Universidad de Las Palmas de Gran Canaria
Grado en Ciencia e Ingenier´ia de Datos

Big Data

Optimized Matrix Multiplication

Zuzanna Furtak

Github link

Contents

1 Abstract

In this report I tested two improved ways of multiplying matrices. I also created a method for multiplying sparse matrices using a hash map structure that holds column index and values under certain row index. The results were significantly better than the basic method, especially for larger matrices, almost four times faster. Of all the techniques, multiplying matrices represented by hash map gave the best results, but only for matrices with a low density (below 0.25). This method allowed me to compute the multiplication of a matrix representing the Markov model of an epidemic, which consists of 525825 rows and columns, much more than the 2000 used to compare other methods. My next plan is to introduce parallel processing into the calculations.

2 Problem statement

In the previous report I successfully benchmarked basic matrix multiplication implemented in 3 programming languages: Python, Java and C. As a result, I found that Java was the fastest option out of these 3. Therefore, I chose Java as the language to use for further comparison. Although the first benchmark showed which language is the fastest, the results were not satisfactory because I only used the basic algorithm to multiply matrices. In this paper I present several ways to optimize this calculation to achieve better performance, more suitable for real problems involving much larger matrices. The methods compared in this paper are blocking, column blocking and a special implementation for sparse matrices. All the results are also compared with basic multiplication to give a better perspective and to clearly see the improvement.

3 Methodology

To measure the performance of each multiplication method, I use a Macbook Pro with an Apple M3 Pro chip and 18GB of RAM. The matrix sizes used to evaluate the performance of the algorithms range from 10x10 to 2000x2000 and additionally Markov model of epidemic matrix with size 525825x525825 used to evaluate the performance of the sparse matrix implementation. As a measurement tool I use Java Microbenchmark Harness (JHM), which allows me to obtain accurate results by using 5 warm-up iterations followed by 10 rounds of 10 iterations each. This allows me to obtain appropriately averaged final results.

4 Experiments

4.1 Basic Multiplication Algorithm

4.1.1 Introduction

I recalled the basic matrix multiplication algorithm to obtain better comparison for optimized methods.

4.1.2 Execution time

As is showed below in the *Table 1* the algorithm is getting significantly slower for bigger matrices. Therefore it would be hard to use this method for realistic problems as the matrices are much bigger than 2000x2000.

Size	Min time	Max time	Mean
10	10^3	0.026	10^3
20	0.003	0.120	0.003
50	0.040	0.079	0.051
100	0.373	1.974	0.405
200	3.265	10.830	3.398
500	60.293	84.148	61.828
1000	934.281	1000.342	949.800
1500	3544.187	3636.462	3572.708
2000	8908.702	9714.008	9261.023

Table 1: Benchmarking results for basic matrix multiplication

4.1.3 Chart

As can be noticed on the chart, execution time is growing rapidly with bigger matrix sizes.

