CPD 1st Project - Performance Evaluation of a Single Core (Matrix Multiplication Analysis)

Afonso Duarte de Carvalho Monteiro - up201907284
Miguel Azevedo Lopes - up201704590
Vasco Marinho Rodrigues Gomes Alves - up201808031

March 27, 2022

Contents

1 Problem & Algorithm Explanation

This report aims to present the results of the study done on the effect on the processor performance of the memory hierarchy, when accessing large amounts of data. To evaluate this we multiplied 2 matrices using three different algorithms and two languages, while also using the *PAPI*[1] to collect relevant information. To simplify the analysis, all the input matrices used are square matrices, stored in row-major order.

Firstly, we implemented a simple iterative method (referred to as simple multiplication throughout this report) of the matrix multiplication on a programming language of our choice (we decided on Java due to its relative proximity to the default language used - C++) and registered the information required. This algorithm uses three loops to multiply one line of the first matrix by each column of the second matrix.

Secondly, the line multiplication method was implemented for both languages and the same information was collected. We also took notes from matrices 4096x4096 to 10240x10240 with intervals of 2048 in the C++ implementation. This version of the algorithm is a simple switch of the order of the loops mentioned in the first algorithm. It multiplies an element from the first matrix by the corresponding line of the second matrix.

Finally, the block multiplication algorithm was implemented in C++, with the same information collected - 4096x4096 to 10240x10240 matrices in intervals of 2048, with different block sizes (128, 256, 512 and 1024). This implementation divides the matrices in blocks and multiplies those blocks using the line multiplication algorithm described earlier.

2 Performance Metrics

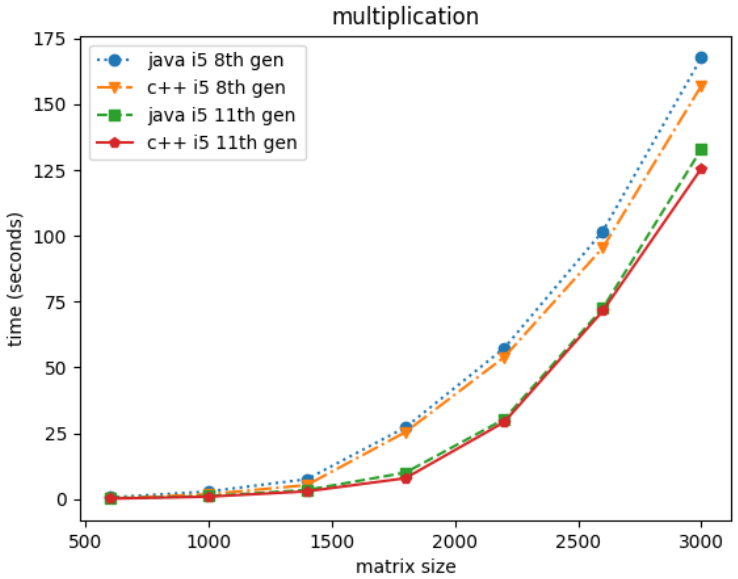
In this report, we will mainly use a combination of the time the algorithm took to finish its iterations, plus the *percentage of L1 Data Cache Misses*[2], and as an inverse consequence, the *percentage of L1 Data Cache Hits*[3], as additional comparison parameters. Those pieces of data are provided by the *API*[4].

3 Results and Analysis

Table and Graphic 1: Simple Multiplication

Matrix Size	Intel Core i5 8th Generation	Intel Core i5 11th Generation	Intel Core i5 8th Generation	Intel Core i5 11th Generation
	Multiplication C++		Multiplication Java	
	Time (seconds)			
600	0.2620	0.1900	0.6604	0.3473
1000	1.8760	0.9430	2.8963	1.1490
1400	5.3530	2.9550	7.5831	3.4821
1800	25.4750	7.9650	27.2954	10.0872
2200	53.9170	29.2240	57.3248	30.3671
2600	95.4850	71.4640	101.7686	72.4915
3000	157.0670	125.6610	167.7108	132.9635

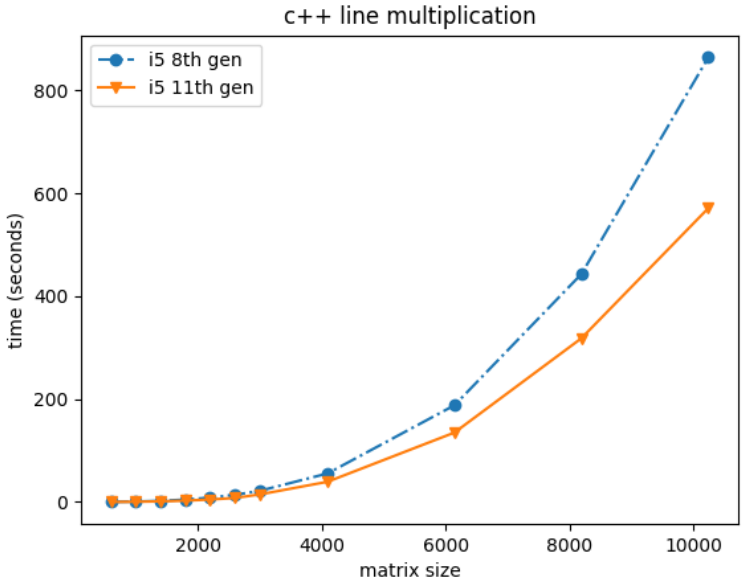
[1] Performance API
[2] *(L1 Data Cache Misses/ Total L1 Data Cache Accesses) * 100*
[3] *100 - (L1 Data Cache Misses/ Total L1 Data Cache Accesses) * 100*
[4] Application Programming Interface

