U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

DEGREE IN INFORMATICS AND COMPUTER ENGINEERING

PARALLEL AND DISTRIBUTED COMPUTING

ADALBERTO TEIXEIRA GUEDES

INÊS SOUSA DA SILVA

RAFAEL AZEVEDO ALVES

PROJECT 1:

PERFORMANCE EVALUATION OF A SINGLE CORE

PORTO
2023

INTRODUCTION

The purpose of this project is to study the effect on the processor performance of the memory hierarchy when accessing large amounts of data. In this study will be using the product of two matrices and will be used Performance API (PAPI) to collect relevant performance indicators of the program execution.

PROBLEM DESCRIPTION

The matrix multiplication problem is the case study of this project and through it, our main goal is to evaluate the impact it has on CPU performance, whenever we want to access a large amount of data kept in memory.

Firstly, it's essential to understand how data is organized in memory and how we can take advantage of that structure to maximize the performance (depending on which region of memory we access, the access time will vary).

In order to test all possible memory accesses and the impact each one has on Performance we implemented three distinct algorithms which make use of CPU resources.

ALGORITHMS EXPLANATION

Simple Matrix Multiplication

The simplest and most intuitive algorithm of matrix multiplication consists of iterating row i of matrix A and column j of matrix B and multiply each element of A with each element of B, add the intermediate results and store it in position (i, j) from a third matrix C, initially null.

Both C++ and Java were the programming languages adopted for this algorithm implementation.

The pseudocode that illustrates the algorithm implemented is:

Algorithm 1 Column Multiplication

```
function ColumnMultiplication(A, B)
    m_ar ← matrix A size
    m_br ← matrix B size
    for i = 0 to m_ar do
        for j = 0 to m_ar do
            temp ← 0
            for k = 0 to m_ar do
                temp += A[i * m_ar + k] * B[k * m_br + j]
            end for
            C[i * m_ar + j] ← temp
        end for
    end for
     return C
end function
```

Complexity Analysis

• Time Complexity: O (N^3), where N is the size of the matrixes

• Space Complexity: O (N^2), where N is the size of the matrixes

Line Matrix Multiplication

The second algorithm differs from the previous one by making a more efficient use of memory, accessing data that is stored in faster memories. Instead of multiplying one line of the first matrix by each column of the second matrix, this version multiplies one single element from the first matrix by the correspondent line of the second matrix. To program this algorithm, it is possible to depart from the previous one and change the order of the nested for loops.

Both C++ and Java were the programming languages adopted for this algorithm implementation.

The pseudocode that illustrates the algorithm implemented is:

Algorithm 2 Line Multiplication

```
function LineMultiplication(A, B)
    m_ar ← matrix A size
    m_br ← matrix B size
    for i = 0 to m_ar do
        for k = 0 to m_ar do
            temp ← 0
            for j = 0 to m_ar do
                temp += A[i * m_ar + k] * B[k * m_br + j]
            end for
            C[i * m_ar + j] ← temp
        end for
    end for
     return C
end function
```

Complexity Analysis

• Time Complexity: O (N^3), where N is the size of the matrixes

• Space Complexity: O (N^2), where N is the size of the matrixes

Despite having the same time complexity, this algorithm is significantly more efficient than the previous one due to the change in the loop's structure.

In the previous algorithm, a cache miss is issued by the CPU for every iteration of the innermost loop. This occurs because the matrices are represented as an array of lines rather than columns. Thus, when the processor needs an element that is not present in the line currently loaded, it must retrieve it from main memory.

For each iteration of the innermost loop, the element from matrix B used is always in a different line. Consequently, the CPU loads a new line from main memory N^3 times for each iteration of the innermost loop.

Block Matrix Multiplication

The block matrix multiplication algorithm consists in partitioning the initial matrix into blocks or sub-matrices of a given size. Finally with the partitioned matrix, we can perform the multiplication of the two initial matrix by multiplying the sub-matrices of smaller size. After dividing the matrices in blocks there are two possible approaches. The first approach is based on the matrix multiplication algorithm, and the second based on the line matrix multiplication algorithm.

