2022/23

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Performance evaluation of a single core

Project 1

Parallel and Distributed Computing

Group 12 – Class 8

Bachelor of Informatics and Computing Engineering

Lia da Silva Vieira	up202005042@fe.up.pt
Marco André Rocha	up202004891@fe.up.pt
Ricardo André Matos	up202007962@fe.up.pt

Index

1. Problem description

This project aims to analyze the memory hierarchy's impact, when accessing large amounts of data, on the performance of the processor. To fulfill this goal, we had to induce high-demand memory and computational operations, so we settled for large matrix product calculations. Therefore, we focused our study efforts on the implementation of three different iterative matrix multiplication algorithms in both C/C++ and Go and, with the use of *Performance API*, we were able to collect these algorithms' performance and cache utilization metrics (namely cache misses), inferring their overall scalability.

2. Algorithms Explanation

2.1 Row-Oriented Algorithm (I J K)

This algorithm comes from applying the straight matrix multiplication formula in a naive way. Hence, in a *l x m* matrix A and a *m x n* matrix B we iterate over the lines, *l*, of A making the dot product with the columns, *n*, of B. Needless to say, this has a huge impact on performance. To make a single line of the result matrix C (*l x n*) we have to iterate over all of matrix B. This is not a problem alone but as the size of the matrices increases, later elements will replace earlier elements making it necessary to reload the matrix to the cache on every iteration. We will discuss that in a more detailed way later in the report.

The above algorithm is O(n^3) in time complexity. In order to make the dot product we have to iterate over every element, *m,* of the columns in the line i of A and row j of B, giving a *l x n x m* performance.

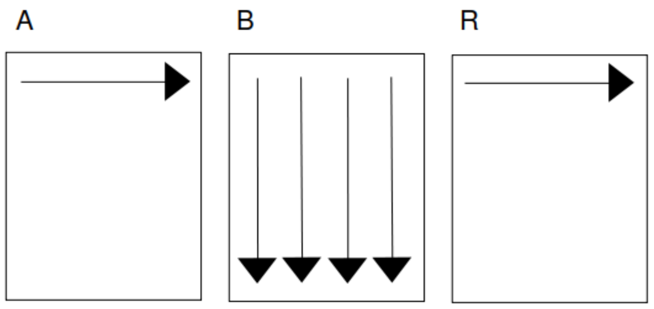


Fig. 1 - Visualization of Row-Oriented Algorithm

2.2 Line-Oriented Algorithm (I K J)

The previous algorithm had yet another major problem. The matrices' elements are stored in cache in a row-major order. Hence, iterating over the rows is not a great idea. In order to take advantage of the efficiency of the cache we should multiply each line from A with a line from B. We can do that by slowing down the iteration that would calculate the dot product, described above. Now, as we first iterate over the rows, we will have vastly decreased cache misses.

The time complexity is the same. To slow down the dot product calculation we just have to swap the two inner loops.

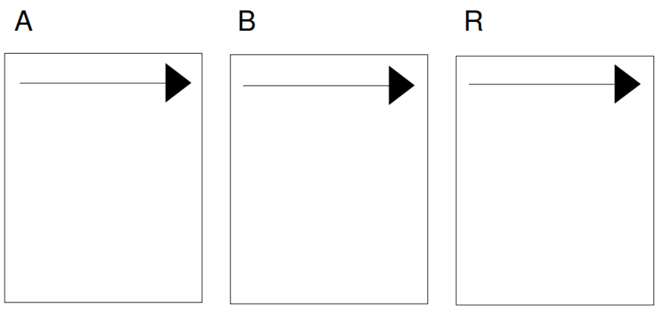


Fig. 2 - Visualization of Line-Oriented Algorithm

2.3 Block-Oriented Algorithm

What if the lines don't fit in the cache? Obviously that would have a great impact on performance, and so we get back to the row-oriented problem. As a line is iterated, earlier elements will be replaced by later elements, resulting in a staggering increase in cache misses.

