Faculdade de Engenharia da Universidade do Porto

Parallel and Distributed Computing

Performance evaluation of a single core

Group T07G12

António Oliveira Santos – 202008004
Pedro Alexandre Ferreira e Silva – 202004985
Pedro Miguel Magalhães Nunes – 202004714

Porto, 10th March 2023

Index

Problem Description

The goal of this project is to study the effect on the processor performance of the memory hierarchy when accessing large amounts of data. The problem used for this study was the product of two same size matrices. In order to collect relevant performance indicators of the program execution, besides execution time, we used the Performance API (PAPI).

Algorithms Explanation

Column Multiplication

Our first approach was a column oriented matrix multiplication algorithm. One line of the first matrix is multiplied by each column of the second matrix. This is a naive algorithm, similar to a more direct manual solution. A C/C++ implementation was already provided by the teaching staff. We then implemented the algorithm in PyPy.

Line Multiplication

Another approach was row oriented matrix multiplication. An element from the first matrix is multiplied by the correspondent line of the second matrix. Changing the order of operations can improve on the performance of the previous algorithm. Matrix elements on the same row are stored in memory consecutively, while elements in the same column are not. Considering this, this is a more cache efficient approach, since the inner loop iterates over a single line of a matrix, therefore data cache misses are expected to be lower. This algorithm was implemented in both C/C++ and PyPy.

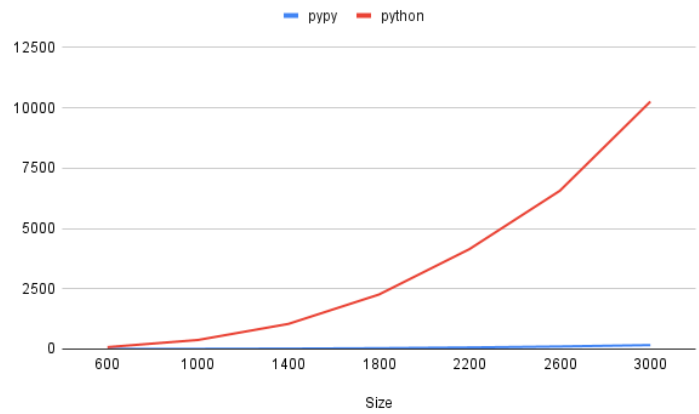
Block Multiplication

Lastly, we followed a block oriented algorithm. If the matrices are large enough, a single line might not fit in cache, which can still lead to a relevant amount of data cache misses. To avoid this, the matrices are divided in blocks and the same sequence of computation as the last algorithm (Line Multiplication) is used. The multiplication is done at a block level, at first. It’s important that the block sizes are small enough that a single line may fit in cache, but large enough that there aren’t many avoidable line changes, resulting in cache misses. We implemented this algorithm in C/C++.

Performance Metrics

To evaluate the performance of each algorithm we used the execution time as well as the data cache misses of the L1 and L2 cache memory, number of cycles and number of instructions, obtained using the performance API (PAPI). These metrics were obtained for different matrix sizes.

Regarding the Python implementations, we started by running our tests with both the traditional Python interpreter and PyPy, an alternative python implementation, which uses a just-in-time compiler (JIT). As expected, we were able to run the program much faster with PyPy, reducing the time necessary for a test run from close to 14 hours to slightly above 2 hours. Considering this, we decided to use PyPy to register metrics. Execution time was the only metric measurable for this language.



Average time of column and line multiplication for 2 python implementations

The approach to gather results was the creation of two bash scripts that ran our algorithms in both languages and saved their results in .txt files or in a .csv format. The final PyPy results that will be subject of study were taken over the course of two sessions both lasting approximately 2 hours where we executed, in each one, all of the different algorithms and sizes combinations 10 times, making up a total of 20 results. All of the C++ values were obtained in a single run lasting approximately 14 hours where we registered once more all of the necessary combinations 8 times, meaning that for each algorithm - matrix size/block size combination - there are only 8 results, this will be discussed further ahead.

