Matrix Multiplication in Java

Elisa-Marie Breeze Rebstock

November 2024

Abstract

This paper analyzes various different algorithms for matrix multiplication in Java, some more optimized than others. The investigation focuses on matrix multiplication, using different methods like loop unrolling, Strassen's algorithm and the difference in performance between dense and sparse matrices, including different sparsity levels and testing these in sparse-specific methods as well. To compare the different algorithms benchmark tests were executed on the different algorithms so that the performance of each method could be analyzed and compared with each other.

1 Introduction

In this paper you will see the implementation and analysis of the performance of various different matrix multiplication algorithms, like for example Loop Unrolling, Strassen's algorithm, CSR, CSC and the difference between dense and sparse matrices, to see which one performs the best of them all. We will focus on the Java programming language, and implement numerous different algorithms, testing them each with benchmarks and memory usage calculations.

2 Methodology

Link to the GitHub Repository of this assignment:
https://github.com/ElisaBreeze/BigData.git

To compare the different algorithms, I created different matrices, dense ones (sparsity level 0.0) and also sparse matrices with sparsity 0.5 and 0.9. I used these matrices in the different algorithms to see which one performed better and I also included two sparse-specific Algorithms to see how well they performed. Based off the first task, I used the same benchmark and memory usage calculation methods, as well as recycling the basic matrix multiplication and the basic optimized multiplication algorithm. The other algorithms used in this investigation were researched thoroughly and then I applied them to the project. Each one of the new algorithms will be explained in a more extensive way in its subsection:

2.1 Basic Matrix Multiplication

Here we can see the basic matrix multiplication code, that uses 3 loops to multiplicate matrix A and B, creating the result matrix, C. This code was taken from the task 1, where I decided to implement a similar code to the example code given by the teachers, but making a few changes that made the code look tidier and simpler: in the creation of the matrix in the testing code, I added the solution matrix (c) to the setup code, in order to have it all in one place and be more organized. I also passed on the size of the matrix as a value to the matrix multiplication function, as it was already initialized for the setup function. This way, the code was more optimized because it didn't have to calculate the matrix size in the matrix multiplication code, but instead it was given to it automatically.

```java
public double[][] multiply(double[][] a, double[][] b, double[][] c,
    int n){
     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];
             }
         }
     }
```

```
        return c;
    }
```

2.2 Optimized Basic Matrix Multiplication

This version is a simple optimization, but very effective, and I also used it in task 1. It involves reducing the number of memory accesses by storing a temporary variable before iterating the next, which reduces the accesses to the matrix A by quite a lot. To do this, I simply changed the order of accessing to a more sequential order by switching k and j and storing the value of the matrix a, to avoid multiple accesses to the same element.

```
public double[][] multiply(double[][] a, double[][] b, double[][] c,
     int n) {
        for (int i = 0; i < n; i++) {
            for (int k = 0; k < n; k++) {
                double temp = a[i][k];
                for (int j = 0; j < n; j++) {
                    c[i][j] += temp * b[k][j];
                }
            }
        }
        return c;
    }
```

2.3 Loop Unrolling

This optimization technique is based on processing multiple elements in each loop, reducing the number of iterations and also improving cache efficiency, making it work better especially for large matrices.

Loop unrolling is a way to make loops run faster by doing more work in each iteration instead of one calculation at a time. In the basic matrix multiplication, you go through each row and each column, multiplying the elements one at a time, which can take a long time because there are so many iterations. This algorithm optimized the whole process by instead of multiplying one element at a time in the inner loop, it does 4 at once, which means the program ends up spending less time checking the loop conditions and more time on the actual calculation, which should reduce the time it takes to multiply the matrices.

After researching how many checks in each loop it should do, it was clear that a good number was 4, because it reduces the loop but keeps the code manageable, having a good balance between improving performance and maintaining readability and simplicity.