Contrarily to the previous algorithms, C++ was the only programming language adopted for this algorithm implementation.

The pseudocode that illustrates the algorithm implemented is:

Algorithm 3 Block Multiplication

```
function BLOCKMULTIPLICATION(A,B,bkSize)
    m_ar ← matrix A size
    m_br ← matrix B size
    bkSize ← block size
    blocks_per_row = m_ar/bkSize
    for block_y = 0 to blocks_per_row do
        for block_x = 0 to blocks_per_row do
          block_y_index = block_y*bkSize*m_ar
          block_x_index = block_x*bkSize
          block_ index = block_y_index*block_x_index
              for block_k = 0 to blocks_per_row do
                block_A_index = block_y_index+block_k*bkSize
                block_B_index = block_k*bkSize+ block_x_index
                    for i = 0 to bkSize do
                        for k = 0 to bkSize do
                            for j = 0 to bkSize do
                              C[block_index + (i*m_ar+j)] += A[block_A_index+(i*m_ar+k)]*B[block_B_index+(k*m_ar+j)]
                            end for
                        end for
                    end for
                end for
            end for
        end for
         return C
end function
```

PERFORMANCE METRICS

In order to correctly evaluate performance and the correlation between memory access and execution time, the *Performance Application Programming Interface* (or PAPI, for short) was used to collect the number of data cache misses on the L1 and L2 cache level.

Among innumerous metrics PAPI API offers, we collected:

- PAPI_L1_DCM -> Cache misses at level 1
- PAPI_L2_DCM-> Cache misses at level 2

In addition to the metrics collected we also calculated Execution Time in our test cases(T(s)) and some derived metrics such as Cycles per Instruction (CPI), the number of Floating-Point Operations per Second and finally Sum of L1 and L2 Cache Misses.

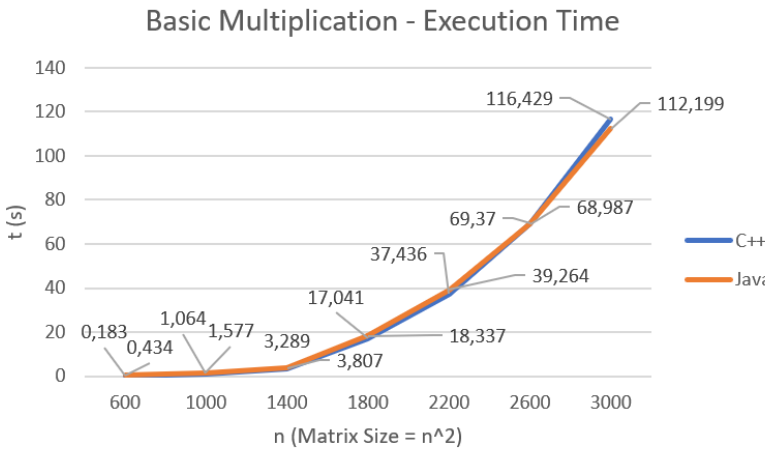
To precisely evaluate the performance of each algorithm we used the collected execution time for each task, allowing us along with the number of Floating-Point Operations ($2N^3 * B^3$ – Block Multiplication, $2N^3$ - Other algorithms) to calculate GFlops/s:

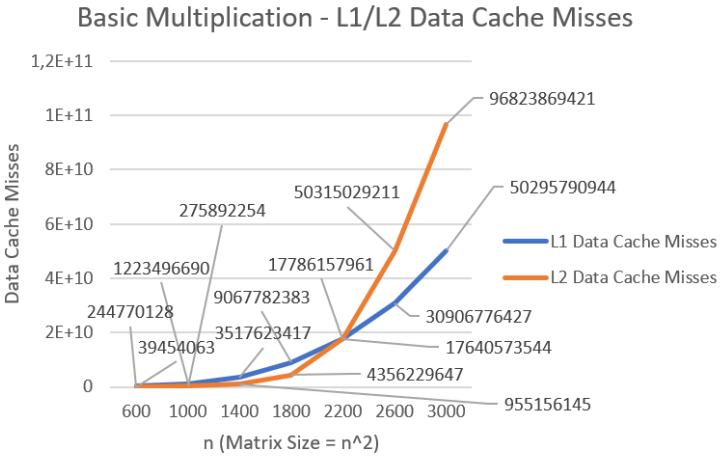
$$\textbf{GFlops/s} = \frac{\text{Number of } Floating-Point\ Ops.}{T(s)\times 10^9}$$

