Problem Description and Algorithms Explanation

Matrix Multiplication

To resolve the problem we will use 3 different algorithms in 2 different programming languages (C++ ant Python).

Algorithms Used

Basic

In this algorithm, we did the standard multiplication, where we selected the n-th line of the first matrix and the n-th column of the second matrix, multiplying each element and adding it to the corresponding cell in the result matrix.

Line

Unlike the previous algorithm, in this one, while still selecting the n-th line of the first matrix, we now go multiply each element of that line by the entire line of the second matrix, updating the results in the result matrix.

Block

With this algorithm, we divide each matrix into blocks of custom sizes, utilizing the line algorithm to calculate each block of the result matrix, adding them in the end.

Performance Metrics

While measuring the performance of the algorithms in c++, we measure the execution time and utilized 6 different performance counters:

- L1_DCM: Level 1 data cache misses – Used to know how many times the value the algorithm wanted wasn't in the first level of the cache memory, which lets us understand how much overhead was introduced to the algorithm due to accessing the main memory;
- L3_LDM: Level 3 load misses – Occurs when the processor needs to fetch memory from the third level of the cache, but the data does not exist in it, causing it to have to get it from the main memory, introducing overhead;
- L1_LDM: Level 1 load misses – Just like L3_LDM, just for the first level of the cache, allowing us to pinpoint with further accuracy the amounts of total load misses and thus the amount of total overhead caused by it;
- L2_LDM: Level 2 load misses – Just like both the L3_LDM and L1_LDM, but for the second level of the cache, providing us with further information with the same goal in mind.
- L2_DCM: Level 2 data cache misses – Just like the same for the first level of the cache, which would allow us to know with more accuracy how many times the main memory had to be accessed during the algorithm;

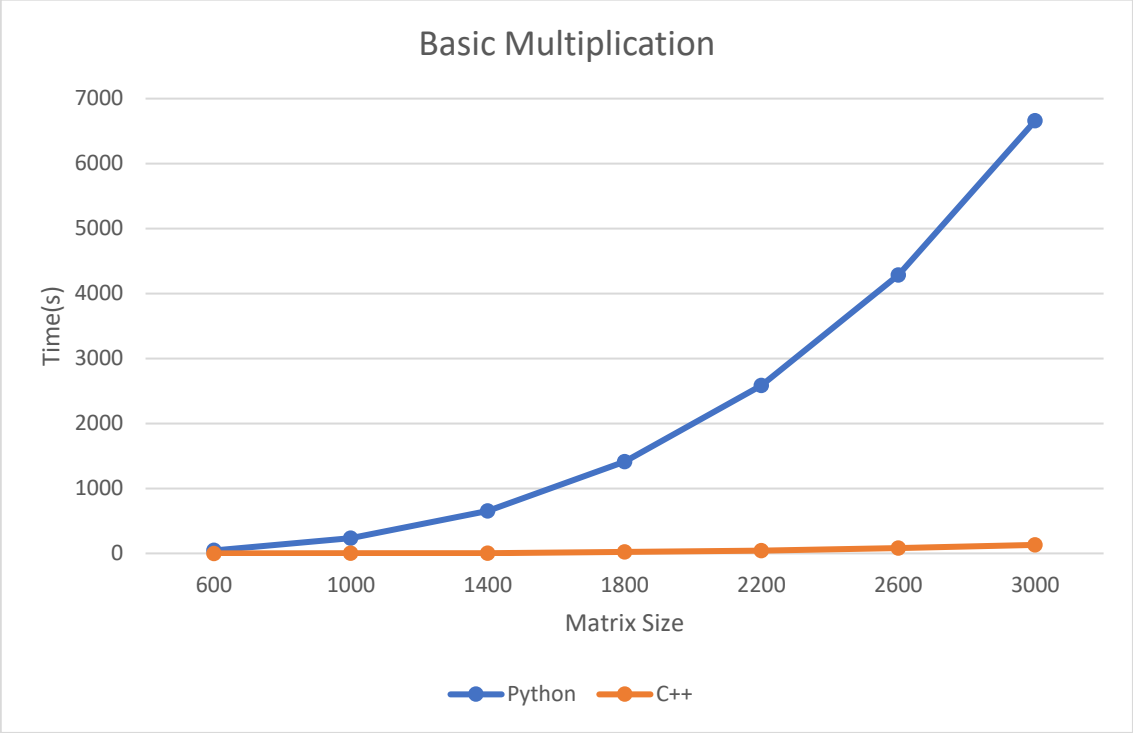
There were other counters which were not available for us due to our own hardware:

- MEM_RCY: Cycles stalled waiting for memory reads – Used to know when the program is waiting to be able to read data, which would allows to understand certain overheads.

