2022/23

Relatório 1º do Projeto 1 de CPD

Performance evaluation of a single core

Computação Paralela e Distribuída

Licenciatura de Engenharia Informática e Computação

Eduardo Ramos - up201906732
Sofia Reis - up201905450
Mónica Araújo - up202005209

Problem description and algorithms explanation

Problem description:

Study the effect on the processor performance of the memory hierarchy when accessing large amounts of data. For visualization of the performance we use the product of two matrices in 3 different ways. The first way is the more simple and unoptimized way in which we multiply one line of the first matrix by each column of the second matrix. The second way we optimize a little bit more is by multiplying an element from the first matrix by the corresponding line of the second matrix. The third way is the most time and space efficient algorithm in this study. We implement a block oriented algorithm that divides the matrices in blocks and uses the same sequence of computation as in the second way.

Algorithms explanation:

In this project we employed 3 different algorithms that achieve the same goal, which is matrix multiplication. The difference in these algorithms stem from the way we multiply the different values to obtain the final matrix.

In the first algorithm we take the common algebraic approach to obtain the desired value. We take the dot operation for every line ‘i’ and column ‘j’ to obtain the element situated in line ‘i’ and column ‘j’ of the resulting matrix. In pseudo code it looks like this:

```
for i in range 0 to m - 1, do
    for j in range 0 to q – 1, do
      for k in range 0 to p, do
        C[i, j] += A[i, k] * A[k, j]
      done
    done
  done
```

As we can see, we use 3 “for’s” to accomplish our task. The first one we use to get the rows of the first matrix, and similarly, the second is used to get the second matrix’s columns. The final for gets the individual values and multiplies them and sums the results getting the value of the final element. This is the method most commonly used in algebra to accomplish matrix multiplication. But because it leaves much to be desired for a computer in terms of performance. As we will see next, there are ways in which we can speed up this process by catering the algorithms to make use of computer architecture that this algorithm does not respect.

In the second algorithm we employ a similar method to the first one but we change the order of the “fors”. Instead of iterating ‘k’ last, if we instead iterate it second we see that the matrix multiplication sees a great speed up. This algorithm, more commonly called “Line Multiplication”, starts to cater to the cache. In the previous algorithm the cache had a lot more misses because we were multiplying a column. Even tho we use a 2d array to store these values, the cache stores them in a simple array. This means that to get to the next value of the column it has to iterate through an entire line which adds significant overhead. With this new method we instead get the values line by line, which means the cache can just iterate a single time to get a new value. In large operations this difference can really add up. In pseudo code the algorithm looks like this:

```
for i in range 0 to m - 1, do
    for k in range 0 to p, do
        for j in range 0 to q – 1, do
            C[i, j] += A[i, k] * A[k, j]
        done
    done
done
```

As we can see the only change is the order of the “fors”. When the algorithm takes into account the computer architecture, the performance is greatly improved. In the third implementation we will see that we can improve further, however we will see that it is not always the way to go.

Finally, with the third algorithm we intend to change the base algorithm even more to the way the cache works. This algorithm is called “Block Matrix Multiplication” and subdivides the 2 matrices to smaller blocks and multiplies them to obtain the final result. This implementation reduces the values the cache has to look up in a “single breath”, instead it can look up a smaller amount of values that are very close in proximity. However this implementation introduces some additional overheads, we now will have 6 “fors” instead of 3, and if the block size is not compatible with the cache and the matrix sizes it can lead to a decrease in performance compared to line multiplication. So for this implementation it is very important to choose the best block size possible. Below is the pseudo code of the block multiplication:

```
for i_block from 1 to n/block_size:
    for j_block from 1 to p/block_size:
        for k_block from 1 to m/block_size:
            for i from (i_block-1)*block_size to i_block*block_size-1:
                for j from (j_block-1)*block_size to j_block*block_size-1:
                    for k from (k_block-1)*block_size to k_block*block_size-1:
                        C_blocks[i_block,j_block] += A_blocks[i_block,k_block] *
B_blocks[k_block,j_block]
                    done
                done
            done
        done
    done
done
```

As we can see this adds significant overhead so the block size must be optimized to the cache and the operation must be large. otherwise the line multiplication method performs better than this.

