Matrix Multiplication Optimizzation

Wojciech Dróżdż

November 11, 2024

Abstract

This paper focuses on speeding up the matrix-multiplication algorithm by tiling the computation into blocks, which keeps the relevant data longer in the CPU cache. The obtained results speed up the execution time for matrix multiplications by around 400%.

The paper also analyses optimizations for multiplying and storing large sparse arrays. The result makes it possible to correlate the execution time with the number of values in the array instead of it's raw size.

The code for this paper can be found in the GitHub repository

1 Introduction

In the previous paper the speed of matrix multiplication was benchmarked in three languages: *Python* *Java* and *C++*. Out of the three languages the performance was the best for Java, so it was selected as the language in which performance improvements will be made.

Although the fastest matrix multiplication algorithm - the *Strassen Algorithm* has asymptotic complexity of $n^{2.8}$, the standard algorithm was kept, but some improvements were made to make the calculations easier for the CPU.

2 Methodology

2.1 Hardware

All of the tests were performed on a 2023 16 inch MacBook Pro. Relevant specifications:

- Apple M3 Pro - 4 GHz 12-core CPU with 6 performance cores and 6 efficiency cores
- 18GB RAM
- 1024GB of internal storage

In order to preserve consistency all of the tests were performed while the device was plugged into a power source.

2.2 Interpreters and Compilers

- *Java* - Java 17 with JetBrains runtime

2.3 Benchmarking Tools

The *Java Microbenchmark Harness (JHM)*, part of the *OpenJDK* project, was chosen for its microbenchmarking features.

2.4 Benchmarking Methodology

The Benchmarks measure only the time it takes to multiply the matricies, all of the memory is preallocated before running the multiplication method. The time of memory allocation is not measured. Testing was started with 10 warmup iterations before full testing, allowing for just-in-time compilations and appropriate cache optimizations. This was followed by 10 testing iterations, during which the shortest, longest, and average execution times were recorded.

3 Experiments

3.1 Unmodified algorithm

The test was run on multiple different matrix sizes, from small ones all the way up to a *2000x2000* matrix.

Benchmark	(size)	Mode	Cnt	Score	Error	Units
Main.standardMultiply	1	sample	144861	10		ms/op
Main.standardMultiply	5	sample	112841	10		ms/op
Main.standardMultiply	10	sample	143499	0.001	± 0.001	ms/op
Main.standardMultiply	50	sample	82245	0.061	± 0.001	ms/op
Main.standardMultiply	100	sample	9416	0.533	± 0.001	ms/op
Main.standardMultiply	200	sample	999	5.033	± 0.015	ms/op
Main.standardMultiply	300	sample	284	17.794	± 0.076	ms/op
Main.standardMultiply	500	sample	64	83.114	± 0.503	ms/op
Main.standardMultiply	1000	sample	10	924.005	± 29.769	ms/op
Main.standardMultiply	1500	sample	5	4038.276	± 198.267	ms/op
Main.standardMultiply	2000	sample	5	13297.621	± 845.536	ms/op

