Performance Evaluation of a Single Core

This project was made by:

- Henrique Santos Ferreira, up202007459
- Pedro Pereira Ferreira, up202004986
- Pedro Manuel Costa Aguiar Botelho Gomes, up202006086

1) Problem Description and Algorithms Explanation

This project consisted in evaluating the performance of a single core while running similar algorithms conceived in different languages. The first language was C++, which was defined as mandatory for the project. The second language was chosen by us. We opted to use Java given the similarities between the two languages. We used three different algorithms to test the performance:

- **1)Matrix Basic Multiplication:** Multiplying two square matrixes by multiplying each line by every column of the matrix.
- **2)Matrix Line Multiplication:** An element in the first matrix is multiplied by its corresponding line in the second matrix.
- **3)Block Multiplication:** Both matrixes are divided in blocks and then the matrixes of blocks are multiplied using the Line Multiplication algorithm.

The first and second algorithms were available for C++ in Moodle and we implemented similar algorithms in Java. The third algorithm was implemented by us in C++ and its implementation in Java wasn't required.

2) Performance Measures

In order to evaluate the performance of these algorithms, we measured:

- **Execution Time:** Total time that the program takes to execute, in seconds.
- **Cache Misses (Levels L1 and L2):** Amount of times that the CPU requests data that is not in Cache L1 (same package as the CPU, so misses are faster to resolve) or in Cache L2 (different packages, which means misses are slower to resolve than Cache L1's).
- **FLOPS:** Number of Floating Point Operations Per Second

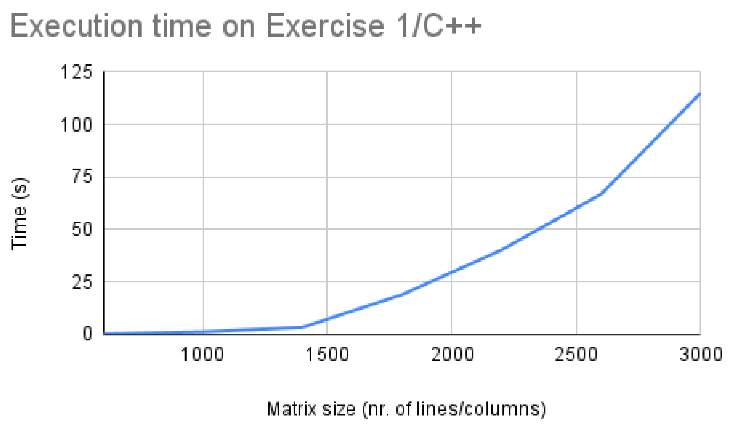
3) Results and Analysis

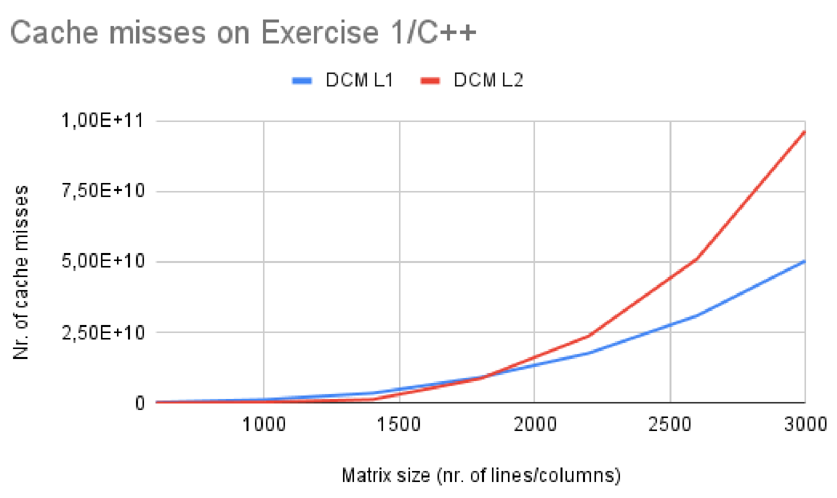
Below are the results we obtained from measuring the data specified above for each algorithm. Following the results of each algorithm is the analysis of the gathered data.