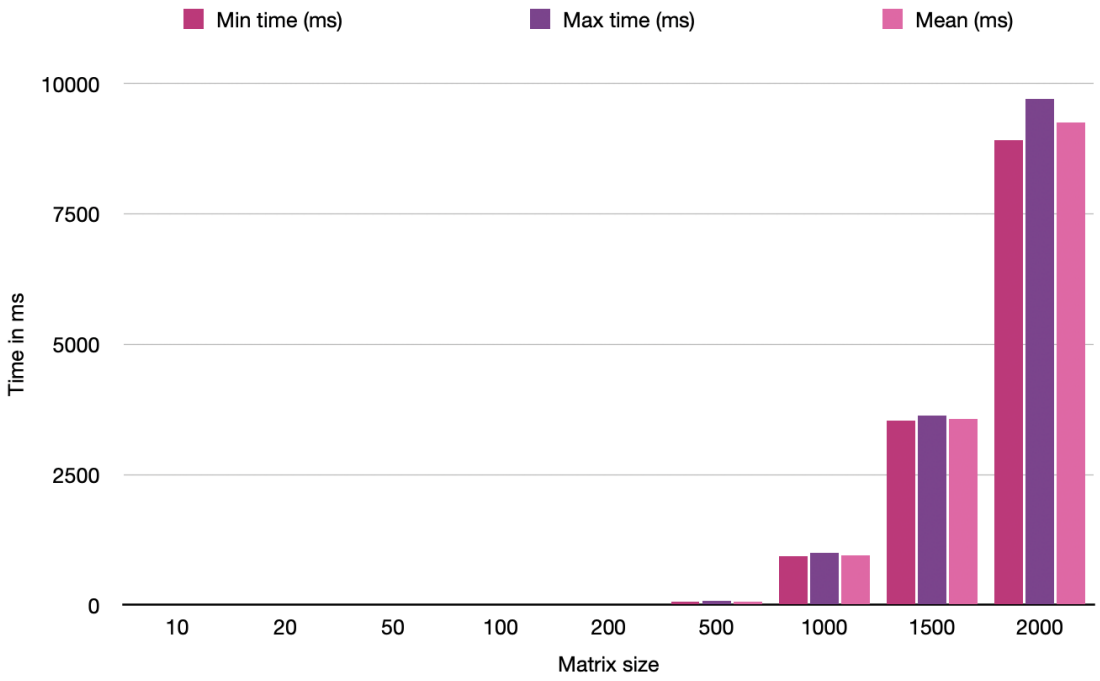

Figure 1: Visualized results for basic matrix multiplication

4.2 Blocking - Cache optimization

4.2.1 Introduction

Blocking (tiling) is a technique used to optimize matrix multiplication by dividing the matrices into into smaller sub-matrices or blocks. This approach improves cache utilization and reduces cache misses by ensuring that data used in computations is more likely to be held in the cache. The computational process of matrix multiplication involves moving data from the larger memory (RAM) to the cache. However, due to the limited size of the cache, it is important to must be managed efficiently to avoid frequent evictions and reloads, which slow down the computation.

4.2.2 Execution time

At first glance, this algorithm appears to be faster than the basic algorithm, but the difference increases as the matrix size increases. For the smallest matrices the blocking algorithm is even slightly slower, but especially for the largest matrices it gets much better. I used tiles of size 256.

For the 1500x1500 and 2000x2000 matrices, I also tried using 1024-blocks, which improved the results even more.

Size	Tile size	Min time	Max time	Mean
10	256	0.001	0.770	0.001
20	256	0.004	0.855	0.005
50	256	0.044	3.543	0.048
100	256	0.308	2.449	0.335
200	256	2.380	9.290	2.609
500	256	41.419	59.441	42.209
1000	256	354.943	378.536	359.085
1500	256	1220.542	1249.903	1230.609
1500	**1024**	1067.450	1111.491	1079.404
2000	256	3196.060	3925.869	3321.050
2000	**1024**	2545.943	2629.829	2589.144

Table 2: Benchmarking results for blocking matrix multiplication

4.2.3 Chart

The graph appears to be very similar to the previous one, but if we look at the y-axis we can see the big difference, where the previous highest value on the axis was 10,000 ms and here it is only 3,000 ms. A further comparison will be made in the next chapter.

