object.

Secondly, the deserialize method is a void function that creates a file containing the content of a given Matrix. This method writes the Matrix object's data into a file, storing it in the desired format for future use.

```java
public interface MatrixSerializer<T> {
    Matrix<T> deserialize(String file) throws FileNotFoundException
    ;
    void serialize(Matrix<T> matrix);
}
```

Listing 11: Matrix Builder Interface

3.7 Matrix Transformation Class

The matrix transformations class is an utility to convert a matrix to an specific format. We will use four method through this research. We should remind that a Coordinate List is the pivot format, as is it will be the middle step between Compressed formats and Dense Matrix.

3.7.1 Transform Dense Matrix to Coordinate List Matrix

The first thing we have to do if we want to create a Coordinate List Matrix will be to invoke a Coordinate Matrix Builder with the size of the Dense Matrix and an empty list. We will iterate through the elements of the dense matrix, checking for each element if is not zero. In this case, we will create a new coordinate using the method set of the builder. This method calls two homonym methods, one creates the coordinate and the other one adds it to a list. An the end we should return a Coordinate Matrix Object with the get method of the builder. We need to cast the result as it return a Matrix object without implementation.

```java
public CoordinateMatrixLong transformCoordinate(DenseMatrixLong
    matrix) {
        CoordinateMatrixBuilderLong builder = new
    CoordinateMatrixBuilderLong(matrix.size(), new ArrayList<>());
        for (int i = 0; i < matrix.size(); i++) {
            for (int j = 0; j < matrix.size(); j++) {
                if (matrix.get(i,j) == 0) continue;
                builder.set(new CoordinateLong(i+1,j+1, matrix.get(
    i,j)));
            }
        }
        return (CoordinateMatrixLong) builder.get();
    }
```

Listing 12: Transform Coordinate List Matrix to Dense Matrix

3.7.2 Transform Coordinate List Matrix to Dense Matrix

Initially, this method may appear somewhat confusing, but its purpose lies in enabling the testing of Compressed Matrix Multiplication results against the conventional multiplication approach we have employed thus far. The methodology employed here closely resembles the one described earlier.

Here is how it works: We create a Dense Matrix Builder based on the size of our Coordinate List matrix. This builder initializes a 2D array filled with zeros, and we modify the values of specific coordinates that are non-zero by iterating through the list of coordinates. Upon completion, we return the result using the ”get” method of the builder and perform any necessary type casting.

```java
public DenseMatrixLong transformDense(CoordinateMatrixLong matrix)
    {
        DenseMatrixBuilderLong builder = new DenseMatrixBuilderLong
    (matrix.size());
        for (CoordinateLong coordinate : matrix.getCoordinates()) {
            builder.set(coordinate.i()-1,coordinate.j()-1,
    coordinate.value());
        }
        return (DenseMatrixLong) builder.get();
    }
```

Listing 13: Transform Dense to Coordinate List

3.7.3 Transform Coordinate List Matrix to Compressed Row Storage (CRS)

We could have directly created the compressed format from a Dense Matrix; however, this approach would have been considerably more laborious. Dense matrices are not typically favored for research purposes, and most downloadable matrices adhere to the COO format.

The process is identical to the one employed when transforming a Coordinate List Matrix into a Dense Matrix. Nevertheless, the key distinction lies in the necessity to pre-sort the coordinates of the list by rows and in the ”set” method of the CRS builder. Here, we must maintain a record of the current row to determine when to add a new element to the row pointer. While the CCS format closely resembles this implementation, we will omit discussing it for brevity.

```java
    public CRSMatrixLong transformCRS(CoordinateMatrixLong matrix){
        CRSMatrixBuilderLong builder = new CRSMatrixBuilderLong();
        for (CoordinateLong coordinate : matrix.
sortCoordinatesByRow()){
            builder.set(coordinate.i(), coordinate.j(), coordinate.
value());
        }
        return (CRSMatrixLong) builder.get();
    }
```

Listing 14: Transform Coordinate List Matrix to CRS

```java
    public void set(int i, int j, Long value) {
        columns.add(j);
        values.add(value);
        if(currentRow < i){
            rowPointers.add(values.size()-1);
            currentRow = i;
        }
    }
```

Listing 15: CRS Builder set method

3.8 Matrix Operations

Previously, we have discussed how our program conducts Compressed Format Matrix Multiplication. However, it’s important to note that we utilize two distinct classes for matrix operations. The first is the RandomMatrix class, employed to generate matrices for our testing purposes. In particular, the Generate Coordinate Matrix method takes a specified size and incorporates a constant to regulate the matrix’s sparsity. While there’s no necessity to display the code, as it closely resembles the generators featured in our previous works, this method’s operation is well understood. The approach of a matrix generator instead of using matrices from Suite Sparse Matrix Collection has been taken due to some difficulties that will be treated later.