In order to avoid differences in the results of the algorithms, related to external factors, we ran all tests in the same machine, ensuring at all times that there were as little processes interfering with our evaluation as possible, while keeping it plugged into a power outlet to avoid CPU battery management interruptions. The machine's specifications are as follows:

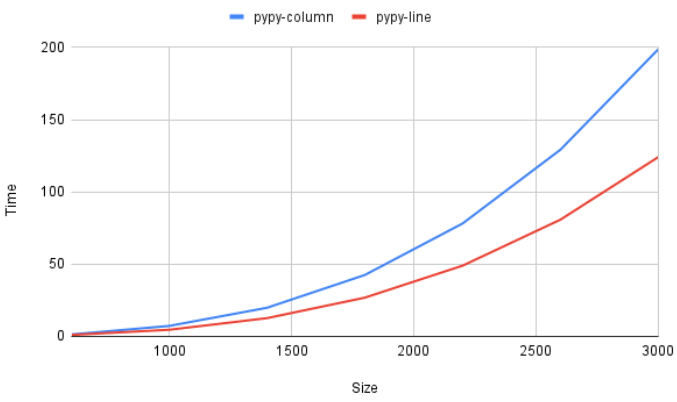
OS	EndeavourOS Linux x86_64 running 6.2.2-arch1-1 kernel
CPU	Intel I7-10510U (8) @ 1.8000GHz
Memory	12GB RAM

Results and Analysis

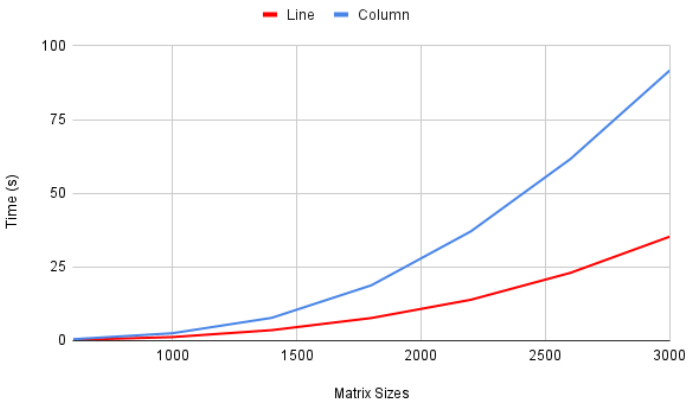
Algorithm Comparison

To directly compare these algorithms, the execution time of each one, for both implementations, were plotted over an increasing matrix size. There is a clear improvement when comparing the first two algorithms, this improvement is clearly related to the way that we had previously postulated the differences in these algorithms, due to the way cache memory is organized and loaded, the CPU has much less need to constantly attempt to fetch more information directly from the registers and instead is able to get a “hit” on the cache registers more often in the “line” algorithm, as it will need the directly next value in the array of the current line that is already cached and quickly available.

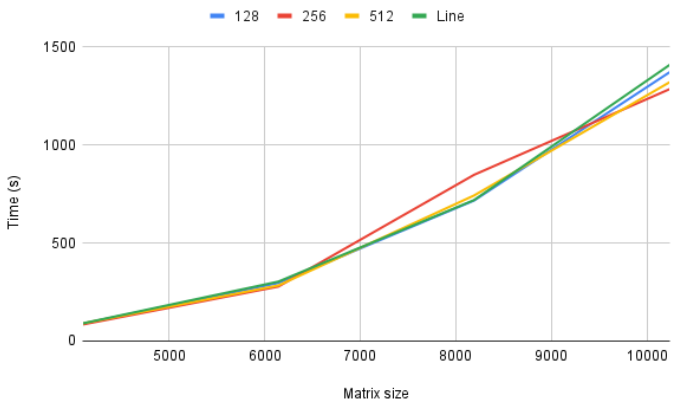
The block algorithm was slightly faster than the line algorithm, when looking at different block sizes, this will be explored further ahead.



