FEUP

LEIC-2022/23

CPD project 1

Performance evaluation of a single core

Diogo Almeida – 202006059

Rafael Morgado – 201506449

Problem description and algorithms explanation

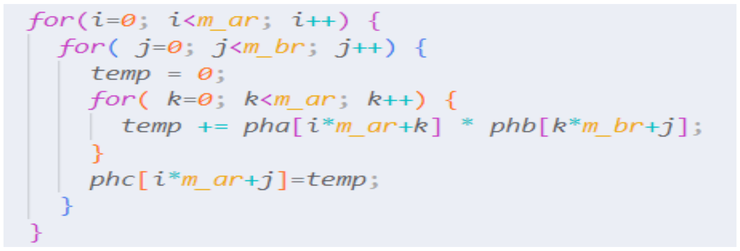
The main objective of this project was to assess the performance of the memory hierarchy of the processor when trying to access large amounts of data.

Three different variants of matrix multiplication were used, alongside the use of the Performance API (PAPI) to collect data.

Simple multiplication

This is a simple linear algebra algorithm that multiplies one row from the first matrix by each column of the second matrix.

This algorithm was implemented in both C++ and Java, they both follow the same pseudo code below.

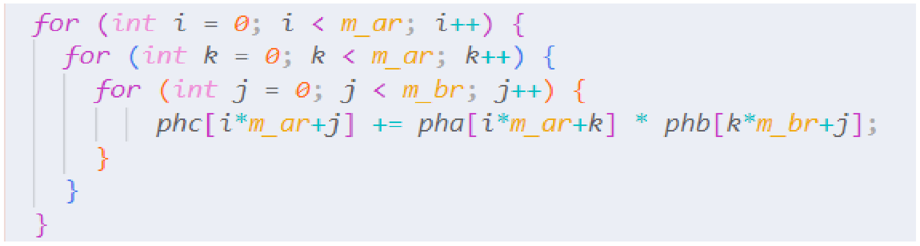


```
for(i=0; i<m_ar; i++) {
    for( j=0; j<m_br; j++) {
        temp = 0;
        for( k=0; k<m_ar; k++) {
            temp += pha[i*m_ar+k] * phb[k*m_br+j];
        }
        phc[i*m_ar+j]=temp;
    }
}
```

The code was tested in both languages using matrixes starting in 600x600 that went up to 3000x3000 in increments of 400 in both dimensions.

Line multiplication

A variation of the previous algorithm where we instead multiply elements from the first matrix by their corresponding line on the second matrix. This algorithm, like the previous, was implemented in both C++ and Java, following the pseudo code below.



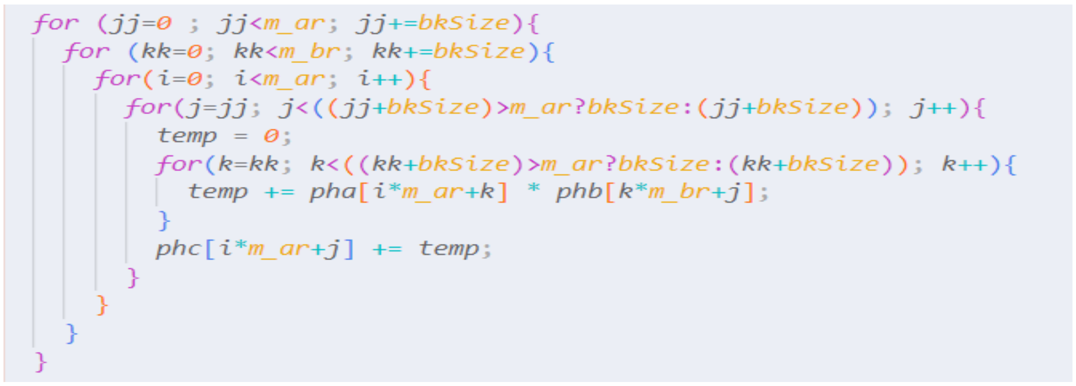
```
for (int i = 0; i < m_ar; i++) {
    for (int k = 0; k < m_ar; k++) {
        for (int j = 0; j < m_br; j++) {
            phc[i*m_ar+j] += pha[i*m_ar+k] * phb[k*m_br+j];
        }
    }
}
```

Besides running the same tests we did on the first algorithm in both languages, for C++, we also tested matrixes starting at 4096x4096 going up to 10240x10240 with increments of 2048.

Block multiplication

In this algorithm, each matrix is divided into a defined number number of blocks, and then the multiplication is carried out the same way it was for the line multiplication.

