U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Performance Evaluation of a Single Core Using Matrix Multiplication

Parallel and Distributed Computing 2021/2022

T09G08

Adriano Soares up201904873@fe.up.pt
Filipe Campos up201905609@fe.up.pt
Francisco Cerqueira up201905337@fe.up.pt

1. Problem Description

Throughout this project, we analyzed the consequences of the performance of the memory hierarchy of the processor when we access large portions of data at a given time with the use of the Performance API (PAPI). To simulate the use of a substantial chunk of memory we've implemented multiple algorithms in C and Python to solve an algebraic operation known for its high memory demands: the product of two matrices. We further analyzed the execution times and Level 1 and Level 2 cache misses of each algorithm in the Results and Analysis section of this report.

2. Algorithms Explanation

2.1. Element Multiplication

This algorithm works by applying the dot product to each line of the first matrix with each column of the second matrix. The pseudo-code is:

```
C = matrix[A.rows, B.columns] // Initialized with zeros
for row in [0, C.rows):
    for column in [0, C.columns):
        for k in [0, A.columns):
            C[row][column] += A[row][k] * B[k][column]
```

2.2. Line Multiplication

Similar to the previous algorithm, it iterates through every row of the first matrix but before looping through each value of the column of the second matrix, the algorithm loops through the entire line of the second matrix, multiplying every element by the corresponding line before moving to the next element. The pseudo-code is:

```
C = matrix[A.rows, B.columns] // Initialized with zeros
for row in [0, C.rows):
    for k in [0, A.columns):
        for column in [0, C.columns):
            C[row][column] += A[row][k] * B[k][column]
```

2.3. Block Multiplication

This last algorithm divides the matrices in blocks and uses the same sequence of computation as in the previous algorithm.

```
C = matrix[A.rows, B.columns] // Initialized with zeros
for block_y in [0, blocks_per_row):
	block_y_idx = block_y * block_size * C.columns
	for block_x in [0, blocks_per_column):
		block_x_idx = block_x * block_size
		block_idx = block_y_idx + block_x_idx
		for block_k in [0, blocks_per_row):
			block_A_idx = block_y_idx + block_k * block_size
			block_b_idx = block_k * block_size * C.columns + block_x_idx
			for row in [0, block_size):
				for k in [0, block_size):
					for column in [0, block_size):
					C[block_idx + (i*C.columns+j)] +=
			A[block_A_idx + (i*C.columns+k)] * B[block_B_idx + (k*C.columns+j)]
```

3. Performance Metrics

To test the performance of the multiple algorithms we've used the following performance metrics: the *time (in seconds)* and *L1/L2 data cache misses*.

4. Results and Analysis

4.1. Results

4.1.1. Element Multiplication

	C++				**Python**	
n	**t (s)**	**GFlops**	**L1 Data Cache Misses**	**L2 Data Cache Misses**	**t (s)**	**GFlops**
600	0.258	1.67442	244,773,953	29,322,622	35.183	0.01228
1000	2.128	0.93985	1,130,940,850	136,273,561	191.203	0.01046
1400	6.805	0.80647	3,555,096,753	615,035,881	576.760	0.00952
1800	15.160	0.76939	7,762,680,820	3,342,669,371	1306.704	0.00893
2200	32.602	0.65321	15,134,537,500	11,426,767,223	2480.985	0.00858
2600	65.582	0.53600	26,645,858,942	28,669,986,533	4190.585	0.00839
3000	119.433	0.45214	43,149,027,021	56,431,603,890	6550.317	0.00824

