Matrix Multiplication Optimization

Diego Navarro Cabrera

2020-02-06

1 Introduction

Multiplying matrices can be a lengthy task, even for a computer. The usual algoithm is $O(n^3)$ because of its 3 nested loops, and even other more complex algorithms like the strassen algorithm don't get to achieve $O(n^2)$. That is why its essential to optimice these algorithms to its fullest, and to do that we have 2 alternatives, we may try to ease the execution taking advantage of how the computer works, or we may use multithreading to process several rows at a time.

1.1 Single thread optimization

The biggest change we can exploit is the use of the cache, usually we access the first matrix by rows and the second one by columns, but given the way that memory is alocated in a matrix, this means that each time we try to access an element of the second matrix, we won't find it in the cache and will have to search it in ram. If we copy the second matrix onto an auxiliar matrix exchanging rows and columns, we can make full use of the cache and speed up the execution time of the function as long as the matrix size is big enough to compensate for the extra time needed for copying the matrix.

Another change we can try is unwinding the loops, each iteration in a loop is followed by a conditional instruction, which has bad repercussions on the execution speed, so if we manage to reduce the number of iterations by putting together several instructions we can usually get a tiny boost in performance. The downside of this is that unwinding makes the code bigger and harder to read.

1.2 Multithreading

Using multithreading libraries like openMP we can create a concurrent version of the multiply matrix function, this way we may overlap the computation of several rows, and while this may not be worth it for small matrices, due to the extra time added by creating and managing threads, we are able to achieve a rather large speedup with respect to the secuential version.

2 Results

The results here mentioned will be obtained using the code present in the directory "/code", compiled using "gcc -fopenmp" using the default compiler optimization, and the binarie will be executed in a computer dual core with 2 threads by core. The speedup of each version is calculated comparing against the basic function.

Size	Basic	Optimized	Parallel	Final
10000	0,0039	0,0036	0,0020	0,0017
40000	0,0348	0,0295	0,0164	0,0139
90000	0,1279	0,1002	0,0580	0,0471
160000	0,3191	0,2356	0,1468	0,1133
250000	0,6520	0,4599	0,2996	0,2167
360000	1,1428	0,7938	0,5317	0,3757
490000	2,0157	1,2645	0,9755	0,5965
640000	3,0107	1,9122	1,6636	0,9018
810000	5,0824	2,7570	2,6853	1,2816
1000000	6,7124	3,8031	3,5879	1,7995
2500	0,0007	0,0007	0,0039	0,0002
2250000	31,839	12,944	13,82	15,901

Figure 1: Execution **time** based on matrix size.

Size	Basic	Optimized	Parallel	Final
10000	1	1,083	2,003	2,249
40000	1	1,180	2,122	2,498
90000	1	1,276	2,205	2,714
160000	1	1,355	2,173	2,815
250000	1	1,418	2,176	3,009
360000	1	1,440	2,149	3,042
490000	1	1,594	2,066	3,379
640000	1	1,574	1,810	3,338
810000	1	1,843	1,893	3,966
1000000	1	1,765	1,871	3,730
2500	1	0,990	0,179	2,684
2250000	1	2,460	2,304	5,396

Figure 2: **Speedup** based on matrix size.

2.1 Optimiced version

The first thing we can notice is that for small sizes, the "optimiced" version is equal or even a bit worse than the basic function, but as size goes up, so does the speedup, achieving astonishing results for the biggest sizes.

2.2 Simple concurrent version

This version seems to be a bit more consistent, with a speedup always around 2 (the number of cores in the machine), which suggest that we may be able to achieve even better results using more cores. It is necessary to point out that

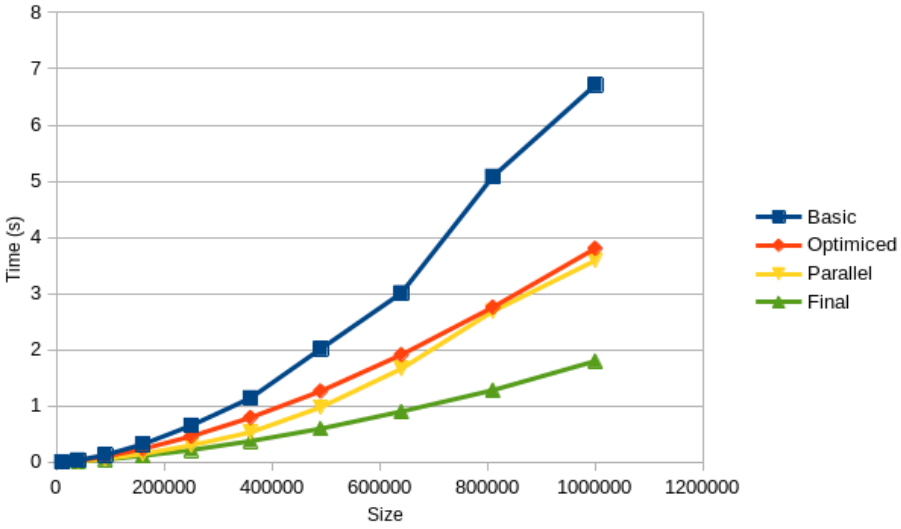

Figure 3: Time comparison.

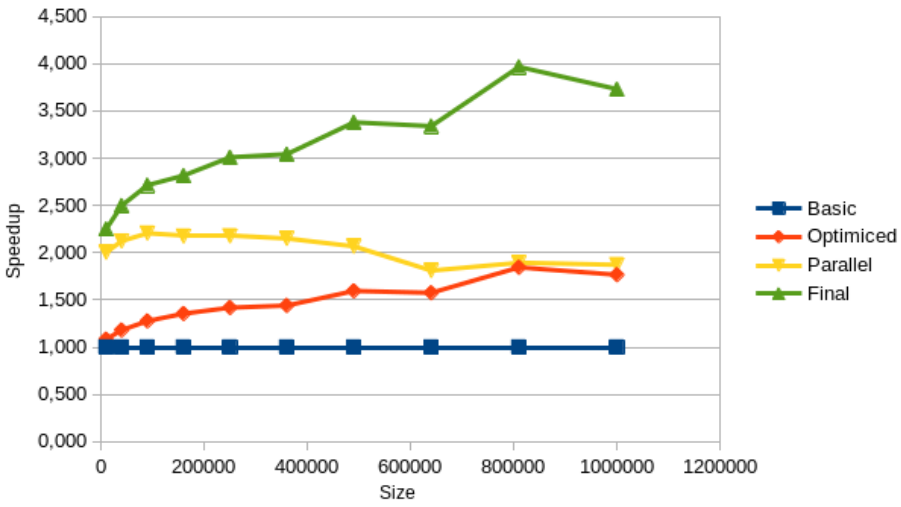

Figure 4: Speedup comparison.

for really small matrix sizes, this version can be much worse than the first two, but that won't usually matter due to how small this times really are.

2.3 Combined version

If we combine the changes made in the past 2 versions we can obtain a much better result for both big and small matrices. As we can see, when the matrix is small, the speedup of this function is around 2, and as the size goes up, the speedup increases. This results can be explained, as combining the improvements made in past versions makes the final speedup to be roughly the sum of their speedups, and it is even better because the use of multithreading accelerates the process of copying the second matrix into an auxiliar one, resulting in a function that can be several times faster than its original version.

3 Conclusion

Has we have seen, given a function which uses a basic algorithm we can obtain much better results, even if we don't change the underlying algorithm, by taking advantage of some low level improvements and multithreading, and while this doesn't change the need for better algorithms to solve certain tasks, it is a great step when no other options are a available.