U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Licenciatura em Engenharia Informática e Computação - Redes de Computadores

Performance Evaluation Of a Single Core

Computação Paralela e Distribuída

Turma 4 - Grupo 10

Adam Nogueira (up202007519@edu.fe.up.pt)

Matilde Silva (up202007928@edu.fe.up.pt)

Vinícius Corrêa (up202001417@edu.fe.up.pt)

Índice

1. Introduction

In this project, we will analyze the performance of a single-core processor, when large amounts of data are accessed. Specifically, this examination will focus on the product between several different sized MxM matrices, whilst using metrics and performance indicators provided by the Performance API (PAPI).

The matrix multiplication acts as a good performance measure, since the operations performed do not occupy a significant part of the run time, in fact, the greatest bottleneck lies within the memory access.

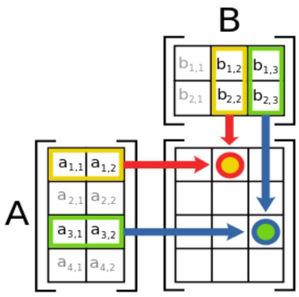
Furthermore, a comparison of the execution time of different programming languages will also be done, to gain a broader understanding of such impact on the processor speed.

2. Algorithm Explanation

2.1. Naive Multiplication

This method of matrix multiplication follows the traditional process used when multiplying matrices by hand. Let A, B and C be matrices of dimensions MxM, where C = A*B.

In this multiplication algorithm, the first row of matrix A is multiplied by the first column of matrix B, the second row of matrix A is multiplied by the second column of matrix B and so forth. Hence, each cell of matrix C is calculated individually. This process is illustrated by the following image:



The values of matrix C can be calculated as, with i, j = [1, M] :

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj},$$

The benefit of this algorithm is the ease of understanding, unfortunately, it performs very poorly as the matrix size increases. The pseudo-code for this algorithm is shown in the following image. The time complexity is O(n^3):

Algorithm 1 Compute the product of two n x n matrices

```
procedure NAIVE-MATRIX-MULTIPLY($(A, B)$)
    $n = A.rows$
    let $C$ be a new $n \times n$ matrix
    for i = 1 to n do
        for j = 1 to n do
            $c_{ij} = 0$
            for k = 1 to n do
                $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
            end for
        end for
    end for
    return $C$
end procedure
```

2.2. Line Multiplication

Let A, B and C be matrices of dimensions MxM, where C = A*B, and i, j = [1, M]. With the line multiplication algorithm, an element of column i from matrix A is fixed, then this element is multiplied by each element of row i of matrix B, computing the value of C_{ij} step by step. Next, follows a simple pseudo-code:

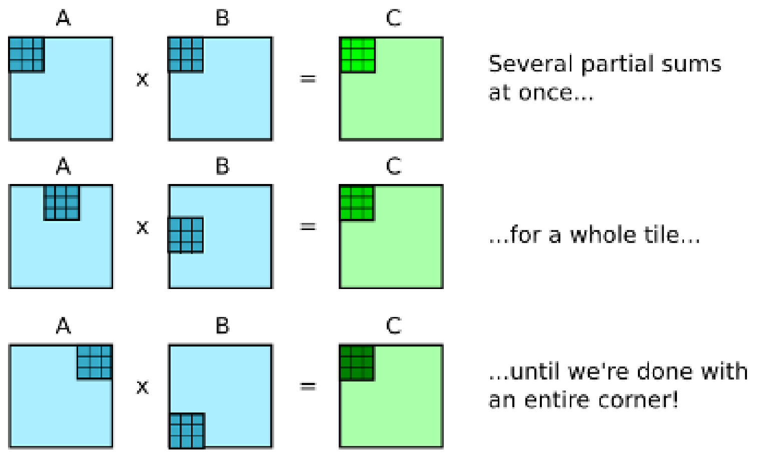
```
1.  for i = 0 to M do
2.       for j = 0 to M do
3.            for k = 0 to M do
4.                 c[i,k] ←c[i,k] + a[i,j] + b[j,k]
```

This algorithm, although not very different from the first, is more efficient, since it does not load onto memory data that will be unneeded soon. The complexity, however, is equal to the Line Multiplication algorithm, $O(n^3)$.