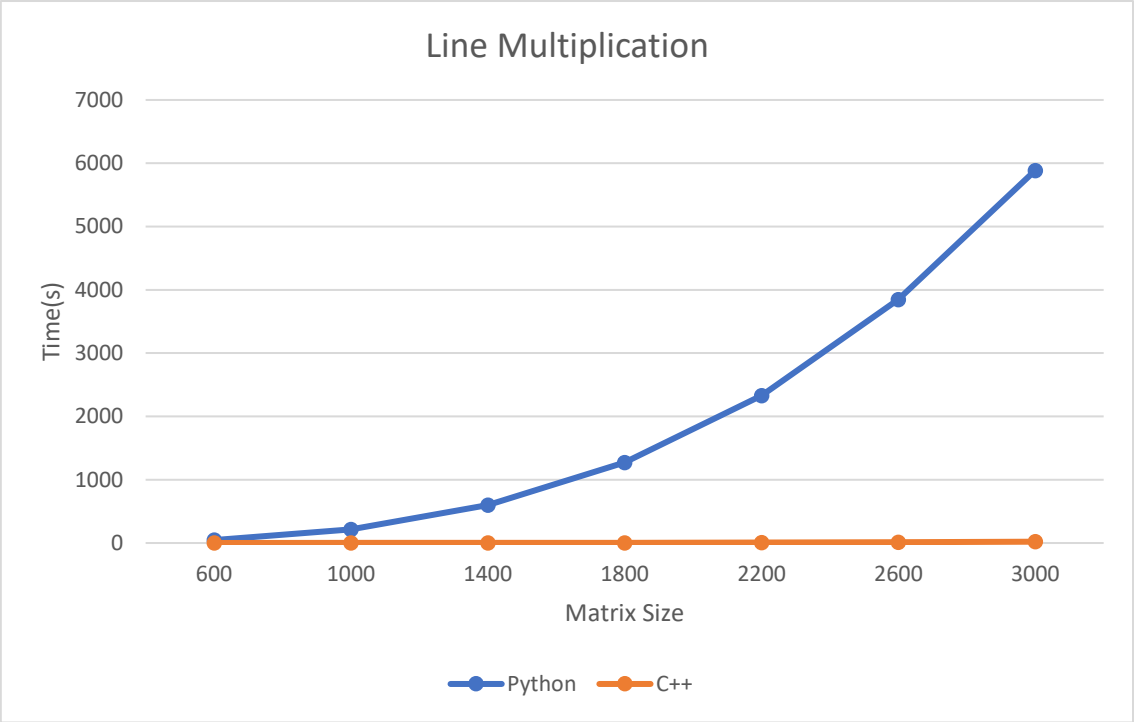
Results and Analysis

We will compare the differences between 2 python algorithms, python and C++ performance, 3 C++ algorithms and block size difference.

Python vs C++



Graph 1



Graph 2

Using both the basic multiplication algorithm and the line multiplication algorithm, with the same matrix sizes, we were able to realize that, just as expected, the algorithm was incredibly faster in C++, especially when escalating the values for the matrix sizes, which yielded extremely long execution times in the Python algorithm.

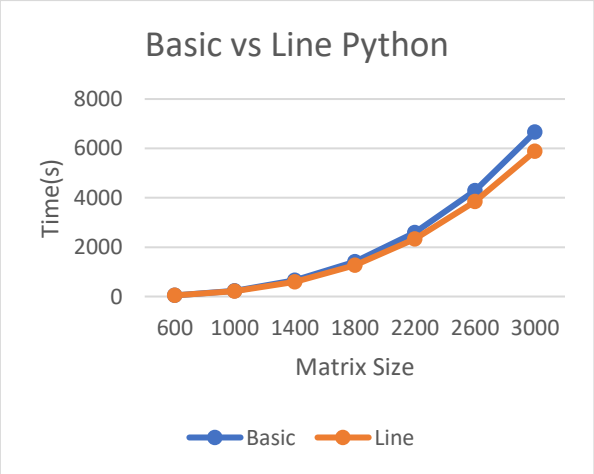
This happens due to Python being an interpreted language, thus needing more CPU cycles to perform a given statement.

Line Algorithm

In both languages, we can see an increase in performance in the line algorithm when comparing to the basic one.

