CPD project 1

Performance evaluation of a single core

Margarida Cosme 201709304
José Pineda 202111200
Tomás Fidalgo 201906743

Index

Problem description and algorithms explanation

In this project, we evaluate the effect on the processor performance of the memory hierarchy when accessing large amounts of data.

To do this we implemented 3 different functions calculating the product of two matrices in two different programming languages, C++ and Java. This report interprets the results in the performance of the two different languages for the three different algorithms.

The Performance API (PAPI) was used to collect relevant performance indicators of the program execution in C++, such as cache memory related indicators, since this API does not work with the Java language, we used the time module to register the execution times in Java, and calculated the CPU GFlops (billions of floating point operations per second).

Basic Algorithm (OnMult)

This algorithm performs three loops which are used to iterate through the elements of each matrix, a and b , as well as the resulting matrix, c.

The first two loops allow an iteration through each line of a and each column of b and the third one allows an iteration through the columns of a and the lines of b,

For each element in c, this algorithm multiplies and adds the elements of the correspondent line in a and the corresponding column in b.

Efficiency

This algorithm isn't very efficient, as for each c element, it is needed to load a line of a and a column of b. As a column i of b consists of all of the elements of each row in the i-th index, there will always be an unnecessary spend on loading each column of the matrix b.

Line Algorithm (OnMultLine)

As the previous algorithm, the line algorithm performs three loops, but with different meanings: the first one iterates through the lines of a, the second through the columns of a/lines of b, and the last one through the columns of b.

This way, the obtaining of each element is done accumulatively, through n_lines_b iterations.

Efficiency

Compared to the basic algorithm, instead of loading a column of b each time for an element of c, it loaded a line of b, which allows us to accumulate n_columns_b results on the corresponding c lines.

Because of this, we assume prior to the results that this algorithm is faster and more efficient that the basic algorithm, as the spending of time made on loading each column in the previous algorithm is eliminated.

Block Algorithm (OnMultBlock)

The block matrix multiplication divides each matrix in blocks of size bkSize, After this division, the multiplication is made using the resulting matrices as matrix elements.

For this to work , we used double the loops of the previous algorithms: one set of loops for iteration between each block and the other for iterations inside each block.

This way, instead of loading lines and columns of each matrix to get one element, we obtain the resulting matrix block by block.

Efficiency

On this algorithm, both matrices are loaded in blocks, so in terms of memory efficiency, it should be slower than the line algorithm, since we continue to load lines and columns of the submatrices.

However, if the code can be parallelized and run in various threads, this algorithm should probably be faster than the latter ones.

Performance metrics

In order to compare the performance of these algorithms both in C++ and in Java we decided do register the following performance metrics:

- **Processing time** in seconds;
- **GFlops -** billions of floating point operations per second.

Additionally, with the help of the PAPI API, we registered the following performance metrics for the C++ algorithms only:

- **L1 DCM** - Data Cache Miss in Level 1, number of CPU data requests that are not in the L1 cache. The L1 cache is in the same package as the processor;
- **L2 DCM** - Data Cache Miss in Level 2. The L2 cache is installed in a different package than the processor;

- **L1 ICM** - Instruction Cache Miss in Level 1, number of instructions that are not in the L1 cache;
- **L2 ICM -** Instruction Cache Miss in Level 2.

