Matrix multiplication

Carlos Santana Esplá.

October 29, 2023

Abstract

As we have seen in other works, one of the most popular tests within benchmarking is the use of matrices, since typically, the computational cost of them tends to be high due to the use of nested loops to traverse them. Therefore, our program will calculate the execution time of a massive matrix with a size of 500.000 * 500.000.

The methods used were compressors, which serve to store in a list the positions where a value is found, the row and column where it is located, and the value itself attributed to that position. With this, our project will not have to traverse the entire matrix, but only where a value is found and then find the multiplication.

What we have found is an exponential improvement in time since, according to other articles, the execution time of a dense matrix on a conventional computer is too high, exceeding two hours, while with our implementation, it does not reach two hours.

1 Introduction

Matrix multiplication is a fundamental operation in computational mathematics and plays a pivotal role in various applications, especially in the realm of programming. In the context of Java programming, the efficiency of matrix multiplication becomes crucial when dealing with large datasets. This is where innovative techniques like Compressed Column Matrix and Compressed Row Matrix come into play.

In traditional matrix multiplication, the computational cost can be prohibitively high due to the need to traverse every element in the matrices. However, compressed representations provide a more efficient alternative. Compressed Column Matrix stores only the non-zero elements along with their row indices, significantly reducing the memory footprint. On the other hand, Compressed Row Matrix optimizes storage by storing non-zero elements along with their column indices.

In this exploration, we delve into the realm of matrix multiplication in Java, exploring the benefits of employing compressed matrix representations to enhance both computational efficiency and memory utilization.

2 Experiments

Our experiment is based on processing data downloaded from this sparse.tamu.edu, which provides a .mtx file. This file contains a matrix where the first two positions in each row indicate the row and column where the value in the third position is located.

Having explained the scope of the problem, let's delve into the implementation. My code first undergoes a preprocessing step to clean the data since it is not initially processed. During this preprocessing, we call the `CoordinateBuilder` class to construct our matrix using the aforementioned coordinates.

“‘java

```
package Bigdata.matrixmultiplication.matrixbuilder;

public class CoordinateBuilder {

    public static String size;
    public static List<Coordinate> createMatrix(BufferedReader br) throws IOException {
        int separationLineCount = 0;
        int firstCoordinate = 0;
        String line;
        while ((line = br.readLine()) != null) {
            if (line.trim().equals("%-----------------------------------------------------
                separationLineCount++;
                if (separationLineCount == 2) {
                    break;
                }
            }
        }
        List<Coordinate> coordinateList = new ArrayList<>();
        while ((line = br.readLine()) != null && !line.trim().isEmpty()) {

            firstCoordinate++;
            if (firstCoordinate == 1) {
                size = line.split(" ")[0];
                continue;
            }
            String[] values = line.split(" ");
            int i = Integer.parseInt(values[0]) - 1;
            int j = Integer.parseInt(values[1]) - 1;
            double value = Double.parseDouble(values[2]);

            coordinateList.add(new Coordinate(i, j, value, size));
        }
        Collections.sort(coordinateList, Comparator.comparingInt((Coordinate c) -> c.i).thenComp
        return coordinateList;
    }
    public static String getSize(){
        return size;
    }
}
```

As we can see, this class returns a list of type `Coordinate` (representing the row, column, and value) to make it easier for us to manipulate the data.

2.1 storage methods

As observed in the code, we store matrices in two different ways using the classes `CompressedColumnMatrix` and `CompressedRowMatrix`. These methods involve storing the matrix using three lists. The first list stores a row/column pointer, indicating, either for a column or a row, how often there is a value different from 0, essentially indicating the presence of a value. The next list stores the row or column where the value is located. Finally, we store the actual value in the order in which it was encountered.

2.1.1 Advantages of using Compressed Row/Column Matrix:

- **Efficiency in Row/Column Access:** Provides fast access to elements in a specific row/column, as data is stored row-wise and pointers indicate the presence of non-zero values.
- **Space Savings in Sparse Matrices:** Particularly efficient for sparse matrices with many zero elements, as only non-zero elements along with their column/row indices are stored.
- **More Efficient Row/Column Operations:** Operations involving row/column manipulations, such as scalar additions or multiplications, can be more efficient due to the data organization.

2.1.2 Disadvantages of using Compressed Row/Column Matrix:

- **Less Efficient Column/Row Access:** Accessing elements in a specific column/row may be less efficient, as the data is organized by rows.
- **More Complex Column/Row Operations:** Operations involving column/row manipulations may require more steps and be less efficient due to the storage structure.