Relevant information regarding this algorithm was sourced from the following websites: - www.geeksforgeeks.org - medium.com - icl.utk.edu

```
public double[][] multiply(double[][] a, double[][] b, double[][] c,
    int n) {
      for (int i = 0; i < n; i++) {
         for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k += 4) {
               c[i][j] += a[i][k] * b[k][j]
                     + (k + 1 < n ? a[i][k + 1] * b[k + 1][j] : 0)
                     + (k + 2 < n ? a[i][k + 2] * b[k + 2][j] : 0)
                     + (k + 3 < n ? a[i][k + 3] * b[k + 3][j] : 0);
            }
         }
      }
      return c;
   }
```

2.4 Strassen's Algorithm

The goal of this algorithm is to multiply two matrices is a more efficient way by reducing complexity. Although the code is more complex, in the inside you can see that it's way of working is that it only uses 7 multiplications and submatrices.

This is how it works: Firstly, we split each matrix into 4 smaller sumbatrices, to then find 7 products, each one involving adding or subtracting smaller submatrices and multiplying the results. Once that was done, the 7 products are used to build the 4 submatrices which are then merged into the final matrix.

This method should make the multiplication fast for large matrices, but on the contrary could make it slower for smaller ones.

Relevant information regarding this algorithm was sourced from the following websites: - www.wikipedia.org - www.geeksforgeeks.org - www.javatpoint.com

```
public double[][] multiply(double[][] A, double[][] B) {
      int n = A.length;
      if (n == 1) {
         double[][] result = new double[1][1];
         result[0][0] = A[0][0] * B[0][0];
         return result;
      }

      int newSize = n / 2;
      double[][] a11 = new double[newSize][newSize];
      double[][] a12 = new double[newSize][newSize];
      double[][] a21 = new double[newSize][newSize];
      double[][] a22 = new double[newSize][newSize];

      double[][] b11 = new double[newSize][newSize];
      double[][] b12 = new double[newSize][newSize];
      double[][] b21 = new double[newSize][newSize];
      double[][] b22 = new double[newSize][newSize];

      split(A, a11, 0, 0);
```

```
        split(A, a12, 0, newSize);
        split(A, a21, newSize, 0);
        split(A, a22, newSize, newSize);
        split(B, b11, 0, 0);
        split(B, b12, 0, newSize);
        split(B, b21, newSize, 0);
        split(B, b22, newSize, newSize);

        double[][] m1 = multiply(add(a11, a22), add(b11, b22));
        double[][] m2 = multiply(add(a21, a22), b11);
        double[][] m3 = multiply(a11, subtract(b12, b22));
        double[][] m4 = multiply(a22, subtract(b21, b11));
        double[][] m5 = multiply(add(a11, a12), b22);
        double[][] m6 = multiply(subtract(a21, a11), add(b11, b12));
        double[][] m7 = multiply(subtract(a12, a22), add(b21, b22));

        double[][] c11 = add(subtract(add(m1, m4), m5), m7);
        double[][] c12 = add(m3, m5);
        double[][] c21 = add(m2, m4);
        double[][] c22 = add(subtract(add(m1, m3), m2), m6);

        double[][] result = new double[n][n];
        combine(c11, result, 0, 0);
        combine(c12, result, 0, newSize);
        combine(c21, result, newSize, 0);
        combine(c22, result, newSize, newSize);

        return result;
    }
```

2.5 Blocking

This method aims to improve the efficiency by dividing large matrices into smaller blocks (in this case, block size was set on 10). These blocks fit into the cache memory, which then should reduce the number of accesses to slower memory. The method consists on iterating over the matrices A and B, converting them into smaller blocks, and within each block it performs the traditional matrix multiplication. The result matrix C is filled block by block. This then should help with large matrices especially.

