U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Performance evaluation of a single core

CPD Project 1

Group T03_G17

- Guilherme Valler Moreira up202007036
- Tomás Pereira Maciel up202006845

Problem Description and Algorithms Explanation

In this project, we aim to study the effect on the processor performance of the memory hierarchy when we access a large amount of data. To do this, we will use algorithms to multiply two matrices and Performance API (PAPI) to collect relevant performance measures of the program execution.

To achieve this goal, the project is divided in three different parts:

1. Download the example file from moodle that contains the basic algorithm in C/C++ that multiplies two matrices, i.e. multiplies one line of the first matrix by each column of the second matrix (matrixproduct.cpp). Implement the same algorithm in another programming language (just one), such as JAVA, C#, Fortran, etc, of your choice.

2. Implement a version that multiplies an element from the first matrix by the correspondent line of the second matrix, using the 2 programming languages selected in 1.

3. Implement a block-oriented algorithm that divides the matrices in blocks and uses the same sequence of computation as in 2, using C/C++.

- **Simple Multiplication**

The algorithm that we implemented to do the simple matrix multiplication consider that each matrix is represented by an array of arrays, since we thought it would be the best option to represent matrices and to access elements in a easy way.

```
public static void OnMult(int[][] m1, int[][] m2, int[][] res, int n) {

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                res[i][j] += m1[i][k] * m2[k][j];
            }
        }
    }

}
```

So, we did 3 for-cycles in which the variant i represents res' and m1's line, j represents res' and m2's column and k represents m1's line and m2's column (res = m1 x m2).

- ## Line Multiplication

This type of multiplication works by multiplying an element from the first matrix (m1) by the correspondent line in the second matrix (m2).

Basically, this type of multiplication is a variation of the simple multiplication (presented in point 1), so we reused the code with a small modification to implement this multiplication correctly.

```
public static void OnMultLine(int[][] m1, int[][] m2, int[][] res, int n) {

    for (int i = 0; i < n; i++) {
        for (int k = 0; k < n; k++) {
            for (int j = 0; j < n; j++) {
                res[i][j] += m1[i][k] * m2[k][j];
            }
        }
    }

}
```

As we can see, the only alteration made to this algorithm was change the 2nd loop with the 3rd of the first algorithm. With this small alteration, this algorithm has a huge increase in his efficiency when compared to the first algorithm.

- ## Block Multiplication

The third segment of this project will involve utilizing Block Matrix Multiplication. This particular method involves breaking a larger matrix into smaller matrices and then performing multiplication on them instead. To execute this algorithm, there are a total of 6 for loops.

The three outer loops are responsible for selecting the submatrices needed for computation, while the three inner loops perform the Multiline Multiplication on the selected submatrices.

```
public static void OnMultBlock(int[][] m1, int[][] m2, int[][] res, int n, int blockSize) {

    if( n % blockSize != 0 || n % blockSize != 0){
        System.out.println("Error");

    }


    for(int i0 = 0; i0 < n; i0 += blockSize ){
        for(int i1 = 0; i1 < n; i1 += blockSize){
            for(int i2 = 0; i2 < n; i2 += blockSize){
                for(int i = i0; i < i0 + blockSize; i++){
                    for(int j = i1; j < i1 + blockSize; j++){
                        for(int k = i2; k < i2 + blockSize; k++){
                            res[i][j] += m1[i][k] * m2[k][j];
                        }
                    }
                }
            }
        }
    }

}
```

The first if-condition verify if the size of the blocks size is valid, since it needs to be a divisor of the matrix's size (for example, a matrix with size 4x4 can have blocks of size 2x2 but can't have blocks of size 3x3).

- **Three first loops (outer loops):**
    - The variable of the first outer for loop determines the range of the fourth outer loop which specifies the lines to be extracted from matrices A and C to create their respective submatrices.
    - The variable of the second outer for loop determines the range of the fifth outer loop, which specifies the columns to be extracted from matrix A and the lines to be extracted from matrix B to create their respective submatrices.
    - The variable of the third outer for loop determines the range of the sixth outer loop (i.e., the third inner loop), which specifies the columns to be extracted from matrices B and C to create their respective submatrices.
- **Three last loops (inner loops):**
    - The variant i represents res' and m1's line, j represents res' and m2's column and k represents m1's line and m2's column (res = m1 x m2).

By splitting the matrix into smaller matrices, this algorithm is even more efficient than the second one (especially in larger matrices) as we will see in the results' analysis.

Performance metrics

These experiences were made all in the same machine, equipped with an Intel Core i7-9700 @ 3.00GHz processor and running on a Linux environment.

To measure the algorithm's performance, besides the two different programming languages, we tested both with different sizes of the matrices as we will see next.