3.1)Basic Multiplication of NxN Matrixes

3.1.1) Code in C++

N	600	1000	1400	1800	2200	2600	3000
Time (s)	0,184	1,146	3,269	18,797	40,149	66,7	114,939
DCM L1	244766701	1219487961	3487120578	9082918864	17640197639	30898411012	50306259669
DCM L2	39726760	289033431	1252881953	8633609190	23716666963	51028948991	96344648970
FLOPS	2347826087	1745200698	1678800857	620524551,8	530424170	527016491,8	469814423,3

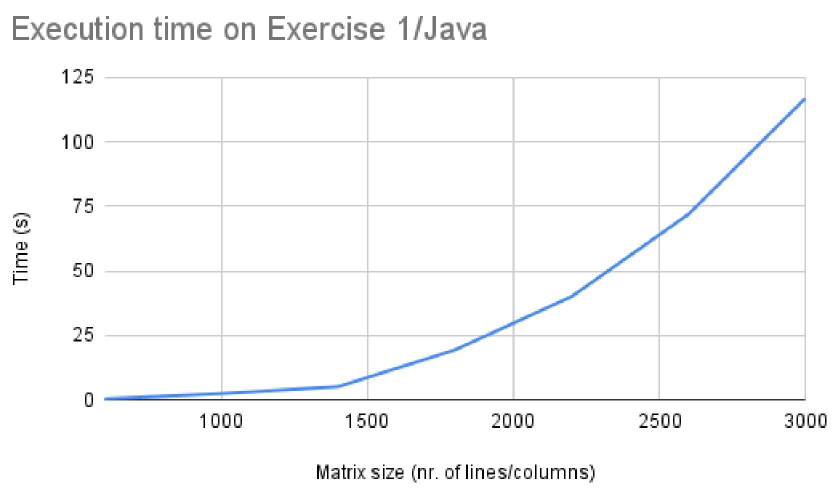




As expected, the execution time is bigger on bigger matrixes, as is the number of cache misses, as the number of memory accesses is bigger when we are working with a bigger matrix.

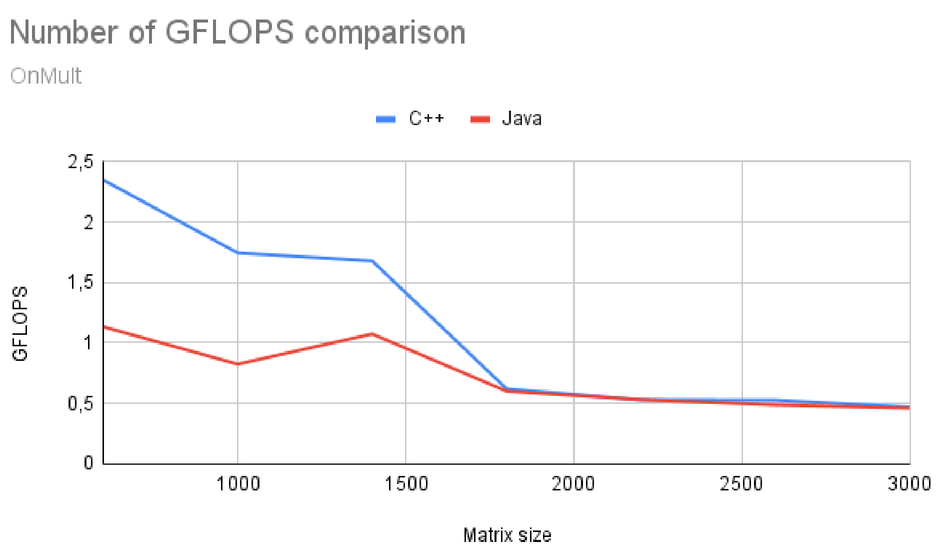
3.1.2) Code in Java

N	600	1000	1400	1800	2200	2600	3000
Time (s)	0,381	2,423	5,109	19,382	40,072	71,956	116,874
FLOPS	1133858268	825423029,3	1074182815	601795480,3	531443401,9	488520762,7	462036038,8



The same happens for the Java algorithm, with slightly bigger execution times.