This can be accounted due to the way the system handles cache memory and how, with this algorithm, it is optimized.

When the computer needs a certain value, the processor will retrieve the value from the main memory, after checking if it exists in the cache, and then copy it to the cache

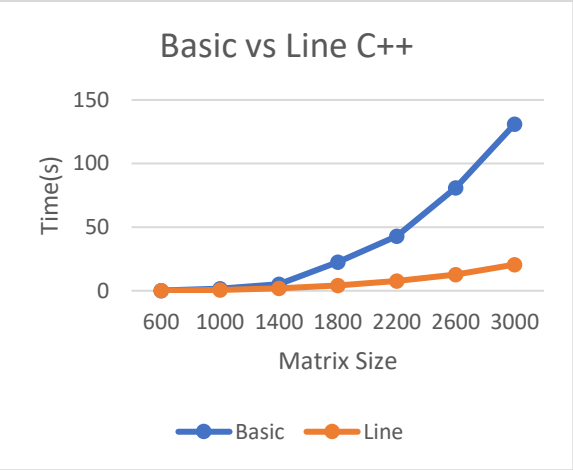


Graph 3

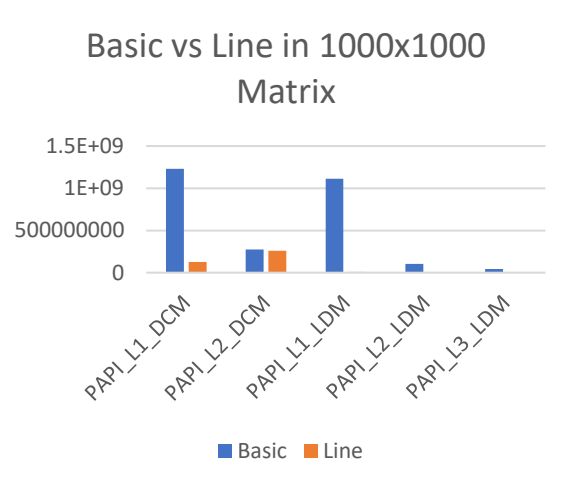
along several values next to it, since it's more likely we'd later on need the values closer to it (spatial locality).

Due to this and since in the line algorithm we go line by line in the second matrix, instead of column by column like in the basic algorithm, the values required for sequential operations will have already been loaded to the cache, resulting in much less accesses to it.

We can observe this behaviour by comparing certain counters, like the ones in the graph to the right (Graph 5). With these, which give us information about data cache misses and load misses, we can conclude that due to taking advantage of spatial locality in the line algorithm, the misses were lower than the ones in the basic algorithm along with the load misses, resulting in a faster execution time.



Graph 4



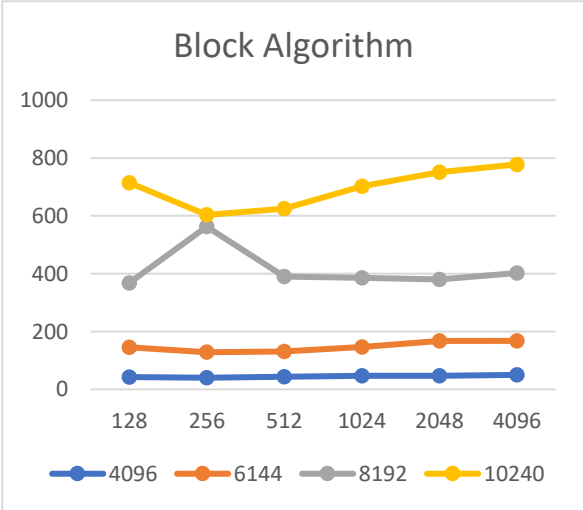
Graph 5

Block Algorithm

In this algorithm, we separate each of the two matrices in submatrices (blocks), which can be fully loaded into the cache, should the memory be enough.

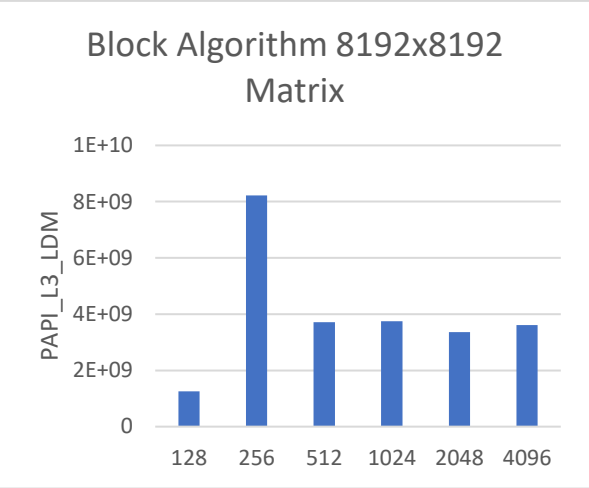
Each of these blocks will then be treated as a single matrix and multiplied with the correspondent one in the other matrix via the line multiplication method, storing the results in the result matrix, to later be incremented by the results of other blocks.

