*C*PD Project 1

Performance evaluation of a single core

up202008462@fe.up.pt José Luís Cunha Rodrigues
up202004421@fe.up.pt Martim Raúl da Rocha Henriques
up202004926@fe.up.pt Tiago Filipe Magalhães Barbosa

March 23

Contents

1 Introduction

The goal of this project is to measure processor performance and relate it to memory accesses when dealing with large amounts of data. To do this, we will use the Performance API (PAPI) to collect relevant project data. The program used to gather information is a calculation of the product of two matrices.

2 Program

The program used to test core performance was the multiplication of matrices. In order to compare memory accesses, the problem was solved in different ways. For each method, the time complexity will be O(n^3) and the spacial complexity O(n^2) because we need to allocate space for the matrices. Nevertheless, the order in which the memory is accessed can be changed to optimize the different strategies, as described below.

2.1 Basic multiplication

The first approach to solving matrix multiplication followed a standard algebraic basic path: find each scalar product between lines and columns of each matrix to get the result. The algorithm is the following :

Create the result matrix C to store the result. For each row i of matrix A go to each column j of matrix B and compute the dot product of row i from A and column j from B and store it in row i and column j of matrix C.

For example :

$$\begin{array}{cc} \text{Matrix } A & \text{Matrix } B \\ \begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix} & \begin{bmatrix}9 & 8 & 7\\ 6 & 5 & 4\\ 3 & 2 & 1\end{bmatrix} \end{array}$$

$$\begin{array}{ccccccccc} \text{Matrix } C1 & & \text{Matrix } C2 & & \text{Matrix } C3 & & \text{Matrix } C4 & & \text{Matrix } C \\ \begin{bmatrix}0 & 0 & 0\\ 0 & 0 & 0\\ 0 & 0 & 0\end{bmatrix} & \rightarrow & \begin{bmatrix}30 & 0 & 0\\ 0 & 0 & 0\\ 0 & 0 & 0\end{bmatrix} & \rightarrow & \begin{bmatrix}30 & 24 & 0\\ 0 & 0 & 0\\ 0 & 0 & 0\end{bmatrix} & ... & \begin{bmatrix}30 & 24 & 18\\ 84 & 69 & 54\\ 138 & 114 & 0\end{bmatrix} & \rightarrow & \begin{bmatrix}30 & 24 & 18\\ 84 & 69 & 54\\ 138 & 114 & 90\end{bmatrix} \end{array}$$

2.2 Line by Line

For the second exercise, the method used was to multiply each element from the first matrix by a line of the second matrix. This way the values in the result matrix are filled incrementally, so as to spare memory access on the first matrix.The algorithm is the following :

Create the result matrix C to store the result. For each element of matrix A go to each row of matrix B and compute the product of element from A and all elements of row from B and store it in row i and column j of matrix C , where i corresponds to the row of element from matrix A and j corresponds to the column of element from matrix B.

For example :

$$\begin{array}{cc} \text{Matrix } A & \text{Matrix } B \\ \begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix} & \begin{bmatrix}9 & 8 & 7\\ 6 & 5 & 4\\ 3 & 2 & 1\end{bmatrix} \end{array}$$

$$\begin{array}{ccccccccc} \text{Matrix } C1 & & \text{Matrix } C2 & & \text{Matrix } C3 & & \text{Matrix } C4 & & \text{Matrix } C \\ \begin{bmatrix}0 & 0 & 0\\ 0 & 0 & 0\\ 0 & 0 & 0\end{bmatrix} & \rightarrow & \begin{bmatrix}9 & 8 & 7\\ 0 & 0 & 0\\ 0 & 0 & 0\end{bmatrix} & \rightarrow & \begin{bmatrix}21 & 18 & 15\\ 0 & 0 & 0\\ 0 & 0 & 0\end{bmatrix} & \rightarrow & \begin{bmatrix}30 & 24 & 18\\ 0 & 0 & 0\\ 0 & 0 & 0\end{bmatrix} & ... & \begin{bmatrix}30 & 24 & 18\\ 84 & 69 & 54\\ 138 & 114 & 90\end{bmatrix} \end{array}$$

2.3 Block multiplication

Lastly, block multiplication tries to take advantage of neighboring memory positions. To do that, the matrix is divided into smaller chunks, so the operations become more lightweight. In the end, the result is the sum of the different smaller matrices.