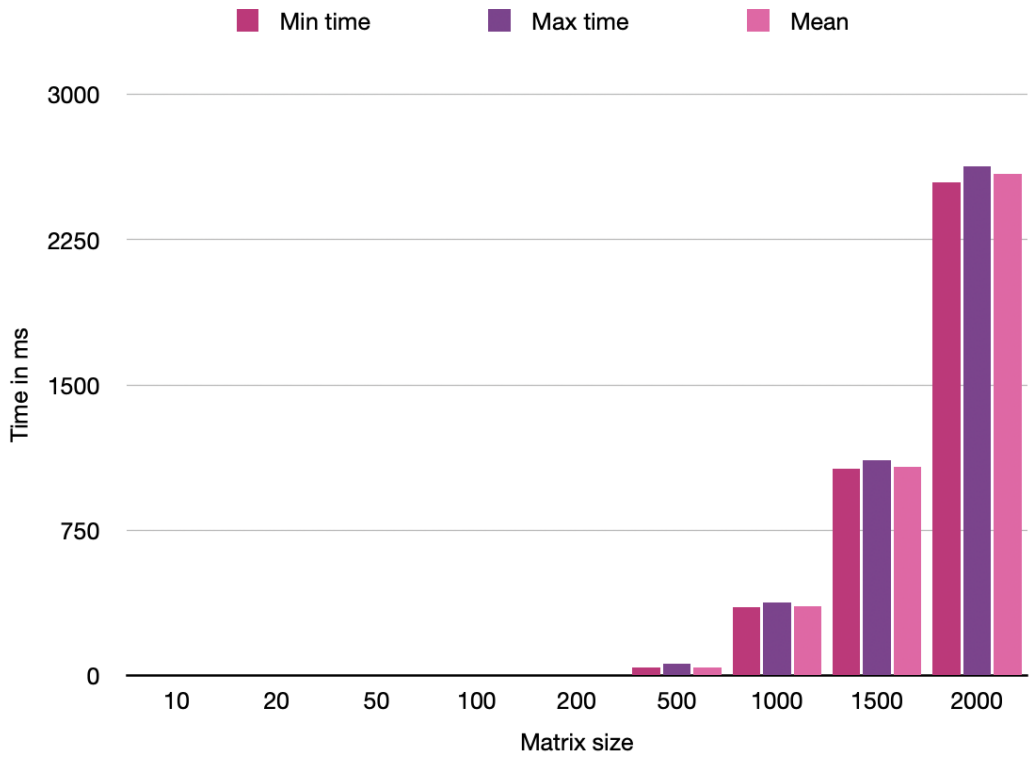

Figure 2: Visualized results for blocking matrix multiplication

4.3 Column Blocking

4.3.1 Introduction

The blocked column algorithm is a method that helps with matrix multiplication by focusing on using memory better. It does this by reusing data from the columns of second matrix instead of the rows of the first matrix as the traditional method does. This algorithm organizes calculations into blocks of columns as the name says. For each group of columns, the algorithm looks at each row, and within each row it processes chunks of elements called blocks or tiles. It then solves each row of tiles by working through the elements in that row. This method successfully reduces the number of times the data has to be loaded from memory, which improves efficiency by reusing the same block of data and making better use of the cache.

4.3.2 Execution time

As can be observed, the execution time for this method is comparable to that of the previous optimization. On closer observation, the timings show slight improvements, especially with larger matrices. In particular, for them, a block size of 1024 shows better performance than a block size of 256.

Size	Tile size	Min time	Max time	Mean
10	256	0.001	1.188	0.001
20	256	0.004	0.569	0.005
50	256	0.044	1.069	0.049
100	256	0.308	1.608	0.334
200	256	2.433	8.372	2.612
500	256	41.419	78.905	41.719
1000	256	354.419	382.206	357.477
1500	256	1207.960	1235.223	1224.946
1500	**1024**	1064.305	1111.491	1075.944
2000	256	2889.875	3007.316	2964.115
2000	**1024**	2503.999	2562.720	2526.229

Table 3: Benchmarking results for column blocking matrix multiplication

4.3.3 Chart

There is not a significant difference between this chart and the previous one.

