# Performance evaluation of a single core

The proposed project aims to show the effect on the processor performance of the memory hierarchy when accessing large amounts of data.

## Index

## Problem description and algorithms explanation

For this study we will approach the problem, **product of two matrices** in different ways (*explained bellow*).

### 1. Basic Multiplication

The basic multiplication algorithm is a linear algebra operation that produces a matrix C from two others A and B.

It applies a row-by-column multiplication, where the entries in the *ith row* of A are multiplied by the corresponding entries in the *jth column* of B and then adding the results.

**PseudoCode:**

```
for line_i in matrix_a:
    for col_j in matrix_b:
        for elem in line_i:
            matrix_c[i][j] = elem * col_j
```

### 2. Line Multiplication

The line multiplication is a variation of the basic multiplication, where the entries in the *ith row* of A are multiplied by the corresponding entries in the *jth row* of B and thus accumulated in the respective position of the matrix C.

**PseudoCode:**

```
for line_i in matrix_a:
    for elem in line_i:
        for line_j in matrix_b:
            matrix_c[i][j] = elem * line_j
```

### 3. Block Multiplication

The block multiplication is a block oriented algorithm that divides the matrices A and B in blocks and takes advantage of the line multiplication for the calculations of the values, of the matrix C.

**PseudoCode:**

```
for block_row in range(0, size_a, block_size):
    for k in range(0, size_a, block_size):
        for block_col in range(0, size_b, block_size):
            matrix_c[block_row][block_col] += OnMultLine(matrix_a[block_row][k], 
matrix_b[k][block_col])
```

## Performance metrics

In order to measure the performance of the processor in the given problem, we decided to compare the results using two different programming languages, and some relevant performance indicators.

The central programming language studied is **C++**, supported by the Performance API (**PAPI**) to collect the values of **data cache misses** in the Level 1 and Level 2 caches.

As for our second programming language we opted for **python**, measuring and comparing the **time performance** against **C++**.

So that our study is more consistent, we decided to take a considerable number of **samples**, 5 for "quick" runs, and 3 for "longer" ones. As well as, making use of **different sized** matrixes as stated by the given proposal.

## Results and analysis

### 1. Basic Multiplication

**C++ Performance:**

| Size | Time(s) | Level 1 DCM | Level 2 DCM |
|---|---|---|---|
| 600 | 0.1974 | 244785701 | 39485267 |
| 1000 | 1.2900 | 1226948406 | 259755344 |
| 1400 | 3.5128 | 3515137318 | 1322263105 |
| 1800 | 18.2572 | 9081694573 | 7534199569 |
| 2200 | 38.8636 | 17648912428 | 23211642005 |
| 2600 | 69.1080 | 30902867592 | 51364020207 |
| 3000 | 115.1938 | 50302210624 | 95153348353 |

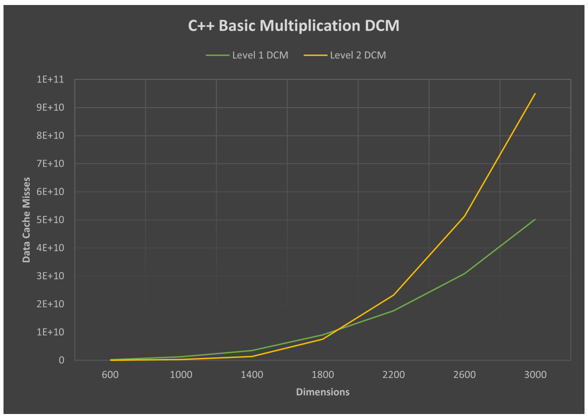


*Fig 1. C++ Basic Multiplication DCM*

From the analysis of the table values we verify an expected increase in the elapsed time on bigger matrix sizes. As for the data cache misses we can follow the same conclusions, knowing that the bigger the matrix is the more memory accesses occur.

**Python Performance:**

| Size | Time(s) |
|---|---|
| 600 | 14.7924 |
| 1000 | 87.9724 |
| 1400 | 254.0206 |

| Size | Time(s) |
|---|---|
| 1800 | 552.8929 |
| 2200 | 1048.3686 |
| 2600 | Time Limit |

As in the previous analysis, as the matrix size increases we verify a notorious increase in the algorithm's elapsed time.