2.3. Block Multiplication

Let A, B and C be matrices of dimensions MxM, where C = A*B, and i, j = [1, M].

The block algorithm, finally, divides matrices A and B into smaller blocks, of coordinates [i, j] and [j, i], respectively, then we treat each block as a smaller matrix and multiply it by the corresponding smaller block. The matrix C block, with coordinates [i, j], is then calculated step by step, until it is complete. The following image illustrates this process:


A
B
C
x
=
Several partial sums at once...
...for a whole tile...
...until we're done with an entire corner!

The code for this method is harder to understand but optimizes the cache memory loads. It has time efficiency $O(n^3)$, as well as the other algorithms. The following pseudo-code was used as an implementation guide for our OnMultBlock function:

Algorithm 3 Block Multiplication

```
1:  procedure FINDSMALLEST(index, blockSize, size)
2:      if $index + blockSize > size$ then
3:          return $size$
4:      else
5:          return $index + blockSize$
6:  procedure BLOCK MULTIPLICATION(a, b, blockSize)
7:      for $jj = 0$ to $length(a)$ in $blockSize$ increments do
8:          for $kk = 0$ to $length(a)$ in $blockSize$ increments do
9:              for $i = 0$ to $length(a)$ do
10:                 for $j = jj$ to $FindSmallest(jj, bkSize, length(a))$ do
11:                     for $k = kk$ to $FindSmallest(kk, bkSize, length(a))$ do
12:                         $c_{i,k} \leftarrow c_{i,k} + a_{i,j} + b_{k,k}$
```

It is important to state that all shown algorithms perform $2n^3$ arithmetic operations, since n elements of row i of matrix A are multiplied by col j of matrix B (n). Then this is repeated for all columns of Matrix B (n) as well as all rows of matrix A (n). Lastly, the factor of 2 accounts for each C[i, j] addition ($2*n*n*n = 2n^3$).

3. Performance Metrics

To evaluate the performance, the algorithms were implemented in two different programming languages: C++ and Lua. In the Lua programming language, only the execution time of Naive Multiplication algorithm and Line Multiplication Algorithm was registered.

With C++, alongside the register of execution time, the Performance API was also used in all 3 algorithms. Data collected includes:

- L1 Data Cache Misses (L1_DCM)
- L2 Data Cache Misses (L2_DCM)
- L2 Data Cache Accesses (L2_DCA)
- Double Precision Floating Point Operations (DP_OPS)

This range of metrics allows us to compare how different cache misses between L1 and L2 are. This is important because, although L1 cache solves cache misses faster, L2 cache is larger in size.

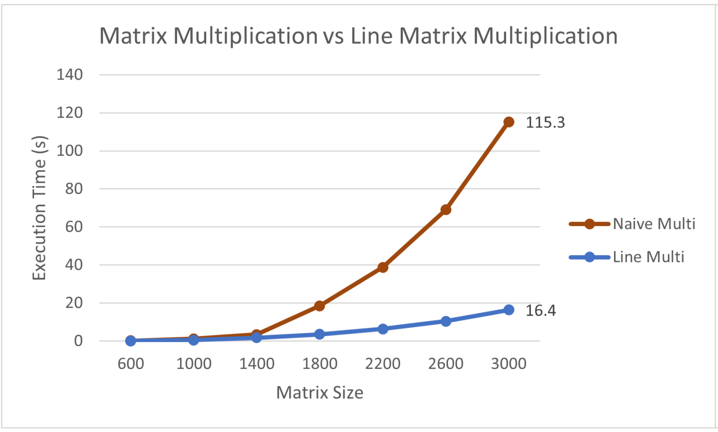
Beyond that, we also register the number of L2 data cache accesses, to better understand the rate of unsuccess when reading information from the L2 cache.
Lastly, and most importantly, the number of Double Precision Floating Point Operations, is also monitored. Since matrix values are of type *double*, the number of DP_OPS assists in calculating the time required to perform a single Floating Point Operation (FLOP / sec = DP_OPS / Execution Time).