This algorithm was only implemented in C++, following the pseudo code below, the tests were from 4096x4096 to 10240x10240 in increments of 2048, the same round of tests was carried out for block sizes of 128, 256 and 512.



```
for (jj=0 ; jj<m_ar; jj+=bkSize){
    for (kk=0; kk<m_br; kk+=bkSize){
        for(i=0; i<m_ar; i++){
            for(j=jj; j<((jj+bkSize)>m_ar?bkSize:(jj+bkSize)); j++){
                temp = 0;
                for(k=kk; k<((kk+bkSize)>m_ar?bkSize:(kk+bkSize)); k++){
                    temp += pha[i*m_ar+k] * phb[k*m_br+j];
                }
                phc[i*m_ar+j] += temp;
            }
        }
    }
}
```

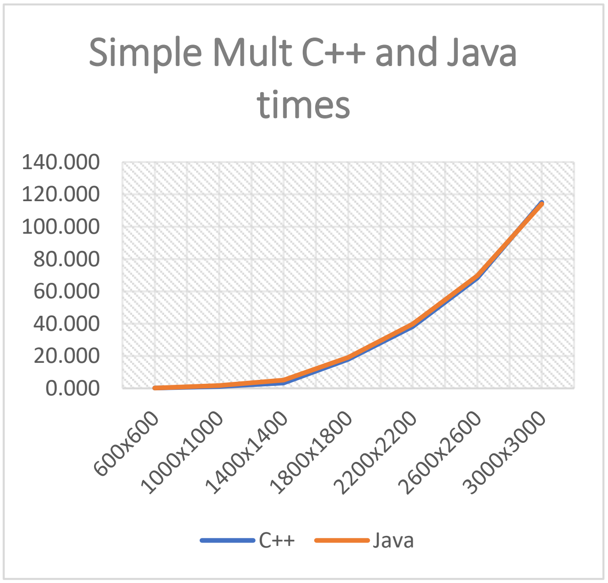
Performance metrics

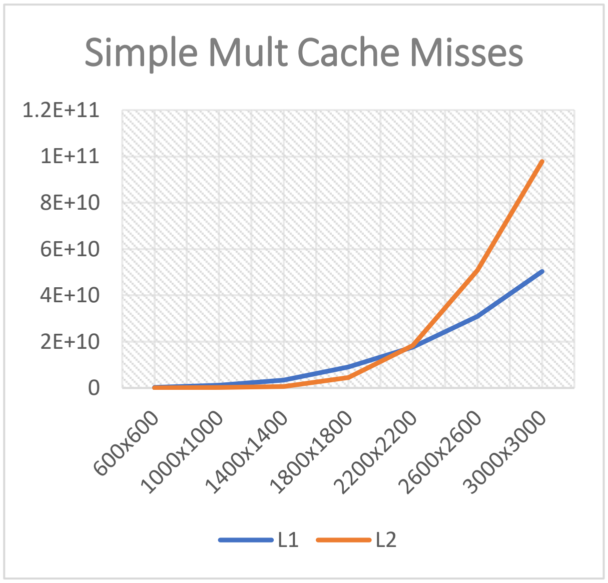
Our analysis on C++ was based on PAPI’s data cache miss metrics, for Level 1 and Level 2 caches.

Since PAPI only works in C/C++, to have a basis for comparison with Java, we also measured the time it took to complete each multiplication using all three algorithms, and different matrix sizes, apart from the block multiplication, that was only implemented in C++.