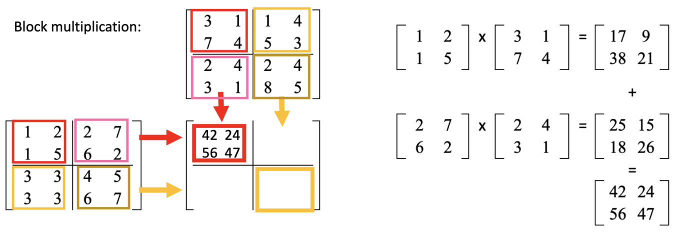


Figure 1: Block Multiplication overview

3 Performance metrics

For each algorithm, several runs with increasingly sized matrices were made as per suggested in the project outline. The tests were all run in C++ and some of them in Java as well. To measure performance, we took into account the execution time, as well as the Data Cache Misses (DCM) on levels 1 and 2 for C++ runs. Furthermore, we used the following metrics from the performance API (PAPI) to get a better insight into the tests' performance: number of cycles, number of instructions, and number of floating point operations. Note that, because the time complexity is O(n^3) and there are two FLOPS per iteration, the total number of FLOPS will be $2n^3$. This value was confirmed empirically with the test's results.

To better understand the data we calculated misses per flop (MPF) and cycles per instruction (CPI):

$$MPF = \frac{L1\_DCM + L2\_DCM}{FLOPS}$$

$$CPI = \frac{Cycles}{Instructions}$$

The former relates the usage of cache with actually performed operations. A higher value means suggests worse cache management. The latter metric serves as an indicator of CPU usage.

All the tests were performed on one of the lab's machines, running an Intel Core i7-9700 CPU at 3.00GHz with 8 cores and 16GB of RAM. The cache for these processors is 12MB.

4 Results and Analysis

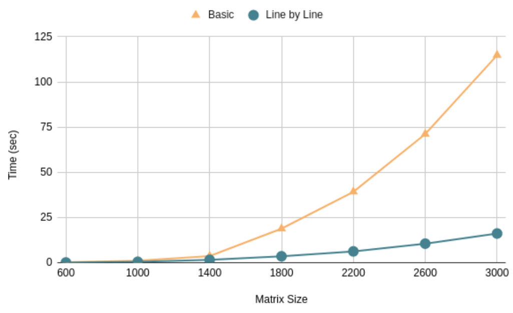
In this section, we provide explanations for the results obtained and some data visualization. For the complete set of raw obtained results, refer to section 5.

4.1 Algorithm Comparison

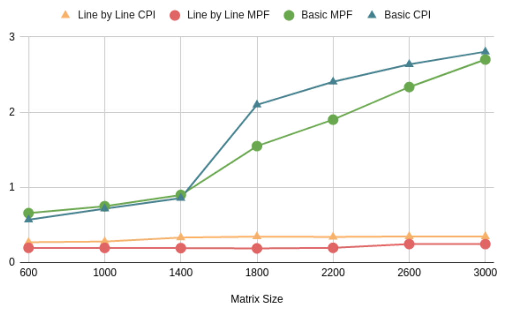
Taking a look at the results obtained for the Basic and Line Multiplications (Figure 2), it can be concluded that the later is more efficient, as suggested. A cache miss will force the CPU down in the memory hierarchy searching for a value and thus spending extra time. Bearing this in mind, it comes to reason that ideally, we would make all the accesses to a memory position as close as possible, so as not to risk it being removed from the cache when it's needed again.

Because the first solution doesn't really take this into account, it makes sense that it would generate more misses and take larger amounts of time. For smaller matrix sizes, this doesn't really matter, as cache misses will still be low (the cache is large enough to store needed values). Only when we increase the matrix size jumps from `1400` to `1800` the cache size won't be enough to store all the values being accessed by the program. It's at this point that Basic multiplication's lack of efficiency starts to have a bigger impact. The results display a significant increase of execution time, CPI, and MPF.

Even though the described increase is also noticeable for the Line by Line approach, it won't overload the cache as fast. This is because the data access is more orderly, thus improving program performance.



(a) C++ Basic and Line Multiplication execution times.



(b) C++ Basic and Line Multiplication Cache Misses per Floating point instruction.

Figure 2: Comparison between Basic and Line Multiplication algorithms.

Furthermore, block multiplication was expected to perform better than the previous algorithms. Not only does it provide a better memory access pattern by using only a small portion of the matrix at a time, but it also cuts down on the total number of operations performed. Oddly, by examining the results in Figure 3, we got to the conclusion that this approach, although it does indeed offers the best results, is not as reliable as Line by Line Multiplication for differently-sized matrices.

