CPD - Project 1
Performance evaluation of a single core

Grupo 14

Mónica Moura Pereira up201905753@edu.fe.up.pt
Rui Pedro Silva up202005661@edu.fe.up.pt
Tiago Marques up201704733@edu.fe.up.pt

Problem description and algorithms explanation

- Normal Matrix Multiplication

We were given a C++ function that multiplies two matrices with the same dimensions. The way the solution matrix is calculated is based on the traditional linear algebra method, this is, each element results from the sum of all the multiplication pairs of all the elements in the correspondent row from the first matrix by all the elements in the correspondent column from the second matrix. So, for example, the first element at the top left is obtained by multiplying the first elements of both matrices, then adding it to the value obtained in the multiplication of the second element of the first row of the first matrix by the first element of the second row of the second matrix, then adding it to the next multiplication and so on until all elements from the first row of the first matrix and all elements from the first column of the second matrix were approached.

We needed to translate the C++ code to another programming language of our choice and we did it in Python. So, below is the translation of the most relevant part of the function described above:

```
for i in range(0, m_ar):
    for j in range(0, m_br):
        temp = 0
        for k in range(0, m_ar):
            temp += pha[i * m_ar + k] * phb[k * m_br + j]
        phc.append(temp)
```

- Line Matrix Multiplication

This algorithm takes a little bit from the first one. Instead of multiplying one line of the first matrix by each column of the second matrix, this version multiplies an element from the first matrix by the corresponding line of the second matrix.

Imagine if we have the following matrices: [[1, 2], [4, 5]] and [[10, 11], [20, 21]]. The algorithm will start by doing the following to the result matrix: start by multiplying “1” to the correspondent line, [[1*10, 1*11], []] and then do the same with “2”, [[1*10+2*20, 1*11+2*21],[]], and so on.

The code made for this algorithm is presented below:

```
for(i=0; i<m_ar; i++)
    {    for( j=0; j<m_br; j++)
         {
              for( k=0; k<m_ar; k++)
              {
                   phc[i*m_ar+k] += pha[i*m_ar+j] * phb[j*m_br+k];
              }
         }
    }
```

- Block Matrix Multiplication

The block multiplication is a block oriented algorithm that is performed in a blocked manner, where the matrices are divided into submatrices of size blockSize x blockSize, and the computation is done on each submatrix. The resulting product phc is updated in a row-major order.

```
for(i=0; i<m_ar; i++)
    {
         for( j=0; j<m_br; j += bkSize)
         {
              for( k=0; k<m_ar; k += bkSize)
              {
                   for( l=i; l<min(i + bkSize, m_ar); l++)
                   {
                        for( m = k; m < min(k + bkSize, m_br); m++)
                        {
                             for( n = j; n < min(j + bkSize, m_ar); n++)
                             {
                                  phc[l*m_ar + n] += pha[l*m_ar + m] *
phb[m*m_br+n];
                             }
                        }
                   }
              }
         }
    }
```

Performance metrics

For performance metrics we decided to base our assumptions mainly in **time**. This was because it’s a simple and effective measure to evaluate the performance of our algorithms. As our Python measures for time were increasing exponentially, we decided to stop measuring once we reached 30 minutes of processing time (about 1800 seconds), as it was becoming almost impossible to take measurements.

We also used the PAPI module for data cache misses analysis, using the **L1_DCM** and **L2_DCM** counters. Likewise, Level 1 (L1) cache is the closest cache to the processor core and has the fastest access time. When the processor needs to access data that is not available in the L1 cache, it needs to fetch it from the next level cache or the main memory. This process is known as a cache miss. With that being said, it’s really important to take into consideration these measures as a high number of L1 cache misses can indicate that the cache size is too small or that the program is not effectively using the cache. This makes these counters really useful to evaluate our algorithms.

Furthermore, adding to the measures mentioned above, we decided to also use **FLOPS**. FLOPS basically stands for "floating-point operations per second." It is a measure of computing performance that quantifies the number of arithmetic operations involving floating-point numbers that a computer system can perform in one second. Because of this reason, we also think that it was an interesting measure to add.

All three functions share the same time complexity, O(N^3), and space complexity, O(N^2). Knowing this and the fact that we only have 2 float operations on each function, more specifically inside the last ‘for’ loop, our formula to calculate the FLOPS was the following: **2 * N^3 / time**, where ‘N’ is the size of a row or column, since they are equivalent, and ‘time’ the time it took for the function to calculate the multiplied matrix of size ‘N’ * ‘N’.

