FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Performance evaluation of a single core

(Report · 1st Project)

Parallel and Distributed Computing · L.EIC028

2022/2023 · 1st Semester

Class: 3LEIC10

Anete Pereira	up202008856
Daniela Tomás	up202004946
Diogo Nunes	up202007895

1 - Introduction

In this project, we will study the effect on the processor performance of the memory hierarchy when accessing large amounts of data.
The product of two matrices will be used for this study. We used the Performance API (PAPI) to collect relevant performance indicators of the program execution.

For this study, three different multiplication algorithms were implemented in C++, with two of them also being implemented in Java, for performance comparison reasons.

2 - Algorithms

2.1 - The Matrix Multiplication Algorithm

This algorithm uses the standard mathematical definition of matrix multiplication, where it multiplies each element of a line by an element of a column, and sums all of the results.

```
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;
    }
}
```

Since this algorithm is just 3 nested for loops, it has a time complexity of $O(n^3)$.

2.2 - The Matrix Line Multiplication Algorithm

This algorithm is a slight variation of the standard matrix multiplication, where it only multiplies a single element of the first matrix with a line of the second.

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

The main difference here, is swapping the order of the for variables, namely swapping k with j.

2.3 - The Matrix Block Multiplication Algorithm

This code uses the block matrix multiplication algorithm to compute the multiplication between two matrices A and B. Firstly, it partitions each matrix into smaller square blocks of the same size, making submatrices, then proceeds to perform the matrix multiplication between each corresponding block of A and B, and accumulates the result in the corresponding block of matrix C.

```
    for(i=0; i<m_ar; i+=bkSize){
        for(j=0; j<m_br; j+=bkSize){
            for(k=0; k<m_ar; k+=bkSize){
                // Multiply block A(i,k) by block B(k,j) and accumulate in block C(i,j)
                for(ii=i; ii < min(i+bkSize, m_ar); ii++){
                    for(kk=k; kk < min(k+bkSize, m_br); kk++){
                        for(jj=j; jj < min(j+bkSize, m_ar); jj++){
                            phc[ii*m_ar+jj] += pha[ii*m_ar+kk] * phb[kk*m_br+jj];
                        }
                    }
                }
            }
        }
    }
```

Since there are three nested for loops iterating over the dimensions of the matrices, and the inner loop performs a constant amount of work, this algorithm has a time complexity of $O(n^3)$.

3 - Performance Metrics

All tests were performed on machine with an i7-9700 CPU.
In order to measure performance we mainly used the 3 following metrics.

3.1 - Time

Time is a fairly self explanatory metric which we will use for comparing different algorithms and languages.

3.2 - Flops

Floating point operations per second is a very useful metric when comparing arithmetic computing speed between 2 different machines, or even 2 different algorithms in the same machine (which is the way we will use it).

3.3 - DCM

Short for *Data Cache Miss*, DCMs occur when data that is requested by the program isn´t located in one of the CPU´s caches(L1,L2) which makes it necessary the fetch it from slower memory sections of the computer(RAM, Hard drives), slowing down the whole process, as the CPU has to wait for the data to keep executing.
Keeping this in mind, minimizing this phenomenon with the use of *Spatial Locality* can often speedup algorithms.

4 - Results and analysis

4.1 - Multiplication

C++

Dimensions	Time(s)	L1 DCM	L2 DCM
600x600	0.187	244718761	41902475
1000x1000	1.389	1235411156	312282492
1400x1400	3.334	3511150113	1402387945
1800x1800	18.026	9087512086	7413532605
2200x2200	38.553	17623982103	23654144405
2600x2600	69.616	30907968749	51588632366
3000x3000	121.019	50293286558	95592567359

Java

Dimensions	Time(s)
600x600	0.443
1000x1000	2.368
1400x1400	7.952
1800x1800	19.369
2200x2200	40.507
2600x2600	71.014
3000x3000	115.873

Java vs C++

In the graphs below we can see the difference between the naive matrix multiplication algorithm in the 2 languages.
We can conclude that both languages are essentially tied with each other, with C++ being just an insignificant bit faster.