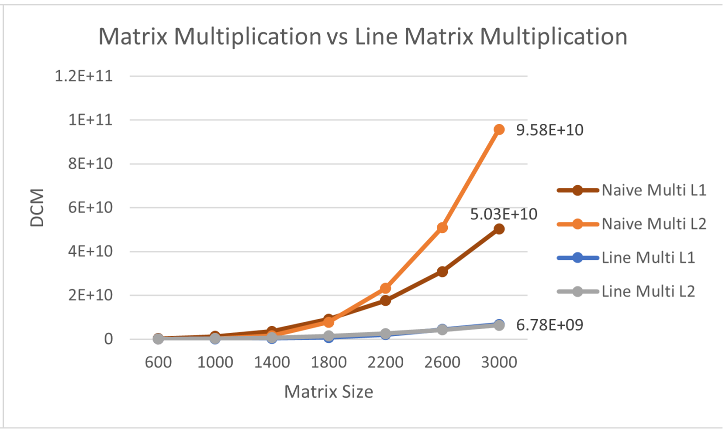
To ensure consistency in the gathered data, we decided to run each test 4 times, for the same matrix and block size. All experiments were tested on a computer running the Ubuntu operating system, with an Intel Core i7-9700 3.00GHz processor.

4. Results and Analysis

4.1. Algorithm Comparison

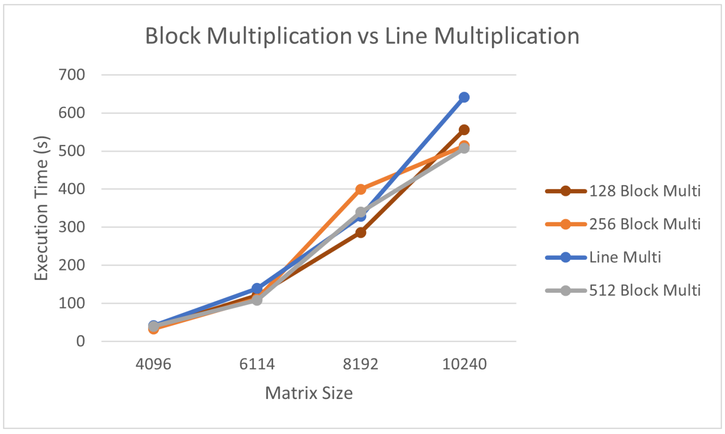
The first following graph compares our results of **Time** for **Naive Multiplication** and **Line Multiplication**. With a single look it becomes very apparent that the Line algorithm is by far the faster one, which can be credited to its use of the cache memory, contracted by Naive's need of getting the values from memory in every iteration. The second graph sustains this statement, as it compares the **Direct Cache Misses** from both algorithms and gives us the same conclusion. From the second graph we can also see the differences between the L1 and L2 caches, which were almost none with the Line algorithm, but show a significantly smaller number of misses in L1 for the Naive algorithm. This is caused by the fact that L2 has a much larger addressing space, which makes it more useful for storing bigger blocks with fewer uses, causing a bigger number of misses.