In the following section we will show the results obtained and do an analysis of the performance metrics chosen. We used these 3 metrics for the 3 different algorithms (in C++) made for the multiplication of matrices. It’s important to note that for the Python versions we only used the time performance measure.

It’s also worth noting that we run the algorithms in the same architecture and every algorithm was run 6 times before we took the mean value. This was mainly done to make sure that the results are consistent. With that being said the details about the architecture used are in the Annexes section.

Finally, we only show in the following section the most relevant graphs that we made, in order to make more of a compact report of our findings. The rest are present in the spreadsheet we used to do our measures (CPD - TP1).

Results and analysis

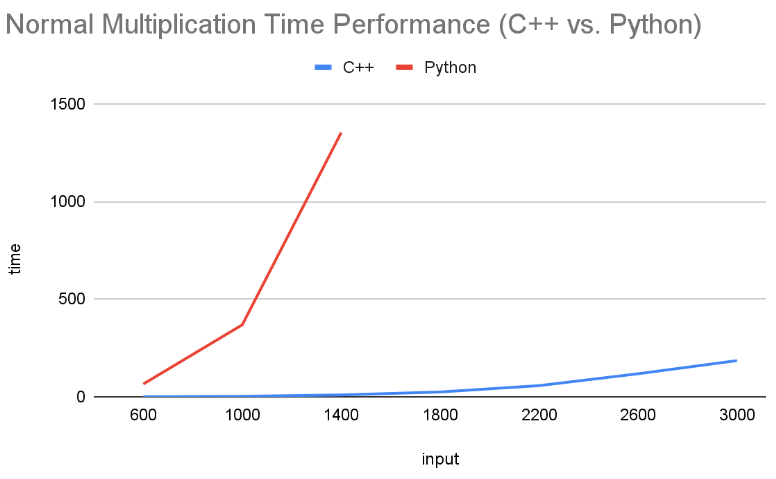
- Normal Matrix Multiplication

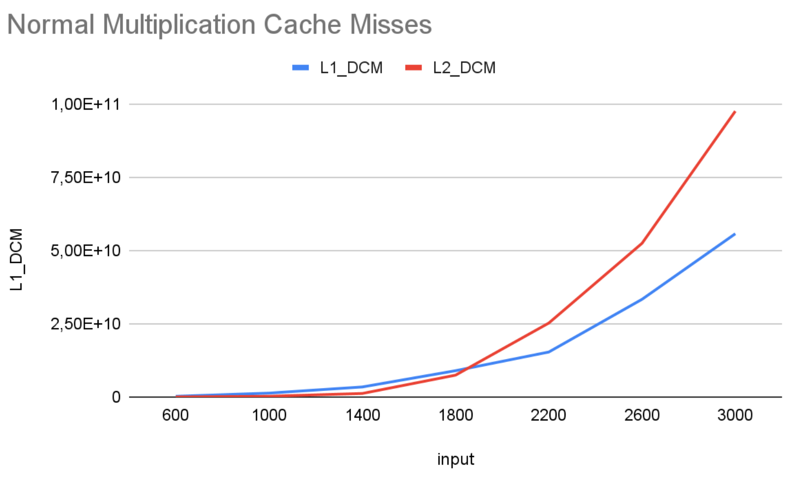
For the first algorithm we registered our previously decided performance measures in the C++ version, for input matrices from 600x600 to 3000x3000 elements with increments in both dimensions of 400. For Python we only registered the time measure.

We almost immediately noticed that C++ is a lot faster than Python. It’s also worth noting that our algorithm started to fail in the Python version, making it almost impossible to make the calculations in a considerable time. This was because, C++ has a lower-level implementation and better memory management. C++ has a faster runtime than Python due to its ability to compile code, while Python is an interpreted language. This means that C++ code can be optimized by the compiler, while Python code is executed line by line.

We also could notice that the number of cache misses increases along with the time it took to run those algorithms. We noticed it mainly on the L2 cache misses.

Below, we also graphs that prove our findings.



