Performance Evaluation on a Single Core

Bachelors in Informatics and Computing Engineering

Paralel and Distributed Computation - Project 1

Class 6 Group 1

up201906086@up.pt	Marcelo Henriques Couto
up201907361@up.pt	Francisco Pinto de Oliveira
up201906355@up.pt	Rui Pedro Mendes Moreira

Março, 2022

Contents

1 Introduction

This project aims to analyze CPU performance on a single core and multiple techniques that can affect it. For this effect, different algorithms for matrix multiplication algorithms will be implemented and there effects on the processor performance analyzed. The programs used to test the algorithms are developped in Java and C++, for comparison purposes. Performance API (PAPI) enables us to fetch data directly related to the CPU activity, allowing us to better analyze the impacts of the different techniques of matrix multiplication.

2 Problem Description

The project consists on applying three different techniques on matrix multiplication and analyzing the impact of each one in the CPU performance. As such, the project was divided into three parts:

1. Download the file containing the standard matrix multiplication in C++; Implement the same algorithm in a different language (we chose Java); Register processing times in both languages for matrixes from 600x600 to 3000x3000 in increments of 400.
2. Implement in both languages a version of the algorithm using line multiplication; Repeat similar tests for this version of the algorithm; Register processing times in C++ for matrixes from 4096x4096 to 10240x10240 in increments of 2048.
3. Implement a C++ a version of the algorithms using block multiplication; Register processing times in C++ for matrixes from 4096x4096 to 10240x10240 in increments of 2048.

In all stages of development, the performance of the programs was also analyzed directly by requesting data from the processor on its execution via PAPI.

3 Algorithms Explanation

3.1 Normal Matrix Multiplication

The algorithm to use in the first part of the project (and that is already given) multiplies two matrixes represented as arrays of arrays (or simply arrays, depending on the implementation). In this project, only square matrixes will be used in the algorithms.

The algorithm is composed of three nested for loops. For the multiplication of matrix A with matrix B and result C ($C = A \times B$):

- The first for loop's variant represents A's and C's line
- The second for loop's variant represents B's and C's column
- The third for loop's variant represents B's line and A's column

Complexity Analysis

- **Time Complexity:** $O(N^3)$, where N is the line size of the matrixes
- **Space Complexity:** $O(N^2)$, where N is the line size of the matrixes

3.2 Multiline Matrix Multiplication

Multiline Multiplication is the algorithm used in the second phase of the project. This algorithm presents itself as an improvement of the previous one in terms of efficiency.

The algorihm is very similar to the previous one, with a slight change in what the loops' variants refer to. For the multiplication of matrix A with matrix B and result C ($C = A \times B$):

- The first for loop's variant represents A's and C's line
- The second for loop's variant represents B's line and A's column
- The third for loop's variant represents B's and C's column

Complexity Analysis

- **Time Complexity:** $O(N^3)$, where N is the line size of the matrixes
- **Space Complexity:** $O(N^2)$, where N is the line size of the matrixes

Although the time complexity remains the same, this algorithm's efficiency is significantly greater than the previous one due to the change in the loop's variants.

Explanation: In the previous algorithm, for every iteration of the most inner loop, a cache miss will be issued by the CPU and a different row of B matrix will be loaded into the processor's cache. This happens because the matrixes are represented as an array of lines rather than columns, which means every time the processor needs an element that is not present in the line currently loaded, it will have to get the one it belongs to from main memory. As for each iteration of the most inner loop the element from B used is always in a different line, the CPU loads a new line from main memory N^3 times.

The **Multiline Matrix Multiplication** algorithm reduces the number of cache misses by changing the order of the loops. With the new structure, the most inner loop's variant is only used in columns rather than lines, which means only for the other two loops' iterations will the processor need to load new lines. The number of cache misses reduces to N^2, thus increasing the efficiency of the process.

Note Cache miss means the variable requested is not present in cache memory

3.3 Block Matrix Multiplication

In the third part of this project, **Block Matrix Multiplication** will be used. This algorithm implements the idea of splitting a matrix into smaller matrixes and multiplying them instead.

This algorithm is composed of 6 nested for loops. The three outer loops account for the submatrixes chosen for the calculation. The three inner loops execute **Multiline Multiplication** between the submatrixes chosen.