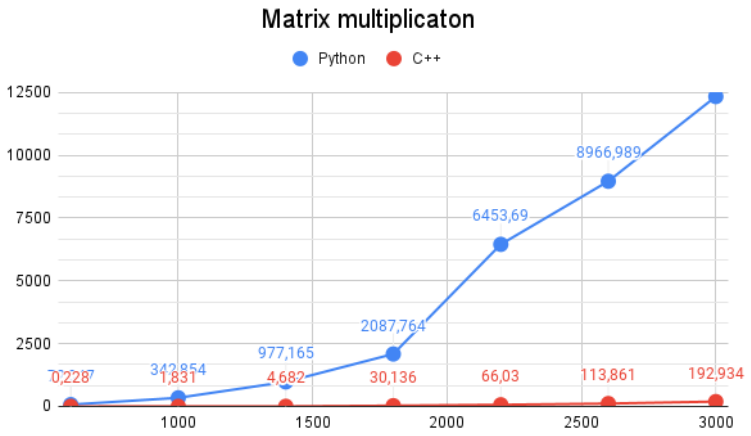
Performance metrics

For this project we decided to use two languages to compare the performance of the algorithms. The algorithm came implemented in C++ and we also used Python to implement and compare the different times.

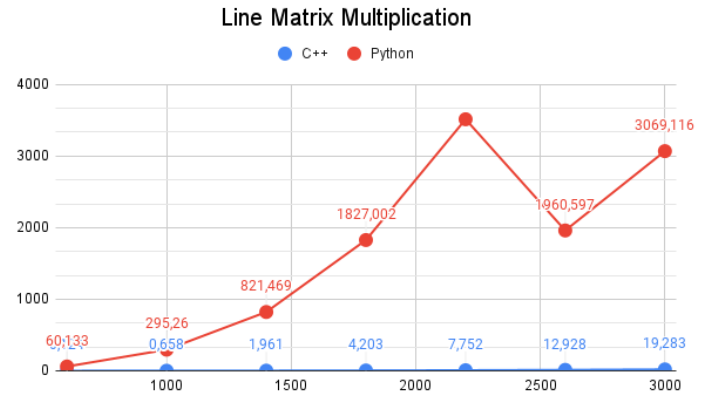
Additionally we measured the DCM for L1 and L2 caches. We ran the code with the Pcs provided in FEUP and compiled the result to create the graphs that we will show below.

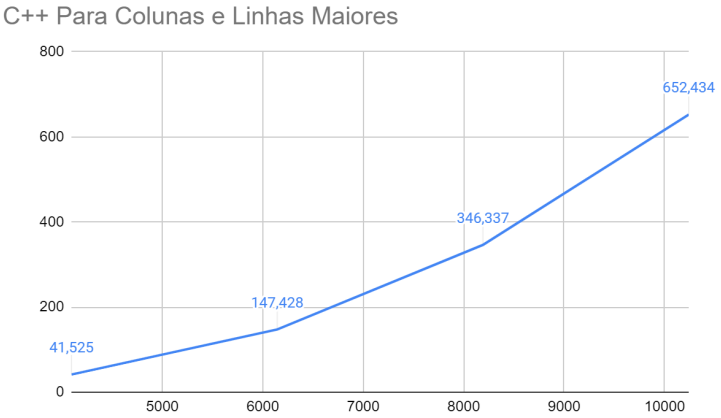
Results and analysis

In the algorithm based on the common algebraic approach the execution time increases significantly (almost exponentially) as the number of lines increases from 600 to 3000 both in c++ and in Python.