3.1.3) FLOPs comparison

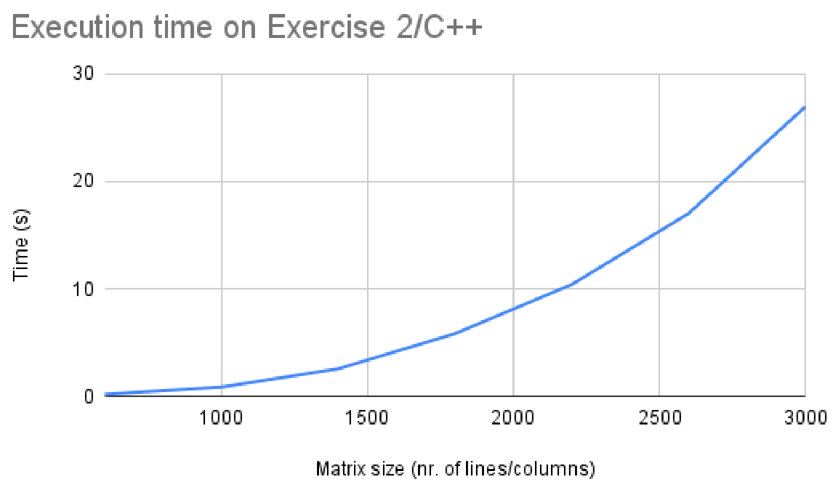


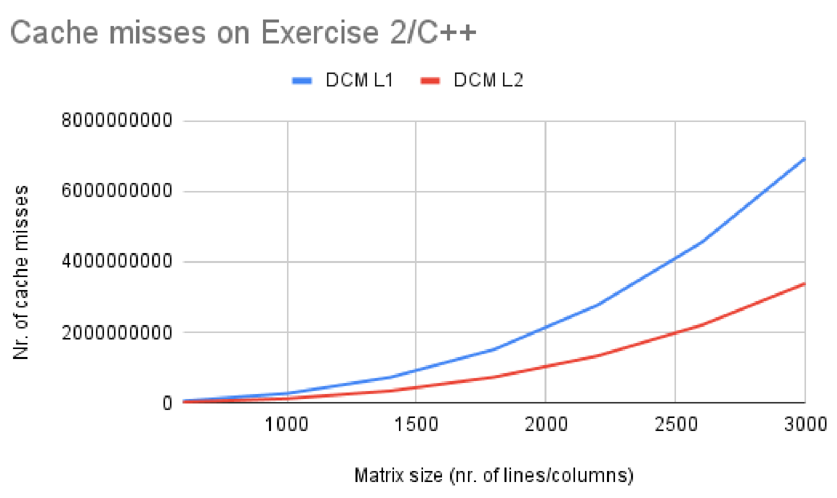
In terms of floating point operations per second, the algorithm seems to have better performance in C++ for matrixes with N <= 1700, but the number of FLOPS from that point upward is pretty similar in both languages.

3.2) Line Multiplication of NxN Matrixes

3.2.1) Code in C++

N	600	1000	1400	1800	2200	2600	3000
Time (s)	0,189	0,866	2,566	5,837	10,409	17,009	26,940
DCM L1	27098374	125547208	344281821	731414577	1335138713	2203454760	3384509661
DCM L2	58144352	270210229	726096184	1517352215	2778638494	4548643623	6938439239
FLOPS	2285714286	2309468822	2138737334	1998286791	2045921798	2066670586	2004454343



