Parallel and Vectorized Matrix Multiplication

Benchmarking

Author:

Carlos Suárez Sauca

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

Contents

1 Abstract

In this case, we investigate parallelization and vectorization applied to matrix multiplication. We compare these algorithms with the performance of the naive matrix multiplication algorithm.

2 Introduction

In the previous work we could observe improvements to the classic (naive) approach to matrix multiplication. There we noticed a performance improvement. In this case we are going to compare two new approaches; parallelization and vectorization.

We are going to benchmark these algorithms and compare their speed performance, and also the use of memory they make.

3 Proposal

The two algorithms we are going to compare on this occasion use parallelization and vectorization.

Parallelization is a way to improve the performance of a program by executing multiple operations at the same time. The previous algorithms we have studied used single-thread performance. This is a limitation, since there are tasks that could be executed simultaneously without affecting each other. With multi-threading, we can cut significantly the time of execution by dividing the task into smaller pieces that are executed in parallel. The total number of operations remains the same, but optimizing the time we have to execute them.

Vectorization improves matrix multiplication in a different way. It transforms the matrix and executes operations based on vectors. While in previous algorithms we used to multiply element by element, and then add them up, vectorization is more similar to the actual way we would multiply to matrices by hand. It applies an operation that multiplies a vector to another vector. This optimizes the loops the algorithm uses, making it more efficient.

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, parallel and vectorized). Each one has its own benchmark class with their respective parameters.

We test the time performance and memory use of each algorithm.

Matrix size (128, 256, 512, 1024)

The parallel algorithm is tested with different values for the number of threads.
After executing the benchmark, we obtain the following data:

Time Performance:

Size (n)	128	256	512	1024
Classic	4	18	185	1923
Parallel(1)	2	19	227	2350
Parallel(2)	1	12	138	1421
Parallel(4)	1	10	113	1195
Parallel(8)	1	9	88	1167
Parallel(16)	1	9	80	2311
Vectorized	1	9	62	528

Table 1: Performance (ms)

Memory use:

Size (n)	128	256	512	1024
Classic	132	142	15	96
Parallel(1)	134	79	69	69
Parallel(2)	72	82	31	31
Parallel(4)	210	75	26	62
Parallel(8)	302	69	30	119
Parallel(16)	277	144	161	167
Vectorized	127	32	77	163

Table 2: Memory use (mb)

5 Conclusions

Comparing the performance of the three algorithms, we can easily notice something: vectorized multiplication is the fastest algorithm. We can see that the difference is wider as the size of the matrix grows. We can also see that the naive algorithm and parallel with 1 thread perform the worst. These are essentially the same algorithm, since the number of operations and the number of cores are the same. The parallel algorithm with 1 thread performs slightly worse because of the logic needed to create the pool of threads, and does not improve, as it is not really using parallelization.
We can also observe that the parallel algorithm improves as the number of

threads increases, but then 16 threads performs worse than with just 8. This is due to the limitations of the system in which it is executed. I count with 8 logical cores. In this kind of problem, matrix multiplication, the algorithm performs the best when the number of threads is equal to the number of logical cores. Having a higher number of threads delays the execution since every time a thread is resumed, there is a context switch, and it takes time.

As for the memory use, we can see that numbers are not directly related to the size of the matrix. There are not many conclusions to obtain from here, although we can point out that in the parallel algorithm, a higher number of threads usually means a higher use of memory.

To sum up, we can say that in terms of performance, vectorized algorithm and parallelization with 8 threads (the number of available logical cores) are the most optimal algorithms.

6 Future Work

In a future work, it would be interesting to pick the best algorithms of this work and the previous one, and try to combine the techniques to obtain a very optimized algorithm that performs better than those seen before.