## 2. Line Multiplication

**C++ Performance:**

| Size | Time(s) | Level 1 DCM | Level 2 DCM |
|---|---|---|---|
| 600 | 0.1066 | 27099162 | 57536539 |
| 1000 | 0.4912 | 125836635 | 264804441 |
| 1400 | 1.7010 | 346408203 | 693423581 |
| 1800 | 3.6562 | 745937934 | 1424375689 |
| 2200 | 6.5332 | 2074927662 | 2562864250 |
| 2600 | 10.5584 | 4413086703 | 4189022075 |
| 3000 | 15.9574 | 6780680802 | 6378165232 |

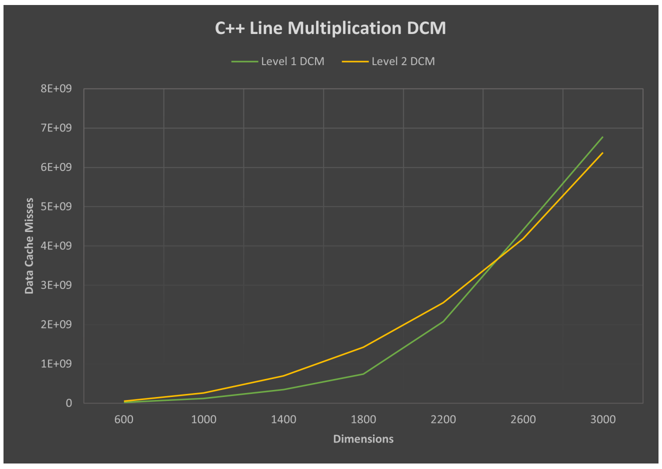


***Fig 2. C++ Line Multiplication DCM***

| Size | Time(s) | Level 1 DCM | Level 2 DCM |
|---|---|---|---|
| 4096 | 43.3510 | 17532306353 | 1606236906 |
| 6144 | 144.2100 | 59125392006 | 54497490331 |
| 8192 | 341.1400 | 1.40058E+11 | 1.31568E+11 |
| 10240 | 675.5400 | 2.73433E+11 | 2.72578E+11 |

There is a clear improvement in all performance metrics comparing the **basic multiplication** with **line multiplication**. This is due to the fact that the second algorithm takes advantage of the **C++** memory allocation.

When accessing a specific matrix position the compiler reads not only the value of that position, as well as the whole block where the value is inserted.

Having that in mind, trying to decrease memory accesses to disk (since it is a time costly operation), the **line multiplication** algorithm makes an effort to make use of all the elements read from the memory block.

**Python Performance:**

| Size | Time(s) |
|---|---|
| 600 | 13.8822 |
| 1000 | 65.2808 |
| 1400 | 180.7721 |

| Size | Time(s) |
|---|---|
| 1800 | 385.8744 |
| 2200 | 704.9206 |
| 2600 | Time Limit |

Being a high level language **python** code does the heavy lifting in a low-level language like **C/C++**, in this degree, an improvement in the **C++** performance would imply an improvement in the **python** program execution, which was corroborated by the achieved values.

