CDP Project 1

Computação Paralela e Distribuída 23/24

João Filipe de Menezes Falcão e Sousa Guedes (up202108711@up.pt)
Lino Branco Vaz (up202108696@up.pt)
Tomás Paiva Ramos (up202108687@up.pt)

Índice

Problem Description

In this project we are tasked to calculate the product of two matrices using different algorithm implementations, in order to study the effect of memory hierarchy on the processor's performance. To achieve this, we developed several algorithms using different programming languages (C/C++ and JAVA), and used the Performance API (PAPI) to register some relevant metrics regarding these algorithms' performance.

Algorithms Explanation

STANDARD MULTIPLICATION

The initial algorithm employs a straightforward standard matrix multiplication approach. In this method, each element of a row in matrix A is multiplied with every element of a column in matrix B, accomplished through three nested loops, resulting in a time complexity of O(n^3) and a spatial complexity of O(n^2). This approach represents a brute-force implementation of matrix multiplication, where memory accesses to positions of matrices A, B, and C occur for every iteration. Unfortunately, these accesses are scattered across memory, leading to a high rate of cache misses. This is because the algorithm fails to exploit spatial locality effectively; each iteration accesses different memory locations, which diminishes cache efficiency. Consequently, the algorithm suffers from poor cache utilization and frequent cache thrashing due to the non-sequential memory access pattern. As a result, its performance is suboptimal compared to more cache-friendly or algorithmically optimized methods. It fails to maximize cache usage efficiency, leading to a higher number of cache misses and slower execution times.

LINE BY LINE

This algorithm represents a significant advancement over traditional matrix multiplication methods, particularly in terms of cache utilization and algorithmic optimization. The algorithm operates by iteratively processing each row of matrix A. For each row, we store its elements in a temporary variable, temp. Then, we proceed to compute the elements of matrix C by multiplying each element of temp with the corresponding column of matrix B. This approach offers a significant improvement in cache efficiency, as it minimizes redundant memory accesses and capitalizes on spatial locality. In terms of computational complexity, the algorithm exhibits a time complexity of O(n^3) and a spatial complexity of O(n^2), where 'n' represents the dimension of the matrices. It's important to note that these complexities are derived from the nested loop structure used in the algorithm, with 'n' being the size of the matrices. The intelligent use of cache in this algorithm is a key advantage. By accessing each element of matrix A only once per column, the algorithm optimizes spatial locality. This means that the likelihood of frequently accessed values being present in the cache is significantly increased, leading to fewer cache misses and improved overall cache utilization. While this algorithm represents a substantial improvement over traditional methods, there is still room for further optimization.

BLOCK BY BLOCK

The block-by-block matrix multiplication algorithm enhances line-by-line multiplication by subdividing the matrices into smaller blocks, thereby transforming a larger problem into more manageable sub-problems. This approach capitalizes on spatial locality and improved cache performance to achieve superior execution times and mitigate cache misses. The algorithm's outer loop iterates over each block of rows in matrix A and columns in matrix B, while the inner loops perform the actual matrix multiplication within each block. It has a time complexity of O(n^3). This strategy significantly enhances cache utilization by operating on smaller blocks that fit more efficiently into the lower level and faster memory (L1 and L2 caches), reducing memory access latency. Furthermore, the elements within each block are reused multiple times, thus enhancing cache residency. The reduction in cache misses contributes to improved data locality (data in the same block are close together in memory) and, consequently, enhanced performance.

STANDARD OMP

For the first solution, we implemented a simple OpenMP *#pragma omp parallel* which creates a team of threads that are executed concurrently.

For the second solution, implemented *#pragma omp parallel for collapse(2) private(i, j, k, temp)*, which tells OpenMP to parallelize the following loops; the collapse(2) combines the i and j loop; the private clause declares that the variables i, j, k, temp should be private to each thread. Then we implemented on the k for loop: *#pragma omp simd reduction(+:temp)*. This enables the use of SIMD (single instruction multiple data) which leads to speedup on hardware that supports it. The reduction clause indicates that *temp* is used in a reduction operation, and it’s used in a way that avoids race conditions.

LINE BY LINE OMP

For the first solution, we implement *#pragma omp parallel for,* it tells to parallelize the following for loop that iterates over i, the iterations of the loop are divided among the available thread.