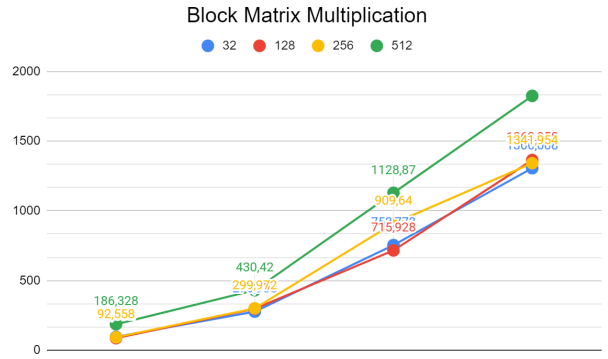


The second algorithm is slightly more optimized by reducing the number of operations required by a significant amount, improving significantly compared to the first algorithm. This doesn't deny that the time, theoretically, increases as the matrix size also increases. However in our tests we observed that with the number of lines 2600 the time, in python, decreases compared to the previous number of lines tested . We thought that this can be due to the hardware and the concurrent use of the computer because the program was tested in personal computers so problems can arrive from a variety of factors such as caching, memory allocation and pipelining.



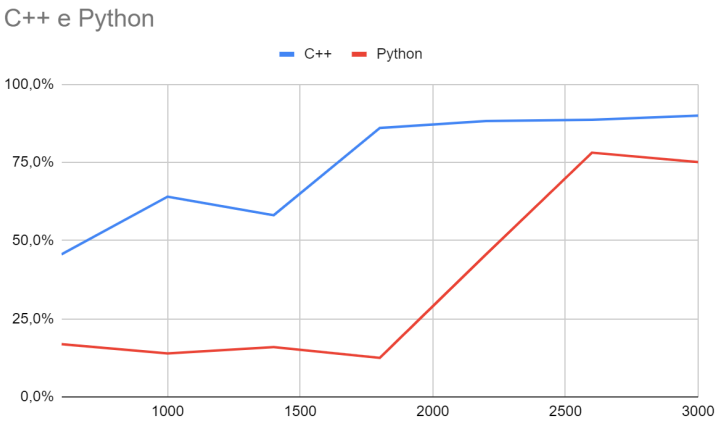


Finally we observe that the third algorithm is the most efficient of all the algorithms, mainly for larger matrix sizes. With the results of the execution we can conclude that the execution time decreases as the block size increases, but only until a certain size is reached. In this case, that size is 128. On the other hand, the blocks of size 512 have the biggest execution time, which could be due to cache limitation or hardware-related factors, like, for example, cache memory size.