Line vs Column algorithm time comparison (PyPy)



Line vs Column algorithm time comparison (C/C++)



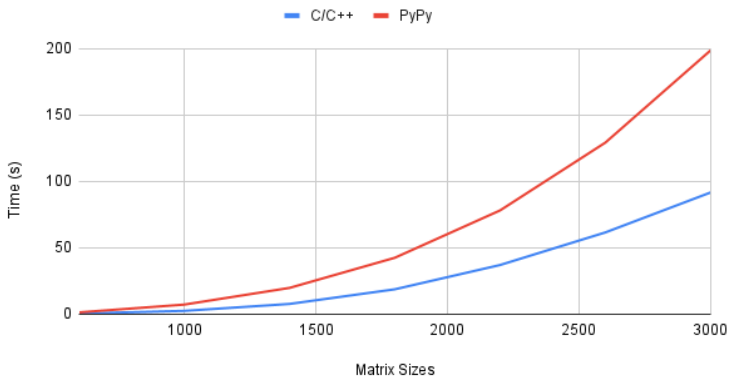
Line vs Block (3 block sizes) algorithm time comparison (C/C++)

Block Size Comparison

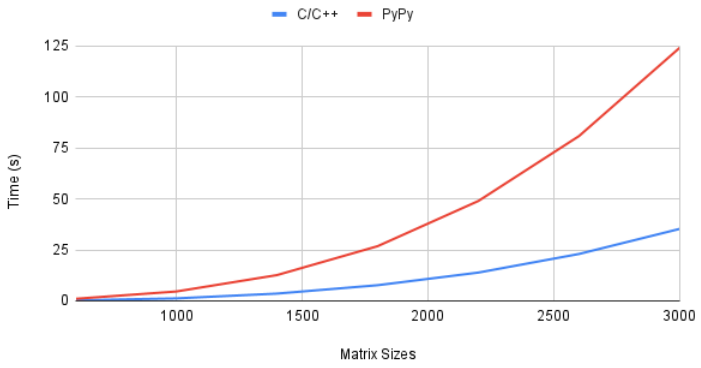
As observed in the previous plot graphic, the time differences for blocks of sizes 128, 256 and 512 were slight. We were unable to draw any clear conclusions from this data, although we can see a modest speed up with block size 256 for large matrices. It is important to take into account that due to the small sample size (only 8), results can be skewed and not entirely representative. In addition, we also encountered two instances where PAPI's returned values for the number of instructions were negative, this can likely be due to a buffer overflow of its counter, as such we are only able to contemplate 7 results for matrix size 10240 with block size 128 and 512.

PyPy and C/C++ Comparison

The execution times of the two algorithms implemented in both languages were plotted, in order to directly compare the performance in different programming languages. C/C++ showed better performance in both algorithms, particularly as the matrix size increased, registering an approximate speed up of 58%.



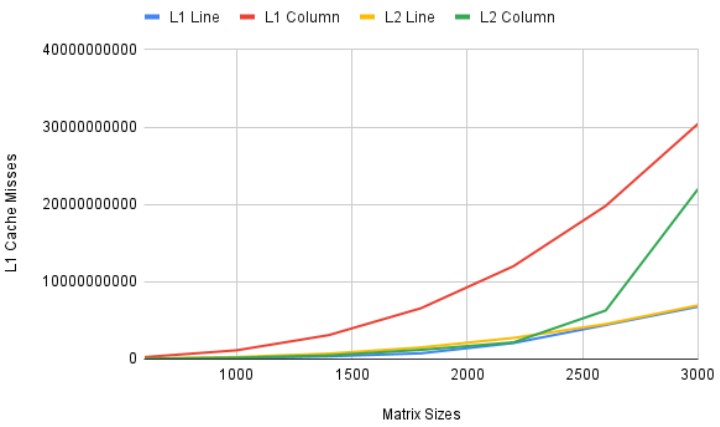
Column algorithm comparison between languages



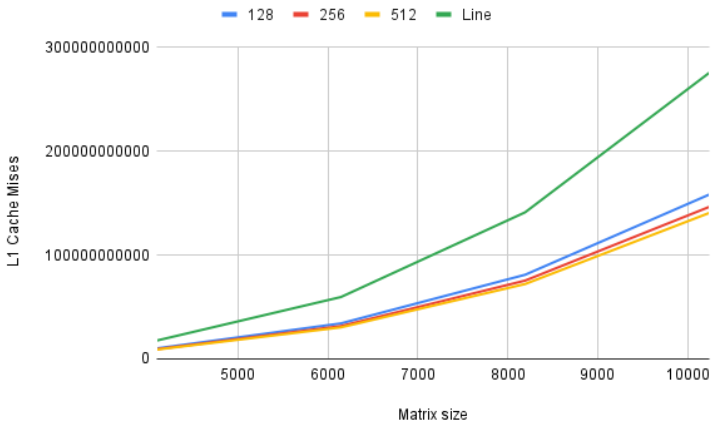
Line algorithm comparison between languages

