Benchmark of matrix multiplication (3)

Joel Clemente López Cabrera

October 29, 2023

1 Introduction

This report focuses on the project of Sparse and Dense matrix multiplication implemented in the Java programming language. The primary objective of the project is to assess and compare the performance of matrix multiplication in these two categories. To perform SparseMatrixMultiplication, we previously implemented the compression of matrices in two different forms: CompressedColumnMatrix (CCS) and CompressedRowMatrix (CRS). A detailed description of the project and its results is presented below.

2 Project description

The project focuses on implementing Sparse and Dense matrix multiplication in Java. Sparse matrices primarily consist of null values, requiring a specialized approach to optimize the implementation. As mentioned earlier, we need to implement CCS and CRS compressions to perform Sparse matrix multiplication correctly, as this type of multiplication is done with one matrix in CRS format and the other in CCS format.

On the other hand, Dense matrices contain a substantial amount of non-null data. Therefore, for matrix multiplication, we simply perform the multiplication of two Dense matrices in a manner similar to basic matrix multiplication.

3 Implementation and Methodology

To carry out this project, various class implementations were developed to complete the work effectively.

Initially, a class named MatrixFileReader was implemented (in its respective reader module). This class is used to read the file *"mc2depi.mtx"*, which contains the matrix in coordinate format and includes metadata that needs to be removed beforehand. This enables us to store the list of coordinates in a CoordinateMatrix.

In the matrix module, we need to implement the CoordinateMatrix class, which collects the matrix's coordinates, the number of rows, and the number of columns. We will also implement a Coordinate class, which contains only the row number, column number, and the specific coordinate value.

In this module, we will create the respective DenseMatrix and SparseMatrix classes, which take the format of the CoordinateMatrix and transform it into Dense and Sparse matrices. These matrices will provide the value retrieved through a 'get' method, along with the number of rows and columns.

Finally, in this module, you will find the CompressedColumnMatrix and CompressedRowMatrix classes, which perform the respective compressions of matrices to be used in the next module, the operations module, where both DenseMatrixMultiplication and SparseMatrixMultiplication implementations can be found.

Additionally, there is a builder module where we construct a builder for each class in the matrix module.

```
for (int i = 0; i < crsMatrix.getNumRows(); i++) {
    for (int j = 0; j < ccsMatrix.getNumCols(); j++) {
        long result = 0L;
        int crsStart = crsRowPointers[i];
        int crsEnd = crsRowPointers[i + 1];
        int ccsStart = ccsColumnPointers[j];
        int ccsEnd = ccsColumnPointers[j + 1];
        int crsIndex = crsStart;
        int ccsIndex = ccsStart;

        while (crsIndex < crsEnd && ccsIndex < ccsEnd) {
            int crsCol = crsColumnIndices[crsIndex];
            int ccsRow = ccsRowIndices[ccsIndex];
            if (crsCol == ccsRow) {
                result += (long) (crsValues[crsIndex] * ccsValues[ccsIndex]);
                crsIndex++;
                ccsIndex++;
            } else if (crsCol < ccsRow) {
                crsIndex++;
            } else {
                ccsIndex++;
            }
        }

        resultBuilder.set(i, j, result);
    }
}
```

Figure 1: SparseMatrixMultiplication

```
DenseMatrixBuilder resultBuilder = new DenseMatrixBuilder(numRowsA,numColsA);

for (int i = 0; i < numRowsA; i++) {
    for (int j = 0; j < numColsB; j++) {
        long sum = 0;
        for (int k = 0; k < numColsA; k++) {
            sum += a.get(i, k) * b.get(k, j);
        }
        resultBuilder.set(i, j, sum);
    }
}

return resultBuilder.get();
```

Figure 2: DenseMatrixMultiplication

4 Experiments

The results of the experiments revealed significant differences in the performance of Sparse and Dense matrix multiplication in Java. These experiments used various matrix examples, although not very large, due to the experiment being conducted with the sample matrix "*mc2depi.mtx*." The execution time reached approximately 6500 seconds due to suboptimal performance in the implementation. Nonetheless, in these diverse experiments, the following trends were observed:

-Sparse matrix multiplication in Java demonstrated remarkable performance in terms of execution speed, even when running multiple executions. There was significant efficiency in handling matrices with null values. For instance, when using a matrix named "*492bus*," as the name suggests, which has a size of 494x494, the multiplication resulted in just 21 ms. This was an almost immediate outcome, highlighting its strength and speed. It was also tested with a matrix of approximately 1000x1000, and the result was also quite immediate, taking only 60 ms.

-On the other hand, Dense matrix multiplication did not excel in terms of fast and efficient performance. The implementation could handle matrices with a substantial amount of non-null data. However, when dealing with a matrix of the example size of 500k x 500k, the DenseMatrixMultiplication exhibited a performance decline. In the case of the 494x494 matrix example, it took a staggering 2480 seconds to execute and In the case of the 1000x1000 matrix example, it took a staggering 4000 seconds to execute. When attempting to use it with a 500k x 500k matrix, an error occurred: **"OutOfMemoryError: Java heap space,"** and the execution time could not be achieved.