Results and analysis

<u>Simple multiplication</u>

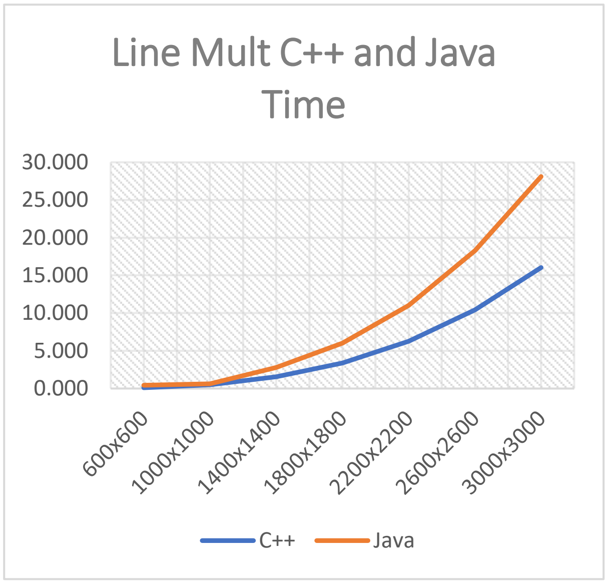


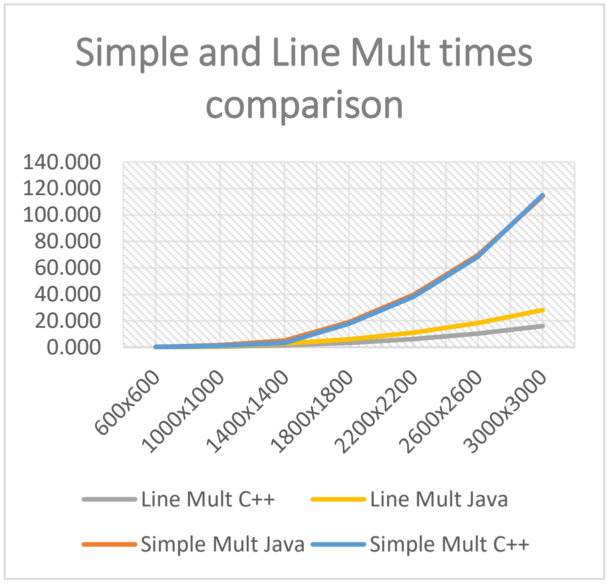


For this algorithm we can conclude that the language is not in itself a determining factor of the performance, since we see that in both languages, C++ and Java, the times were pretty much identical all throughout the tests.

As for the amount of cache misses, their amount and growth with the increasing size of each matrix, is to be expected, since more memory accesses occur.

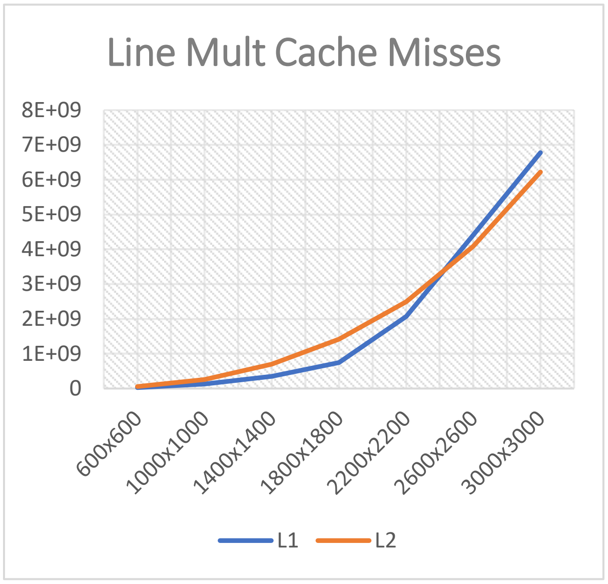
Line Multiplication

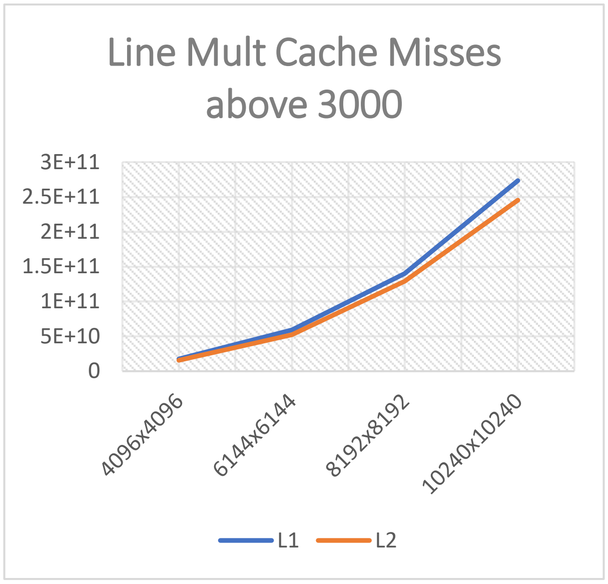




With the above graphics for this algorithm's implementations, we can see that with an increasing size of the matrixes there is a time discrepancy between the C++ and java implementations, this can, maybe, be attributed to the fact that Java runs on a virtual machine, whereas C++ doesn't, meaning it has an easier time accessing memory.

Even with this in mind, we can also see a tremendous decrease in times for the line multiplication compared to the simple multiplication, which is to be expected, since we are now using memory allocation, meaning that during execution the processor doesn't only use the value on a single position but also the neighbor's values, the principle of memory locality.