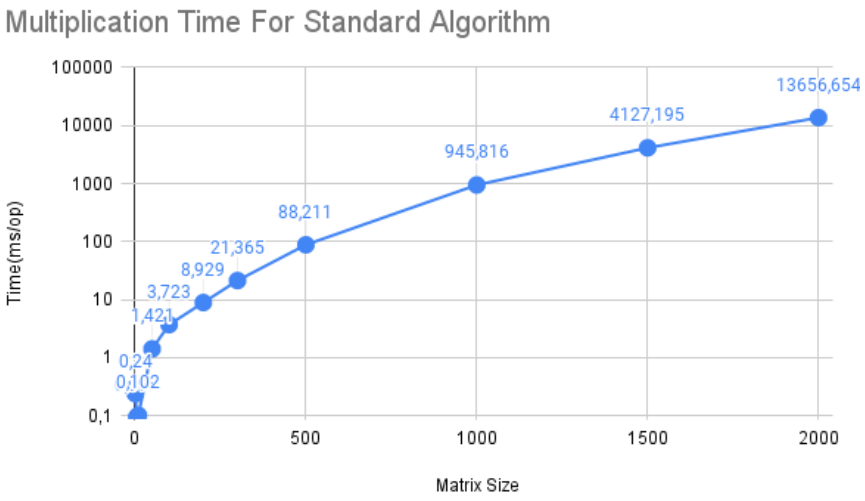

Figure 1: The average excecution times for the standard Algorithm

3.2 Blocked Algorithm

In standard matrix multiplication, elements from two input matrices, usually denoted as A and B, are processed individually, oftentimes following a row-by-column approach. For each row in matrix A and each column in matrix B, the corresponding elements are multiplied together and summed to form an element in the result matrix, C. This element-wise multiplication and summation is repeated systematically until all elements of the new matrix have been computed. However, this approach fails to account for processor caching mechanics, leading to inefficient memory use. To remedy this, a variant known as the blocked algorithm, or tiling, has been developed to optimize the utilization of the processor cache.

The blocked algorithm, or tiling, enables optimized utilization of the processor cache by processing a block of data elements at a time rather than a single element. This approach is designed to retain data in the cache for longer, reducing the need for unnecessary memory fetch operations. In a blocked algorithm, computation is decomposed into blocks. Each row of matrix C is processed, one at a time. Subsequently, instead of processing elements one column at a time, elements are processed a block at a time, allowing for reuse of elements fetched from matrix B. Each block being solved is then processed in chunks, making it more likely to utilize cache memory effectively. Finally, the computation for each element is executed. This strategy enhances performance by optimizing cache utilization. Further explanation of this approach: Coffee Brunch Page

Benchmark	(size)	Mode	Cnt	Score	Error	Units
Main.blockedMultiply	1	sample	118424	10		ms/op
Main.blockedMultiply	5	sample	114047	10		ms/op
Main.blockedMultiply	10	sample	156488	0.001 ±	0.001	ms/op
Main.blockedMultiply	50	sample	60415	0.041 ±	0.001	ms/op
Main.blockedMultiply	100	sample	15819	0.317 ±	0.001	ms/op
Main.blockedMultiply	200	sample	2027	2.480 ±	0.017	ms/op
Main.blockedMultiply	300	sample	532	9.469 ±	0.042	ms/op
Main.blockedMultiply	500	sample	115	44.013 ±	0.346	ms/op
Main.blockedMultiply	1000	sample	15	377.977 ±	4.819	ms/op
Main.blockedMultiply	1500	sample	5	1496.108 ±	38.980	ms/op
Main.blockedMultiply	2000	sample	5	6831.682 ±	950.730	ms/op

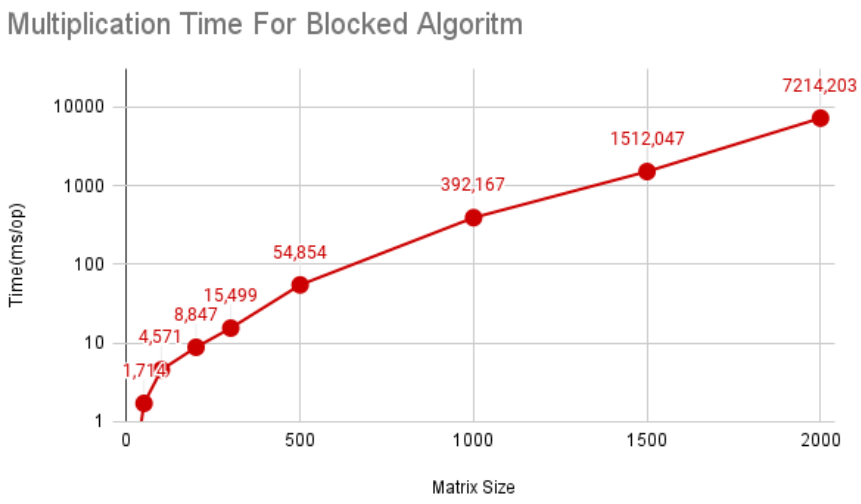

Figure 2: The average excecution times for the blocked algorithm

3.3 Blocked-Column Algorithm

The blocked column algorithm handles matrix multiplication in a more memory-friendly way by focusing on reusing data from columns of matrix B, as opposed to rows of matrix A used in the row-wise blocking. The blocks of computation are organized around column chunks.

For each designated block of columns, the algorithm goes through each row. Within each row, it processes tiles — blocks of elements. Each tile row is then solved for, proceeding across elements in the tile row.