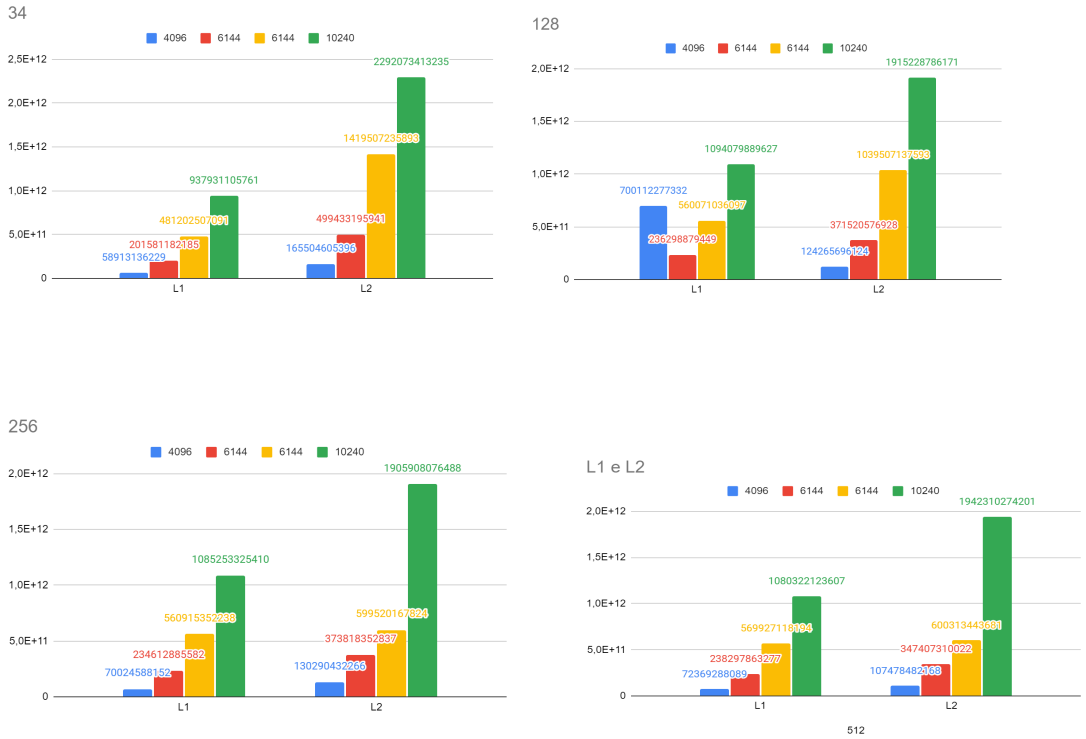


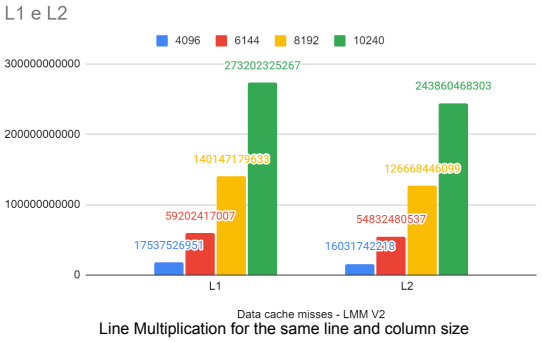
In terms of comparing between the two languages, it's noticeable that the c++ language is significante faster than the python for all algorithms and matrix sizes tested. This can be attributed to the fact that c++ is a low level compiled language, instead of a high level

interpreted language like python, making it better for computationally intensive tasks where speed is critical. As we can see in the graph below, the speed ups we get after applying the line multiplication is very significant, being even greater for C++. We can also see that these speedups are even greater for higher column and line values. For C++ the performance almost doubled and in the case of python the result is not far behind.



Finally, we analyzed the number of cache misses for the line multiplication and block multiplication for blocks of size 32, 128, 256 and 512. This metric can help us see the efficiency regarding the cache operations that our algorithms perform and miss.





Line Multiplication for the same line and column size

As we can see in all graphs the cache misses are higher the bigger the number of columns and lines. However we see that for the block multiplication of 128 there is an outlier for 4096 blocks and lines. This might be because we measured this particular number on a different machine than the rest of the values. We also see a trend upwards when we increase the size and that for block multiplication the misses on L2 are always higher than on L1.

The graph for line multiplication is a bit different, L1 misses are higher than L2 misses and those values are significantly lower than those of the block multiplication. This might explain why the performance is better when using this algorithm compared to the block algorithm. We think this happened because of 2 reasons: The block size was not the best for the cache that was being used and second the cpu was not parallelizing the code in an efficient manner as this would speed up the block multiplication very significantly.

Conclusions

From these results, we can conclude that the simpler matrix multiplication algorithm is not efficient for large matrices and that the C++ implementation is significantly faster than the Python implementation. This highlights the importance of algorithm design and language selection in achieving optimal performance in computing applications. We also found the importance of algorithms that cater to the architecture of the computer. With the data we collected, it is clear to see that the algorithms with line multiplication and block multiplication are much faster than the simple matrix multiplication. We can also see that with a sufficient parallelization the block matrix multiplication can be sped up even further. We can now also see how we could improve on this work. I think it would serve us better by having measured some more metrics apart from cache misses, like cache hits and number of operations per second. That being said, the results we obtained are very conclusive in proving the importance of parallelization and catering algorithms to computer architecture.