Complete Performance Metrics

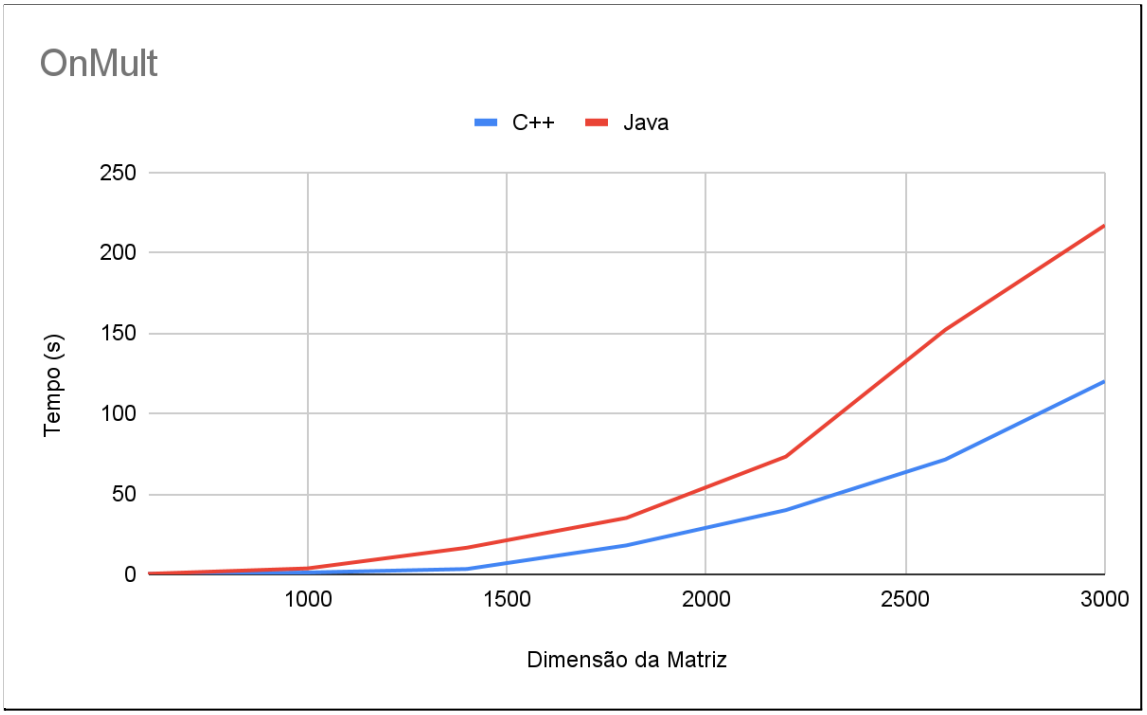

Figure 1: Processing time from the OnMult function in C++ and Java

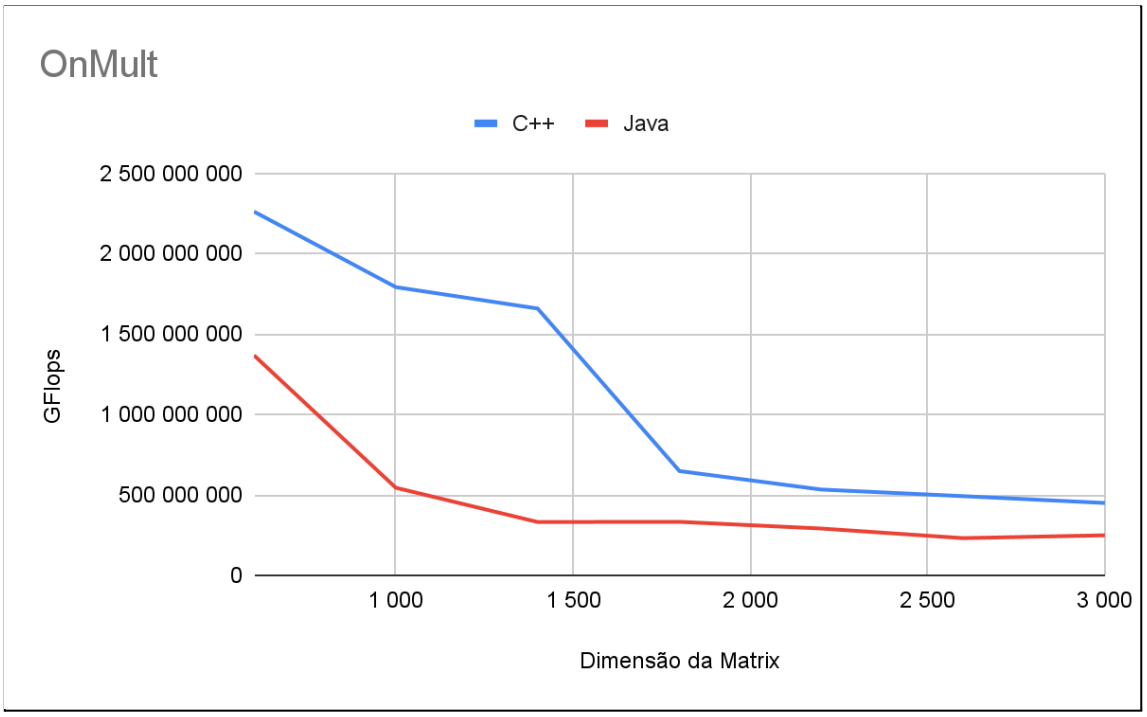

Figure 2: GFlops from the OnMult function in C++ and Java

Dimension	L1 DCM	L1 ICM	L2 DCM	L2 IMC
600	244 764 502	4 640	40 194 367	1 992
1 000	1 231 744 254	5 035	289 438 745	2 574
1 400	3 433 509 015	7 606	1 354 503 302	5 011
1 800	9 081 454 428	14 254	7 697 593 941	11 481
2 200	17 630 217 217	28 776	23 602 818 739	25 904
2 600	30 906 504 969	52 521	51 352 820 176	49 671
3 000	50 293 403 881	76 407	94 320 908 619	73 432