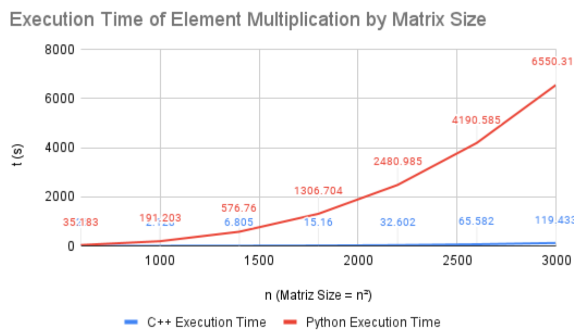
4.1.2. Line Multiplication

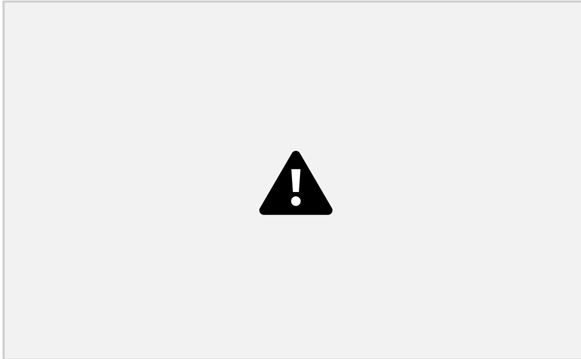
	C++				Python	
n	**t (s)**	**GFlops**	**L1 Data Cache Misses**	**L2 Data Cache Misses**	**t (s)**	**GFlops**
600	0.167	2.58683	27,186,273	30,223,616	41.050	0.010524
1000	0.900	2.22222	127,312,501	184,896,835	189.908	0.010531
1400	3.470	1.58156	394,678,371	504,461,139	523.493	0.010483
1800	7.767	1.50174	931,168,001	1,099,572,588	1117.866	0.010434
2200	14.421	1.47674	2,209,901,234	2,006,783,275	2036.448	0.010457
2600	23.402	1.50209	4,394,128,495	3,250,097,304	3382.683	0.010392
3000	36.459	1.48112	6,773,836,547	4,972,763,371	5294.650	0.010199
4096	99.769	1.37757	17,513,091,865	12,519,345,650		
6144	296.698	1.56340	59,148,590,680	42,037,363,292		
8192	574.727	1.91310	140,126,383,189	98,948,004,971		
10240	1072.302	2.00269	273,598,959,749	195,059,329,533		

4.1.3. Block Multiplication

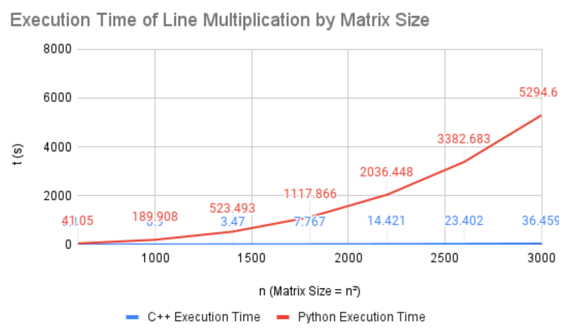
n	**block size**	**t (s)**	**GFlops**	**L1 Data Cache Misses**	**L2 Data Cache Misses**
4096	128	61.153	2.24746	9,691,212,983	17,662,146,027
	256	56.416	2.43617	9,102,172,943	13,276,842,928
	512	57.474	2.39132	8,857,472,547	10,172,218,117
6144	128	204.489	2.26837	32,725,668,530	60,138,407,543
	256	189.711	2.44507	30,725,990,968	44,913,026,892
	512	205.404	2.25826	29,954,348,576	36,899,310,576
8192	128	643.252	1.70930	78,063,759,562	145,675,644,840
	256	591.940	1.85748	73,380,999,289	119,643,036,552
	512	576.505	1.90720	71,003,386,677	103,179,843,333
10240	128	953.944	2.25116	151,510,491,393	275,907,467,427
	256	1034.153	2.07656	142,785,021,472	219,120,380,524
	512	1085.102	1.97906	139,105,698,984	202,753,921,169

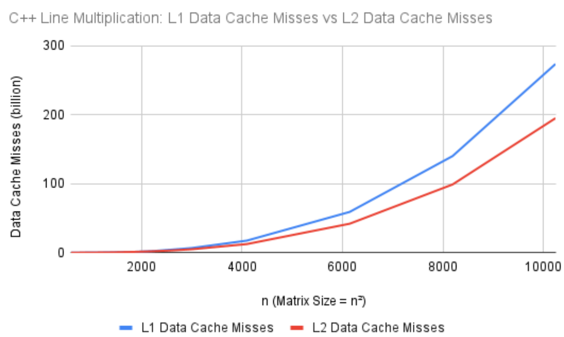
4.2 Analysis





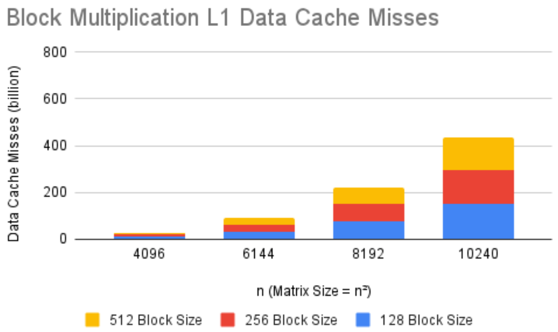
As we can observe, the C++ implementations of all algorithms are much more efficient than our Python counterparts, mostly due to it being an interpreted language rather than a compiled one. The implementation of the Element Multiplication algorithm proved to be the least efficient, due to the large amount of cache misses. This happens since it needs to access values that are spaced apart in memory when accessing the column values of the second matrix. When a new value is needed a block of contiguous memory containing the value will be copied to cache. Each block with a determined size (usually 32, 64 or 128 byte[1]). The larger the matrix, the lower the possibility of the next element of the column being in that block of memory, resulting in cache misses. We should note that the number of L2 data cache misses was higher than L1 data cache misses for larger matrices, we considered that as an anomaly and tried to explain it in the ***Note*** present below.