DCM and Gflops

Here we can see that DCMs rise with matrix size and particularly L2 DCM overtakes L1 DCM, which mean an increase in data fetching from slower memory subsystems, leading to a decrease in flops that we can see in the graph below.

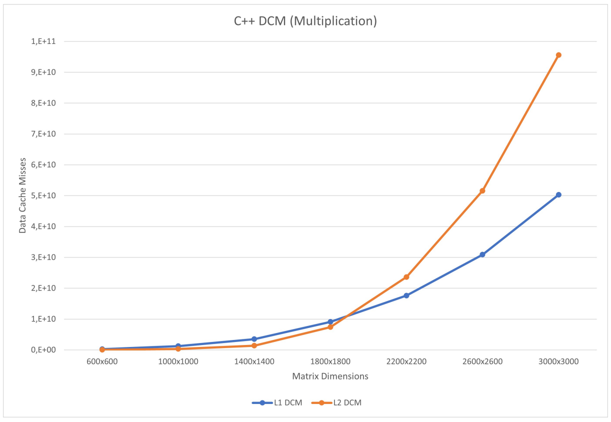

Figure 1: *C++ DCM for multiplication*

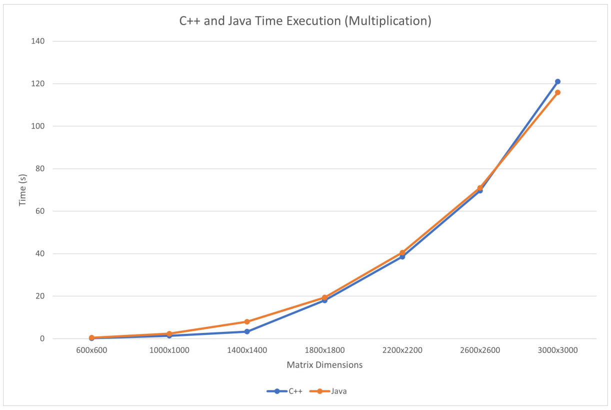

Figure 2: *C++ and Java time execution for multiplication*

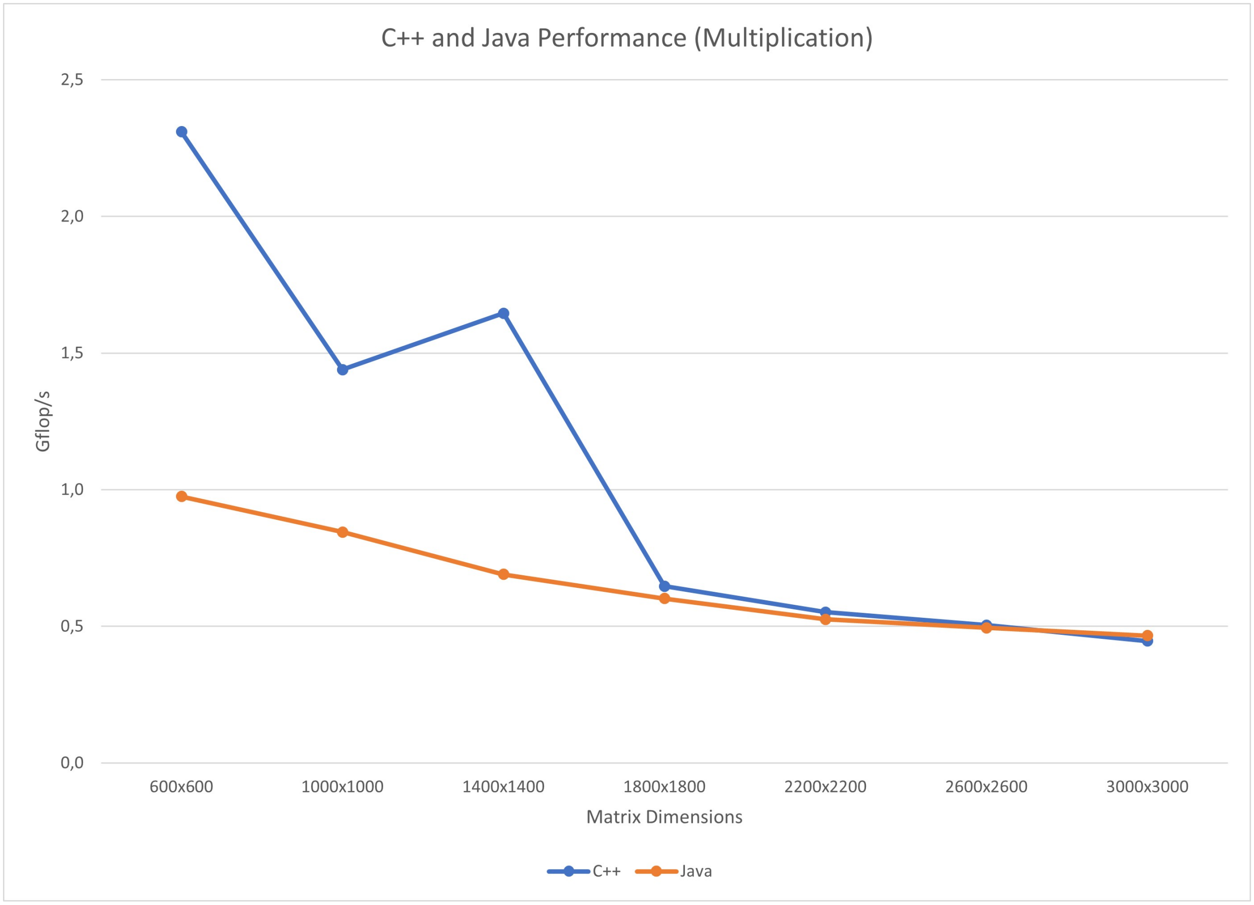

Figure 3: *C++ and java performance for multiplication*

4.2 - Line Multiplication

C++

Dimensions	Time(s)	L1 DCM	L2 DCM
600x600	0.100	27114212	58101280
1000x1000	0.470	125772373	268539718
1400x1400	1.561	346331640	713792916
1800x1800	3.420	746419914	1453999434
2200x2200	6.250	2075600141	2576239637
2600x2600	10.370	4413262686	4216332536
3000x3000	15.984	6780816623	6412745757

Java

Dimensions	Time(s)
600x600	0.355
1000x1000	1.617
1400x1400	4.653
1800x1800	9.980
2200x2200	14.846
2600x2600	24.858
3000x3000	38.106

Comparison To Previous Algorithm

Here we can see that, despite having the same theoretical time complexity this algorithm is consistently much faster than the previous naive implementation, by a good margin.
For example we can see that in C++, when using a 3000x3000 matrix the algorithm completed in only 13.2% of the original implementation´s time.
We can also note that for most matrix sizes, L1 DCM decreases when compared to naive multiplication, which makes sense in correlation with the speedup received, meaning we are making better use of spatial locality than before.

Java vs C++

Here we can see that, the gap between languages has widened, now making C++ the clear winner in terms of speed.
Still, both languages became significantly faster than with the naive algorithm.