For the second solution, we implemented: *#pragma omp parallel for private(j, k)*, it parallelizes the loop that iterates over *i*, and declares that the variables j and k should be private to each thread. On the innermost loop that iterates over j, we implement: *#pragma omp parallel for private(j),* it tells to parallelize the innermost loop, then declares that the variable *j* should be private for each thread. The use of several *#pragma omp parallel for* inside another one can lead to the creation of many more threads than cores available, which impacts the performance.

Performance Metrics

To evaluate the performance of these algorithms, we compared several metrics collected during the execution of these algorithms.

Firstly, we registered processing time for the first two algorithms, and compared it to their respective implementations in JAVA. Afterwards, we measured the average number of L1 and L2 cache misses with the use of PAPI (Performance API). Both the C++ and JAVA implementations of these first two algorithms were measured for matrices of different sizes, ranging from 600 to 3000, in intervals of 400 elements.

For the block oriented algorithms, the same values were taken into account, but for a different range of matrix sizes, from 4096 to 10240 elements, with intervals of 2028 each. To experiment with the difference in block sizes, this algorithm was tested three times for block sizes of 128, 256 and 512. To compare the results obtained by exploring different block sizes, the previous line algorithm was also tested with these matrix size ranges.

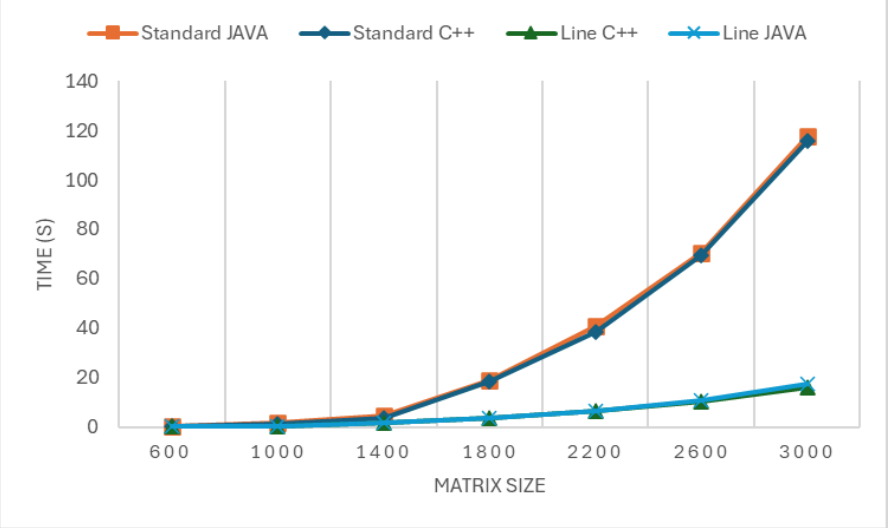
Finally, for the parallel versions of the first two algorithms, the processing time, Flops and speedup values were measured for both solutions of each algorithm.

To ensure precision on the collected measures, all these tests were performed on identical computers available at FEUP, and on new processes to guarantee the independence of the acquired results. Additionally, all registered values result from a calculated average from 3 different executions.