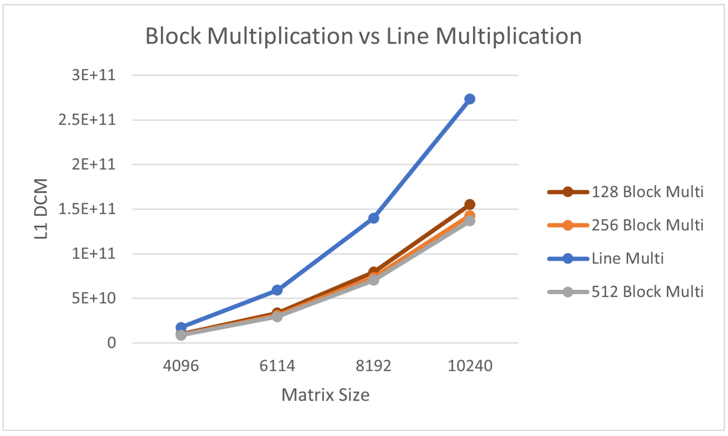




The next two graphs compare the **Line** and **Block Multiplication** algorithms in terms of **Time** and **L1 Direct Cache Misses**, respectively, and a line for each Block Size tested: 128, 256 and 512. We have removed the L2 DCMs for the second graph as it shows proportional results to L1. It is easy to see the advantage of Block Multiplication over Line Multiplication, as it has a better use of cache hits, with lower cache misses shown in the second graph.

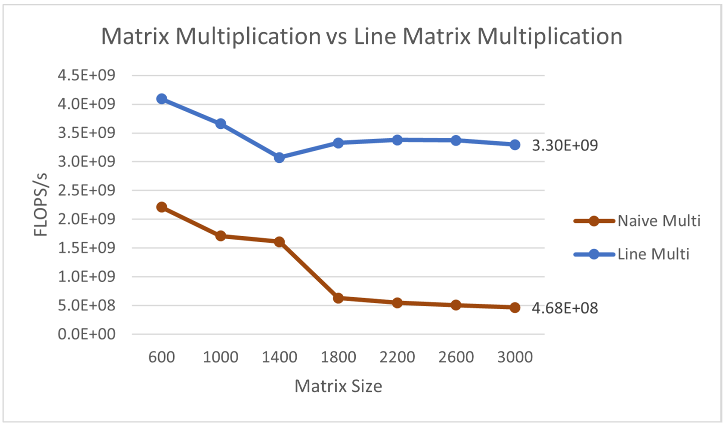
It is also noticeable that the **block size** is reversibly proportional to the number of cache misses, which translates into a faster time for the **256** and **512** blocks for the 10240 sized matrix, but doesn't seem to be relevant enough with smaller matrices, such as the 8192 sized one, where the **128** block algorithm out-performed the other two in terms of time, even with a higher number of cache misses.

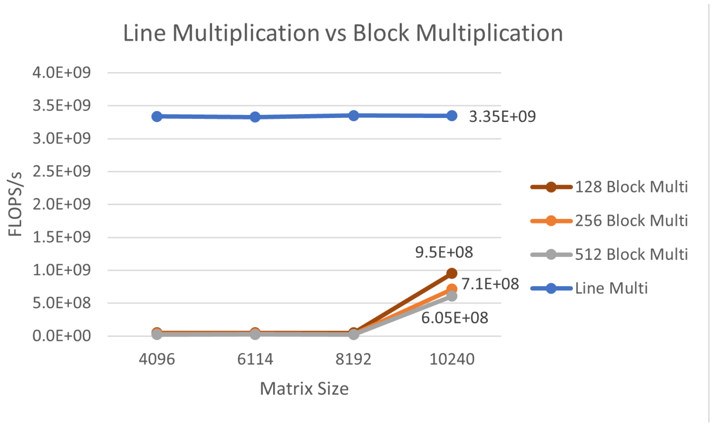




For these next two graphs, we used the **DP_OPS** and divided it by the **Execution Time** in order to get the **FLOPs/s** as stated previously, and compared it for the different algorithms (and block sizes). As shown, the Line Multiplication algorithm does far more operations on average per second in every case, even when compared to the block algorithms that out-performed it in terms of time. This leads us to conclude that fewer operations over time can be better (when comparing Naive with Line), but does not ensure faster execution time (when comparing Line with Block), because it can be an indicator of underuse of cache and repetition of calculations.