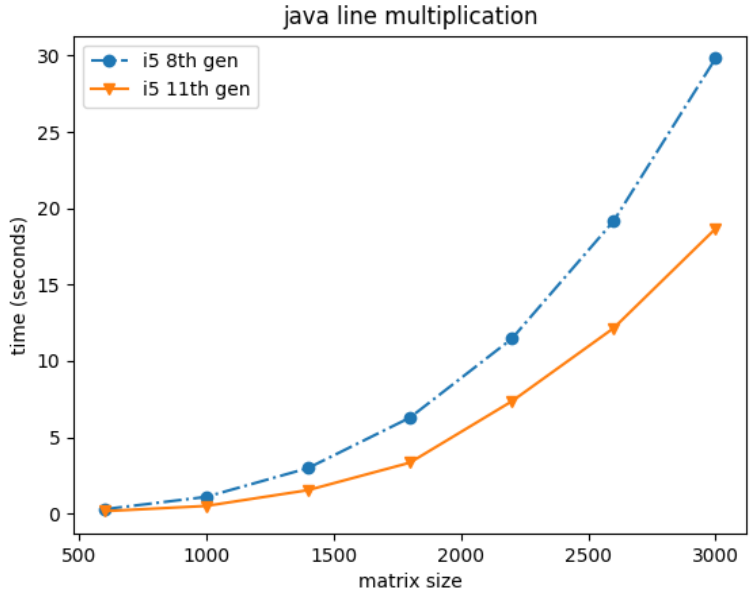


As we can observe from the graphic the normal multiplication performs well for small matrices but gets exponentially worse when we are dealing with bigger matrices. It is also important to notice that the C++ version performs better than its counterpart on Java. As expected, the 11th Generation Intel processor also outperforms the 8th Generation one, especially when the matrices start to go above the size of 1500.

Table and Graphic 2: Line Multiplication

Matrix Size	Intel Core i5 8th Generation	Intel Core i5 11th Generation	Intel Core i5 8th Generation	Intel Core i5 11th Generation
	Line Multiplication C++		Line Multiplication Java	
	Time (seconds)			
600	0.1350	0.0730	0.2870	0.1619
1000	0.7960	0.3640	1.0972	0.5045
1400	2.1350	1.2070	2.9816	1.5392
1800	4.5650	2.5720	6.2863	3.3395
2200	8.4070	4.7100	11.4576	7.3583
2600	13.9470	7.7950	19.1742	12.1574
3000	21.4810	14.5490	29.8319	18.6726
4096	55.402	39.236	-	-
6144	187.9	134.859	-	-
8192	442.784	318.029	-	-
10240	863.779	570.863	-	-