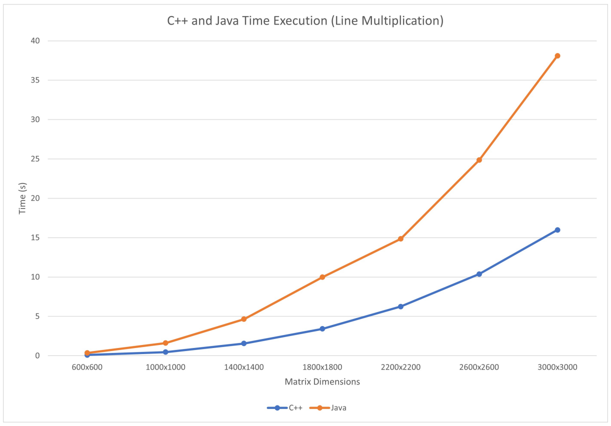

Figure 4: *C++ and java time execution for line multiplication*

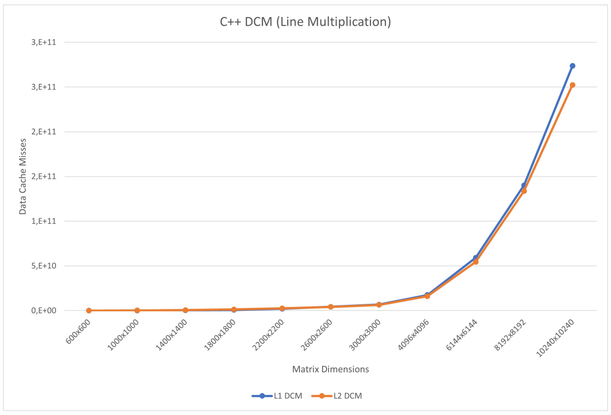

Figure 5: *C++ DCM for line multiplication*

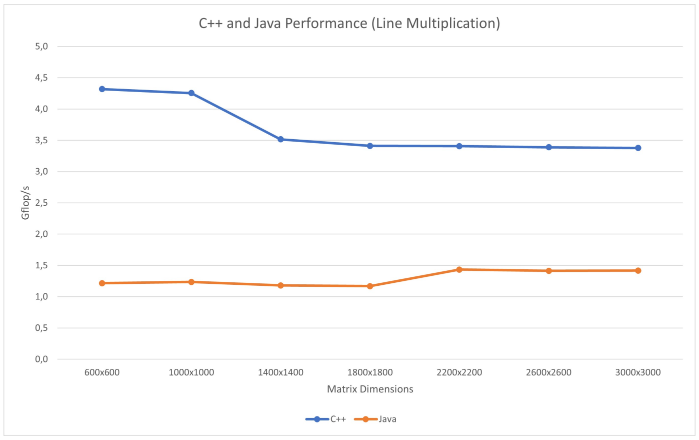

Figure 6: *C++ and java performance for line multiplication*

4.3 - Block Multiplication (C++)

Line Multiplication

Dimensions	Time(s)	L1 DCM	L2 DCM
4096x4096	40.697	17541636687	16190690724
6144x6144	140.566	59137220919	54504610835
8192x8192	341.137	140166998314	133727816732
10240x10240	648.109	273729501559	252408569867

Block Size = 128

Dimensions	Time(s)	L1 DCM	L2 DCM
4096x4096	34.250	9680256809	32602616307
6144x6144	116.651	32577777594	111533277601
8192x8192	310.356	74438585371	259553964368
10240x10240	575.867	150700435536	503800161645

Block Size = 256

Dimensions	Time(s)	L1 DCM	L2 DCM
4096x4096	27.565	9070988565	23263665193
6144x6144	99.681	30592449742	78551730746
8192x8192	380.954	72158343887	175955648756
10240x10240	475.594	141688358033	350381128618

Block Size = 512

Dimensions	Time(s)	L1 DCM	L2 DCM
4096x4096	26.272	8778821001	19775022694
6144x6144	97.542	29673533047	66648766795
8192x8192	331.375	70520938460	153628742576
10240x10240	433.33	137020608506	303831306641

Comparison to previous algorithm

Due to the fact that the Block Multiplication Algorithm takes advantage of the memory layout and divides the matrices into smaller submatrices, in large matrices it is more efficient than the Line Multiplication Algorithm. The Block Multiplication Algorithm allows a better utilization of the CPU cache reducing the number of memory accesses, which reduces the processing time and number of data cache misses.

Different Block Sizes

For different block sizes (128, 256 and 512) we can see that the performance, measured in Gflops, increases when the block size increases. This happens because larger block sizes reduces the number of blocks needed to cover the entire matrix, so the number of block multiplications required to compute the final result decreases.