Cache Performance Comparison

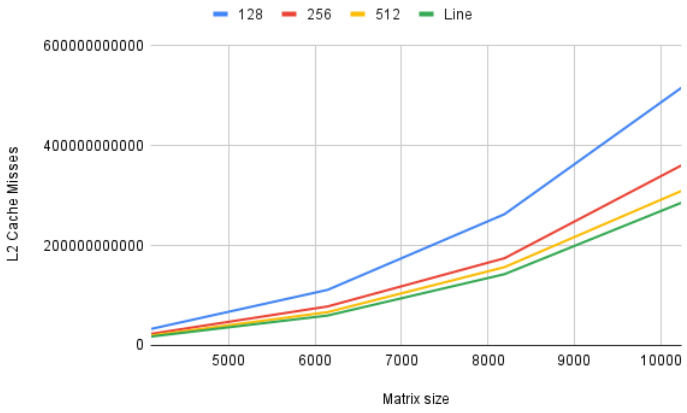
In order to verify the impact of the optimizations in cache usage, the L1 and L2 cache misses were plotted for all 3 algorithms. Again, the block algorithm was plotted for 3 different block sizes. We observed that the number of cache misses significantly decreases for both L1 and L2 when using the line algorithm compared to when using the naive column algorithm, for the motives previously mentioned. As for the block multiplication algorithm, the number of L1 cache misses is reduced when compared to line multiplication, however the opposite is observed when it comes to L2 cache.



Line vs Column L1 and L2 cache misses comparison



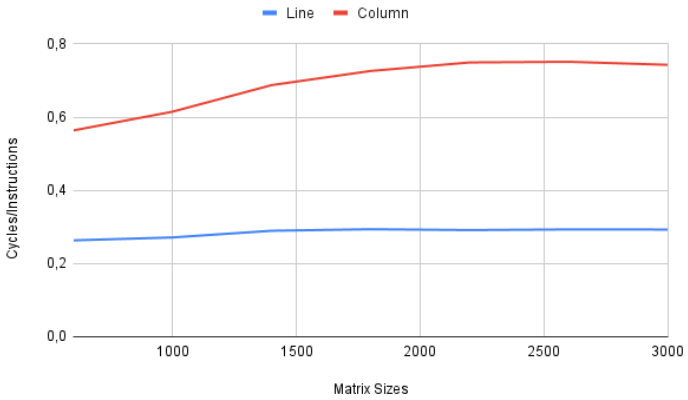
Line vs Block (3 block sizes) L1 cache misses comparison



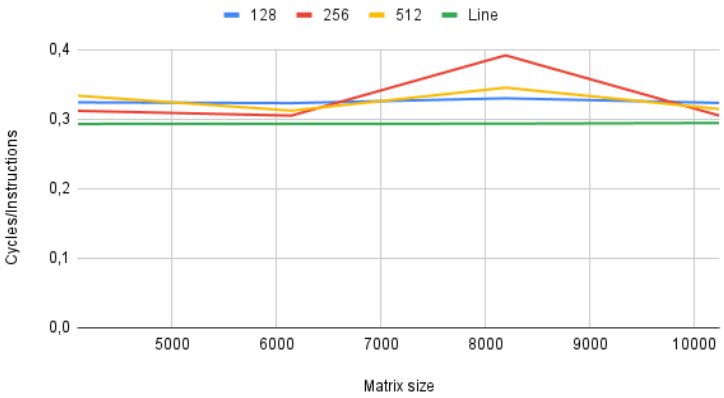
Line vs Block (3 block sizes) L2 cache misses comparison

Cycles/Instruction Comparison

At last, we compared all algorithms regarding the average number of cycles per instruction, plotting these metrics for all 3, with 3 different block sizes for block multiplication. Again, as expected, there is a clear improvement when going from the original column oriented algorithm to the line algorithm, with instructions of the line algorithm taking less cycles to be executed, reinforcing the superiority of this algorithm. When it comes to block multiplication the values are very similar but we are able to conclude that there isn't an improvement, there is however a bump in CPI around matrix size 8000, this phenomenon is not expected and could be due to the small sample size of our study as well as the previously reported incident regarding the registered number of instructions.