For the multiplication of matrix A with matrix B and result C ($C = A \times B$):

- **3 Outer Loops:**
  - The first outer for loop's variant represents the fourth outer loop (first inner loop)'s variant limits, which are the lines from A and C to be selected to the submatrix A and submatrix C

  - The second outer for loop's variant represents the fifth outer loop (second inner loop)'s variant limits, which are the columns from A and lines from B to be selected to the submatrix A and submatrix B
  - The third outer for loop's variant represents the sixth outer loop (third inner loop)'s variant limits, which are the columns from B and C to be selected to the submatrix B and submatrix C

- **3 Inner Loops:**
  - The first for loop's variant represents A's and C's line
  - The second for loop's variant represents B's line and A's column
  - The third for loop's variant represents B's and C's column

Complexity Analysis

- **Time Complexity:** $O(N^3 \div B^3 \times B^3 = N^3)$, where N is the line size of the matrixes
- **Space Complexity:** $O(N^2)$, where N is the line size of the matrixes

Although the time complexity is the same, once again this algorithm is a big improvement from the previous one, specially in multiplication of larger matrixes.

Explanation: This algorithm, once again, is more efficient due to the optimization related with the number of cache misses. Some matrixes are so big that a whole line can't fit in the processor's cache. By splitting the matrix into smaller chunks that fit in it, the number of cache misses can be drastically reduced.

4 Performance Metrics

4.1 PAPI events

As previously stated, and as indicated by the project guidelines, the **Perfomance API** (PAPI) was used to collect data directly from the CPU, which we used to better analyze the performance of each algorithm. PAPI is an API that collects information on the processor's execution and performance in the form of events, such as data cache misses and cache accesses. As it uses CPU hardware counters for this purpose and so do each event, the number of events we can capture and study is limited. The CPU we worked with had 10 counters and PAPI itself uses 4, meaning we only had 6 left. For this reason, we chose to capture 4 events:

- **PAPI_L1_DCM -** Level 1 cache misses
- **PAPI_L2_DCM -** Level 2 cache misses
- **PAPI_L2_DCA -** Level 2 cache accesses
- **PAPI_L3_DCA -** Level 3 cache accesses

Our choice was based on the necessity to capture the cache miss and cache access variations from each algorithm. As explained in the previous section, the performance improvements from each algorithm to the next one base themselves mostly in reducing cache misses of the elements of the matrixes.

PAPI_L2_DCM and **PAPI_L3_DCA** are derived events, meaning PAPI needs to collect data on to other events to calculate the values for these ones. For that matter, these two events require the use of two counters each, meaning that we could only register two more normal events. We opted to register Level 3 cache accesses because there were no events available to register Level 1 cache accesses.

Note Level 1 2 and 3 cache allude to the level of proximity of the cache to a processor core.

4.2 Hardware

All the benchmarks were ran using a Intel Core i7-9750H

5 Results and Analysis

In order to analyze the algorithms, we used both PAPI events and timed the algorithms for differently sized matrixes. The data collected was saved in .csv files and transformed into graphics for better analysis.

5.1 PAPI events inconsistencies

At the beginning of the project we wanted to collect data about data cache accesses to calculate miss to access ratio. As we analysed our data we found that most of the time data cache accesses were lower than misses. Later we found that PAPI gives no guaranties that data cache misses, accesses and hits have any correlation [1] so our analysis was limited do only data cache misses. We didn't used level 3 data cache accesses because this metric is the same as level 2 data cache misses as can be seen in PAPI event description.

5.2 Matrixes from 600x600 to 3000x3000

5.2.1 Execution time and FLOPS

When comparing the different algorithms that were implemented regarding execution times and GFLOPS we can see that, as matrix size grows, **Multi-line Matrix Multiplication** is an order of magnitude more efficient than **Normal Matrix Multiplication**.

The comparison between algorithms regarding GFLOPS shows that for the same matrixes, the computer performed more almost double the operations per second when running **Multi-line Matrix Multiplication** when compared with it's counterpart. This indicates that **Multi-line Matrix Multiplication** is using the computer resources more efficiently as we will discuss below.

The tendencies described above were verified both on C++ and Java. Java being a high-level language performed worse than C++ in **Normal Matrix Multiplication**. This tendency was not verified in **Multi-line Matrix Multiplication** for this matrix sizes with Java performing as good as C++. This can be explained because the algorithm is more efficient so performance trade-offs were not observed.