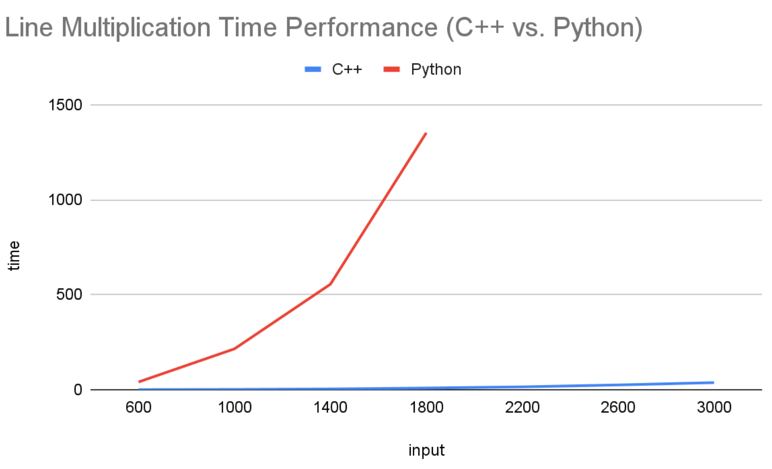


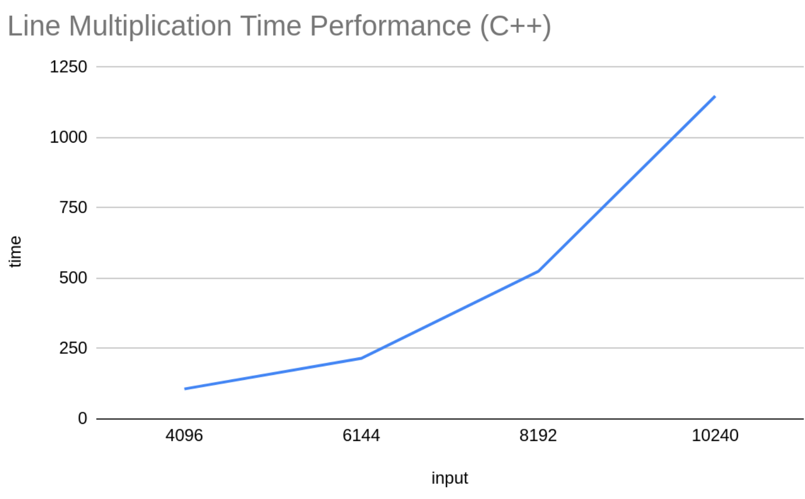
- Line Matrix Multiplication

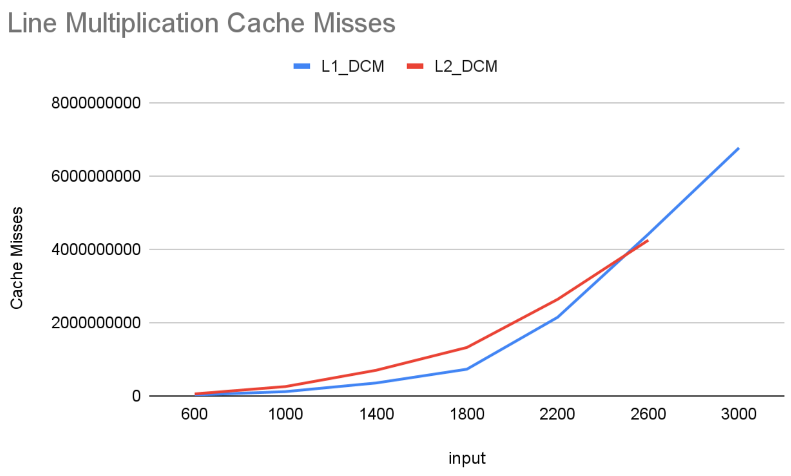
For the Line Matrix Multiplication we registered our previously decided performance measures in the C++ version, for input matrices from 600x600 to 3000x3000 elements with increments in both dimensions of 400. We also registered these measures for matrices with 4096x4096 to 10240x10240 with intervals of 2048 in the C/C++ version. It’s worth noting that for the Python version we only measured time.

Like the first algorithm, we came to the conclusion that the Python version is a lot slower than the C++ version.

Likewise, the number of cache misses is also increasing over the input increase, but the L1_DCM and the L2_DCM are much similar.