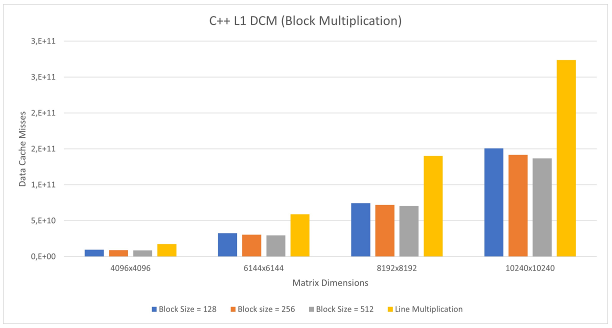

Figure 7: *C++ L1 DCM for block multiplication*

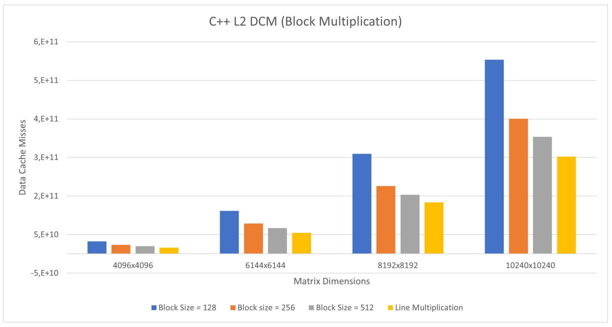

Figure 8: *C++ L2 DCM for block multiplication*

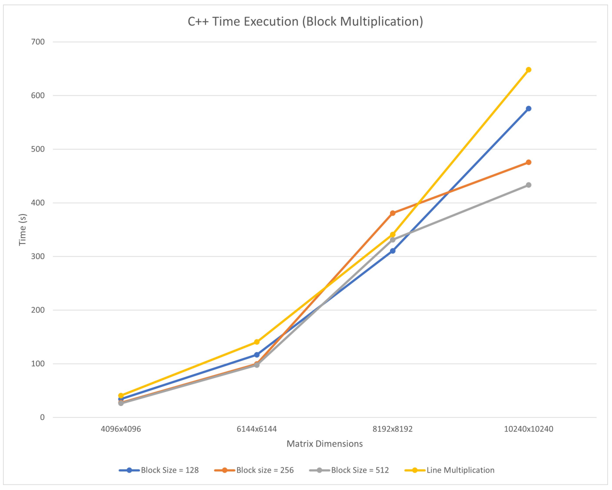

Figure 9: *C++ time execution for block multiplication*

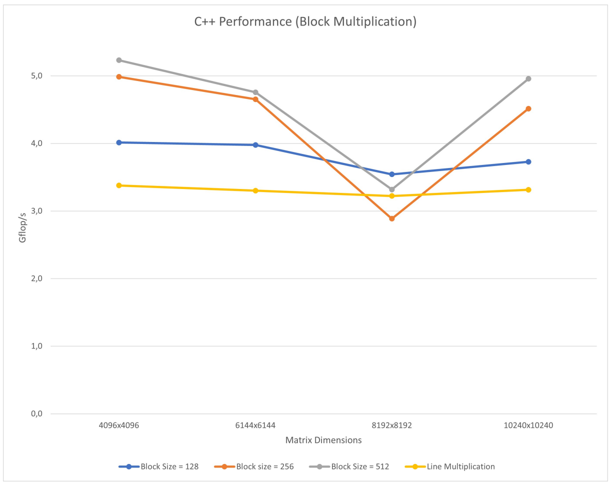

Figure 10: *C++ performance for block multiplication*

5 - Conclusion

The goal of this project was to compare different versions of the same algorithm in their efficiency, the effect of data locality and impact of chosen language.

As expected, more clever algorithms that make good use of CPU cache can offer better real world speed, even though theoretically they have the same time complexity and also, lower level languages, closer to the hardware can often offer speed advantages.