Optimized Matrix Multiplication
Approaches and Sparse Matrices

Benchmarking

Author:

Carlos Suárez Sauca

https://github.com/CarlosSuaSau/BDBenchmarking.git
October 2024

Contents

1 Abstract

In this study we explore different matrix multiplication algorithms, and compare them with the basic one we studied in the previous project. Then we will see how these algorithms perform when oriented towards sparse matrix of different sizes.

2 Introduction

As we could observe in the previous work, matrix multiplication is a task that can rapidly increase the amount of time it takes to execute when we increase the matrix size. This is why we are going to explore alternative versions to the classic (naive) matrix multiplication algorithm. These versions try to optimize one way or another the aspects of the classic algorithm that makes it take long time of execution.

We are going to benchmark the classic algorithm and other two optimized versions, and compare their performance and memory usage. Also, we are going to try them with certain sparsity levels, to proof how do these algorithms behave.

3 Proposal

There are many different optimized algorithms for matrix multiplication. Most of them perform better than other in certain circumstances. But we are looking for algorithms that perform better than the classic, under similar circumstances. After all, I decided to try with Strassen´s algorithm, and Blocking multiplication algorithm, two solutions that improve the classic version but still make use of its core to multiply.

Strassen´s algorithm is based on a divide-an-conquer strategy. It divides in submatrices the matrix is going to multiply, recursively. When the most recent submatrix size is equal or lower to a fixed threshold, the multiplication is solved with the classic algorithm. Strassen reduces the number of multiplications needed to build the solution, which helps improve the performance if we set a good threshold. This means, a decent size submatrix to avoid more recursiveness, but that still has a good performance when solved with the classic matrix multiplication algorithm.

Blocking algorithm has some similarities with Strassen´s. Both divide the matrix into submatrices, and accomplish their task by using the classic algorithm. However, blocking has a different approach. Instead of dividing in four submatrices every time, it divides the matrix in a number of blocks, given by its block size. The ideal block size is difficult to calculate, as it depends on the cache memory we have. Blocking is not a recursive algorithm. It follows a loop structure to complete the result matrix. It reduces the use of memory

in execution, but does not reduce the number of operations compared to the classic algorithm. This means that in the worst case scenario, the performance will not be improved.

The code used to test the algorithms is in GitHub:
https://github.com/CarlosSuaSau/BDBenchmarking.git

4 Experiments

We have defined the three algorithms in Java (classic, strassen and blocking.

We have adjusted each algorithm to accept an optional parameter to set the sparsity mode, which will save the time of multiplying when it finds a zero in the matrix.

We benchmark them with the JMH library, the same used in the previous work. We test time performance and memory usage for each algorithm with the following parameters:

Matrix size (128, 256, 512, 1024)

Sparsity level (0.0, 0.2, 0.5)

The strassen algorithm is tested with a threshold of size 32, and the block size in the blocking algorithm is set to 64.

After executing the benchmark, we obtain the following data:

Time Performance:

Size (n)	128	256	512	1024
Classic	4	18	185	1923
Classic(0.2)	3	31	268	3222
Classic(0.5)	2	22	197	2276

Table 1: Classic algorithm performance (ms)

Size (n)	128	256	512	1024
Strassen	1	12	86	640
Strassen (0.2)	2	12	102	735
Strassen (0.5)	2	15	109	790

Table 2: Strassen´s algorithm performance (ms)

Size (n)	128	256	512	1024
Blocking	1	14	117	962
Blocking(0.2)	1	13	113	962
Blocking(0.5)	1	13	105	853

Table 3: Blocking algorithm performance (ms)

Memory use:

Size (n)	128	256	512	1024
Classic	132	142	15	96
Classic(0.2)	78	70	88	149
Classic(0.5)	82	79	104	173

Table 4: Classic algorithm memory usage (mb)

Size (n)	128	256	512	1024
Strassen	82	299	396	628
Strassen(0.2)	137	371	416	305
Strassen(0.5)	233	203	321	1205

Table 5: Strassen algorithm memory usage (mb)

Size (n)	128	256	512	1024
Blocking	36	66	74	47
Blocking(0.2)	124	121	108	171
Blocking(0.5)	48	75	32	73

Table 6: Blocking algorithm memory usage (mb)

5 Conclusions

If we compare the time performance of the three algorithms, there is something clear; Both Strassen and Blocking algorithms are faster than the classic algorithm in all cases. We can see that specially Strassen gets the task done in the fastest time. But it is also important to notice how this algorithm is behaving with sparse matrices. Strassen, aswell as the classic algorithm, are performing worst when they have to check for the zeros in the matrix. This means they are not properly adjusted to sparse matrices. In the other hand we have the Blocking algorithm, which is better than the classic, but not faster than Strassen. However, is the only of the three making a good use of sparse matrices, as it improves its performance the larger proportion of the matrix is composed of zeros. As expected, Strassen is the fastest, as it is the only that consistently

has to calculate less operations.

Then we can observe the memory use of these algorithms. Although these numbers are not as consistent, we can see a clear trend. The blocking algorithm is the one that makes the lower use of memory while executing, and Strassen is the one that makes the higher use of it. The classic algorithm is right in the middle. This makes sense with the expected outcome, as Blocking is more focused on optimizing the use of cache, while Strassen relies heavily on making submatrices and storing them.

To sum up, Strassen´s algorithm has the best time performance, while Blocking is better in making a lower use of memory.

6 Future Work

In this work we have implemented and tested some optimized algorithms for matrix multiplication, although we have stayed in single-threaded programming. In the future, it would be interesting to test with parallelization, and proof if these algorithms work for that purpose or we have to use different ones.