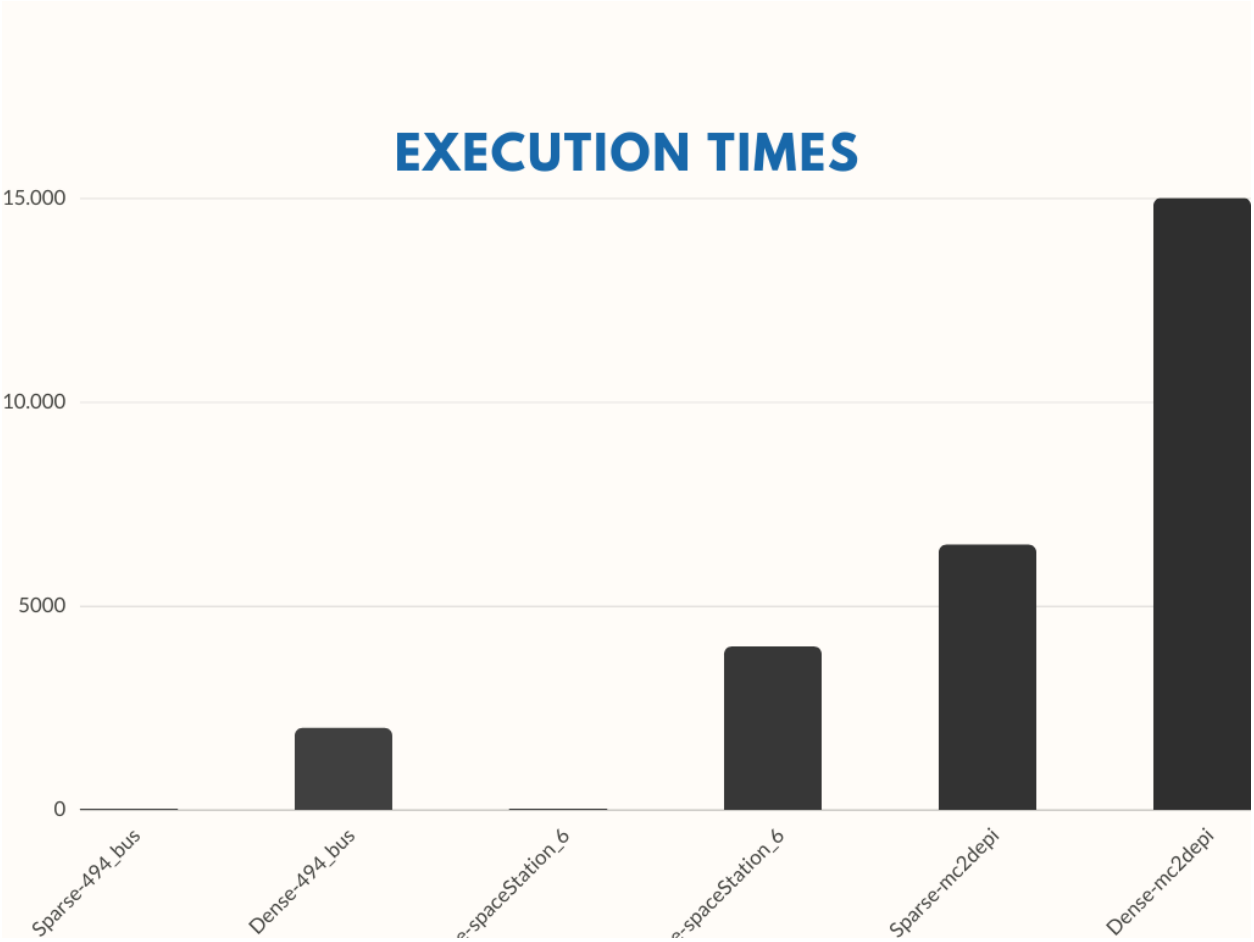

Figure 3: Execution Times

5 Conclusions

In summary, the results indicate that SparseMatrixMultiplication is a robust choice for matrix multiplication. The implementation has shown fast and efficient performance compared to DenseMatrixMultiplication. This makes the former implementation an outstanding choice for operations in projects requiring high performance and efficiency.

Nevertheless, the algorithm used and the project's approach promise the potential for further optimization to achieve even faster matrix multiplications.

6 Future Work

In future research, additional optimization of the implementation and the exploration of advanced techniques to enhance performance, particularly in both multiplication implementations, could be considered. Furthermore, there is potential to investigate the application of these matrix multiplication algorithms in practical projects across various disciplines.

7 Code on GitHub

In this last section, you can find the link to my GitHub, where there will be a repository the scientific paper and the code of the project.
`https://github.com/JoelClemente/Benchmark-of-matrix-multiplication-3`