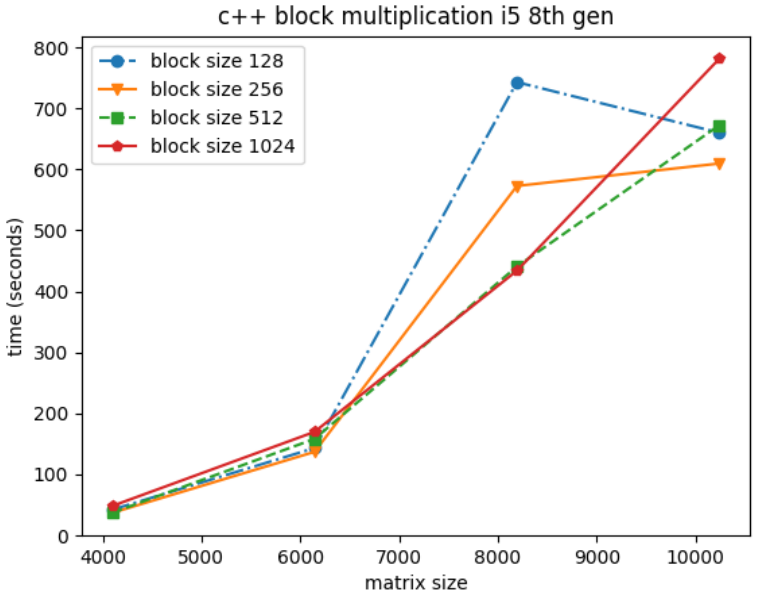


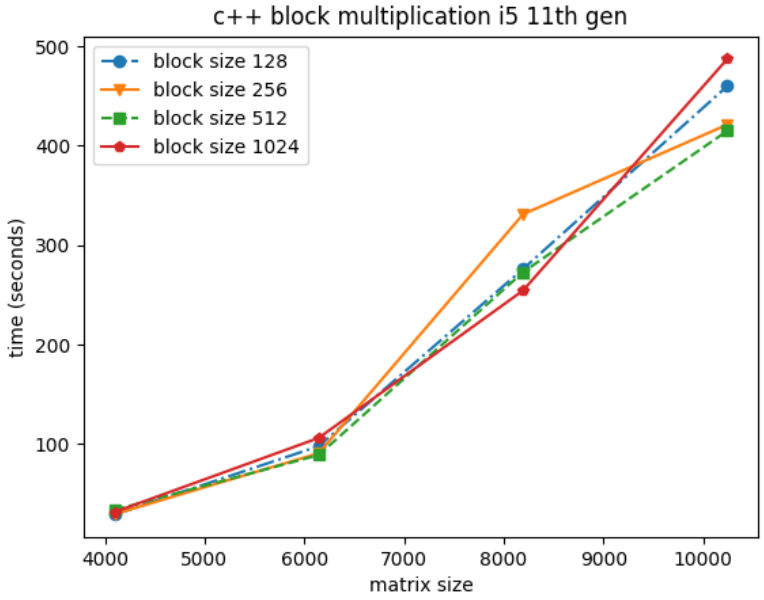


The C++ version of the line multiplication algorithm performs significantly better than the normal one. For example, for a matrix of size 3000, the multiplication algorithm takes around 150 seconds while with this implementation it only takes about 20 seconds. Once again, the 11th Gen processor outperforms the 8th Gen one. For Java, the line multiplication algorithm was only tested up to matrices of size 3000, and it was outperformed by the C++ version.

Table and Graphic 3: Block Multiplication

Block Size	Matrix Size	Intel Core i5 8th Generation	Intel Core i5 11th Generation
		Block Multiplication C++	
		Time (seconds)	
128	4096	42.161	29.2140
	6144	143.275	97.9690
	8192	742.818	276.0810
	10240	659.889	459.9310
256	4096	37.054	29.6220
	6144	137.066	90.9040
	8192	572.808	331.1230
	10240	609.576	421.2420
512	4096	37.203	33.8460
	6144	158.319	89.0810
	8192	440.790	272.8740
	10240	672.380	414.9930
1024	4096	48.302	31.7130
	6144	169.832	106.2760
	8192	434.031	254.4580
	10240	781.443	487.2600





The block multiplication algorithm was implemented only in C++. It was tested with four different block sizes and on two different processors, just like the others. We can observe that for a matrix with the size of 4096, this algorithm outperforms the normal multiplication and line multiplication versions, for every block size we tested. The exact same thing happens for the size 6144 matrix. However, the line multiplication method outperforms the block multiplication one for a block size of 256 when using them on a matrix with size 8192. For the multiplication of matrices with size 10240, the block multiplication outperforms the other methods for every block size.

Table 4: L1 Data Cache Statistics

Intel Core i5 8th Generation			
	Multiplication	Line Multiplication	Block Multiplication
Average L1 Data Cache Misses	77,63%	8,37%	4,34%
Average L1 Data Cache Hits	22,37%	91,63%	95,66%

The simple multiplication method was the most inefficient of all the methods, with an average of 77.63% of L1 data cache misses and exponential growth in time used to complete the operation. The line multiplication implementation was in every way incomparably more efficient, as the average percentage of cache misses was way lower, at 8.37%. Despite this, the most reliable algorithm, at least for matrices of size 4096 and up, was the block multiplication method. It boasts an average of cache misses of only 4.34%, making it the most efficient out of the 3 methods.

4 Conclusion

The most important objectives of this project were achieved. The PAPI was successfully utilized to collect relevant performance indicators of the program execution, to deduce the efficiency on the L1 Cache Access and if the algorithms were well implemented.

By coding the three different methods of matricial multiplication, on 2 different languages, and on top of it comparing the time results on 2 different processors, we were able to have the perception of what plays into the efficiency of a program. When comparing the simple multiplication with the line multiplication, two very similar algorithms composed of the 3 loops, we realized the impact that a change in the order of a loop can have on practical performance. We also verified that the size of the matrices, the sequence in which they are stored, the language in which we code, the algorithm and the processor used all have an impact on performance.

It should also be noted that in an ideal situation, we would've tested these algorithms multiple times, in order to create a more reliable statistical analysis, however due to time constraints associated with the delivery date of this report, we were only able to run these algorithms and tests once.