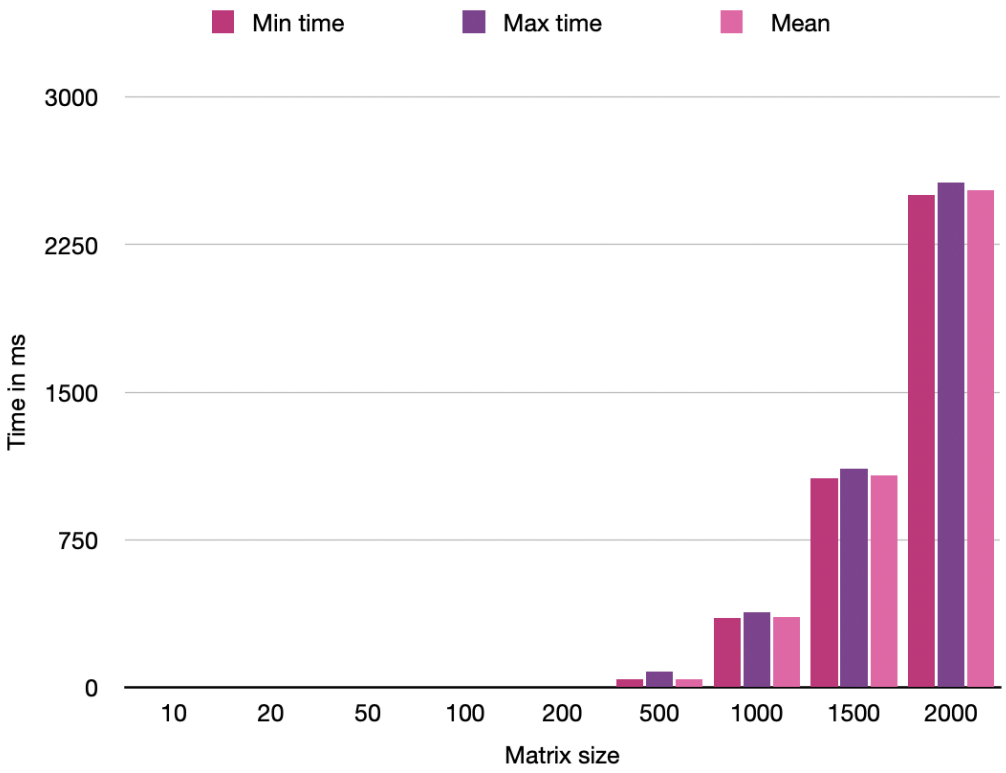

Figure 3: Visualized results for column blocking matrix multiplication

4.4 Sparse Matrix

4.4.1 Introduction

So far I have used the "basic" matrix representation to compare all multiplication methods. I created a BasicMatrix object which has size and data as a two-dimensional array. But in realistic cases matrices are much bigger and much emptier. So I created another representation for sparse matrices. Instead of storing a whole array almost full of zeros, I stored it as a hash map, where the key is a row index and a value is an array of tuples, each containing the column index where the value is placed, with that value.

4.4.2 Execution time

Using a multiplication suitable for sparse matrix representation, I was able to compute a much larger matrix - a Markov model of an epidemic matrix of size 525825x525825. This is an input that could not be computed with previous methods because it would take (based on the time complexity $O(n^3)$) more or less 1.45 * 10e17 seconds, which is 4610171531 years.

Min time	Max time	Mean
11643.388	12700.353	12039.330

Table 4: Results for sparse matrix multiplication

4.4.3 Different density comparison

In the case of the matrix representing the Markov model of an epidemic, sparse matrix multiplication worked really well, as the density ratio is 0.000007595972133. I've also tested this method for different densities, to check up to which value it's better to use it instead of other optimized methods.

And what I noticed while analyzing *Table 5*, is that density ratios above 0.125 give worse timings than simple matrix multiplication. But as the density of the matrix increases, the results get better, with amazing results for matrices like the Markov's.