The metrics that we used to evaluate the performance of the algorithms are:

- Time – execution time of the algorithm
- L1_DCM – Level 1 data cache misses
- L2_DCM – Level 2 data cache misses
- L2_DCA – Level 2 data cache acesses
- L3_DCA – Level 3 data cache acesses

Then, to compare the different algorithms the metric that we use is GFLOPS (number of floating-point operations per second, which is obtained by the formula: $\frac{2*matrixSize^3}{TIme*10^9}$

Results and analysis

Simple Multiplication and Line Multiplication

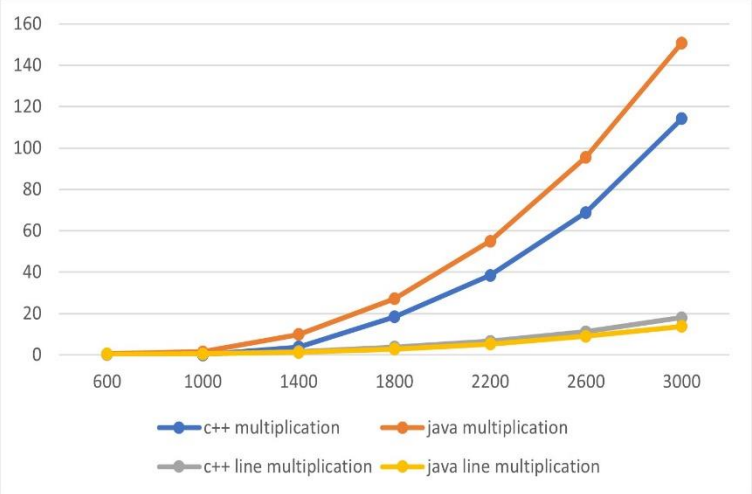

Figure 1: Execution Time of Simple Multiplication and Line Multiplication Algorithms in seconds

As we can see by analyzing the graphic represented in Figure 1, with the growth of the matrix's size the Line Matrix Multiplication is more efficient to multiplicate matrices than the Simple Matrix Multiplication in both programming languages.

If we compare the efficiency of the programming language, we can see that for the Simple Matrix Multiplication C++ was faster than Java while for the Line Matrix Multiplication the execution time was similar. In the first case, this happens because C++ is a compiled language while Java is an interpreted language. Also, C++ is a low-level language and Java is a high-level language and so Java performed worse in this algorithm. The second case in which the execution time was similar in both languages can be explained due to the fact that the Line Matrix Multiplication algorithm is more efficient than the Simple one so performance differences were not able to be recognized.

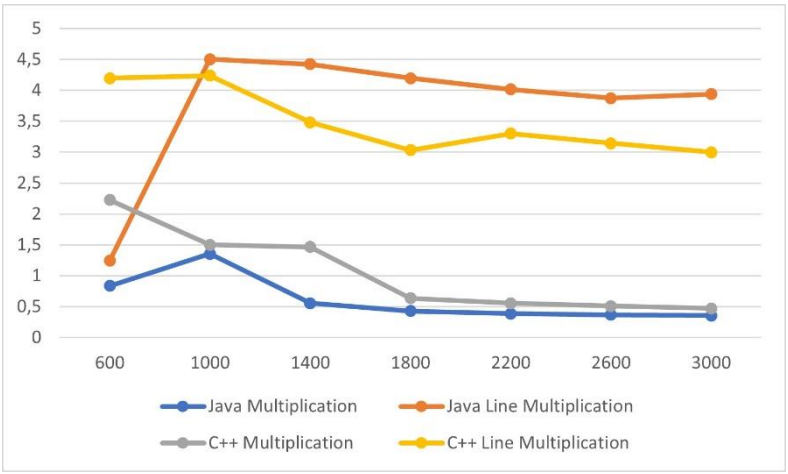

Figure 2: GFLOPS of Simple Multiplication and Line Multiplication Algorithms

When comparing algorithms in terms of GFLOPS, it was found that Line Matrix Multiplication performs more operations per second than the Simple Matrix Multiplication when using the same matrices. This suggests that Line Matrix Multiplication is utilizing computer resources more effectively.