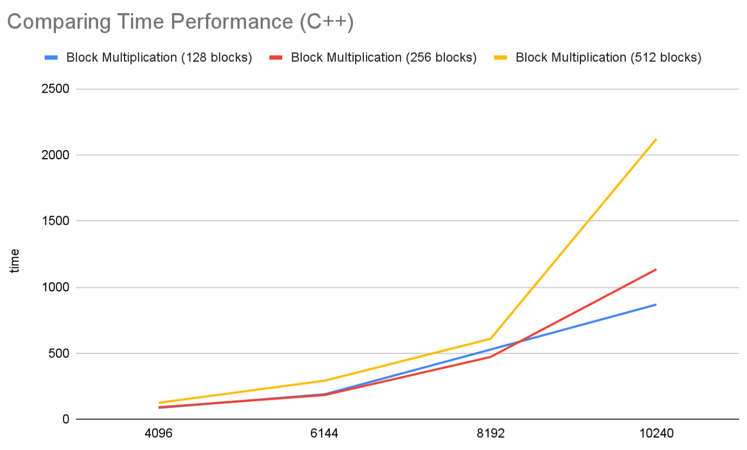


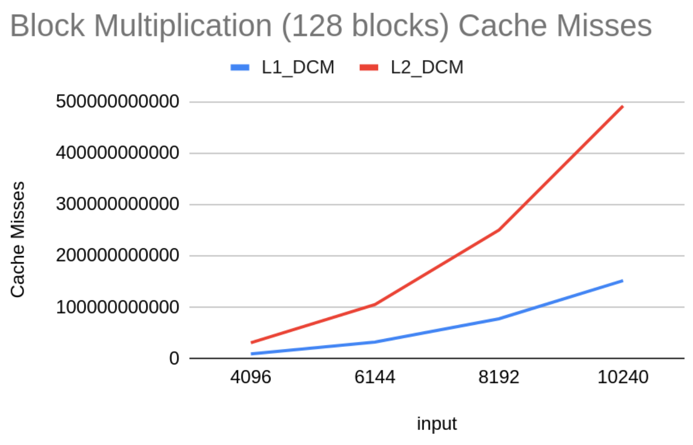
- Block Matrix Multiplication

We took measures of the performance metrics for 3 different block sizes (128 blocks, 256 blocks and finally 512 blocks). We show the results in the graphs below and then make a comparison between the number of blocks. We found out that when increasing the block size, the time it took to run the algorithm didn't change all that much, although it was better than the Line Multiplication algorithm.

The memory is divided into blocks, and accessing one block at a time is the norm. When dealing with a large amount of data, it is important to minimize these accesses for optimal program performance. The block algorithm takes advantage of memory layout to reduce the number of memory calls, while the line multiplication algorithm reduces data cache misses. A larger block size can improve performance, but it is limited by the size of the Level 1 cache block. We can conclude this by examining the values in each table.

Finally, we came to similar conclusions as previously discussed. The number of cache misses increases over input increases and like so the time increases also. We present the number of cache misses for blocks of 128, the other graphs are very similar and are present in the spreadsheet provided. We conclude that the L2_DCM values are higher and increase more than the L1_DCM.





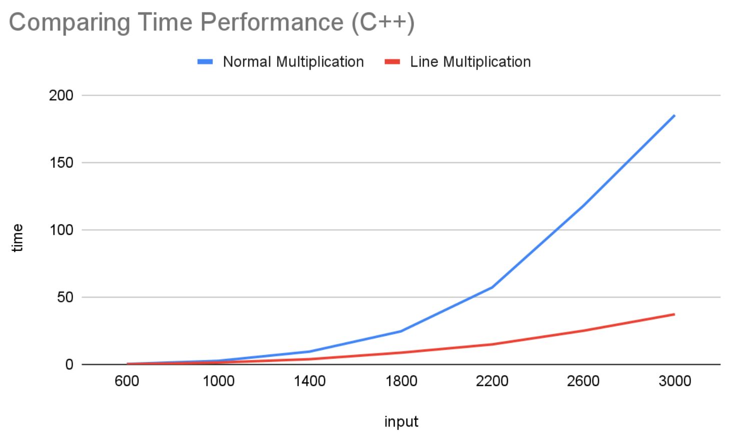
Conclusions

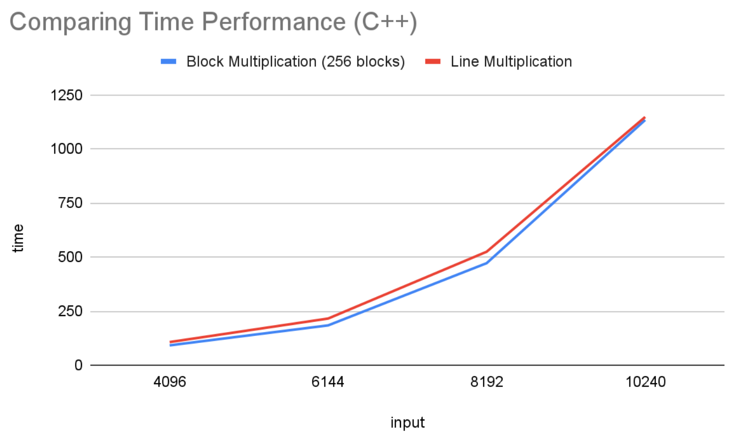
Finally, to come to realistic conclusions about the performance of the 3 algorithms that we made, we decided to compare the time performance along with the number of FLOPS. With that being said, we compared the Normal Multiplication algorithm with the Line Multiplication, and then we compared the Line Multiplication algorithm with the Block Multiplication (with 256 blocks).