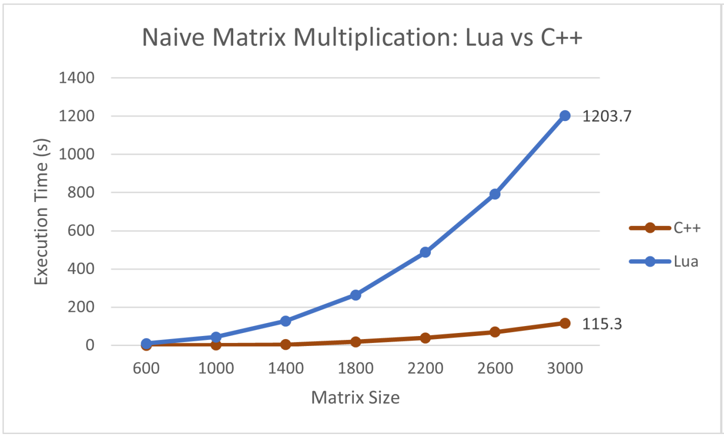
It is also noticeable that in the first graph the FLOPs/s get smaller as the matrix size increases, especially for the smaller matrices. This contrasts with the second graph, where it is clear that the use of a much larger matrix causes the average operations per second to increase significantly for the block algorithms, which can no longer rely on the cache to store most of the results it's going to need.

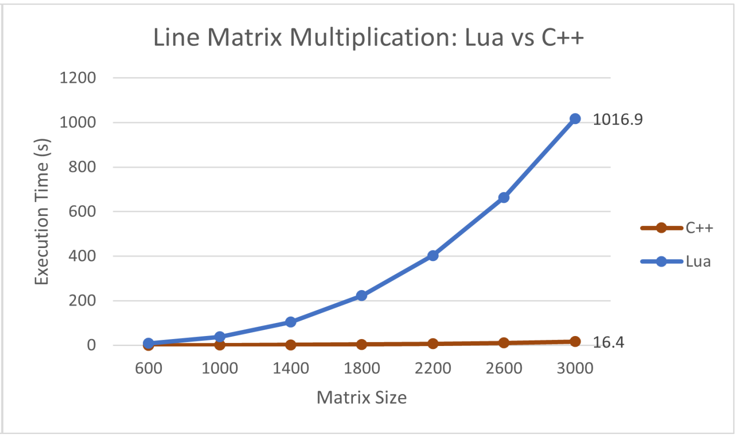




4.2. Algorithm Speed Comparison: C++ vs Lua

To finish our data analysis, we have two graphs that compare **C++** and **Lua** in terms of **Execution Time** for the **Naive** and the **Line** multiplication algorithms respectively. Before starting this project we had known that Lua tends to perform similarly to Python, so it is no surprise that C++ is the faster language, as the Lua compiler causes the execution time to increase exponentially with the increase of the matrix size for both of the tested algorithms.





5. Conclusions

The study of the performance of a single core was an insightful task, giving us a better understanding of the impact of cache memory on execution time, and the different types of cache memory. The order at which a programmer writes code and computations are made are great influences over the overall measures of performance.

Furthermore, the impact of the programming language is also to be noted, since traditionally compiled languages like C++ are faster and make use of memory more efficiently than JIT compilation languages, such as Lua.

Lastly, we now have a better understanding of the limitations of the Big-O Notation, because although all algorithms have O(n^3) complexity, they perform very differently under the tested conditions.

6. References

https://medium.com/@kilichbekhaydarov/toward-an-optimal-matrix-multiplication-algorithm-4f024baa1206

https://en.wikipedia.org/wiki/Matrix_multiplication

https://www.quora.com/What-is-the-best-way-to-multiply-two-matrices-in-C++

https://iitd-plos.github.io/col729/lec/matrix_multiplication.html

https://malithjayaweera.com/2020/07/blocked-matrix-multiplication/

https://www.differencebetween.com/difference-between-l1-and-vs-l2-cache/