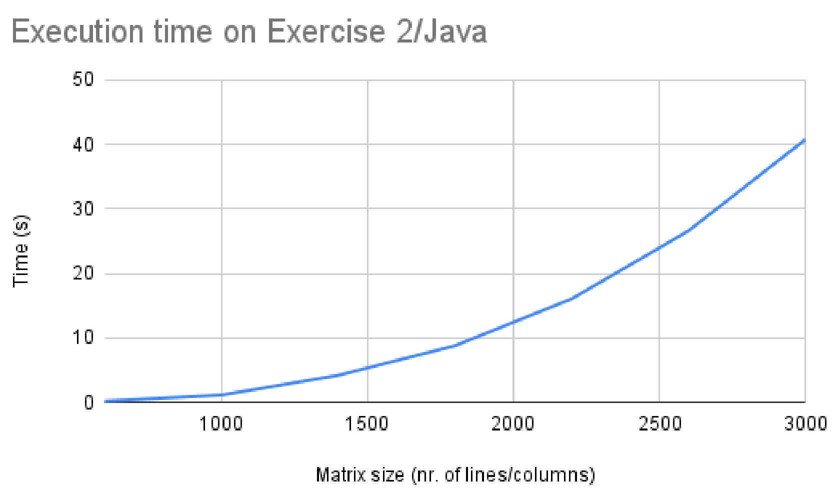


In this new algorithm, it was possible to notice how a relatively simple change in the code reduced the execution time in about a third. Of course, the execution time still varies exponentially as before, but the execution times measured for maximum N are significantly lower than the execution time of the first algorithm for the same N.

When it comes to cache misses, there is also a significant decrease, which can be explained by the fact that this second algorithm takes advantage of C++'s memory allocation. What happens is that everytime the algorithm accesses a value in the matrix, it accesses the whole block where the value is inserted (in this case, the line). The difference between the previous algorithm and this one is the fact that every value read from the line is used, making a better use of the time spent accessing the memory and decreasing the need to access it so often.

3.2.2) Code in Java

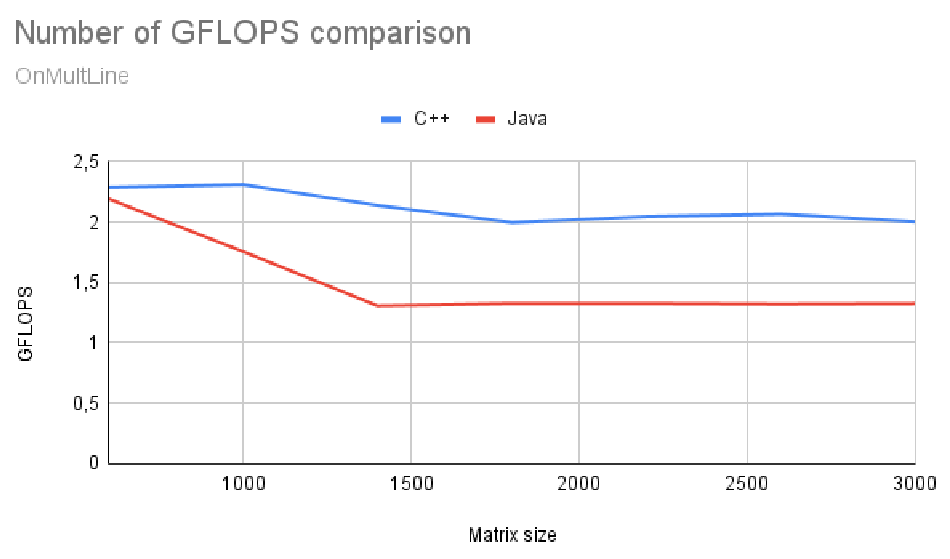
N	600	1000	1400	1800	2200	2600	3000
Time(s)	0,197	1,138	4,195	8,792	16,058	26,601	40,744
FLOPS	2192893401	1757469244	1308224076	1326660601	1326192552	1321454081	1325348518



Analysing the execution times of the Java algorithm, we can conclude it also effected performance positively, despite the improvement not being as good as the C++ algorithm. This can be justified by the fact that Java and C++ work differently in terms of memory allocation. While in C++ it is possible to allocate arbitrary blocks of memory, Java only allocates memory via object instantiation.