The other class is Matrix Operations, housing methods that directly impact matrix multiplication. Within this class, we’ve incorporated a Dense Matrix Multiplication method along with the functionality to generate a support vector. This support vector serves a vital role in the subsequent multiplication tests, functioning in the following manner:

- $A \in \text{CRS}$
- $B \in \text{CCS}$

- $C \in$ COO
- $V \in$ Vector of size CCS

$A * BV = CV$

The code will be the following one, the only difference if we multiply the B matrix in Coordinate List format with the vector before using the CCS format:

```java
public CoordinateMatrixLong multiplySupportVector(List<Long>
supportVector, CoordinateMatrixLong matrix){
    CoordinateMatrixBuilderLong matrixBuilderLong = new
CoordinateMatrixBuilderLong(matrix.size(), new ArrayList<>());
    for (int i = 0; i <= matrix.size(); i++){
        for (int j = 0; j <= matrix.size(); j ++) {
            if (matrix.get(i, j) * supportVector.get(j) != 0) {
                matrixBuilderLong.set(i, j, matrix.get(i, j) *
supportVector.get(i));
            }
        }
    }
    return (CoordinateMatrixLong) matrixBuilderLong.get();
}
```

Listing 16: Multiply by support vector

3.9 Test Module

In the test as mentioned before we are going to do three comparisons to check our results:

- A x B = ExpectedC
- A x Bv = ExpectedCv
- DenseMatrixMultiplication = Compressed Matrix Multiplication

The test cases are situated within the MatrixMultiplicationLong class, residing in the test module. We employ JUnit and the assertEquals method to compare the content of two matrices during the testing process. Before executing the tests, JUnit enables us to set up specific variables using the "@BeforeClass" decorator. In this instance, we initialize a support vector and deserialize a matrix from the resources folder.

After this preparation, we proceed to create functions responsible for generating the essential components of our test, such as A, B, BV, and the results of each multiplication. To formally declare a test, we utilize the "@Test" decorator. An illustrative example of a test is presented below:

```java
@Test
public static void testMatrixMultiplicationSupportVector(){
    // Calculate the result C using your matrix multiplication
function
    MatrixOperationsLong matrixOperationsLong = new
MatrixOperationsLong();
```

```
        CoordinateMatrixLong C = matrixOperationsLong.
    multiplyCompressed(initializeMatrixA(), initializeMatrixBV());

        // Assert that the actual result is equal to the expected
    result
        assertEquals(C.getCoordinates(), initializeExpectedResultCV
    ().getCoordinates());
        System.out.println("Completed test with support vector! (
    AxBv = ExpectedCv)");
}
```

Listing 17: Test Support Vector

In this illustrative example, we carry out Compressed Matrix multiplication by supplying A (in CRS format) and BV (in CCS format) as arguments, subsequently assessing the results against our expected outcomes. The test can be initiated from within the primary function housed in this class. Within this function, we also take measurements of the execution time for each test. In order to assess the benefits of the compressed format, we generate random matrices with an 80% sparsity rate (i.e., 80% of values are non-zero). The primary function is structured as follows, and since I couldn't implement parameterized testing, I manually substitute the file string for each run in System.setProperty():

```
    public static void main(String[] args) throws
    FileNotFoundException {
        System.setProperty("matrixFileAPath", "/matrix1024.mtx");

        // Call setUpCoordinateMatrix to initialize
    matrixCoordinates
        setUpCoordinateMatrix();
        setUpSupportVector();


        // Measure and display execution time for
    testMatrixMultiplicationNormal() (1st time)
        long startTime = System.currentTimeMillis();
        testMatrixMultiplicationCompressed();
        long endTime = System.currentTimeMillis();
        long executionTime = endTime - startTime;
        System.out.println("testMatrixMultiplicationCompressed()
    executed in " + executionTime + " milliseconds");

        // Measure and display execution time for
    testMatrixMultiplicationNormal() (2nd time)
        startTime = System.currentTimeMillis();
        testMatrixMultiplicationSupportVector();
        endTime = System.currentTimeMillis();
        executionTime = endTime - startTime;
        System.out.println("testMatrixMultiplicationSupportVector()
     executed in " + executionTime + " milliseconds");

        // Measure and display execution time for compareWithDense
    ()
        startTime = System.currentTimeMillis();
        compareWithDense();
        endTime = System.currentTimeMillis();
        executionTime = endTime - startTime;
        System.out.println("compareWithDense() executed in " +
    executionTime + " milliseconds");
    }
```