Relevant information regarding this algorithm was sourced from the following website: - csapp.cs.cmu.edu

```
  public double[][] multiply(double[][] A, double[][] B, int blockSize)
        {
        int n = A.length;
        double[][] C = new double[n][n];

        // Divide into Blocks and multiply
        for (int i = 0; i < n; i += blockSize) {
            for (int j = 0; j < n; j += blockSize) {
                for (int k = 0; k < n; k += blockSize) {
                    for (int ii = i; ii < Math.min(i + blockSize, n); ii
                        ++) {
```

```
                for (int jj = j; jj < Math.min(j + blockSize, n);
                     jj++) {
                    for (int kk = k; kk < Math.min(k + blockSize,
                         n); kk++) {
                        C[ii][jj] += A[ii][kk] * B[kk][jj];
                    }
                }
            }
        }
    }
  }
  return C;
}
```

2.6 Sparse Matrix CSC Multiplication

This Algorithm handles sparse matrices, using the Compressed Sparse Column (CSC) format. The multiplication method used efficiently iterated over non-zero elements of both matrices, computing the result and then returning the resulting matrix in CSC format. It includes a method called convertToCSC with converts the dense matrix into the CSC format by storing the non-zero values, row indices and column pointers. It optimized the algorithm by avoiding operations on zero elements, which therefore makes it a lot more efficient for sparse matrices.
This code was taken from the teachers uploaded material, and includes the conversion to the sparse matrix format as well as the multiplication. For simplicity and cleanness reasons I will only include the multiply function in code below.

```
public CSCMatrix multiply(CSCMatrix B) {
          if (this.cols != B.rows) {
              throw new IllegalArgumentException("Matrix dimensions do
                   not match for multiplication.");
          }

          List<Double> resultValues = new ArrayList<>();
          List<Integer> resultRowIndices = new ArrayList<>();
          List<Integer> resultColPointers = new ArrayList<>();
          resultColPointers.add(0);

          // Temporary array to store result for a single column in C
          double[] colResult = new double[this.rows];

          // Perform CSC matrix multiplication (this * B)
          for (int jB = 0; jB < B.cols; jB++) {
              // Clear colResult
              Arrays.fill(colResult, 0.0);

              // For each non-zero element in column jB of matrix B
              for (int k = B.colPointers[jB]; k < B.colPointers[jB +
                   1]; k++) {
                  int rowB = B.rowIndices[k]; // Row index in matrix B
                  double valB = B.values[k]; // Value of B at rowB and
                       column jB
```

```
            // Multiply column jB of B by corresponding row of
                 this matrix
            for (int i = this.colPointers[rowB]; i < this.
                 colPointers[rowB + 1]; i++) {
                int rowA = this.rowIndices[i];
                double valA = this.values[i];
                colResult[rowA] += valA * valB;
            }
        }

        // Save the result of column jB in CSC format
        int nonZeroCount = 0;
        for (int i = 0; i < this.rows; i++) {
            if (colResult[i] != 0.0) {
                resultValues.add(colResult[i]);
                resultRowIndices.add(i);
                nonZeroCount++;
            }
        }

        resultColPointers.add(resultColPointers.get(
             resultColPointers.size() - 1) + nonZeroCount);
    }

    // Convert lists to arrays
    double[] resultValuesArray = resultValues.stream().
         mapToDouble(Double::doubleValue).toArray();
    int[] resultRowIndicesArray = resultRowIndices.stream().
         mapToInt(Integer::intValue).toArray();
    int[] resultColPointersArray = resultColPointers.stream().
         mapToInt(Integer::intValue).toArray();

    return new CSCMatrix(resultValuesArray,
         resultRowIndicesArray, resultColPointersArray, this.rows
         , B.cols);
}
```

2.7 Sparse Matrix CSR Multiplication

This Algorithm is similar to the CSC algorithm, but performs the multiplication using the Compressed Sparse Row (CSR) format. The multiplication is done by iterating over the non-zero elements in rows of the first matrix, and the corresponding rows in the second matrix, updating the result in CSR format. It also includes a convertToCSR method to convert it into the CSR format by storing non-zero elements and their column indices, and updating the row pointers. It optimized by avoiding operating on zero elements, therefore making it more efficient for sparse matrices as well.