The changes made from the Basic Multiplication Algorithm to the Line Multiplication one make better use of the dynamic memory allocation of C++, hence why the algorithm improvement is more significant in C++.

3.2.3) FLOPs comparison

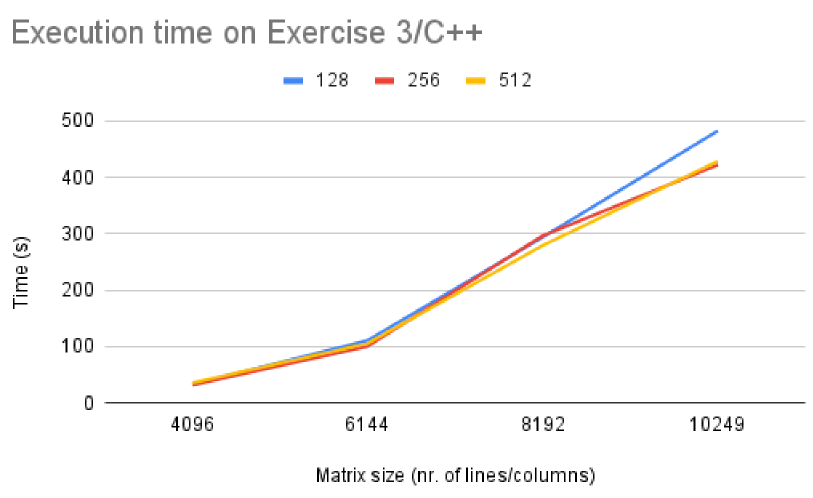


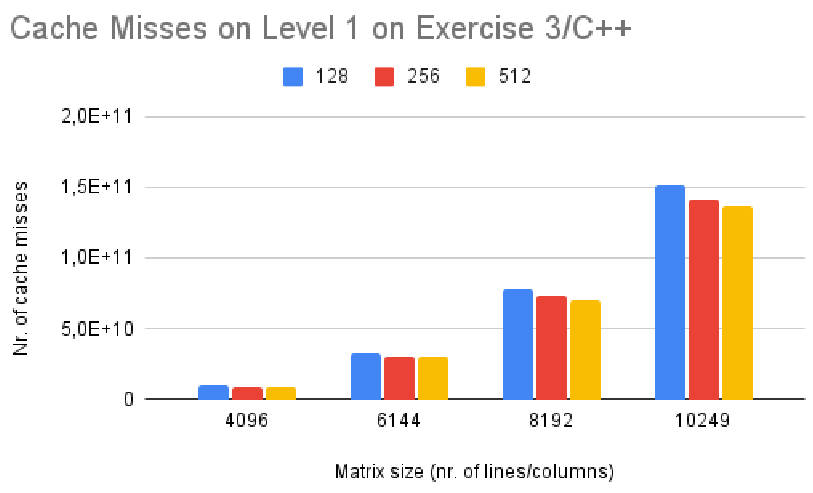
In comparison to the first algorithm, it is easily observed that the number of floating point operations per second doesn't decrease as much with the increase of the matrix size. Still, as the execution time already hinted, the C++ code has better performance and maintains a bigger amount of FLOPS for bigger matrix sizes, thanks to the way memory allocation is handled in C++, as explained earlier.

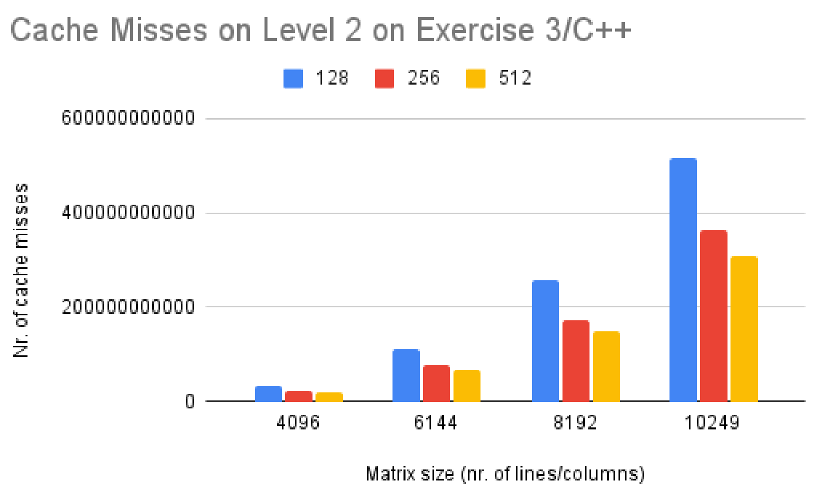
3.3) Block Multiplication of NxN Matrixes