RESULTS

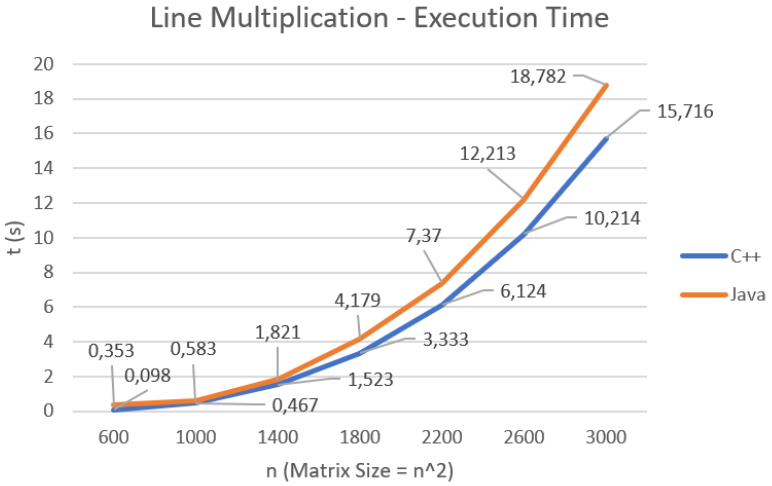
In this section we have the graphs obtained by collecting data from the implemented algorithms. The analysis of the results is in section “Analysis” and the tables with values obtained are in the section “Annexes”.