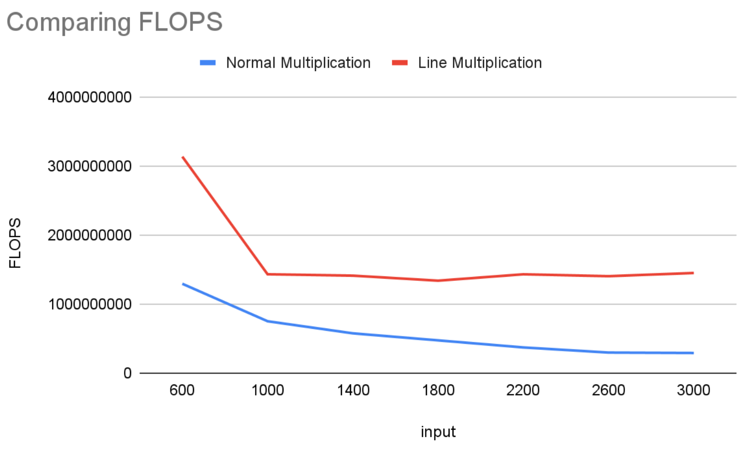
Likewise, we noticed that there was a significant difference in time performance between the first and second algorithms, which is shown below. Now comparing the Line with the Block Multiplication we can see that the Block Multiplication improved the time performance but not significantly. Thus, the best performing algorithm in terms of time is the last one.

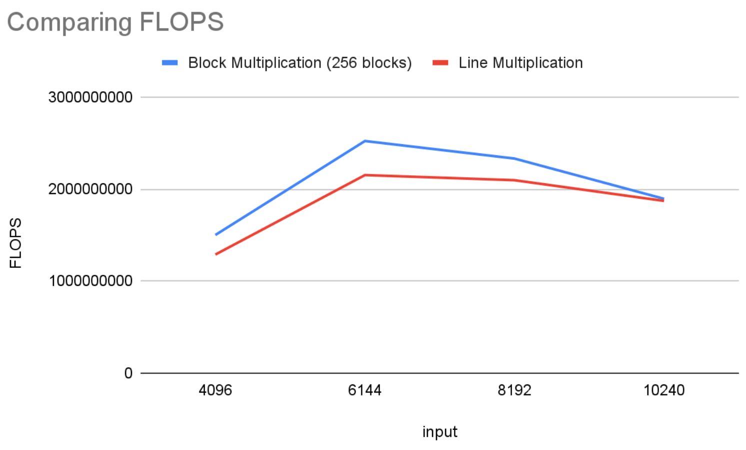
When it comes to Floating Point Operations per second, we noticed that for all 3 algorithms they fluctuate (but are more so consistent) over input size. This came to no surprise, as the time increases over input size (number of operations). So with both increasing, the FLOPS should stay more less constant. We found out that the algorithm with less FLOPS is the Normal Multiplication and the algorithm with more FLOPS is the block multiplication.

Likewise, we present our findings in the graphs below.









Annexes

Architecture Used:

```
Arquitetura:                 x86_64
Modo(s) operacional da CPU: 32-bit, 64-bit
Ordem dos bytes:                      Little Endian
CPU(s):                      4
Lista de CPU(s) on-line:              0-3
Thread(s) per núcleo:                 2
Núcleo(s) por soquete:                2
Soquete(s):                  1
Nó(s) de NUMA:                        1
ID de fornecedor:            AuthenticAMD
Família da CPU:                       23
Modelo:                      24
Nome do modelo:                       AMD Ryzen 3 3200U with Radeon Vega Mobile Gfx
Step:                        1
CPU MHz:                     1407.760
CPU MHz máx.:                         2600,0000
CPU MHz mín.:                         1400,0000
BogoMIPS:                    5189.88
Virtualização:               AMD-V
cache de L1d:                32K
cache de L1i:                64K
cache de L2:                 512K
cache de L3:                 4096K
CPU(s) de nó0 NUMA:                   0-3
```

Papi:

```
Cache Information.

L1 Data Cache:
  Total size:                32 KB
  Line size:                 64 B
  Number of Lines:           512
  Associativity:             8

L1 Instruction Cache:
  Total size:                32 KB
  Line size:                 64 B
  Number of Lines:           512
  Associativity:             8

L2 Unified Cache:
  Total size:                256 KB
  Line size:                 64 B
  Number of Lines:     4096
  Associativity:       4

L3 Unified Cache:
  Total size:          8192 KB
  Line size:           64 B
  Number of Lines:   131072
  Associativity:       16
```