The solution is simple: we should divide the matrices into equally sized blocks and perform the calculations on the block level, using the line-oriented algorithm.

In order for this to be fully harnessed, the block sizes must be small enough so that a single row fits in cache. But not the smallest, as that could bring an overhead to the performance of the algorithm.

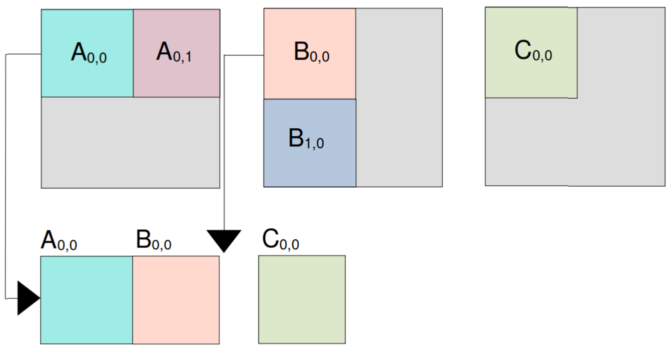


Fig. 3 - Visualization of Block-Oriented Algorithm

3. Performance metrics

To test the performance of the processor in the given problem, we compared the results in two different programming languages, C++ and Go, and with different sized matrices. We measured some relevant performance indicators such as execution time and data cache misses in the Level 1 and Level 2 caches, using the Performance Application Programming Interface (PAPI). L1 and L2 cache misses were monitored since they give useful information on how cached memory is being used.

The CPU used for benchmarking has a 256 Kilobyte L1 cache, a 2MiB L2 cache and a 3MiB L3 cache. That means the L1 cache can hold up to 32, 768 matrix elements, as an element is a double precision floating point that takes 8 Bytes of memory. Hence, 181 x 181 is the maximum matrix size that can possibly fit in the L1 Cache.

We also planned on calculating the GFLOPS. However the PAPI's parameter for calculating the PAPI_FP_INS (Floating point instructions) was not available so we resorted to using the following formula:

$$FLOPS = 2 * \frac{matrix\ size^{3}}{CPU\ Time}$$

4. Results and analysis

4.1 Time performance analysis

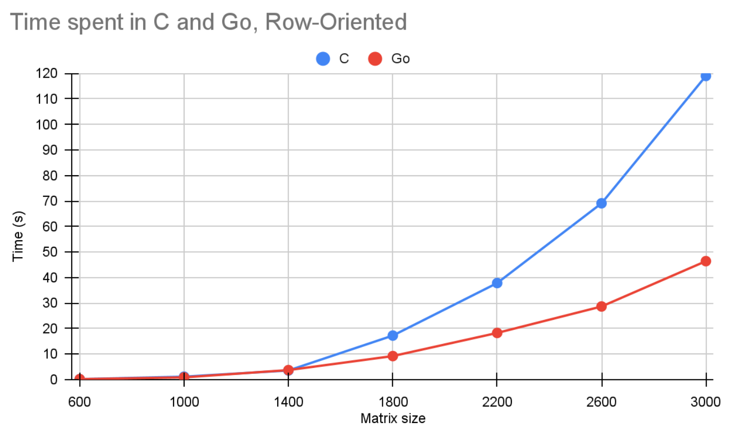


Fig. 4 - Row oriented time graph

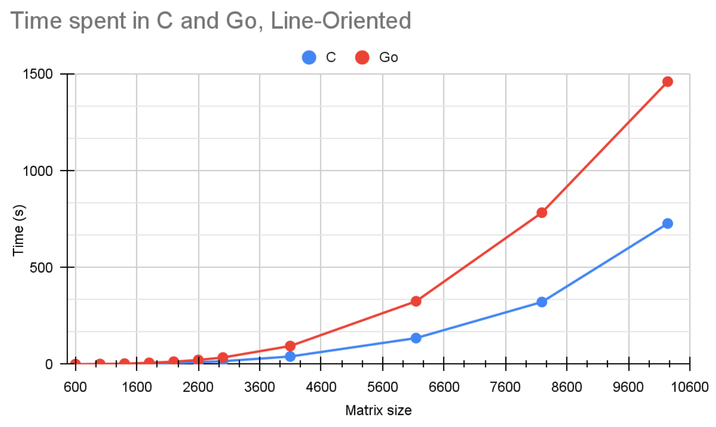


Fig. 5 - Line Oriented time graph

4.1.1 Algorithm comparison

From the graphs it is clear the time evolution that was described in section 2. The row-oriented algorithm has by far the worst performance of them all. The optimization done for the line-oriented algorithm represents a huge improvement to the latter. As we talked, iterating over the row reduces the number of cache misses. Cache misses in lower-level caches, such as L1, have a greater impact on performance because they require accessing slower memory locations. Block sized oriented algorithms represent an even better improvement in performance. Not only reduces the cache misses and execution time but also keeps the GFLOPS stable over the increasingly block sizes. We will discuss that later.

As it may be noticeable, C++ has considerably better time performance than Go in all but one case, the row-oriented algorithm. This happens due to the fact that Go has a set of features that offer automatic dynamic memory management. One of them is the slice traversal feature, which in a nutshell optimizes the "for range" loop to iterate over slices. With that, data can be kept together in cache, improving performance.

4.1.2 Blocks Size Impact on time performance

It is expected that different block sizes have a different effect on performance. When we have bigger matrix sizes, we get an overhead to the performance if we subdivide them in smaller blocks. As we can see in figure 3 for matrix sizes of 10240x10240, smaller blocks will harm the performance of the multiplication.

We can also see the opposite effect, for matrix sizes of 8192x8192, even though cache misses are lower, that does not make bigger blocks sizes have a greater impact on performance than smaller blocks sizes, which is surprising. Furthermore, we should aim to have blocks whose rows are large enough to fit in memory but do not represent an overhead to the corresponding matrix size.

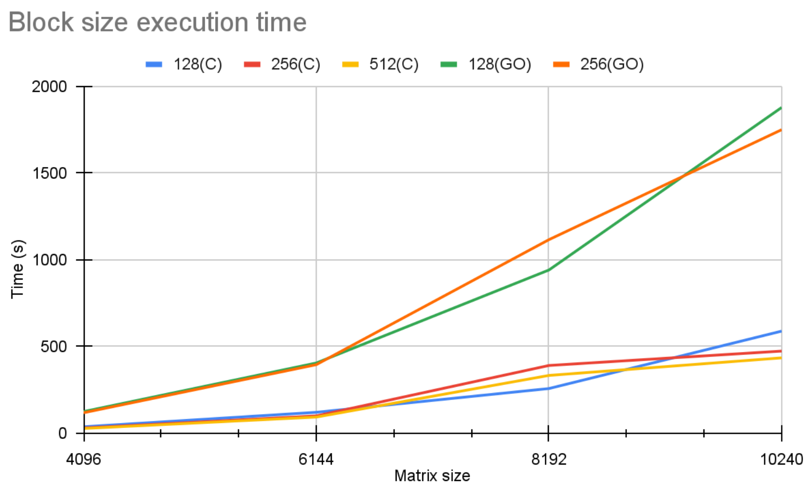
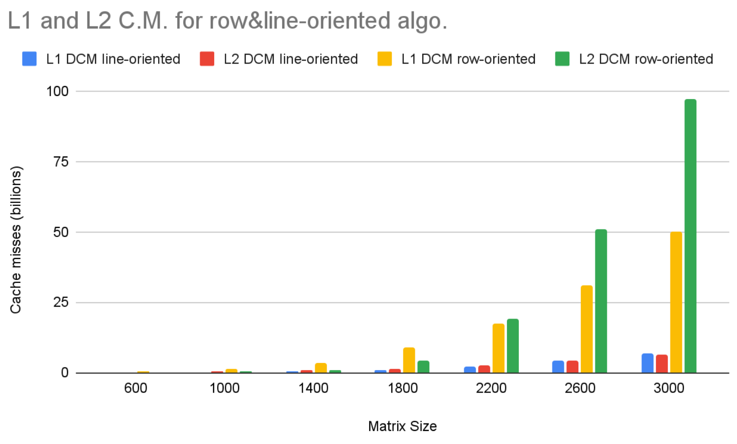