3.3.1) Code in C++

Size		4096	6144	8192	10240
Time (s)	Block 128	32,5621	110,635	293,535	481,981
	Block 256	31,7325	100,255	296,2	421,757
	Block 512	35,5202	105,263	278,95	428,209
DCM L1	Block 128	9730902823	32843784869	78050234918	151972611631
	Block 256	9087884775	30665496872	72822173076	141827954816
	Block 512	8766051901	29607613185	70185948989	136904392692
DCM L2	Block 128	32923750341	110686253564	256582643073	515431636005
	Block 256	23036198163	77372230133	173602049401	362032973580
	Block 512	19220293936	66216447548	148003917885	307479992520
FLOPS	Block 128	4220825852	4192673819	3745759883	4455535899
	Block 256	4331241443	4628151339	3712058163	2606978966
	Block 512	3869318120	4406643056	3941608273	2567698549

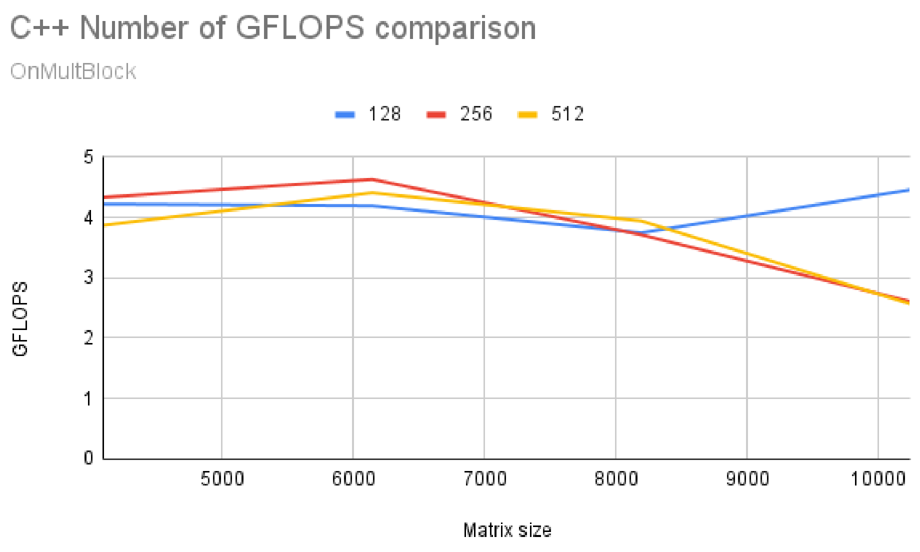






As the amount of data increases, the previous solutions become ineffective, making it necessary to create more efficient algorithms. In the block multiplication algorithm, we divide the problem into smaller ones by arranging the memory in consecutive blocks. In this way, we can ensure the reduction of memory calls and cache misses, which leads to an increase in program performance.

3.3.2) FLOPs comparison



In comparison with the previous algorithms, the amount of information processed in relation to time is immensely superior. It is also possible to note that an increase in block size leads to an increase in performance, although it is capped by the Level 1 cache block size.

4) Conclusions

This project allowed us to understand that time complexity isn't the only concern when it comes to the performance of our program, as there are several factors that can effect the performance of our code positively or negatively without making time complexity vary at all. It also helped us finding out there are better performance metrics than performance time, like the number of cache misses and the number of floating point operations per second that give us more information about program performance than execution time does.