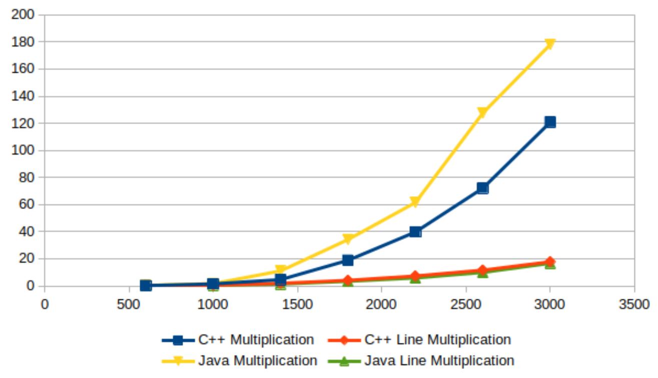

Figure 2: Execution Time (s)

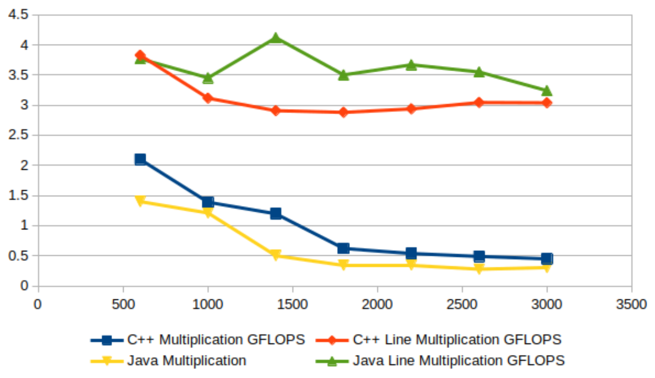

Figure 3: Float Operations (GFLOPS)

5.2.2 PAPI events analysis

The comparison of the **data cache misses** in L1 and L2 (PAPI_L1_DCM and PAPI_L2_DCM) between **Normal Matrix Multiplication** and **Multi-line Matrix Multiplication** draws us corroborate our theories: the second method of matrix multiplication avoids many cache misses. Although we expected to see a higher number of L1 cache misses than L2 cache misses, because there is some data that might not be present in L1 cache but might be in L2 cache, our data contradicts this fact.

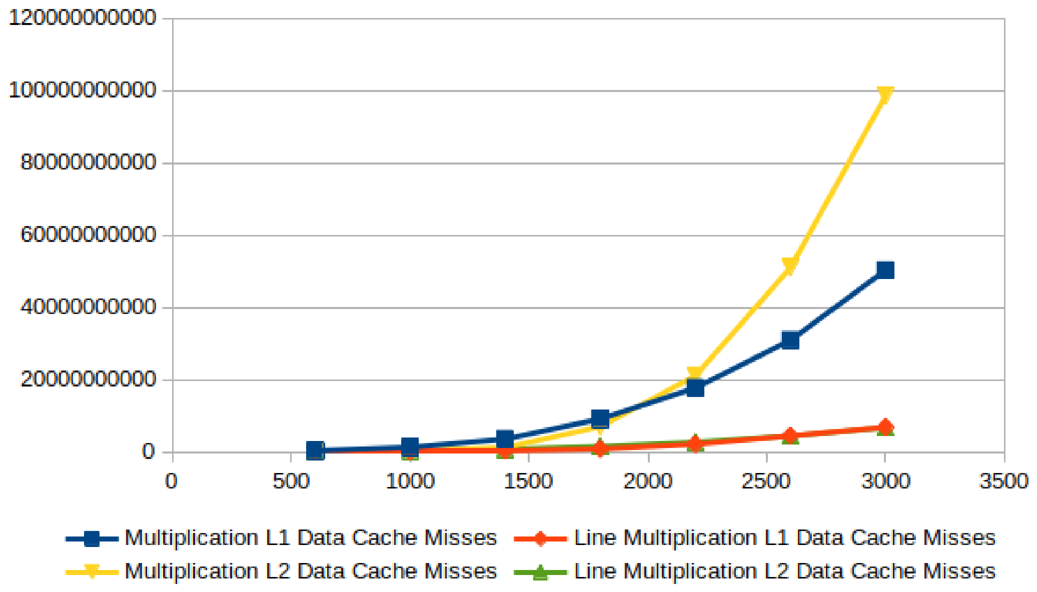

Figure 4: Number of cache misses related to matrix size

5.3 Matrixes from 4098x4098 to 10240x10240

5.3.1 Execution time and FLOPS

The comparison between **Multi-line Matrix Multiplication** and **Block Matrix Multiplication** in terms of execution time reveals that the second algorithm is more efficient independently of matrix and block size. Between different block sizes we observed that the execution time decreases when comparing 128x128 with 256x256 matrixes, and then it starts increasing again. We can conclude that 256x256 blocks seem to be the ideal size of block for this configuration. The analysis of FLOPS proved that **Block Matrix Multiplication** with a block size of 256x256 yielded the best result.