Line vs Column cycles/instruction comparison



Line vs Block (3 block sizes) cycles/instruction comparison

Conclusions

From the metrics analysed we can see that cache-aware algorithms can lead to a significant improvement in the processor's performance, particularly regarding tasks making use of large amounts of data. Taking advantage of the cache memory's organization allows usage of the much faster access times provided by this type of memory, decreasing the overall run time of these tasks.

Moreover, we can see a clear discrepancy between the algorithm run times using PyPy and C++. Since PyPy uses just-in-time compilation (compiling the code while it is executed) and C++ is a compiled language, it is only natural that the latter is much faster at computing the aforementioned algorithms, especially when the sizes of the matrices are considerably high. Using PyPy, column multiplication lasts more than 2 times longer than C++, and more than 4 times longer in the line multiplication.

Annex

Block size	Matrix Size	Time (s)	L1 Cache Misses	L2 Cache Misses	Cycles	Instructions
128	4096	88,08995338	10127882875	32879184964	158085605451	487685257582
	6144	296,2197644	34201219551	111363498020	531680868856	1645786508613
	8192	717,1679314	81059398929	262994896010	1287305978519	3900944430062
	10240	1372,589861	158340210576	516241706296	2463350896692	7618822194303
256	4096	84,21496963	9378797228	23086121279	151112565442	484409855889
	6144	277,9133599	31637093090	78124998547	498798688202	1634732037423
	8192	848,2239346	75455278204	174770570177	1518119793352	3874741303492
	10240	1285,921399	146454091861	360712201495	2308248889141	7567644119156
512	4096	89,79648963	8995920557	19698491687	161128050098	482785787024
	6144	283,467272	30376393090	66756439547	508726110833	1629250803935
	8192	742,8109269	72119516824	156882789722	1333309953523	3861748718155
	10240	1321,444836	140624924542	309594943726	2371925515430	7542268050283

C++ Block Algorithm Results

Matrix Size	Time (s)	L1 Cache Misses	L2 Cache Misses	Cycles	Instructions
600	0,256571875	27117946,38	55804203,63	456932647,1	1735259017
1000	1,216042375	125634685,6	258321443,9	2175521277	8020073053
1400	3,564278	345969786,8	705879212,5	6367035744	21991288420
1800	7,67857025	743524933,9	1506369229	13725326828	46720906048
2200	13,87590413	2071121684	2732302616	24854090077	85280925141
2600	23,00220513	4414347154	4525617486	41265259425	140743344930
3000	35,30664725	6777964802	6949035570	63322854276	216180165931
4096	89,89761775	17678757835	17692856975	161278161052	550091594027
6144	303,243666	59606564355	59916830607	544288085111	1856181255812
8192	719,3400706	141208750845	142816116926	1291272905629	4399389348667
10240	1409,788523	275645480455	285927143730	2530795157063	8592032754273

C++ Line Algorithm Results

Matrix Size	Traditional	Line
600	1,379616239	1,01382184
1000	7,240278938	4,592338714
1400	19,69846279	12,62351043
1800	42,47950281	26,81930009
2200	78,18875488	48,9771823
2600	129,304029	80,8231862
3000	198,8443023	124,1323263

PyPy Algorithm Results

Matrix Size	Time (s)	L1 Cache Misses	L2 Cache Misses	Cycles	Instructions
600	0,48505775	244523075,5	39188083,38	855620465,6	1518180029
1000	2,50817925	1129359450	175867559	4314047297	7017072434
1400	7,744207875	3096639328	469860622,9	13227388044	19241407390
1800	18,77073938	6579232753	1190971118	29683923442	40879185761
2200	37,07437413	12003241949	2156071707	55925481261	74618404906
2600	61,60865788	19807053381	6281990405	92512782970	123147065989
3000	91,72098238	30421709940	21992909941	140530008924	189153168461

C++ Traditional Algorithm Results