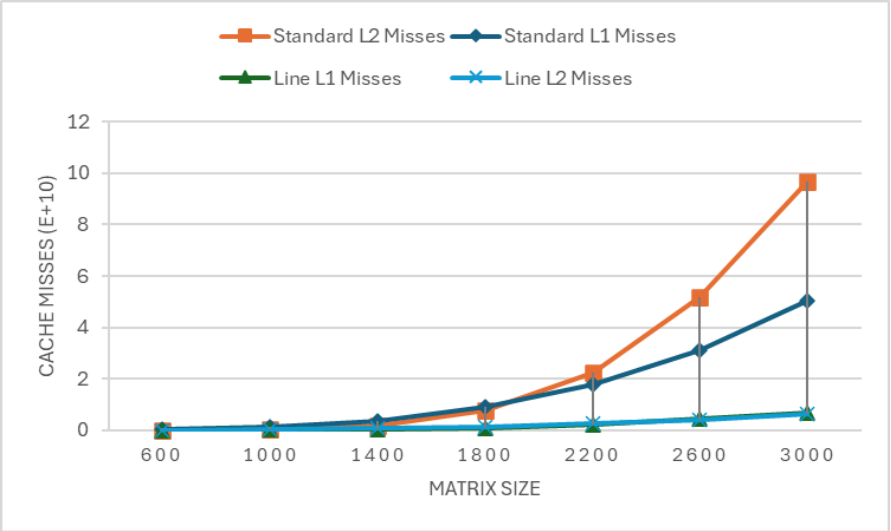
Results and Analysis

The following graphs contain metrics from different algorithms whose comparison is relevant to analyze. Note that the matrix size ranges are different for some graphs, and the units used for both cache misses and GFlops are labeled on the vertical axis of the graphs. (For more detailed information, the complete tables with all obtained values are available in the Annexes section.)

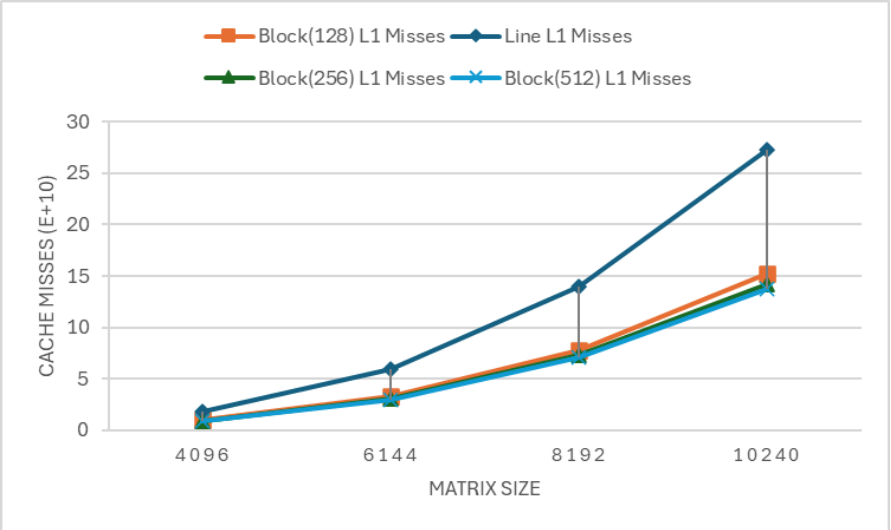
In this graph, we see that for these algorithms, either in Java or C++, the execution times are relatively similar for each matrix size.



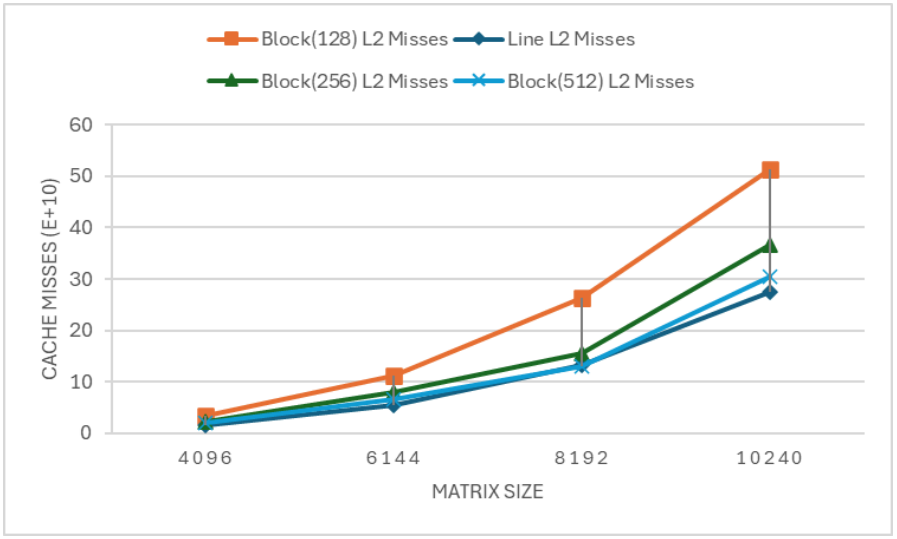
For the cache misses, in relation to the standard algorithm, there's a large difference when we get to a larger matrix, the misses in cache L2 become greater than L1. But that's not the case for the line multiplication where the cache misses stay quite similar, meaning it's a more efficient algorithm that allows for more data availability in low level memory.