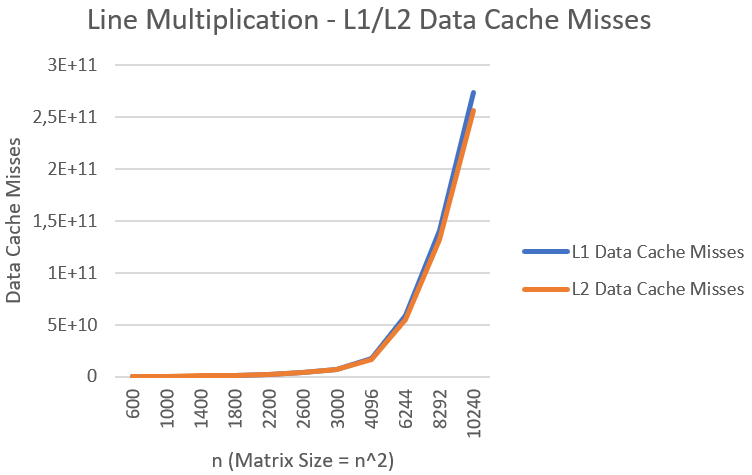
1. BASIC MULTIPLICATION



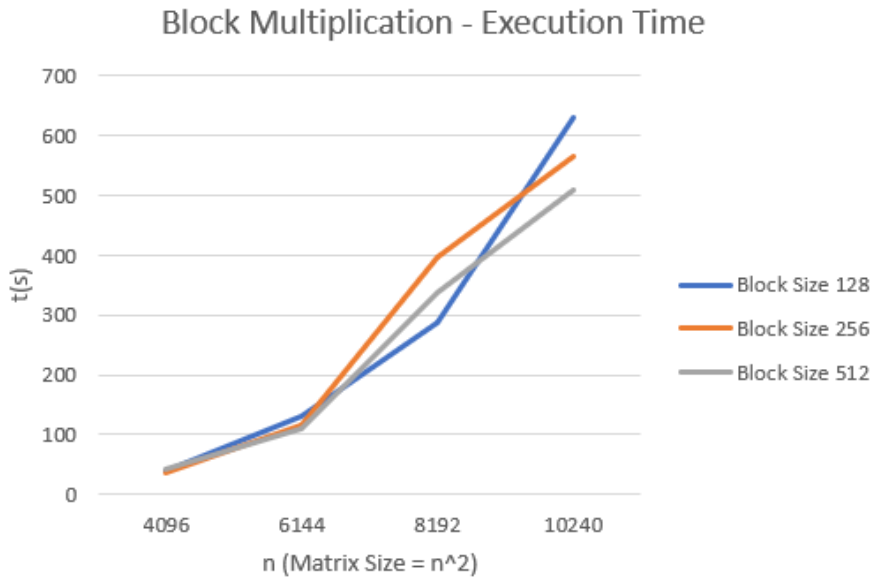


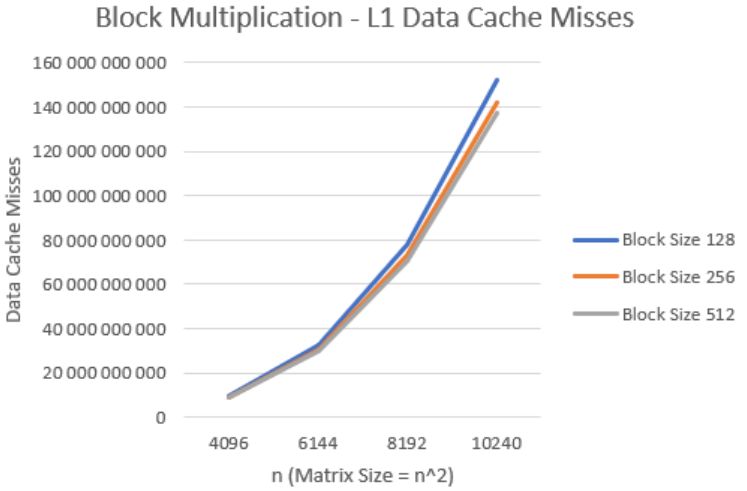
2. LINE MULTIPLICATION

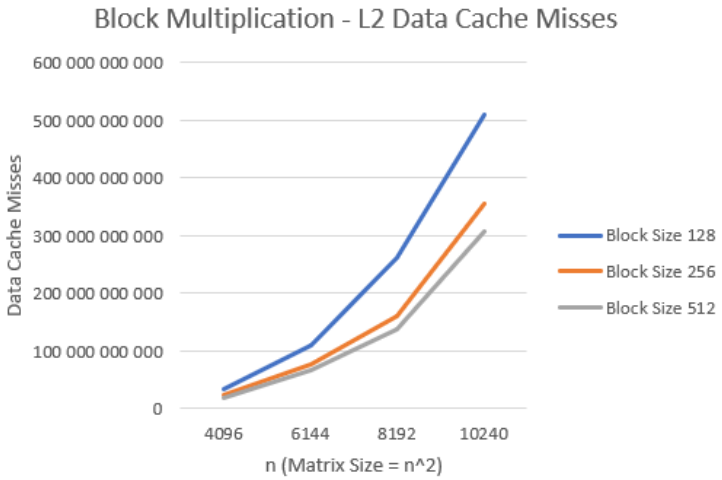




3. BLOCK MULTIPLICATION







ANALYSIS

After analyzing the results, we verified that the implementation of the program in both programming languages did not directly affect the program's efficiency. We also verified that the Basic Multiplication Algorithm gave worse results compared to the Line Multiplication Algorithm. These results confirm the known fact that reading memory in contiguous locations is faster, as explained above in the explanation of the Line Multiplication Algorithm. Comparing the graphics “Line Multiplication – Execution Time” and “Basic Multiplication – Execution Time” we can see that the execution time when using the Line Multiplication Algorithm is much shorter than in the Basic Matrix Multiplication of matrices. The time decreases drastically in both C++ and Java programs. So, it can be said that the algorithm plays a decisive role regarding the efficiency of the program. Therefore, the Line Multiplication Algorithm is much more efficient than the algorithm that does the Basic Multiplication of matrices. Better algorithms generate better results and shorter execution times.