This code was also taken from the teachers uploaded material, and includes the conversion to the sparse matrix format as well as the multiplication. For simplicity and cleanness reasons I will only include the multiply function in code below.

```java
public CSRMatrix multiply(CSRMatrix B) {
          if (this.cols != B.rows) {
              throw new IllegalArgumentException("Matrix␣dimensions␣do␣
                   not␣match␣for␣multiplication.");
          }

          List<Double> resultValues = new ArrayList<>();
          List<Integer> resultColumnIndices = new ArrayList<>();
          List<Integer> resultRowPointers = new ArrayList<>();
          resultRowPointers.add(0);

          // Temporary array to store result for a single row in C
          double[] rowResult = new double[B.cols];

          // Perform CSR matrix multiplication (this * B)
          for (int i = 0; i < this.rows; i++) {
              // Clear rowResult
              Arrays.fill(rowResult, 0.0);

              // For each non-zero element in row i of this matrix
              for (int j = this.rowPointers[i]; j < this.rowPointers[i
                   + 1]; j++) {
                  int colA = this.columnIndices[j]; // Column index in
                       matrix A
                  double valA = this.values[j];   // Value of A at row i
                        and column colA

                  // Multiply row of A by corresponding row in B (which
                        is stored in CSR format)
                  for (int k = B.rowPointers[colA]; k < B.rowPointers[
                       colA + 1]; k++) {
                      int colB = B.columnIndices[k]; // Column index in
                           matrix B
                      double valB = B.values[k];    // Value of B at
                           column colB
                      rowResult[colB] += valA * valB;
                  }
              }

              // Save the result of row i in CSR format
              int nonZeroCount = 0;
              for (int j = 0; j < B.cols; j++) {
                  if (rowResult[j] != 0.0) {
                      resultValues.add(rowResult[j]);
                      resultColumnIndices.add(j);
                      nonZeroCount++;
                  }
              }

              resultRowPointers.add(resultRowPointers.get(
                   resultRowPointers.size() - 1) + nonZeroCount);
          }

          // Convert lists to arrays
          double[] resultValuesArray = resultValues.stream().
              mapToDouble(Double::doubleValue).toArray();
          int[] resultColumnIndicesArray = resultColumnIndices.stream
```

```
            ().mapToInt(Integer::intValue).toArray();
        int[] resultRowPointersArray = resultRowPointers.stream().
            mapToInt(Integer::intValue).toArray();

        return new CSRMatrix(resultValuesArray,
            resultColumnIndicesArray, resultRowPointersArray, this.
            rows, B.cols);
    }
```

3 Experiments and Results

3.1 Benchmark Methodology

All the algorithms are tested through benchmark and memory usage calculation, using various matrices of varying sizes, and also tested with various different sparsity levels. The results were taken from the benchmark output, and the memory usage of each algorithm was saved in a CSV, so that once the benchmark process was finished, I had the information in one place for easier information retrieval. The results are shown in the result section below, where I show a table for each algorithm with the results for the different values tested.