Fig. 6 - Block time graph

4.2 Cache performance comparison

As mentioned above, the big difference between the algorithms is the way they make use of the cache. As we can see in the graphic below, the number of cache misses is according to what we expected in section 2. The naive algorithm described in 2.1 is by far the worst algorithm and the improvement made by the line-oriented algorithm is noticeable. Furthermore, the block oriented algorithm represents an improvement on the latter one.



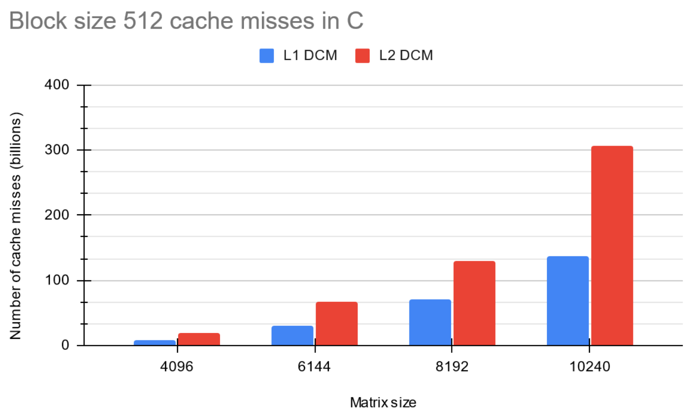


Fig. 7 - Cache misses in 2.1 and 2.2 algorithms

Fig. 8 - Cache misses for block-oriented algorithm

It's also important to note that the L2 cache miss counter is flagged as "derived" in PAPI, having a substantial effect on the reliability of the results obtained, thus being advised to be looked at with skepticism, while focusing on the results obtained from L1 DCM counter.

4.3 GFLOPS graph evolution

The efficiency of the languages and the algorithms they implement, can be measured with quality via the average FLOPS achieved by C++ and Go in all algorithms. These calculations were made according to the formula presented in section 3 and then we plotted the GFLOPS graph for both programming languages as it can be seen below:

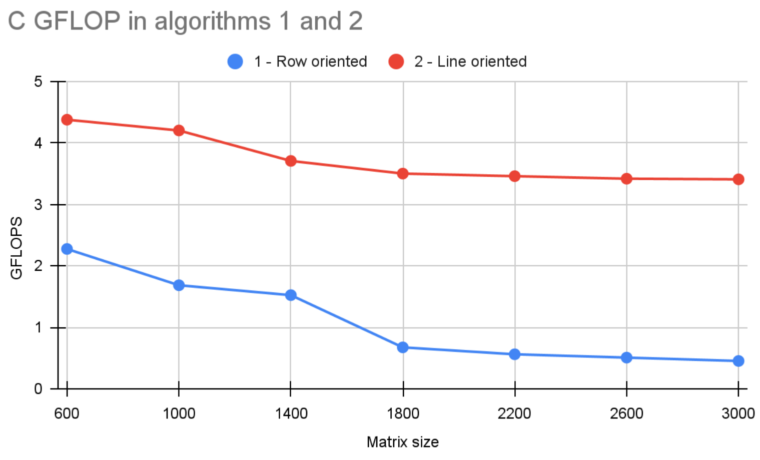


Fig. 9 - C GFLOPS for algorithms 1 and 2

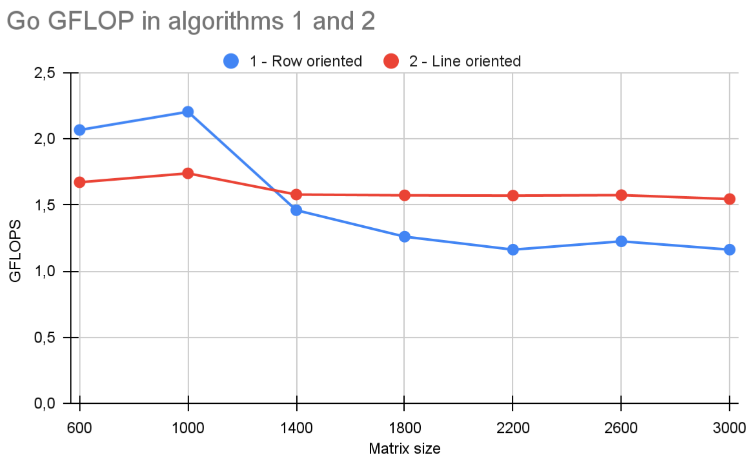


Fig. 10 - Go GFLOPS for algorithms 1 and 2

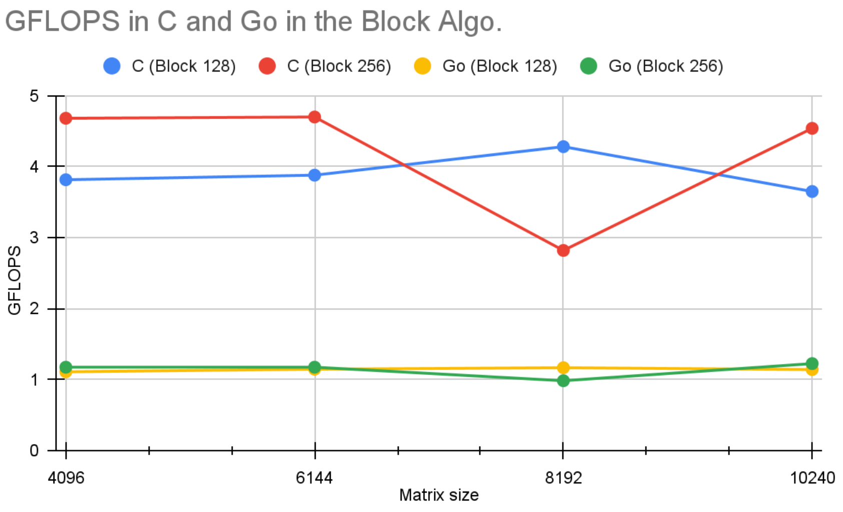


Fig. 11 - GFLOPS graph in both C and Go for different matrix block sizes

It is evident that C++ exhibits and maintains much higher GFLOPS on all three algorithms but the first where, as previously stated, Go does some compiling optimizations for better memory management and manages to beat C. It should be noted that the GFLOPS C provides a clear improvement as we go from the first to the third algorithm, due to the strategy used being progressively more efficient and refined.

There is also a noticeable dropout in the GFLOPS as the matrix sizes increase in the non-block oriented algorithms. The block oriented algorithm is capable of maintaining high GFLOPS and its not noticeable dropout. If we look at the graphic, for small matrices there are similar GFLOPS values between the line-oriented algorithm and the block algorithm , however for bigger matrix sizes the difference between the block oriented algorithm and the line-oriented starts to be noticeable, this is obviously related to the increase in cache misses.

5. Conclusions

After conducting this study, we gained a clearer comprehension of the important metrics to consider when assessing program performance. It is not just the time complexity that matters, but also how we take advantage of memory layout and location. Algorithms can be improved and their performance increased by knowing the internal operations of the processor and the temporal and spatial location of the cache.

If one conclusion should be retained from this entire project, is that the memory access is the bottleneck for the processor performance.

This project also allowed us to compare the impact of the memory hierarchy in two different programming languages, C++ and Go, as well as their performance.
Overall, our assignment helped us understand how the cache stores data and the impact of memory management on code efficiency.

To retain this report's brevity, we only included the graphs deemed essential and with statistical relevance, opting not to include many other graphs used in our in-depth study and that helped us reach the conclusions presented all throughout this document. To have a closer look at the data tables used for the graphs here presented as well as many other graphs, please consult this Google Sheets link.