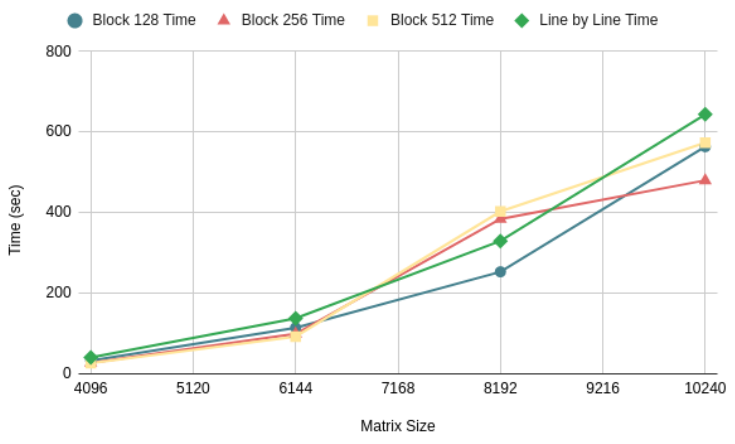


Figure 3: Execution time comparison between different block sizes and line multiplication.

More in detail, if we take a look at Block Multiplication MPF and CPI in Figure 4, we will notice a spike of these metrics for block sizes of 256 and 512 on 8192-sized matrices. This means that these conditions will overload the cache and cause an impact on performance.

We get to the conclusion that using larger-sized blocks will generally be worse in performance. For this processor's architecture, it is best to keep the blocks smaller in order to get a more efficient use of cache.

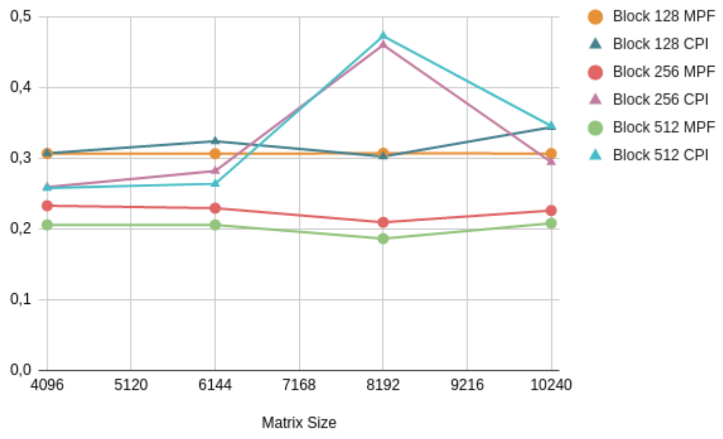


Figure 4: CPI and MPF comparison between different block sizes in Block multiplication.

4.2 C++ vs Java

Generally, it is safe to assume that C++ can outperform Java. For starters, C++ is a compiled language. Java, on the order hand, is interpreted. This just-in-time compilation (JIT) brings overhead to a program's execution the same can be said for the Java Virtual Machine (JVM), on which Java code runs on. Additionally, C++ provides more direct and less protected memory access (e.g. no garbage collection), which can improve performance.

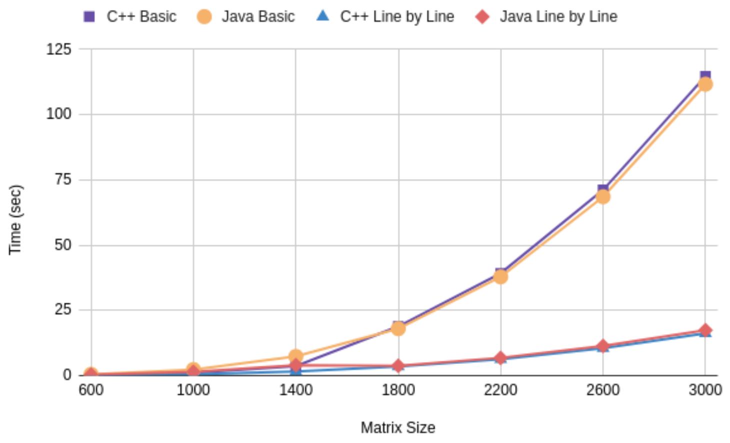


Figure 5: C++ and Java Basic and Line Multiplication execution times.

For the tested matrix sizes, the execution times obtained for each language, got relatively close (Figure 5). Nevertheless, C++ did generally outperform Java with some exceptions.

5 Detailed Results

Table 1: Block multiplication results.