Listing 18: Test main

4 Experiment

Following our testing, we meticulously recorded the execution times for each test type using matrices of specific sizes. It's essential to note that this testing was limited to a select few matrices. Any matrices with more than 1024 rows or columns proved impractical due to prohibitively long execution times. We will delve into potential causes later, but first, let's examine our data:

Size	C (ms)	CV (ms)	CD (ms)
128	875	3436	199
256	4724	9733	1892
512	60415	190567	33311

- C = Matrix Multiplication Compressed
- CV = Matrix Multiplication Compressed with vector
- CD = Matrix Multiplication Compressed and Dense

We have successfully completed all the tests, providing strong evidence that our compressed matrix multiplication operates effectively. However, it's important to note that the execution time of the Dense Matrix test significantly outperforms the others. In fact, we can create a speed-up measure for each of the sizes, yielding the following results

	Speed Up indicator based for CD
Matrix Multiplication Compressed (ms)	1.864697
Matrix Multiplication Compressed with vector (ms)	5.754929

As observed, CD exhibits an execution time nearly six times faster than CV and almost twice as fast as C. This outcome is rather unexpected when we consider our initial hypothesis. The primary concept behind the compressed matrix was to economize space based on the presence of non-zero elements, a principle that doesn't align with the current results. We can visualize the progression of time graphically, though it serves to reaffirm our findings rather than introduce any new insights. 1.

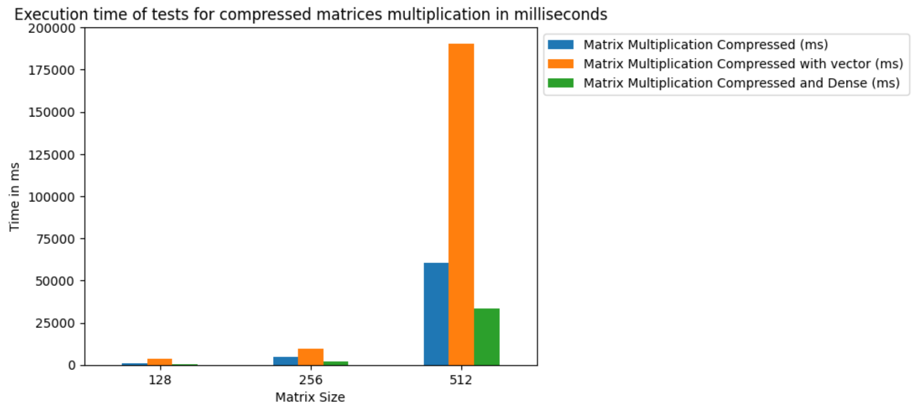

Figure 1: Execution Time test

5 Conclusion

We can conclude that our compressed matrix multiplication has performed as expected, especially when compared to vector multiplication and traditional dense matrix multiplication. However, it's evident that the ArrayList implementation falls short of achieving the performance levels seen with Dense Matrix multiplication.

In our previous studies, we utilized a 2D array to represent dense matrices. This choice was reiterated throughout our work, as the inefficiency of the ArrayList design became increasingly apparent. Notably, after running tests for approximately 40 minutes, it became clear that the Compressed Matrix Multiplication with vectors did not complete successfully for a matrix size of 1024.

As a result, it would have been more advisable to implement our compressed matrix using arrays instead of ArrayLists. However, making this transition would have necessitated several significant modifications, which unfortunately I was unable to undertake. This limitation arises from the reliance on specific utilities that ArrayList can support, primarily for sorting elements by columns or rows before converting them into a compressed format using Coordinate List.

Also the use of an HDD disk is not recommended for this type of tasks. The experiments were ran also in a second disk(SSD) and the execution time was reduced to a half. Nevertheless, the matrices of huge size still being a problem.

In conclusion, when it comes to optimizing space usage, primitive data types like arrays prove to be the most efficient choice. While more advanced data structures offer additional functionality and convenience, they often come at the cost of increased memory and processing time. For the management of large volumes of data, opting for lower-level data structures is the more pragmatic approach.

6 Future Work

In future research, it is evident that implementing the compressed matrix format using ArrayLists does not fully unlock its potential. Instead, opting for primitive Array types would be a more appropriate choice. However, once the code had been designed around ArrayLists, I encountered difficulties when attempting to transition to the use of primitive types. My concern about potentially breaking the code made me more cautious about releasing these results, despite their unfavorable impact on execution time.

Additionally, I had a strong desire to test these functions with matrices from the Suite Sparse Matrix Collection. Overcoming the aforementioned challenge would enable me to work with larger and more complex matrices found in this repository, instead of using my own generated matrices.