3.1.1 Benchmark

```
public CSRMatrix multiply(CSRMatrix B) {
        if (this.cols != B.rows) {
            throw new IllegalArgumentException("Matrix␣dimensions␣do␣
                not␣match␣for␣multiplication.");
        }

        List<Double> resultValues = new ArrayList<>();
        List<Integer> resultColumnIndices = new ArrayList<>();
        List<Integer> resultRowPointers = new ArrayList<>();
        resultRowPointers.add(0);

        // Temporary array to store result for a single row in C
        double[] rowResult = new double[B.cols];

        // Perform CSR matrix multiplication (this * B)
        for (int i = 0; i < this.rows; i++) {
            // Clear rowResult
            Arrays.fill(rowResult, 0.0);

            // For each non-zero element in row i of this matrix
            for (int j = this.rowPointers[i]; j < this.rowPointers[i
                + 1]; j++) {
                int colA = this.columnIndices[j]; // Column index in
                    matrix A
                double valA = this.values[j];   // Value of A at row i
                     and column colA

                // Multiply row of A by corresponding row in B (which
                     is stored in CSR format)
                for (int k = B.rowPointers[colA]; k < B.rowPointers[
                    colA + 1]; k++) {
                    int colB = B.columnIndices[k]; // Column index in
                        matrix B
                    double valB = B.values[k];    // Value of B at
                        column colB
                    rowResult[colB] += valA * valB;
                }
            }

            // Save the result of row i in CSR format
            int nonZeroCount = 0;
            for (int j = 0; j < B.cols; j++) {
                if (rowResult[j] != 0.0) {
                    resultValues.add(rowResult[j]);
                    resultColumnIndices.add(j);
                    nonZeroCount++;
                }
            }

            resultRowPointers.add(resultRowPointers.get(
                resultRowPointers.size() - 1) + nonZeroCount);
        }

        // Convert lists to arrays
        double[] resultValuesArray = resultValues.stream().
            mapToDouble(Double::doubleValue).toArray();
        int[] resultColumnIndicesArray = resultColumnIndices.stream
            ().mapToInt(Integer::intValue).toArray();
        int[] resultRowPointersArray = resultRowPointers.stream().
            mapToInt(Integer::intValue).toArray();

        return new CSRMatrix(resultValuesArray,
            resultColumnIndicesArray, resultRowPointersArray, this.
            rows, B.cols);
    }
```

3.2 Performance Results

The performance of each algorithm is measured in terms of execution time and memory usage. Each test was executed various times to ensure consistency. Below you can see the different algorithms with their respective results, for the chosen values and executed on the same laptop (MacOS M3) and environment (IntelliJ IDEA):

3.2.1 Basic Multiplication results

Matrix Size	Sparsity	Execution Time (ms)	Memory Usage (MB)
16x16	0.0	0.004	2.89
16x16	0.5	0.004	3.33
16x16	0.9	0.004	3.77
128x128	0.0	2.222	9.76
128x128	0.5	2.220	10.88
128x128	0.9	2.221	15.54
1024x1024	0.0	3054.763	51.10
1024x1024	0.5	3148.569	280.94
1024x1024	0.9	3085.585	281.82

3.2.2 Access Optimized Multiplication Results

Matrix Size	Sparsity	Execution Time (ms)	Memory Usage (MB)
16x16	0.0	0.002	3.33
16x16	0.5	0.002	3.33
16x16	0.9	0.002	3.77
128x128	0.0	0.557	10.25
128x128	0.5	0.558	10.88
128x128	0.9	0.558	15.54
1024x1024	0.0	285.990	51.14
1024x1024	0.5	285.966	280.94
1024x1024	0.9	287.242	281.92

3.2.3 Blocking Multiplication results

With Blocks of size 10

Matrix Size	Sparsity	Execution Time (ms)	Memory Usage (MB)
16x16	0.0	0.006	2.89
16x16	0.5	0.006	4.0
16x16	0.9	0.006	4.88
128x128	0.0	2.563	6.0
128x128	0.5	2.806	10.5
128x128	0.9	2.604	15.0
1024x1024	0.0	1371.231	33.51
1024x1024	0.5	1376.474	288.01
1024x1024	0.9	1396.696	290.65

3.2.4 Loop Unrolling Multiplication results

Matrix Size	Sparsity	Execution Time (ms)	Memory Usage (MB)
16x16	0.0	0.004	3.33
16x16	0.5	0.004	3.33
16x16	0.9	0.004	3.77
128x128	0.0	1.554	9.76
128x128	0.5	2.138	10.88
128x128	0.9	1.555	15.54
1024x1024	0.0	3007.980	51.27
1024x1024	0.5	3020.045	280.94
1024x1024	0.9	3004.301	281.92