Size	Density ratio	Min time	Max time	Mean
10		0.026	1.065	0.051
20		0.287	2.159	0.423
50		6.021	7.512	6.394
100		79.823	114.426	81.912
200		1032.847	1113.588	1061.369
500		20803.748	21307.064	20984.942
1000		-	-	-
2000	1	-	-	-
10		0.014	2.146	0.027
20		0.162	1.311	0.255
50		4.440	21.692	4.690
100		47.186	49.480	47.940
200		607.126	617.611	611.058
500		11861.492	12113.150	11945.378
1000		117708.947	143612.969	123660.958
1500		-	-	-
2000	0.75	-	-	-
10		0.006	0.490	0.013
20		0.077	1.380	0.112
50		2.474	4.743	2.613
100		23.331	25.723	23.644
200		282.067	290.980	285.291
500		5838.471	6014.632	5945.006
1000		54693.724	69793.219	59746.761
1500		-	-	-
2000	0.50	-	-	-
10		0.001	0.381	0.002
20		0.015	0.487	0.024
50		0.899	6.341	0.979
100		9.126	9.978	9.338
200		83.493	129.106	86.159
500		1669.333	1725.956	1689.466
1000		16424.894	16458.449	16438.182
1500		-	-	-
2000	0.25	-	-	-
10		10	0.343	10
20		10	0.419	10
50		0.001	0.405	0.001
100		0.003	0.485	0.005
200		0.033	2.134	0.052
500		3.924	10.076	4.133
1000		44.171	56.492	45.466
1500		178.782	251.396	189.308
2000	0.1	479.724	588.251	520.618
10		10	0.009	10
20		10	0.345	10
50		0.001	0.333	0.001
100		0.004	0.431	0.006
200		0.035	4.350	0.051
500		3.858	13.713	4.115
1000		42.533	84.410	44.597
1500		164.102	182.714	169.349
2000	0.01	177.471	230.425	185.768

Table 5: Results for sparse matrix multiplication with different densities

5 Conclusions

5.0.1 Charts

The graphs below clearly show that the optimisation methods speed up the computations compared to the basic algorithm. However, when we consider sparse matrices (0.1 density), the sparse matrix representation is much better and does not grow exponentially.

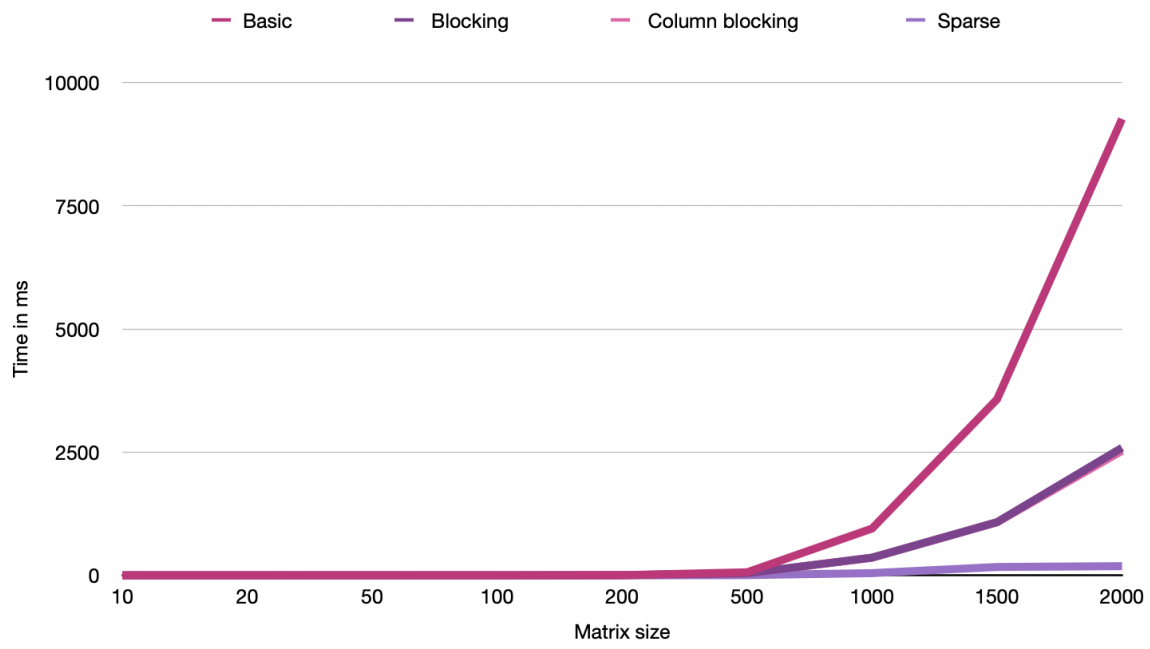

Figure 4: Average time comparison for all the methods

For a better visual comparison, in the *Figure 5* I only put optimized methods, but as on the *Figure 4* we can see that the sparse matrix multiplication is much faster than the others and the timing of both blocking optimizations are very similar with a small advantage of column blocking.