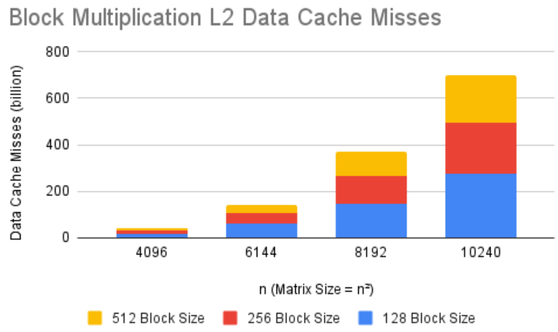


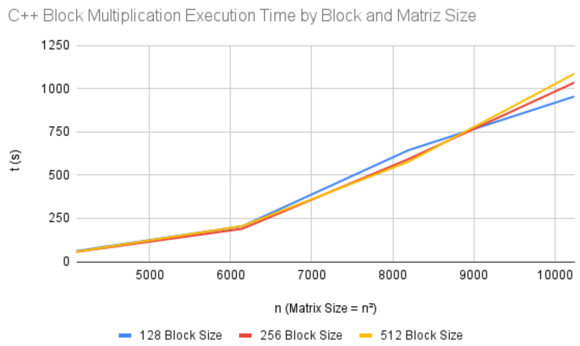


As the graphics above demonstrate, the Line Multiplication algorithm proved to have a greater efficiency than the previous algorithm. This happens because the values of the seconds matrix in memory are now accessed consecutively, leading to a better usage of the cache memory as more values present at the cache will be used, resulting in less cache misses and a higher performance.

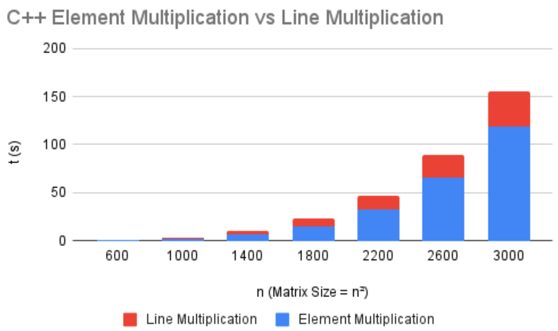
[1] http://www.nic.uoregon.edu/~khuck/ts/acumem-report/manual_html/ch03s02.html

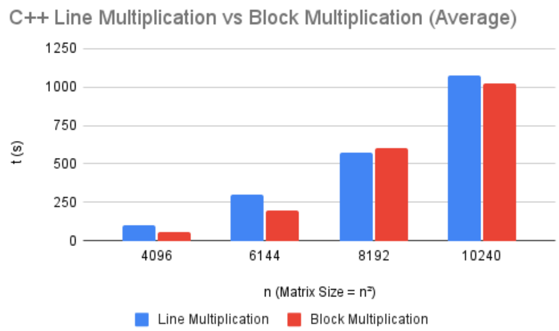


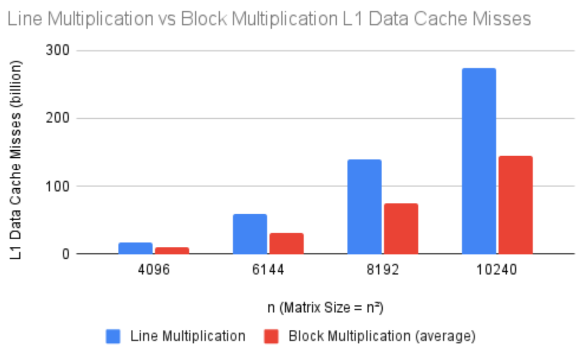




To further reduce the number of cache misses the last implementation divides the matrix into blocks of lower size so that the cache can reference a whole block at a time. This results in fewer cache misses and an average lower execution time when compared to all previous algorithms. To illustrate these results:







Note: We took notice that PAPI derived the L2_DCM counter with LLC_REFERENCES and L2_RQSTS:CODE_RD_MISS counters. LLC_REFERENCES is responsible for counting the amount of instructions and data references to the last level cache, in our case, L3 cache, excluding cache line fills due to hardware prefetch. L2_RQSTS:CODE_RD_MISS counts the amount of L2 cache misses when fetching instructions. With that in mind, and with our extensive research, we advise the reader to adopt a skeptical attitude towards L2 data cache misses values.

5. Conclusion

In this project we had the opportunity to measure the performance of a single core, using matrix multiplication techniques to compare and evaluate different behaviors that we concluded to be caused by the memory management, mainly by the cache memory. We've managed to complete all project objectives.