Performance evaluation of a single core

Parallel and Distributed Computing
Bachelors in Informatics and Computer Engineering

3L.EIC04_G16
Gonçalo Jorge Soares Ferreira up202004761
Gonçalo José Vicente Domingues up202007914
Pedro Nuno Ferreira Moura de Macedo up202007531

Introduction

This project intends to show and evaluate the performance on the processor when accessing large amounts of data. In this study, the product of two matrices was used as the benchmark.

Finally, performance measures were made using the **Performance API (PAPI)**.

Problem Description

The main goal of this project was to measure the performance of the processor when submitted to different methods of the matrix multiplication problem. This allows for a more accurate measurement of the time spent by the processor accessing memory, as well as the impact of cache hits and misses on the program execution.

To improve the performance of the matrix multiplication problem, two techniques were used. Firstly, the **line multiplication** technique, and then the **block multiplication**.

Algorithm Analysis

Normal Multiplication

This first technique is normally used. Consists of a dot product of rows of the first matrix with columns of the second matrix.

Below, we can see an example of a pseudocode that describes this first implementation.

The following pseudocode details how the algorithm works. Here, a is the first Matrix and b the second one, and they produce the result on matrix c:

Algorithm 1 Normal Multiplication

```
1: procedure NORMAL MULTIPLICATION(matrix_a, matrix_b)
2:     for $i = 0$ to $length(matrix\_a)$ do
3:         for $j = 0$ to $length(matrix\_b)$ do
4:             $temp \leftarrow 0$
5:             for $k = 0$ to $length(a)$ do
6:                 $temp \leftarrow temp + matrix\_a_{i,k} + matrix\_b_{k,j}$
7:             $matrix\_c_{i,j} \leftarrow temp$
```

Although this implementation is simple and very easy to understand, the execution time increases dramatically with the increase of matrixes size.

Line Multiplication

This second technique changes the orders that the operations ate performed. Consists of multiplying each element of a line from the first matrix with all elements from the correspondent line in the second matrix.

To further clarify this, the following pseudocode shows how it works. Once again, a represents the first matrix, b the second one, and the output will be stored in c:

To further clarify this, in the below pseudocode we can see the phases of this implementation. Where *matrix_a* corresponds to the first matrix, *matrix_b* corresponds to the second matrix and, finally, $matrix_c$ corresponds the resulted matrix from the multiplication of the first two matrixes.

Algorithm 2 Line Multiplication

```
1: procedure Line Multiplication(a, b)
2:     for $i = 0$ to $length(matrix\_a)$ do
3:         for $j = 0$ to $length(matrix\_b)$ do
4:             for $k = 0$ to $length(matrix\_a)$ do
5:                 $matrix\_c_{i,k} \leftarrow matrix\_c_{i,k} + matrix\_a_{i,j} + matrix\_b_{j,k}$
```

Block Multiplication

The final algorithm used was the block multiplication method. Furthermore, this algorithm divides the matrixes in blocks of the same size and multiplies block by block, using a technique similar to the **Line Multiplication** algorithm.

The following pseudocode can help with understanding how such calculations are done. Again, a is the first matrix, b the second one, and c will store the result of executing the instructions. Also, this time, the size of the blocks is included, as it can affect speed of computations:

Below, you can see an implementation of a pseudocode. Where *matrix_a* corresponds to the first matrix, *matrix_b* corresponds to the second matrix and $matrix_c$ corresponds the resulted matrix from the multiplication of the first two matrixes.

Even though the number of loops increased, we can conclude that it's the fastest algorithm (proved later on).

Performance Evaluation

Metrics Used

In order to evaluate the performance of the processor, 6 metrics were used. To measure the speed of execution, the execution time was used, in seconds. The other 5 metrics are relative to the memory access (using the PAPI API), to measure the data cache misses (DCM), on layers 1