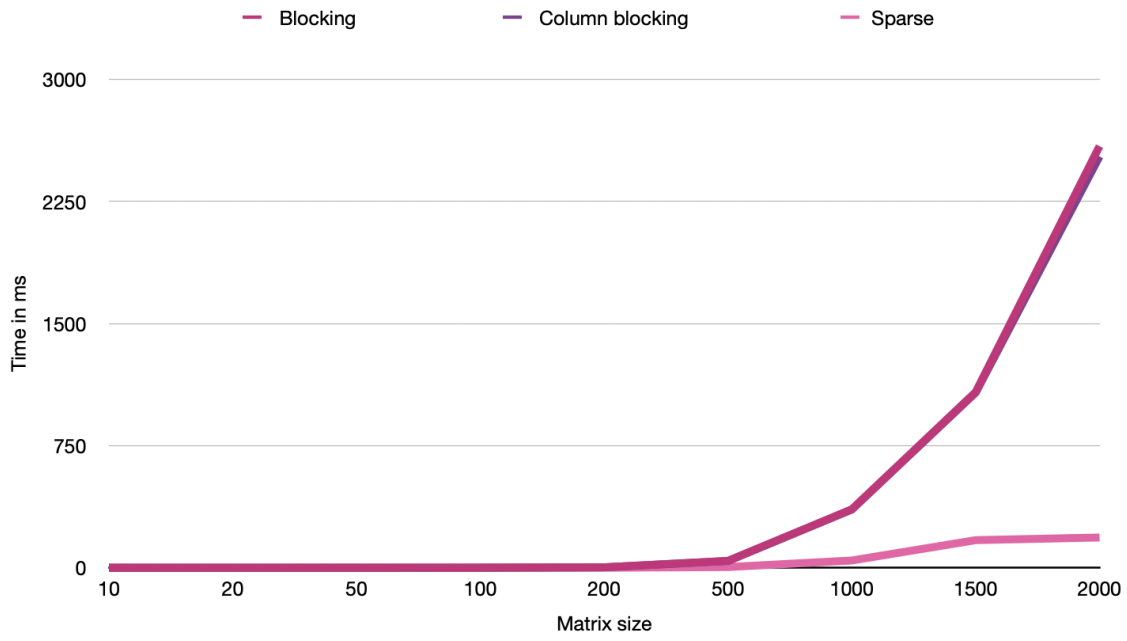

Figure 5: Average time comparison for all the optimization methods

5.0.2 Average execution time comparison

It can clearly be seen that for large matrices the time gain is huge. The code is executed almost 4 times faster. As we can see, Blocking and Column Blocking give similar results with a slight advantage for Column Blocking. I have not included sparse matrix multiplication in this comparison because the basic matrix multiplication is not performed on low-density matrices.

Size	**Basic**	**Blocking**	**Speed up**	**Column blocking**	**Speed up**
10	0,001	0,001	1	0,001	1
20	0,003	0,005	0,6	0,005	0,6
50	0,051	0,048	1,06	0,049	1,04
100	0,405	0,335	1,21	0,334	1,21
200	3,398	2,609	1,3	2,612	1,3
500	61,828	42,209	1,46	41,719	1,48
1000	949,800	359,085	2,65	357,477	2,66
1500	3572,708	1079,404	3,31	1075,944	3,32
2000	9261,023	2589,144	3,58	2526,229	3,67

Table 6: Comparison between optimized methods and the basic algorithm

6 Future work

In future work I would consider introducing parallel processing to the matrix multiplication calculation. A great idea would be to use Strassen's algorithm, which split matrix and calculates smaller parts of it. If we could parallelize these calculations, it should be a huge step towards the optimal solution.