		Algorithm				
		Line Multiplication	Block Multiplication (128)	Block Multiplication (256)	Block Multiplication (512)	Block Multiplication (1024)
Matrix Size	4096	49.3273	39.7087	36.7971	46.8632	40.9488
	6144	162.744	134.351	122.605	127.923	139.881
	8192	414.092	345.589	370.028	389.386	342.124
	10240	763.512	635.891	578.957	612.065	679.643

Table 1: Execution Time (s)

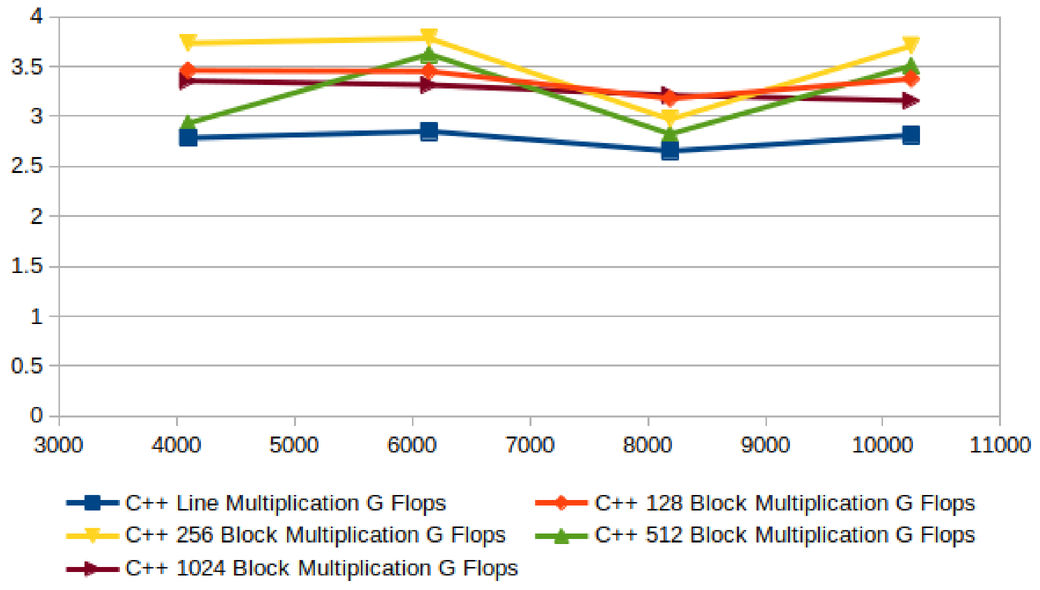

Figure 5: Float Operations (GFLOPS)

5.3.2 PAPI events analysis

As we observed with smaller matrix sizes the number of cache L2 cache misses is greater than the number of L1 cache misses. The number of cache misses in both level one and level two caches in **Multi-line Matrix Multiplication** is the double of **Block Matrix Multiplication** this justifies why **Multi-line Matrix Multiplication** is much slower than **Block Matrix Multiplication**. Between different block sizes we observed that the number of cache misses in both level of caches is decresing as the size of the block increases. This fact does not explain why execution time increases as block size increases.

		Algorithm				
		Line Multiplication	Block Multiplication (128)	Block Multiplication (256)	Block Multiplication (512)	Block Multiplication (1024)
Matrix Size	4096	17.647795712	9.613625662	9.053586218	8.766963868	8.785341456
	6144	59.561360444	32.437636037	30.540354075	29.638354628	29.690638422
	8192	141.119280713	76.943793304	72.472340236	70.217429601	70.384445555
	10240	275.441489151	150.101146037	141.423714683	137.100929723	137.52190264

Table 2: L1 Cache Misses (Miss$\times 10^9$)

		Algorithm				
		Line Multiplication	Block Multiplication (128)	Block Multiplication (256)	Block Multiplication (512)	Block Multiplication (1024)
Matrix Size	4096	17.209304763	30.272506811	22.718508038	19.089596015	18.635256131
	6144	57.841281211	107.174007311	30.540354075	64.757090769	64.494440561
	8192	136.827183508	247.599027107	177.171570622	150.843420658	150.278383529
	10240	289.717224388	495.942550318	356.950322647	305.120720325	292.778686636

Table 3: L2 Cache Misses (Miss$\times 10^9$)

6 Conclusion

In conclusion, the objectives of the work were achieved as we got to test and prove our hypothesis. To start with, the compiler isn't able to detect some data affinities and at compile time. For instance, the matrix multiplication algorithms prove further optimizations can be done in the code development when it comes to memory management. Not only that but we were also able to detect the improvement **Multi-line Matrix Multiplication** and **Block Matrix Multiplication** promess and, through the analysis of the events captured through PAPI, confirm the source of the optimizations lies on the reduction of cache misses that happen on the matrix calculation algorithm.

7 Bibliography

- [1] DCM, DCH and DCA correlation
- [2] Paralell and Ditributed Computation in FEUP
- [3] Papi Documentation