Algorithm 3 Block Multiplication

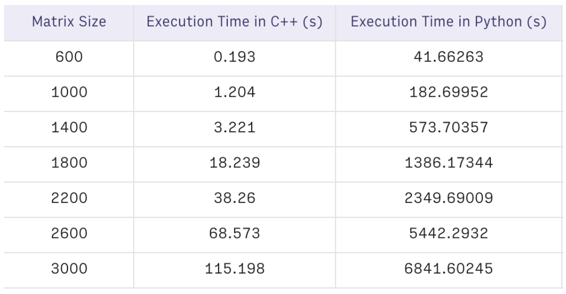
```
1: procedure BLOCK MULTIPLICATION(a, b, blockSize)
2:     for $line\_block\_a = 0$ to $length(matrix\_a)$ in $blockSize$ increments do
3:         for $col\_block\_a = 0$ to $length(matrix\_a)$ in $blockSize$ increments do
4:             for $col\_block\_b = 0$ to $length(matrix\_b)$ in $blockSize$ increments do
5:                 for $line\_a = 0$ to $bkSize$ do
6:                     for $col\_a = 0$ to $bkSize$ do
7:                         for $col\_b = 0$ to $bkSize$ do
8:                             $matrix\_c_{(line\_block\_a+line\_a)*bkSize,(col\_block\_a+col\_block\_b)} \leftarrow matrix\_c_{(line_block\_a+line\_a)*bkSize,(col\_block\_a+col\_block\_b)} + matrix\_a_{(line\_block\_a+line\_a)*bkSize,(col\_block\_b+col\_a)} + matrix\_b_{(col\_block\_b+col\_a)*bkSize,(col\_block\_a+col\_b)}$
```

and 2, the data cache accesses (DCA), on layers 2 and 3, and the GFLOPS. The purpose of DCM metrics is to evaluate the memory efficiency of the 3 algorithms by counting the number of data cache misses during its execution. Measuring this metrics is important because cache performance is a crucial factor in determining the system performance. A high cache miss rate can lead to slow performance, and that makes the processor to wait for the data to be retrieved from other levels of memory. In this order, system designers can ensure that their system are giving the user a pleasant experience.

Results and Analysis

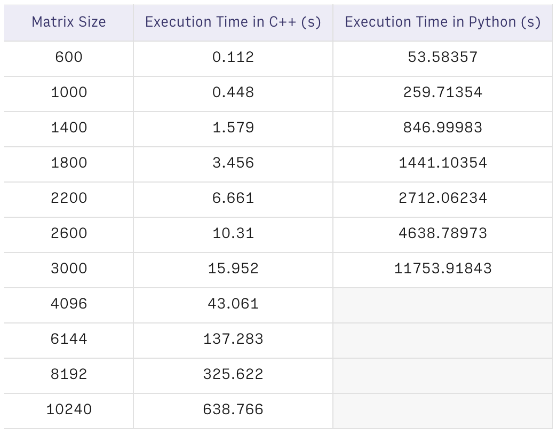
We chose to compare the values obtained between the C++ and Python languages to better demonstrate the efficiency differences. In general, we can observe that the execution times obtained in python are much slower then the results obtained in C++.

Comparison between first method and second method



Matrix Size	Execution Time in C++ (s)	Execution Time in Python (s)
600	0.193	41.66263
1000	1.204	182.69952
1400	3.221	573.70357
1800	18.239	1386.17344
2200	38.26	2349.69009
2600	68.573	5442.2932
3000	115.198	6841.60245

Figure 1: Normal Multiplication execution times[1]

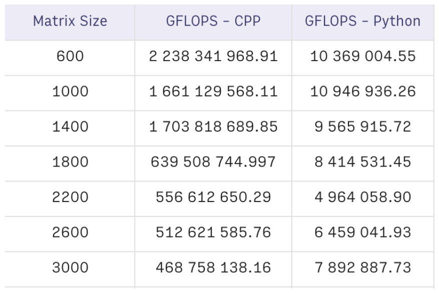


Matrix Size	Execution Time in C++ (s)	Execution Time in Python (s)
600	0.112	53.58357
1000	0.448	259.71354
1400	1.579	846.99983
1800	3.456	1441.10354
2200	6.661	2712.06234
2600	10.31	4638.78973
3000	15.952	11753.91843
4096	43.061	
6144	137.283	
8192	325.622	
10240	638.766	

Figure 2: Line Multiplication execution times[2]

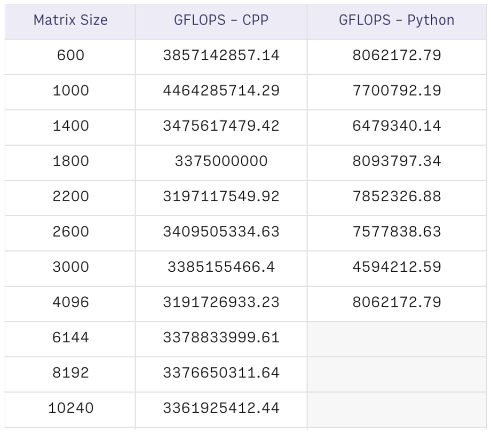
In the first algorithm, we can see C/C++ will always run faster than Python (this will persist throughout every algorithm). We also can observe that Python is slower than CPP. This can be due to the fact that Python is an interpreted language, which means that the python code is processed at runtime by the interpreter. While CPP converts the script to executable machine code before executing it. As for the DCM, it's expected that the DCM will increase with the increasing matrix size. This happens because the bigger the matrix is, more data the processor needs to access. When the data exceeds the capacity of the cache, it needs to be fetched from the main memory, what leads to the increasing of cache misses.