## C++ vs Python:

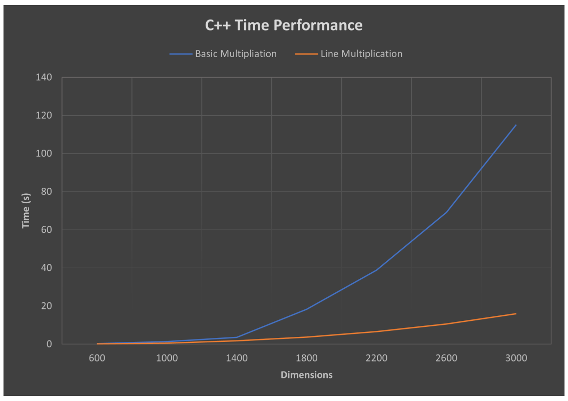


*Fig 3. C++ Time Performance*

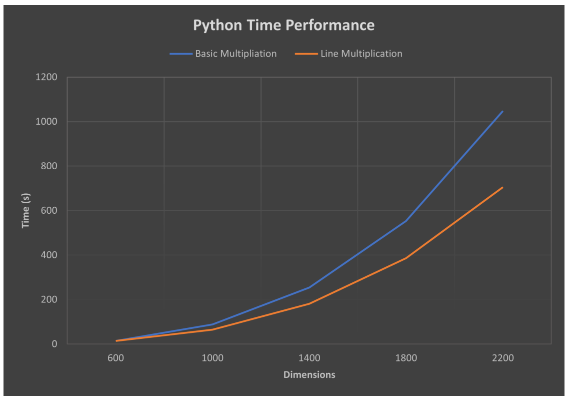


*Fig 4. Python Time Performance*

From the graphs above, and the results obtained with different runs, we can conclude that **python** has a worst performance in comparison to **C++** regarding the given problem.

The reason for such differences can be explained by the fact that **python** uses the interpreter and performs lots of run time operations while **C++** programs are precompiled, thus only executing the produced program. Furthermore, no libraries were used in the development of the algorithms from which **python** would have greater benefits.

## 3. Block Multiplication

**C++ Performance:**

| Block Size = 128 | Size | Time(s) | Level 1 DCM | Level 2 DCM |
|---|---|---|---|---|
| | 4096 | 36.6276 | 9832748918 | 32935278304 |
| | 6144 | 129.4356 | 33176432529 | 1.12414E+11 |
| | 8192 | 303.9380 | 78627735611 | 2.65393E+11 |
| | 10240 | 622.3020 | 1.53525E+11 | 5.16219E+11 |
| **Block Size = 256** | **Size** | **Time(s)** | **Level 1 DCM** | **Level 2 DCM** |
| | 4096 | 34.5753 | 9133260397 | 23278123986 |
| | 6144 | 118.8180 | 30815161405 | 76811797988 |
| | 8192 | 405.4706 | 73352067948 | 1.62399E+11 |
| | 10240 | 561.1250 | 1.42613E+11 | 3.50658E+11 |
| **Block Size = 512** | **Size** | **Time(s)** | **Level 1 DCM** | **Level 2 DCM** |
| | 4096 | 36.4226 | 8758854262 | 19290855969 |
| | 6144 | 76.8863 | 29606575477 | 66240145866 |
| | 8192 | 341.0196 | 70239707870 | 1.36267E+11 |
| | 10240 | 512.3696 | 1.36869E+11 | 3.06304E+11 |

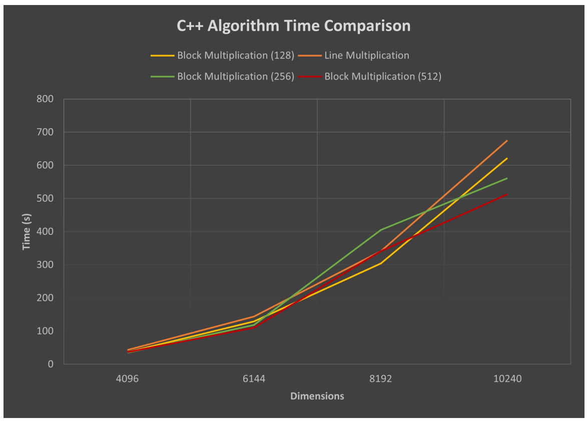
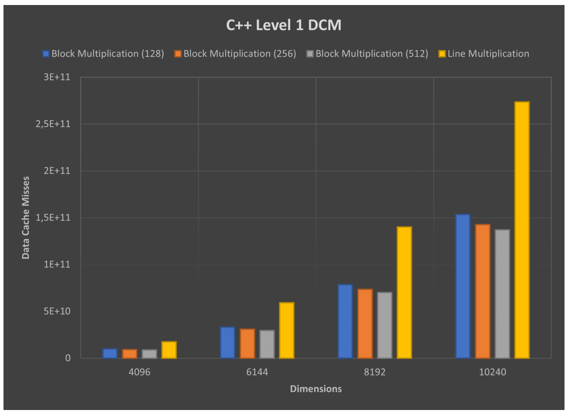