The Block Matrix Algorithm gave the best results overall, especially for larger matrices, having the least amount of data cache misses and executing in the least amount of time among all block sizes. The larger the block size the more efficient the algorithm was, however, the block must fit in the cache (architecture dependent), so we cannot make these blocks arbitrarily large. These results occurred because the Block Algorithm takes advantage of the memory cache block system. By using smaller blocks that can fit in the L1 and L2, the access of data, compared to accessing further level cache, achieved smaller execution times.

CONCLUSIONS

In summary, the results of the analysis show that the choice of matrix multiplication approach has a significant impact on CPU performance. The block multiplication approach was found to be the most efficient due to its ability to reduce cache misses and increase the cache hit rate. On the other hand, the simple approach was the least efficient, while the line approach showed intermediate performance. However, the line approach may be more suitable for smaller matrices.

When selecting a matrix multiplication approach, it is important to consider the size and complexity of the matrices being used, as well as the specific goals of the computation. While the block approach may be the most efficient in many cases, the line and simple approaches may be more appropriate for certain scenarios.

ANNEXES

1. BASIC MULTIPLICATION - RESULTS TABLE

	C++				**Java**	
n	**t (s)**	**GFlops/s**	**L1 Data Cache Misses**	**L2 Data Cache Misses**	**t (s)**	**GFlops/s**
600	0.183	2.361	244 770 128	39 454 063	0.434	0.99539
1000	1.064	1.878	1 223 496 690	275 892 254	1.577	1.26823
1400	3.289	1.668	3 517 623 417	955 156 145	3.807	1.44156
1800	17.041	0.684	9 067 782 383	4 356 229 647	18.337	0.63609
2200	37.436	0.569	17 640 573 544	17 786 157 961	39.264	0.54238
2600	69.370	0.506	30 906 776 427	50 315 029 211	68.987	0.50955

3000	116.429	0.463	50 295 790 944	96 823 869 421	112.199	0.48129

2. LINE MULTIPLICATION - RESULTS TABLE

	C++				Java	
n	**t (s)**	**GFlops/s**	**L1 Data Cache Misses**	**L2 Data Cache Misses**	**t (s)**	**GFlops/s**
600	0.098	4.408	27 108 892	37 402 801	0.353	1.224
1000	0.467	4.278	225 735 664	268 126 577	0.583	3.431
1400	1.523	3.604	346 078 090	704 477 388	1.821	3.014
1800	3.333	3.499	745 330 466	1 444 642 688	4.179	2.791
2200	6.124	3.483	2 074 234 149	2 546 448 080	7.370	2.89
2600	10.214	3.439	4 412 874 037	4 166 671 443	12.213	2.878
3000	15.716	3.436	6 780 588 352	6 426 832 733	18.782	2.875
4096	40.795	3.368	17 550 841 861	16 316 125 318		
6244	239.628	2.156	59 191 253 655	55 074 778 324		
8292	338.002	3.859	140 376 049 506	132 152 171 831		
10240	647.837	331.771	273 707 064 222	255 812 009 228		

3. BLOCK MULTIPLICATION - RESULTS TABLE

	C++				
n	**block size**	**t (s)**	**GFlops/s**	**L1 Data Cache Misses**	**L2 Data Cache Misses**
4096	128	40.039	3.433	9 724 563 072	32 631 116 574
	256	35.820	3.837	9 092 201 897	23 028 977 096

	512	40.833	3.366	8 764 990 644	18 963 273 456
6144	128	132.191	3.509	32 822 940 829	110 251 653 065
	256	117.147	3.96	30 675 152 678	77 346 525 103
	512	110.531	4.197	29 621 129 201	66 430 264 334
8192	128	287.608	3.823	77 775 127 740	262 752 097 640
	256	396.882	2.77	72 961 310 529	159 658 651 705
	512	338.435	3.249	70 219 926 742	137 902 142 961
10240	128	628.919	3.415	151 963 707 747	510 681 171 026
	256	565.468	3.798	142 036 316 744	354 068 913 338
	512	507.924	4.228	137 000 601 942	308 167 320 698