Now the second algorithm **Line Matrix Multiplication** registers better time executions in the tests provided, with the same temporal complexity ($O(n^3)$) has the last algorithm seen **Normal Matrix Multiplication**. This is due to the change in order of calculation of the result matrix. Since now it will make the calculations of one element from the matrix with the entire correspondent row on the other matrix, and store the value. In this order, the program accessed the memory less times to perform the calculations. We were expecting the second algorithm to be more efficient than the first one, but that wasn't the case for the python implementations, where the first method was more efficient. This may have happen because memory management in Python involves a private heap containing all Python objects. The memory of this heap is ensured by the the Python memory manager. As the second algorithm is always accessing the same result matrix indexes, it will always need to search for them in the heap (while the first algorithm only access each result matrix index once). And this can increase the execution time of the program.



Matrix Size	GFLOPS – CPP	GFLOPS – Python
600	2 238 341 968.91	10 369 004.55
1000	1 661 129 568.11	10 946 936.26
1400	1 703 818 689.85	9 565 915.72
1800	639 508 744.997	8 414 531.45
2200	556 612 650.29	4 964 058.90
2600	512 621 585.76	6 459 041.93
3000	468 758 138.16	7 892 887.73

Figure 3: Normal Multiplication $GFLOPS_3$

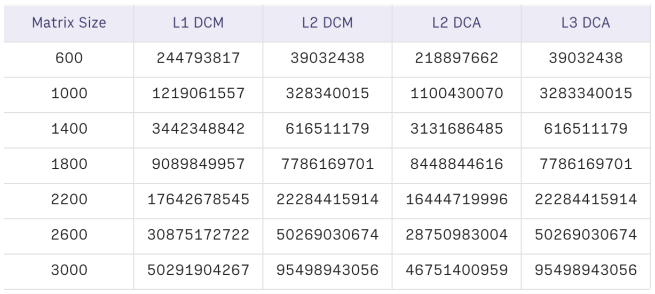


Matrix Size	GFLOPS – CPP	GFLOPS – Python
600	3857142857.14	8062172.79
1000	4464285714.29	7700792.19
1400	3475617479.42	6479340.14
1800	3375000000	8093797.34
2200	3197117549.92	7852326.88
2600	3409505334.63	7577838.63
3000	3385155466.4	4594212.59
4096	3191726933.23	8062172.79
6144	3378833999.61	
8192	3376650311.64	
10240	3361925412.44	

Figure 4: Line Multiplication $GFLOPS_4$

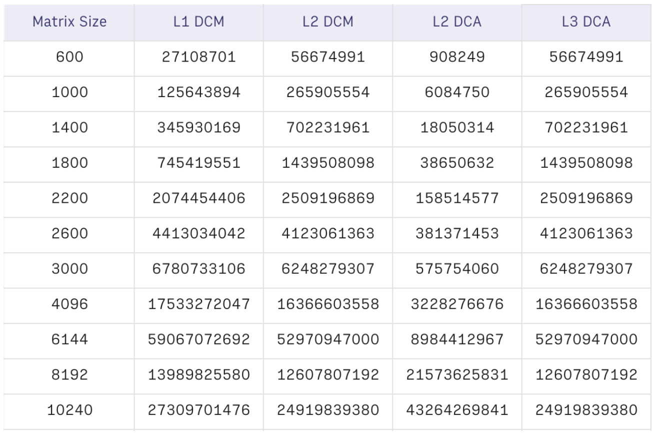
GFLOPS represents the number of floating-point operations a computer can perform in one second ($2 * N^3/T_exec$). With the results provided regarding the GFLOPS in both languages, we can observe that the Python language registered bigger values when compared to the CPP results. As the execution time increases, the GFLOPS will decrease because the total number of operations that the computer can perform in a given time also decrease.