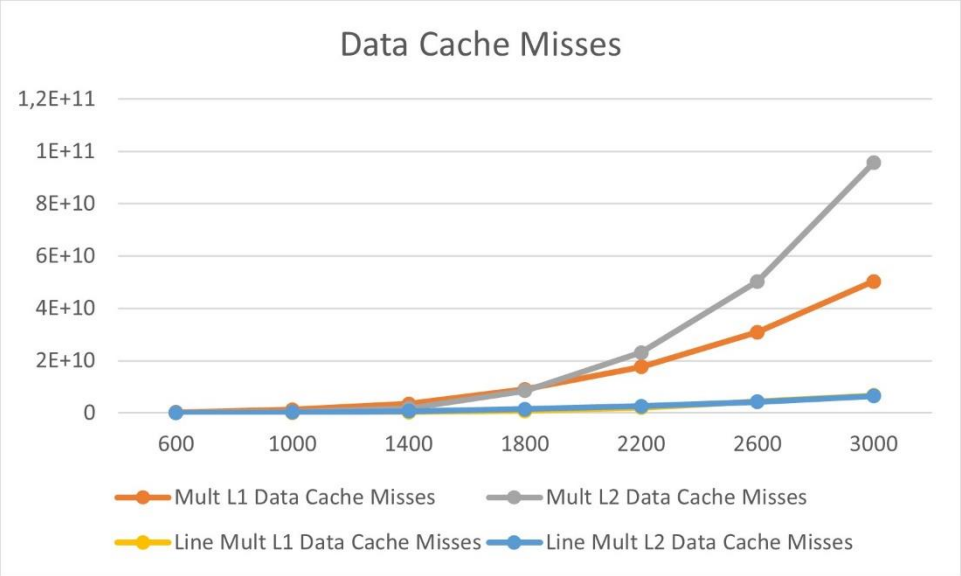

Figure 3: Number of cache misses for each algorithm and matrix's size

Looking at the number of data cache misses in L1 and L2, we can see that the Line Matrix Multiplication avoids many more cache misses than the Simple Matrix Multiplication. This is explained by the fact that the Line Matrix Multiplication uses the cache more efficiently and accesses values closer to each other in memory.

Block Matrix Multiplication

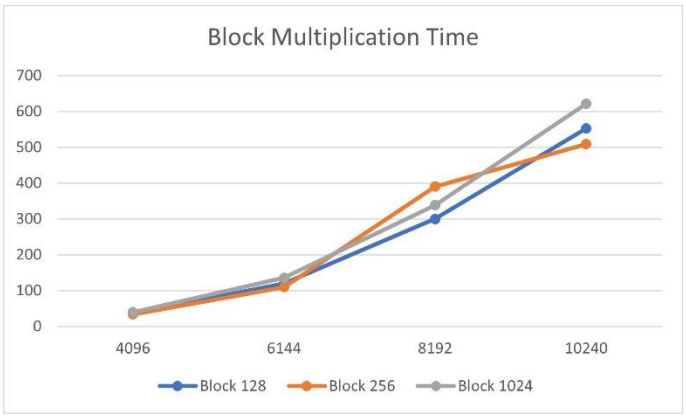

Figure 4: Block Multiplication execution time in seconds

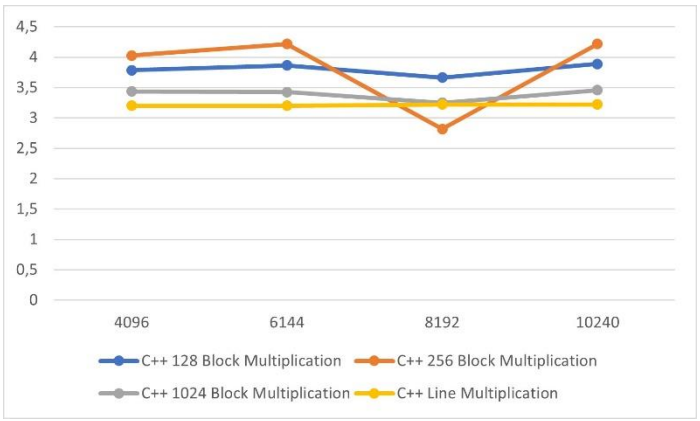

Figure 5: GFLOPS of Block Matrix Multiplication

As expected, this algorithm is the more efficient of all that we tested for every matrix size, executing much faster than the Line Matrix Multiplication for the same

dimensions of matrices, which is exactly what we expected because this technique divides the matrix in smaller blocks.

Analyzing the GFLOPS, we can conclude that for this algorithm the best results were achieved with blocks of 256x256.

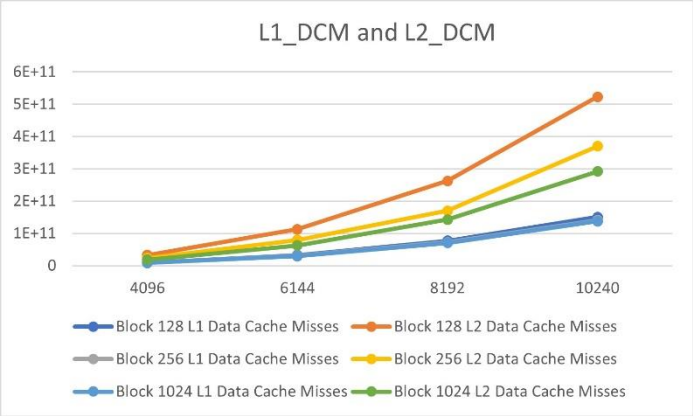

Figure 6: Number of cache misses for Block Multiplication

Looking at the graphic, it's clearly that the number of L2 cache misses is higher than the number of L1 cache misses. Also, we can see that as the size of the block increases, the number of cache misses decreases (it's more visible for the L2 cache misses but happens in L1 caches misses also).

Conclusions

With this project, we understood the importance of writing efficient code and the impacts that this has in the performance. If we analyze all the 3 algorithms implemented, we can see that small changes (like change 1 line per other in Line Matrix Multiplication) have an huge impact in performance. At the end we are happy because we were able to achieve the final goal of this project and we learnt important concepts and ideas that will be very useful for our lives.