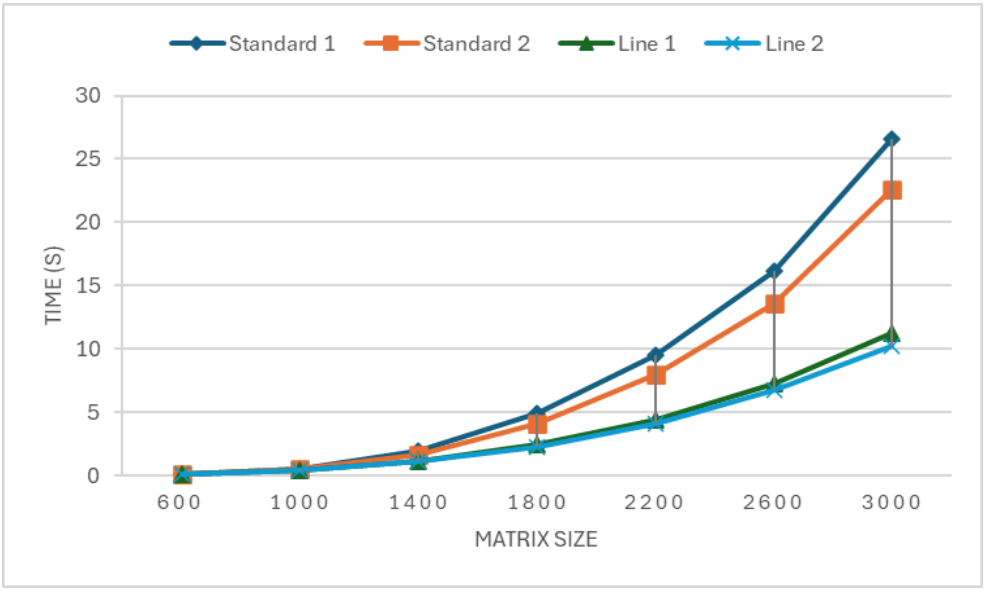
For L1 cache misses, we observe that the cache misses for the block by block multiplication remain similar for matrix size 4096, 6144, 8192, 10240 and block sizes 128, 256 and 512. In comparison to the line L1 misses that has way more misses for those same matrix sizes – which goes to show an improvement in data locality and temporal location.



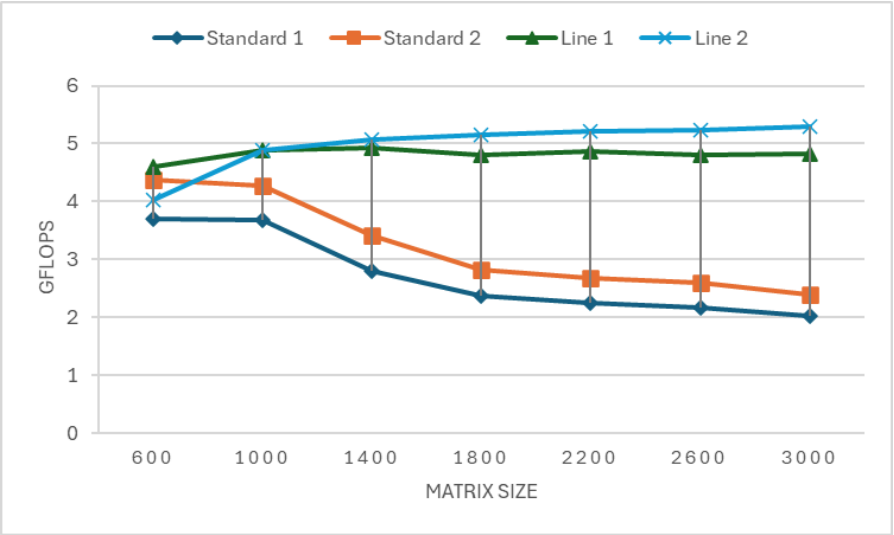
But when we go over to the cache L2 misses, we find that there are more misses for the block multiplication than the line.



As for the parallel versions of the first two algorithms, we immediately notice a huge reduction in processing time values for all solutions, when compared to their corresponding single-core implementations.



The number of FLOPS (Floating Point Operations per Second) manages to stay rather consistent for both solutions of the line algorithms, but the same is not true for the standard algorithm, which shows to be clearly more affected by the growth of the matrix dimension.