3.2.5 Strassen's Algorithm Multiplication results

Matrix Size	Sparsity	Execution Time (ms)	Memory Usage (MB)
16x16	0.0	1.144	3.77
16x16	0.5	0.764	4.88
16x16	0.9	0.653	5.76
128x128	0.0	236.521	53.72
128x128	0.5	269.193	15.58
128x128	0.9	226.560	115.77
1024x1024	0.0	80856.955	206.07
1024x1024	0.5	879420.532	126.09
1024x1024	0.9	79665.765	271.71

3.2.6 Sparse Matrix CSC Multiplication results

Matrix Size	Sparsity	Execution Time (ms)	Memory Usage (MB)
16x16	0.0	0.027	3.33
16x16	0.5	0.017	3.33
16x16	0.9	0.006	3.77
128x128	0.0	2.712	7.05
128x128	0.5	1.331	12.79
128x128	0.9	0.514	16.52
1024x1024	0.0	1081.153	234.79
1024x1024	0.5	387.304	143.90
1024x1024	0.9	86.287	146.50

3.2.7 Sparse Matrix CSR Multiplication results

Matrix Size	Sparsity	Execution Time (ms)	Memory Usage (MB)
16x16	0.0	0.026	3.33
16x16	0.5	0.016	3.77
16x16	0.9	0.005	3.77
128x128	0.0	2.652	10.44
128x128	0.5	1.329	15.09
128x128	0.9	0.501	17.38
1024x1024	0.0	1052.208	303.17
1024x1024	0.5	383.960	319.13
1024x1024	0.9	86.340	225.78

4 Result Discussion

Evaluating the results of the different algorithms across different matrix sizes and sparsity levels, it is possible to discuss several important aspects:

Firstly, the execution times:

When looking at the dense matrices results, these are where the sparsity level is 0.0, it is clear that the Access Optimized Multiplication outperformed the other algorithms, especially in the case of the large matrix (1024x1024).

On the other hand, we can see a big difference in the highly sparse matrices with sparsity 0.9, when using the CSR (Compressed Sparse Row) and the CSC (Compressed Sparse Column) algorithms. They both prove efficiency, especially for large matrices, where they significantly reduces execution time. With this, it is clear that sparse matrices show clear time advantages when using sparse-specific algorithms, and especially with high sparsity levels, due to handling the zero elements efficiently.

Secondly, we can see the differences in memory usage:
In the case of dense matrices, the basic one and loop unrolling seem to require higher memory usage as the matrix size increases, and in the case of large matrices, the CSR and CSC algorithms showed the most memory efficiency when the sparsity was high, which therefore demonstrates that sparse-specific methods, once again, can optimize in a better way.

In general, when analyzing the algorithm performances, the Basic Multiplication one proves a baseline and shows the highest memory usage, especially as the matrix size grows. The Access Optimized Multiplication reduces the execution time in all levels but seems to be the most beneficial for dense matrices, as for sparse matrices this one does not seem to offer such a good performance when comparing with the sparse-specific algorithms. The Blocking Algorithm, while reducing the memory footprint, has moderate improvements in execution time compared to the Access Optimized Algorithm, but this method is still better than the Strassen's Algorithm, which was much slower than all the others, especially in in dense matrices, possibly due to the added complexity. And lastly, the two sparse-specific methods CSC and CSR outperformed in an important way the other algorithms in both memory usage and execution time for the sparse matrices, especially as the sparsity increased and the matrix size got bigger.

5 Conclusion

In this paper, various different algorithms for matrix multiplication were explored and various interesting results were found, each showing the strengths of the algorithm depending on matrix size and sparsity. We can conclude that for dense matrix in this case the best results came from the Access Optimized Multiplication, which consistently offered the best performance in execution time and also manageable memory usages. And for sparse matrices, the CSR and CSC Algorithms were a clear winner in both execution time and memory usage, especially for big matrices and as the sparsity level increased.

Overall, we can conclude that if it's possible to work with sparse matrices, the best choices are CSR and CSC, which balance memory usage and speed, but if you are using dense matrices, from the algorithms tested the best option would be the Access Optimized one.