*Fig 5. C++ Algorithm Time Comparison*



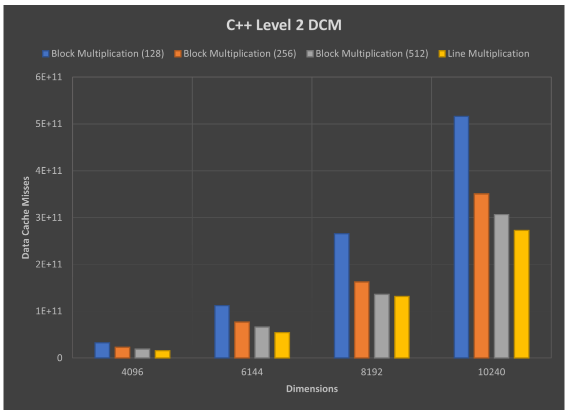


*Fig 6. C++ Level 1 DCM*

*Fig 7. C++ Level 2 DCM*

Since the memory is divided into consecutive blocks each access to memory reads one block at a time. When dealing with large amounts of data, trying to reduce these accesses becomes essencial for a better performance program.

To this extent, the **block algorithm** takes advantage of the memory layout by reducing memory calls (divides the problem into smaller ones) in contrast to the **line multiplication algorithm**, which results in a data cache miss reduction.

Increasing the block size leads to a greater performance, capped by the Level 1 cache block size, since it is the smaller of the various caches. By analysing the values in each of the tables, we can conclude the stated above.

### 4. Floating Point Operations Comparison

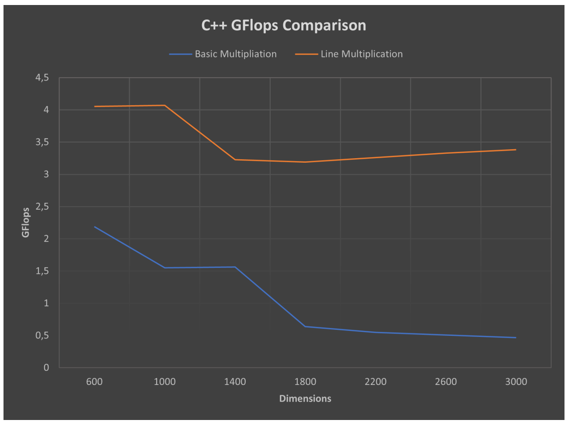


***Fig 8. C++ Number of Floating Point Operations Comparison***

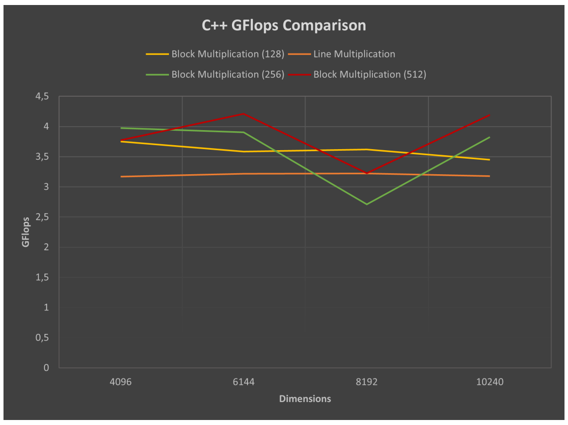


***Fig 9. C++ Number of Floating Point Operations Comparison***

Concerning the number of floating point operations per second (useful work), the algorithms which perform better (higher values) are the ones who present superior time performances, as seen previously.

As a consequence of bad memory use, much of the work done by the processor revolves around memory accesses and cache coherence, decreasing, this way, the useful work.

Comparing the 3 algorithm versions, in order, and as the size of the matrixes increase, there is a significant decrease in the number of operatins per second, for the first algorithm, due to the fact that for each calculation there is a need for extra load/store operations. Contrastingly, for the second and third variants, for each useful operation, as there is an improved memory cache reuse, the number of load/store operations is much lower, therefore substantially better results.

## Conclusions

To sum up, this study allowed us to perceive other important metrics when taking program performance into account, not only the time complexity, since each variation of the matrix multiplication has the same Big O, but the way we take advantage of memory layout and locality.