# 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 let’s 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;
* PRF\_DM: Data prefetch cache misses – Used to know how accurately the computer can predict the data that will be needed by the algorithm before it’s needed, by loading it into the cache;
* MEM\_WCY: Cycles stalled waiting for memory writes – Used to know when the program is waiting to be able to write data;
* 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.

Due to unforeseen problems that left us unable to run the algorithm once again, we couldn’t measure the performances for the following 2 counters:

* 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;
* TLB\_DM: Data translation lookaside buffer misses – Used when the computer needs to calculate the physical address where a certain variable is located based on the virtual memory address, instead of just looking up the corresponding table inn the translation lookaside buffer, adding overhead to the program.

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++

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 cache, and then copy it to the cache, alongside 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.

## Block Algorithm

#### Python Algorithms

Chart, line chart

Description automatically generatedWe can observe that Line algorithm is better than Basic Algorithm.

## Conclusions