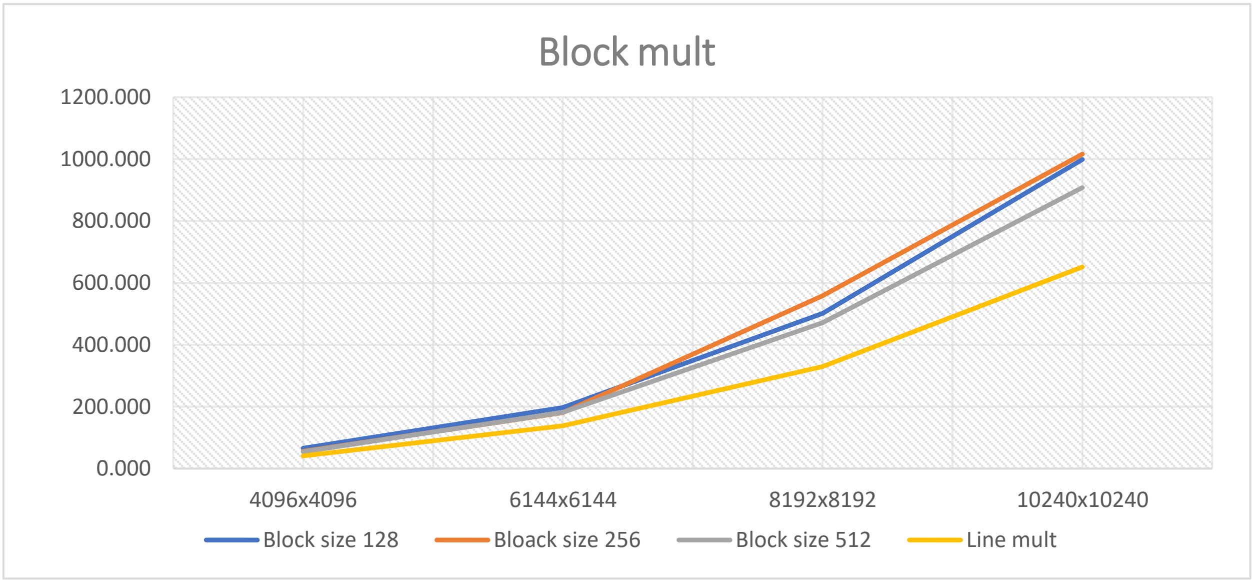




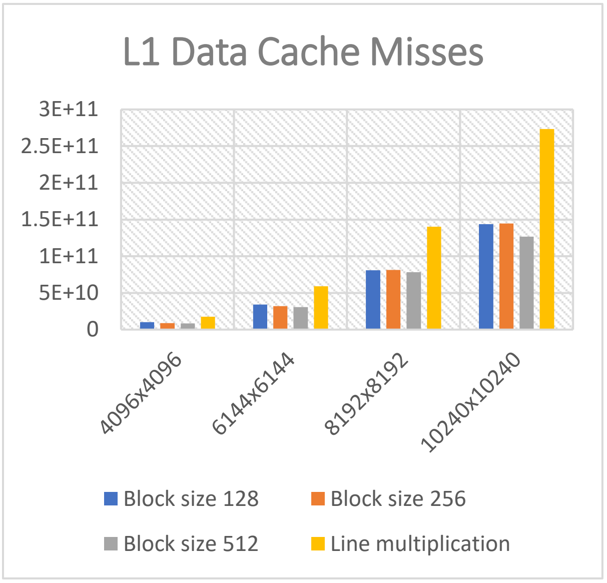
In terms of cache misses, there was a significant improvement going from the simple multiplication to the line multiplication, this is still due to the fact, that, we use memory allocation for the later one.

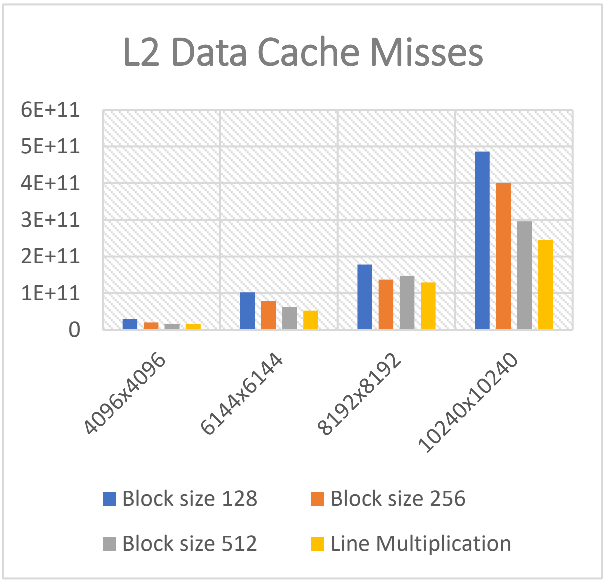
To reach the same level of data cache misses as the simple multiplication, we have to expand the matrixes to 4096x4096 and beyond.

Block multiplication



Using different sized blocks for each matrix doesn't seem to have improved the performance, time wise, but we can see a slight regression when compared to the line multiplication, this could be due to our implementation of the block multiplication not being as optimized as we would have hoped so.





With the above graphics we can see a definitive improvement in the data cache misses, this is to be expected since we are dividing the matrixes in blocks, hence reducing the amount of memory calls made by the processor, in contrast to the line multiplication, even if inside each block we are using an algorithm similar in nature to the line multiplication one.

Conclusion

In review, we can conclude that the language doesn't in of itself define the efficiency or performance of a program, taking into consideration different algorithms is a must when developing programs that deal with large amounts of data, since better ways of dealing with memory access, like taking into account the layout of memory and using the principle of memory locality, will inevitably lend us a better result.