Table 1: Performance metrics on the OnMult function in C++

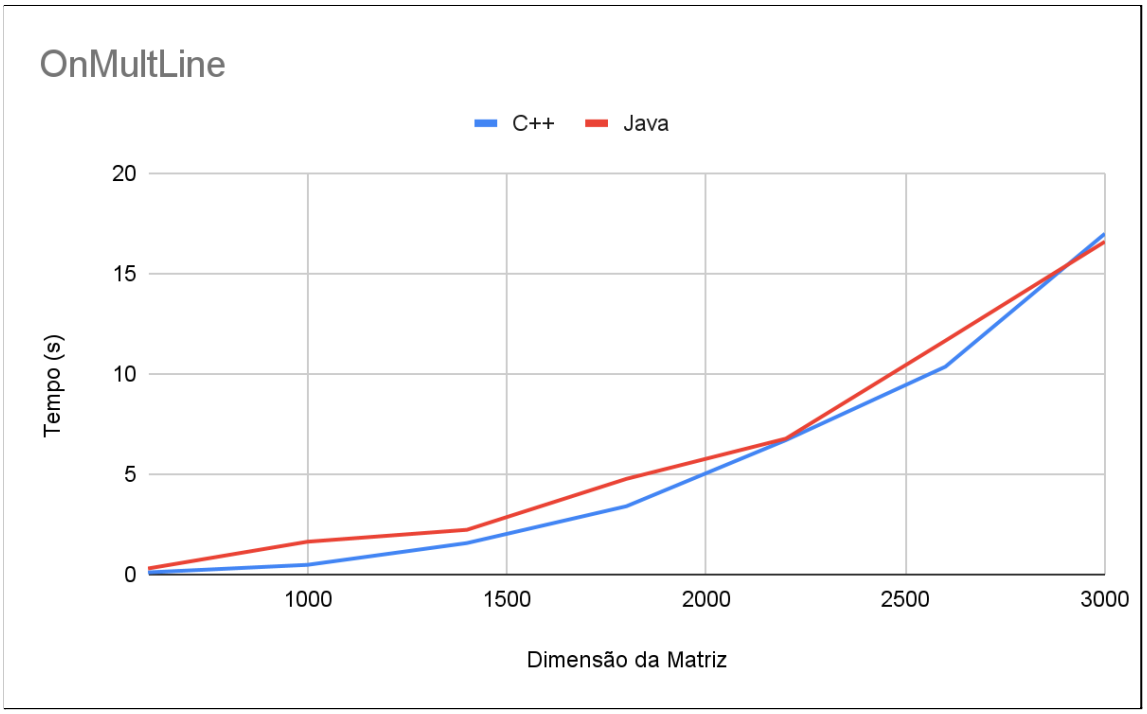

Figure 3: Processing time from the OnMultLine function in C++ and Java

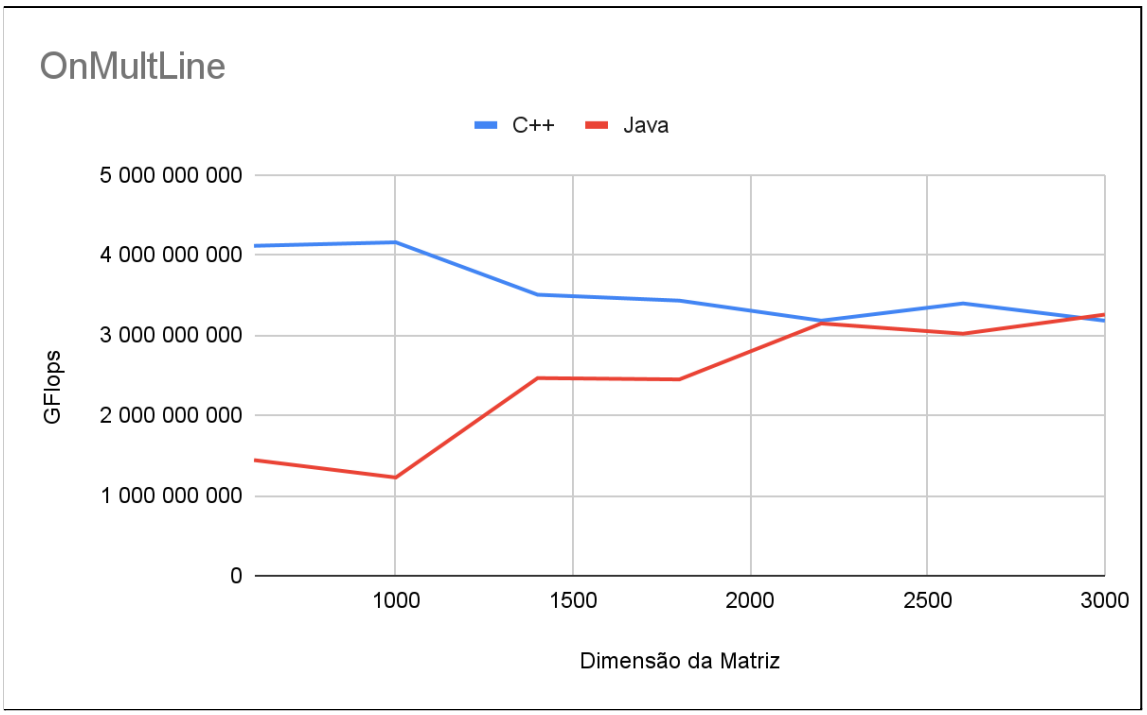

Figura 4: GFlops from the OnMultLine function in C++ and Java

Dimension	L1 DCM	L1 ICM	L2 DCM	L2 IMC
600	27 107 689	4 381	56 445 293	1 966
1 000	125 876 488	4 634	261 273 231	2 228
1 400	346 581 400	5 208	700 198 850	2 644
1 800	746 118 246	6 141	1 438 411 293	3 299
2 200	2 074 628 004	7 468	2 541 439 663	4 473
2 600	4 413 526 577	6 606	4 123 686 755	3 483
3 000	6 781 745 787	8 544	6 255 905 347	5 868
4 096	17 530 833 378	8 534	15 837 423 014	5 863
6 144	59 153 865 040	16 794	53 846 455 342	14 211
8 192	140 351 419 232	39 527	130 678 004 560	36 814
10 240	273 717 035 195	89 144	253 284 131 714	86 064