Size	C++ time	Java time	Level 1 DCM	Level 2 DCM	Cycles	Instructions	Double flops	MPF	CPI
600	0.191	0.449	2.45e+8	3.88e+7	8.64e+8	1.52e+9	4.32e+8	0.656	0.569
1000	1.098	2.299	1.23e+9	2.70e+8	5.03e+9	7.02e+9	2.00e+9	0.749	0.717
1400	3.62	7.31	3.51e+9	1.42e+9	1.65e+10	1.92e+10	5.49e+9	0.898	0.857
1800	18.806	17.969	9.08e+9	8.97e+9	8.58e+10	4.09e+10	1.17e+10	1.548	2.098
2200	39.193	37.82	1.77e+10	2.28e+10	1.79e+11	7.46e+10	2.13e+10	1.899	2.404
2600	71.143	68.53	3.09e+10	5.11e+10	3.25e+11	1.23e+11	3.52e+10	2.333	2.635
3000	114.75	111.671	5.03e+10	9.54e+10	5.30e+11	1.89e+11	5.40e+10	2.699	2.804

Table 2: Line by Line multiplication results (C++).

Size	Time	Level 1 DCM	Level 2 DCM	Cycles	Instructions	Double flops	MPF	CPI
600	0.105	2.71e+7	5.70e+7	4.68e+8	1.73e+9	4.32e+8	0.195	0.270
1000	0.485	1.26e+8	2.61e+8	2.23e+9	8.02e+9	2.00e+9	0.194	0.278
1400	1.589	3.46e+8	7.05e+8	7.29e+9	2.20e+10	5.49e+9	0.192	0.332
1800	3.534	7.46e+8	1.44e+9	1.61e+10	4.67e+10	1.17e+10	0.188	0.344
2200	6.245	2.08e+9	2.08e+9	2.90e+10	8.53e+10	2.13e+10	0.195	0.340
2600	10.515	4.41e+9	4.21e+9	4.88e+10	1.41e+11	3.52e+10	0.245	0.347
3000	16.127	6.78e+9	6.43e+9	7.50e+10	2.16e+11	5.40e+10	0.245	0.347
4096	40.799	1.75e+10	1.58e+10	1.90e+11	5.50e+11	1.37e+11	0.242	0.345
6144	137.478	5.92e+10	5.42e+10	6.40e+11	1.86e+12	4.64e+11	0.244	0.345
8192	329.646	1.40e+11	1.30e+11	1.53e+12	4.40e+12	1.10e+12	0.246	0.349
10240	643.744	2.74e+11	2.57e+11	3.00e+12	8.59e+12	2.15e+12	0.247	0.349

Table 3: 128 Block sized multiplication (C++).

Size	Time	Level 1 DCM	Level 2 DCM	Cycles	Instructions	Double flops	MPF	CPI
4096	32.487	9.74e+9	3.24e+10	1.50e+11	4.87e+11	1.37e+11	30.7	30.7
6144	114.424	3.29e+10	1.09e+11	5.33e+11	1.64e+12	4.64e+11	30.7	32.4
8192	253.115	7.79e+10	2.60e+11	1.18e+12	3.90e+12	1.10e+12	30.7	30.3
10240	563.625	1.52e+11	5.07e+11	2.62e+12	7.61e+12	2.15e+12	30.7	34.4

Table 4: 256 Block sized multiplication (C++).

Size	Time	Level 1 DCM	Level 2 DCM	Cycles	Instructions	Double flops	MPF	CPI
4096	2.70e+1	9.08e+9	2.29e+10	1.26e+11	4.84e+11	1.37e+11	0.23	0.26
6144	9.92e+1	3.07e+10	7.58e+10	4.61e+11	1.63e+12	4.64e+11	0.23	0.28
8192	3.84e+2	7.30e+10	1.57e+11	1.78e+12	3.87e+12	1.10e+12	0.21	0.46
10240	4.80e+2	1.42e+11	3.44e+11	2.23e+12	7.56e+12	2.15e+12	0.23	0.29

Table 5: 512 Block sized multiplication (C++).

Size	Time	Level 1 DCM	Level 2 DCM	Cycles	Instructions	Double flops	MPF	CPI
4096	26.75	8.77e+9	1.95e+10	1.25e+11	4.83e+11	1.37e+11	0.21	0.26
6144	92.354	2.96e+10	6.58e+10	4.30e+11	1.63e+12	4.64e+11	0.21	0.26
8192	402.842	7.03e+10	1.35e+11	1.83e+12	3.86e+12	1.10e+12	0.19	0.47
10240	573.447	1.37e+11	3.10e+11	2.61e+12	7.54e+12	2.15e+12	0.21	0.35

6 Conclusion

Throughout the project, we interacted with different approaches to the same problem and how they can affect performance. Guided by measurements of different metrics, we got to conclude how advantageous different algorithms could be for cache usage. A major takeaway of this project is that programs should be diligent in terms of memory access, so as to spare resources. A good heuristic for optimization is to keep accesses to the same memory location close to each other.

In more specific terms, it may be inferred that Line by Line Multiplication and Block multiplication for small blocks are reliable methods for Matrix Multiplication. In the same light, it can be stated that C++ offers better performance than Java.