A notable advantage of this approach is that each column chunk uses data fetched from matrix B multiple times before moving onto a new set of columns, reducing cache misses. Hence, it significantly improves computational efficiency by repeating the use of the same data block, enabling better cache utilization.

The structure of the blocked column algorithm essentially translates into more efficient data access patterns, accommodating data reuse and granting a performance boost in matrix computations.

Further explanation of this approach: Coffee Brunch Page

Benchmark	(size)	Mode	Cnt	Score	Error	Units
Main.blockedColumnMultiply	1	sample	117273	10		ms/op
Main.blockedColumnMultiply	5	sample	117138	10		ms/op
Main.blockedColumnMultiply	10	sample	147550	0.001	± 0.001	ms/op
Main.blockedColumnMultiply	50	sample	60265	0.042	± 0.001	ms/op
Main.blockedColumnMultiply	100	sample	15850	0.317	± 0.001	ms/op
Main.blockedColumnMultiply	200	sample	2004	2.509	± 0.012	ms/op
Main.blockedColumnMultiply	300	sample	533	9.451	± 0.097	ms/op
Main.blockedColumnMultiply	500	sample	115	44.264	± 0.496	ms/op
Main.blockedColumnMultiply	1000	sample	15	378.431	± 5.868	ms/op
Main.blockedColumnMultiply	1500	sample	5	1278.004	± 25.280	ms/op
Main.blockedColumnMultiply	2000	sample	5	3066.875	± 130.911	ms/op

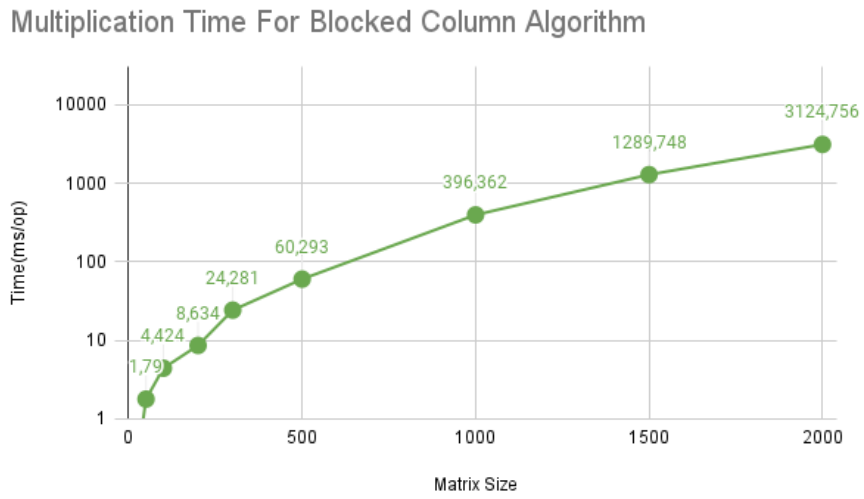

Figure 3: The average excecution times for the blocked-column algorithm

4 Sparse Martix Multiplication

In the real word, in many cases, it is necessary to multiply very large matricies, that are very sparse. For example lets look at the Williams/mc2depi matrix. We will try to multiply the matrix by itself. The matrix has the size of *525825x525825*, which means that at the n^3 operation cost of matrix multiplication we would have to do roughly 1,45e17 to calculate the result. This is not possible on a consumer computer, even with the best optimizations. This is where we can use the properties of the matrix to our advantage. Even though the matrix can hold up to 2.7e10 values it actually only holds around two million values! If we can successfully ignore the zero values, the execution time should be comparable to multiplying a 1500x1500 dense matricies.

4.1 The optimization

For each row of the array we will hold a list of *Tuple* values. Each tuple will store the number of the column where the value is stored for the index and the value. This way we will use a lot less memory and will ignore the unnecessary zeroes. All of the values are stored in a hash map, where the key is the row number (this way we don't have to store any empty rows).

We can apply a regular matrix multiplication algorithm, modified to support it, to the resulting structure.