The way we create and store these matrices is as follows:

```
public class CompressedColumnMatrixBuilder {
    public static int matrixColumnSize;

    public static ArrayList getColumnPointer(List<Coordinate> coordinateList, String size) {
        ArrayList columnPointer = new ArrayList<>();
        matrixColumnSize = Integer.parseInt(size);
        int pointer = 0;
        int rangeCounter = 0;
        for (Coordinate coordinate : coordinateList) {
            if (coordinate.j != pointer) {
                for (int j = 0; j < (coordinate.j - pointer); j++)
                    columnPointer.add(rangeCounter);
                pointer = coordinate.j;
            }
            rangeCounter++;
        }

        for (int j = 0; j < matrixColumnSize; j++) {
            columnPointer.add(rangeCounter);
        }
        return columnPointer;
    }
    public static ArrayList rows(List<Coordinate> coordinateList){
        ArrayList rows = new ArrayList<>();
        for (Coordinate coordinate : coordinateList) {
            rows.add(coordinate.i);
        }
        return rows;
    }
    public static ArrayList columnsValues(List<Coordinate> coordinateList){
        ArrayList columnValues = new ArrayList<>();
        for (Coordinate coordinate : coordinateList) {
            columnValues.add(coordinate.value);
        }
        return columnValues;
    }
    public static int getMatrixColumnSize(){
        return matrixColumnSize;
    }
    public static List<CompressedColumnMatrix> ccs(ArrayList columnPointer, ArrayList rows, Arra
        List<CompressedColumnMatrix> ccslist = new ArrayList<>();
        ccslist.add(new CompressedColumnMatrix(columnPointer, rows, columnsValues, matrixColumnS
        return ccslist;
    }

}
```

2.2 Operation

Finally, the class where we calculate the matrix multiplication follows the following implementation.

```
public class MatrixMultiplication {

    public static List<Coordinate> multiplyMatrices(List<?> crsList, List<?> ccsList) {
        List<Coordinate> result = new ArrayList<>();
        for (Object crsOrInteger : crsList) {
            for (Object ccsOrInteger : ccsList) {
                for (int i = 0; i < CompressedRowMatrixBuilder.getMatrixRowSize(); i++) {
                    double sum = getSum((CompressedRowMatrix) crsOrInteger, i, (CompressedColumn
                    if (sum != 0) {
                        result.add(new Coordinate(i, CompressedColumnMatrixBuilder.getMatrixColu
                    }
                }
            }
        }
        return result;
    }

    private static double getSum(CompressedRowMatrix crs, int i, CompressedColumnMatrix ccs) {
        double sum = 0;
        int rowStart = crs.getRowPointer().get(i);
        int rowEnd = (i == crs.getMatrixRowSize() - 1) ? ccs.getMatrixColumnSize() : crs.getRowP

        for (int j = rowStart; j < rowEnd; j++) {
            int col = ccs.getColumnPointer().get(j);
            double crsValue = crs.getRowsValue().get(j);
            double ccsValue = ccs.getColumnsValue().get(j);
            sum += crsValue * ccsValue;
        }
        return sum;
    }
}
```

In this final operation, we return a list of type `Coordinate` again because it is the most optimal way I found for storing the resulting matrix. If we were to construct a dense matrix, we would need to add zeros in every position where there is no value in the matrix, which would incur a high computational cost.

For my code, the runtime was 6238 seconds, which is equivalent to 1 hour and 44 minutes. The entire code was executed on the following terminal: Asus Zenbook.

3 Conclusion

In conclusion, when addressing our problem, we implemented matrix multiplication using Compressed Row/Column Matrix storage methods. While we observed variations in execution times for a 500,000 by 500,000 matrix (falling within the range of 4000-6000 seconds), it is crucial to emphasize the significance of these storage methods in the realm of Big Data.

The choice of compressed matrices is essential not only for efficient data filtering and processing but also for optimizing performance compared to dense matrices. In a dense matrix, execution time would increase significantly due to the need to traverse each row and column, position by position, even when many values are zero.

The implementation of pointers in compressed matrices improves the speed and efficiency of the process, which is particularly valuable in Big Data scenarios where effective resource management and reducing computational complexity are paramount. These methods provide an efficient solution for large matrices by minimizing unnecessary processing of null elements and offering a faster and more efficient approach to matrix operations.