By analysing our results, we concluded that blocks of size 256 worked better for all matrix sizes, except for 8192 matrices, where it registered, over multiple attempts, the worse results out of them all.



Graph 6

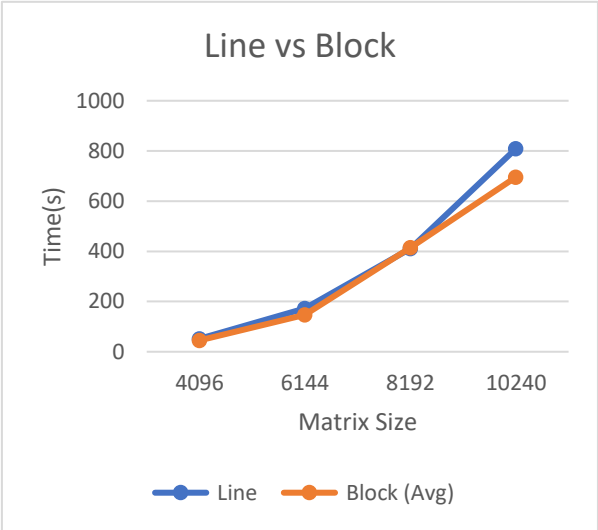
From looking at Graph 7, we realized that in the size 256 for the blocks, the load misses in level 3 of the cache were more than double of any amount for the same matrix size, which could explain the overhead it added to the algorithm in terms of execution time.



Graph 7

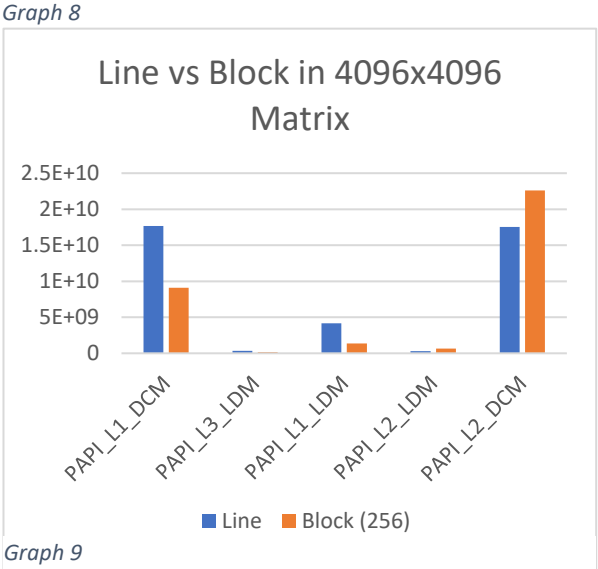
By comparing the execution times of both the line and block algorithms, we noticed a slight decrease in execution time, although not as drastic as when comparing to the basic algorithm.

However, in the 8192-size matrix, due to the anomaly of the execution time in blocks of size 256, the average of the execution time of the block algorithm ended up being just 4 seconds higher than the line algorithm, even if the other values aside from 256 were lower.



Graph 8

By analysing Graph 9, where we compare the counters of the two algorithms, we can see that the data cache misses of the first level were nearly halved from the line to the block algorithm, while in second level they rose up by a small amount.



Graph 9

Conclusions

We concluded that C++, even without maximum optimization, is much more efficient when compared to Python, especially once the values start escalating.

Regarding the algorithms, we determined that the line algorithm is much more efficient than the basic algorithm, yielding much better results as the matrix sizes increased, while the block algorithm was only slightly better than the line one.

However, as the values increased, we noticed an increasing amount in the difference between efficiency when comparing the block and line algorithms, which leads us to conclude that in even higher values, ones we did not test, the block algorithm might prove to be even better.

The main efficiency increase from the basic to the line and then to the block algorithms is due to lowering the cache miss rate at the first level. In the line algorithm, we see it by going through the second matrix in a line, instead of column, and in the block algorithm, alongside with the improvement from the line one, there's also the fragmentation of the matrix into blocks small enough to take better advantage of the cache memory.