4.2 Importing and optimizing the matrix format for multiplication

The provided matrix is in matlab format, thanks to the *JMatIO* java library the matrix can be imported, but is stored in the *CSC Sparse* format. Even though the multiplication can be performed with this format, it is much easier to implement a solution based on the custom multiplication-optimized format specified above. Included below, is a code snippet of the conversion function.

```
public static HashMap<Integer, ArrayList<Tuple>> preprocessSparseMatrix(MLSparse matrix) {
  int[] jc = matrix.getJC();
  int[] ir = matrix.getIR();
  Double[] data = matrix.exportReal();
  HashMap<Integer, ArrayList<Tuple>> processedData = new HashMap<>();

  for (int col = 0; col < jc.length - 1; col++) {
    for (int idx = jc[col]; idx < jc[col + 1]; idx++) {
      int row = ir[idx];
      double value = data[idx];

      if (processedData.get(row) == null) {
        processedData.put(row, new ArrayList<>());
      }
      ArrayList<Tuple> rowList = processedData.get(row);
      rowList.add(new Tuple(col, value));
    }
  }
  return processedData;
```

This should be a really quick process, unfortunately due to shortcomings of the library, the execution of the *matrix.getJC()* method takes around 30 seconds for the matrix (the array should be pre-calculated instead of being derived from an internal TreeSet). Due to this the preprocessing was excluded from the benchmarking time.

4.3 Multiplication Algorithm, Output Format and Execution Times

The multiplicaiton algorithm is very similar to a regular matrix multiplication. The array is saved into a *MLSparse* structure from the *JMatIO* library.

```
for (Integer i : procMat1.keySet()) {
  List<SparseMatrixUtils.Tuple> row = procMat1.get(i);
  for (SparseMatrixUtils.Tuple aElement : row) {
    int k = aElement.columnIndex;
    if (procMat2.containsKey(k)) {
      List<SparseMatrixUtils.Tuple> bRow = procMat2.get(k);
      for (SparseMatrixUtils.Tuple bElement : bRow) {
        int j = bElement.columnIndex;
        double currentValue;
        Double storedValue = (Double)result.get(i, j);
        if(storedValue == null)
          currentValue = 0;
        else
          currentValue = storedValue;

        double addValue = aElement.value * bElement.value;
        result.set(currentValue + addValue,i, j);
      }
    }
  }
}
```

Thanks to the optimizaiton, instead of the problem being impossible to solve on the hardware mentioned at the beginning of a paper the algotighm has multiplied the matricies in ***9637ms***, landing in between the time it took to multiply a dense 1500x1500 and 2000x2000 matricies with a regular, unoptimized, matrix multiplication algorithm.

5 Conclusion

Correct CPU access optimization allowed us to archive 4x speed improvement over the regular matrix multiplication algorithm. This result shows, that even though the underlying algorithm doesn't change optimizing the order of operations might have a huge impact on the execution performance.

Moreover the paper has shown how significant is the correct representation of a matrix. Taking into consideration characteristics of a matrix can save us orders of magnitude in computational power, therefore it is important to always carefully analyze a problem, before implementing a naive solution.

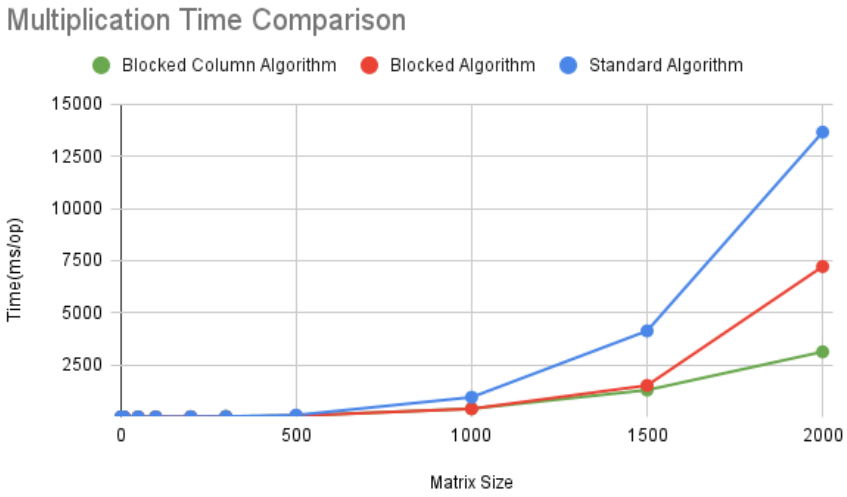

Figure 4: Execution time comparison for the algorithms

6 Future work

The speed of multiplication can be further improved by using parallelization and distributed computing. Thanks to the characteristics of matrix multiplication, the speed improvments should be more-or-less linear compared to the number of computing units used.