Comparison between the 3 methods



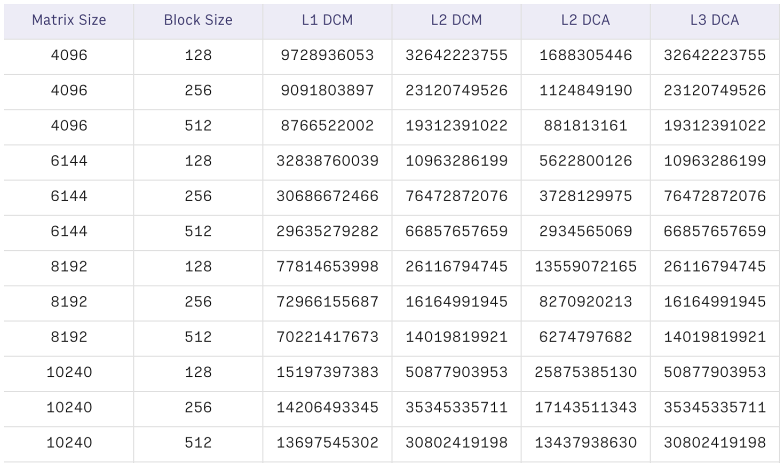
Matrix Size	L1 DCM	L2 DCM	L2 DCA	L3 DCA
600	244793817	39032438	218897662	39032438
1000	1219061557	328340015	1100430070	3283340015
1400	3442348842	616511179	3131686485	616511179
1800	9089849957	7786169701	8448844616	7786169701
2200	17642678545	22284415914	16444719996	22284415914
2600	30875172722	50269030674	28750983004	50269030674
3000	50291904267	95498943056	46751400959	95498943056

Figure 5: Normal Multiplication PAPI parameters$_5$



Matrix Size	L1 DCM	L2 DCM	L2 DCA	L3 DCA
600	27108701	56674991	908249	56674991
1000	125643894	265905554	6084750	265905554
1400	345930169	702231961	18050314	702231961
1800	745419551	1439508098	38650632	1439508098
2200	2074454406	2509196869	158514577	2509196869
2600	4413034042	4123061363	381371453	4123061363
3000	6780733106	6248279307	575754060	6248279307
4096	17533272047	16366603558	3228276676	16366603558
6144	59067072692	52970947000	8984412967	52970947000
8192	13989825580	12607807192	21573625831	12607807192
10240	27309701476	24919839380	43264269841	24919839380

Figure 6: Line Multiplication PAPI parameters$_6$



Matrix Size	Block Size	L1 DCM	L2 DCM	L2 DCA	L3 DCA
4096	128	9728936053	32642223755	1688305446	32642223755
4096	256	9091803897	23120749526	1124849190	23120749526
4096	512	8766522002	19312391022	881813161	19312391022
6144	128	32838760039	10963286199	5622800126	10963286199
6144	256	30686672466	76472872076	3728129975	76472872076
6144	512	29635279282	66857657659	2934565069	66857657659
8192	128	77814653998	26116794745	13559072165	26116794745
8192	256	72966155687	16164991945	8270920213	16164991945
8192	512	70221417673	14019819921	6274797682	14019819921
10240	128	15197397383	50877903953	25875385130	50877903953
10240	256	14206493345	35345335711	17143511343	35345335711
10240	512	13697545302	30802419198	13437938630	30802419198

Figure 7: Block Multiplication PAPI parameters$_7$

Since the matrix is divided into blocks of the same size, we calculate the resultant matrix using the multiplication between 2 blocks. This implementation takes advantage of the cycles order, as so it can reduce the memory calls made. We can see that the values obtained in L1 DCM are bigger than the ones obtained in L2 DCM. This is due to the L2 cache being larger, and slower, than the L1 cache. Therefore, programs are more likely to experience data cache misses in the L2 cache, which results in higher values of L2 DCM.

Observing the results provided, we can conclude that the L2 DCM (data cache misses) is bigger than the L1 DCM. On reason for this to happen is because the L2 cache is larger and slower than the L1 cache, and is more likely to experience cache misses. On the other hand, we can see that the L2 DCA values are larger than the L3 DCA values, indicating that the L2 cache is accessed more frequently.

Conclusion

The use of the best algorithms to perform in a single processor, can have a great impact in the performance and in the execution time. Even more, we can notice the difference between the 2 languages, Python and C++.

While examining the results of the implementations, we concluded that the first algorithm (Normal Matrix Multiplication) is the least efficient in C++, but the most efficient in Python. Actually, C++ run substantially faster when compared to the Python implementations.

Additionally, we observed that the implementations have different execution times depending on the programming language used. As reference before, Python has several levels of memory management that needs to follow before accessing the variables. And this causes a bigger execution time.

With this project, it helped us to understand the impact of memory management in the processor ad to understand the way it affects data caches.

If you would like to access the graphs of the data collected, you can get them in the Annexes section.

Annexes (Graphs of the collected data)

References

1) https://www.honeybadger.io/blog/memory-management-in-python/

2) https://www.simplilearn.com/tutorials/cpp-tutorial/cpp-memory-management

3) https://hazelcast.com/glossary/cache-miss/

4) https://towardsdatascience.com/how-fast-is-c-compared-to-python-978f18f474c7