Conclusions

In conclusion, our experiments deepened our understanding of cache memory's role in code efficiency. We optimized data and temporal locality through various strategies, notably loop restructuring and data reorganization. However, while parallel implementations showed promise, their impact on cache performance was less significant than anticipated. This underscores the importance of targeted optimizations tailored to cache dynamics for maximizing computational efficiency.

Annexes

Part 1: Performance evaluation of a single core

1. Standard

C++

Standard Multiplication	Time(s)	L1 Cache Misses (un E+10)	L2 Cache Misses (un E+10)
600	0.199	0.022	0.004
1000	1.100	0.123	0.026
1400	3.471	0.350	0.151
1800	18.350	0.910	0.774
2200	38.337	1.763	2.238
2600	69.288	3.091	5.175
3000	115.626	5.029	9.681

JAVA

Standard Multiplication	Time(s)
600	0.231
1000	1.547
1400	4.626
1800	19.054
2200	40.913
2600	70.613
3000	117.55

2. Line

C++

Line Multiplication	Time(s)	L1 Cache Misses (un E+10)	L2 Cache Misses (un E+10)
600	0.112	0.003	0.005
1000	0.530	0.013	0.026
1400	1.610	0.035	0.070
1800	3.443	0.075	0.145
2200	6.338	0.207	0.256
2600	10.515	0.441	0.410
3000	16.097	0.678	0.637

JAVA

Line Multiplication	Time(s)
600	0.124
1000	0.537
1400	1.657
1800	3.638
2200	6.563
2600	10.787
3000	17.398

C++ (4096x4096 to 10240x10240)

Line Multiplication	Time(s)	L1 Cache Misses (un E+10)	L2 Cache Misses (un E+10)
4096	41.205	1.754	1.619
6144	139.171	5.907	5.359

8192	330.958	14.01	13.234
10240	654.806	27.316	27.415

3. Block

C++

Block Size 128

Block Multiplication	Time(s)	L1 Cache Misses (un E+10)	L2 Cache Misses (un E+10)
4096	30.265	0.971	3.337
6144	104.130	3.278	11.099
8192	244.283	7.771	26.248
10240	480.345	15.179	51.363

Block Size 256

Block Multiplication	Time(s)	L1 Cache Misses (un E+10)	L2 Cache Misses (un E+10)
4096	26.704	0.907	2.350
6144	90.599	3.063	7.923
8192	391.607	7.303	15.636
10240	417.612	14.180	36.700

Block Size 512

Block Multiplication	Time(s)	L1 Cache Misses (un E+10)	L2 Cache Misses (un E+10)
4096	26.520	0.876	1.968
6144	93.133	2.959	6.567
8192	326.525	7.036	12.975
10240	418.023	13.689	30.460

4. OMP

First solution for standard multiplication

Standard Multiplication	Time(s)	GFLOPs	Speedup
600	0.117	3.706	1.701
1000	0.542	3.690	2.03
1400	1.961	2.798	1.77
1800	4.938	2.362	3.716
2200	9.481	2.246	4.044
2600	16.147	2.177	4.291
3000	26.559	2.033	4.354

Second solution for standard multiplication

Standard Multiplication	Time(s)	GFLOPs	Speedup
600	0.099	4.372	2.01
1000	0.469	4.269	2.345
1400	1.607	3.416	2.16
1800	4.123	2.829	4.451
2200	7.965	2.674	4.813
2600	13.582	2.588	5.101
3000	22.60	2.389	5.116

First solution for MultLine

Line Multiplication	Time(s)	GFLOPs	Speedup
600	0.094	4.597	1.191
1000	0.409	4.887	1.296
1400	1.113	4.930	1.447

1800	2.409	4.801	1.429
2200	4.382	4.860	1.446
2600	7.302	4.814	1.44
3000	11.204	4.820	1.437

Solution 2 with nested parallelism

Line Multiplication	Time(s)	GFLOPs	Speedup
600	0.100	4.031	1.12
1000	0.409	4.885	1.296
1400	1.084	5.061	1.485
1800	2.259	5.162	1.524
2200	4.092	5.204	1.549
2600	6.718	5.233	1.565
3000	10.215	5.286	1.576