Table 2: Performance metrics on the OnMultLine function in C++

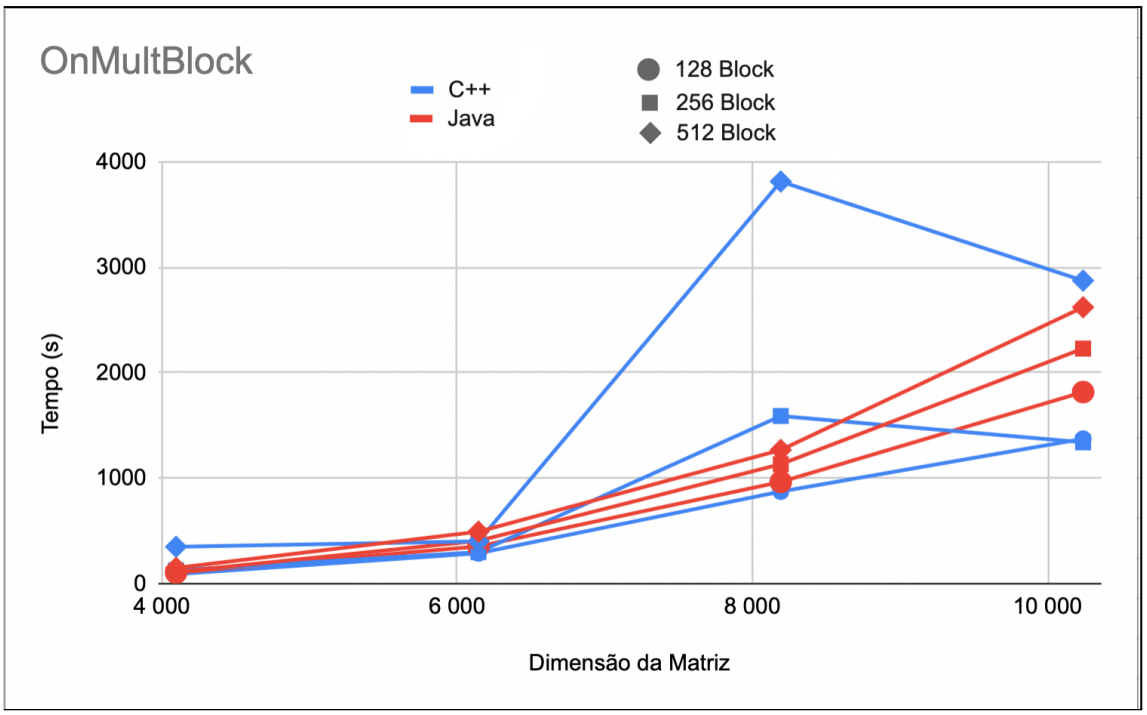

Figure 5: Processing time from the OnMultBlock function in C++ and Java

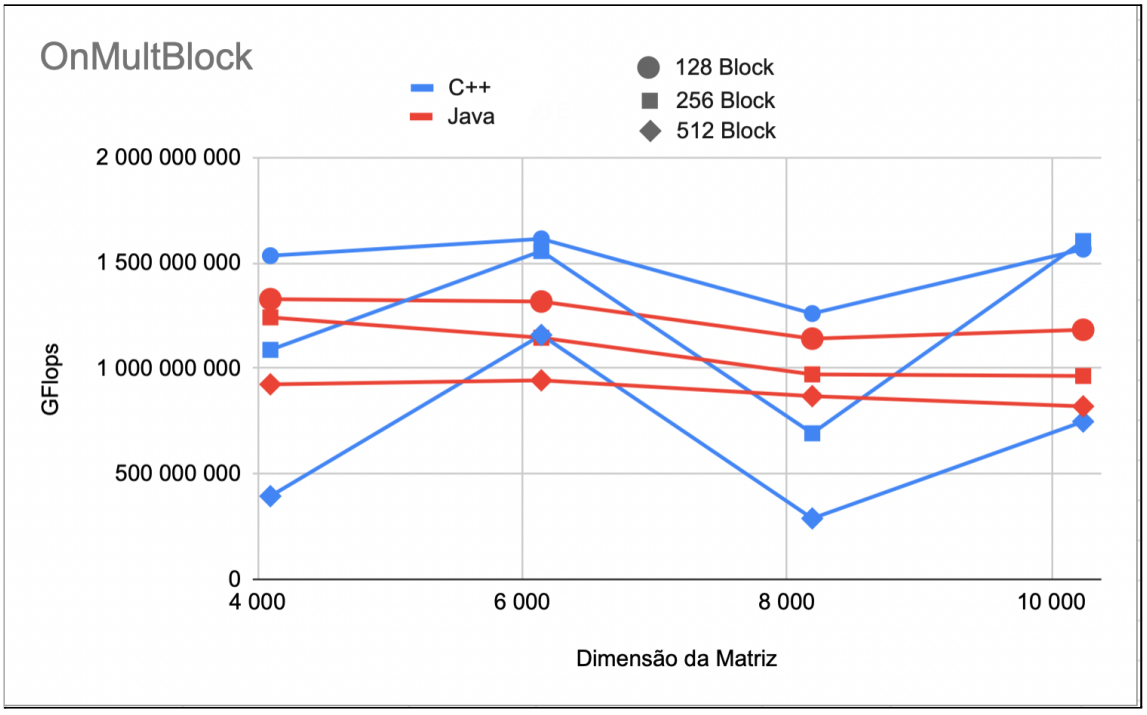

Figura 6: GFlops from the OnMultBlock function in C++ and Java

Block	Dimension	L1 DCM	L1 ICM	L2 DCM	L2 IMC
128	4 096	70 060 096 446	15 677	103 272 622 911	13 655
	6 144	236 427 756 694	46 466	425 114 569 735	37 466
	8 192	585 305 114 433	754 859	974 212 767 017	637 346
	10 240	1 094 604 028 961	369 121	1 939 908 758 947	308 760
256	4 096	73 440 003 493	112 175	125 984 142 412	109 545
	6 144	234 454 326 395	81 207	423 953 708 742	76 584
	8 192	570 373 799 131	1 271 389	919 729 678 034	1 208 707
	10 240	1 085 478 488 725	105 821	1 897 292 795 848	94 913
512	4 096	71 166 148 708	371 570	85 745 208 745	349 340
	6 144	238 878 913 195	131 643	366 677 906 589	104 561
	8 192	557 300 079 304	3 156 959	603 481 574 550	2 866 071
	10 240	1 121 014 255 578	2 282 787	1 657 261 943 575	2 257 253

Table 3: Performance metrics on the OnMultBlock function in C++

Results and analysis

The three algorithms implemented were exactly the same for C++ and Java, however, there are considerable differences in their performances running on different compilers.

For the first algorithm, *OnMult*, the C++ version is significantly faster than the Java one (Figure 1). From the analysis of the results we can conclude that the algorithm *OnMultLine* is significantly faster than the *OnMult* (Figure 1 and 3), as was theorized previously. Because of this, for the *OnMultLine* function, the difference in processing time between C++ and Java is less perceptible, even though the C++ version still runs faster.

In Figure 4 we can see that the C++ *OnMultLine* algorithm performs significantly more floating point operations per second with smaller arguments, decreasing its performance as the size of the matrix increases; the opposite happens for Java, which is very unusual.

The third and final function *OnMultBlock*, for blocks of 128, runs faster in C++. For blocks of 256 and 512, the C++ algorithm has a very strange behavior (Figure 5 and 6), its performance appears to be related to the matrix size in a non-linear way, being faster for sizes of 6 144 and 10 240, and slower for blocks of 4 096 and 8 192. This happens because, for the latter matrix sizes, the instruction cache miss (L1 and L2) increases significantly and disproportionately (Table 3), even though the data cache miss has a predictable and regular increase. An ICM happens when the processor

tries to run an instruction that has not recently been run, meaning that the instruction is not loaded into the cache, or has been flushed out because the cache got filled. Even though we cannot analyze the values of the Java ICM, we can assume that the problem that is occurring in the C++ version is not occuring in the Java one. This is explained by the fact that the allocation of memory in C was done by us, while in Java it is made by the compiler.

The number of data cache misses and instruction cache misses on the *OnMult* and *OnMultLine* (Table 1 and 2) increases in an expected and regular way.

Comparing the three algorithms, the *OnMultLine* one is the fastest, executing 4.5 times faster than *OnMult*, and 2.2 times faster than the *OnMultBlock*.

We conclude that the C++ algorithms are executed faster than the Java ones, having better execution times and GFlops, with the exception of the *OnMultBlock* function, that has an indexing issue.

Analysis and Conclusions

Concluding, excluding the *OnMultBlock* algorithm, the results according to the exoected, and the *OnMultLine* algorithm is the most efficient.

From comparing the results of the *OnMult* algorithm with the *OnMultLine* algorithm, we can see a big improvement in the efficiency of the multiplication, so, as we assumed, we can conclude that loading a matrix line by line is way more efficient than loading a column for each element.

In the *OnMultBlock* algorithm, we expected better results, however, through the analysis of the performance measures we were able to understand what went wrong. Additionally, when comparing the Java *OnMult*, *OnMultLine* and *OnMultBlock* algorithms, (Figure 1, 3 and 5) we can conclude that the memory allocation problem on the *OnMultBlock* was not the only reason for its poor performance. We consider that it is not optimized.

We can also conclude that the *OnMultBlock* algorithm performs better with lower block sizes (Figure 5 and 6).

In addition, between both programming languages Java and C++, we can verify that C++ runs, on average, faster. This happens probably because C++ is a compiled language and differentiates execution from compilation, converting the program into machine code, and Java does not.