Performance evaluation of a single core

Computação Paralela e Distribuída - Project 1

T05 G10

1. João Pinheiro (up202008133@up.pt)
2. João Matos (up202006287@up.pt)
3. Ricardo Cavalheiro (up202005103@up.pt)

Problem Description

The main goal of this project is to access the performance of a single core in a processor while accessing large amounts of data in a memory hierarchy. In this study, we will analyze how the system behaves using different algorithms to calculate the product of two matrices.

The Performance API (***PAPI***) will be useful to collect all the relevant information about the performance metrics related to the behavior of the different algorithms and the underlying hardware.

This project consists on applying three different techniques to matrix multiplication and analyzing the impact of each one in the CPU performance:

- 1st Task: Implement a simple algorithm that multiplies one line of the first matrix by each column of the second matrix. We choose Rust as the selected programming language.
- 2nd Task: Implement a different version of the matrix multiplication, which multiplies an element from the first matrix by the correspondent line of the second matrix. This algorithm was designed in both C/C++ and Rust.
- 3rd Task: Implement matrix multiplication algorithm which uses a block-oriented technique by dividing the matrices into blocks.

In each assignment, we used matrices with different sizes to analyze and compare the results.

The results of these tests will provide insights on how well the CPU performs when accessing large volumes of data stored in a memory hierarchy. The performance metrics collected using PAPI will be used to determine which matrix multiplication implementation and method is the most effective. This project's overall goal is to evaluate the impact of the memory hierarchy on processor performance, which is crucial for applications that work with large datasets.

Algorithms Explained

1. Simple Matrix Multiplication (Version 1)

The **first** algorithm used was simple matrix multiplication where each cell *i, j* of a matrix C is the scalar product of the line i of matrix A with column j of matrix B. We iterate over the lines of matrix A and over the columns of matrix B. Then we iterate both one of those lines and one of those columns at the same time accumulating the result in one cell of matrix C.

```
FOR i = 1 to MATRIX_SIZE
      FOR j = 1 to MATRIX_SIZE
            FOR k = 1 to MATRIX_SIZE
                  MATRIX_C[i·MATRIX_SIZE+j]
                        +=
                  MATRIX_A[i·MATRIX_SIZE+k]
                        *
                  MATRIX_B[k·MATRIX_SIZE + j]

            END FOR
      END FOR
END FOR
```

As a small optimization, the result of the last inner cycle is accumulated into a temporary variable and then at the end of each inner cycle, the result is written onto the corresponding cell of matrix C. This way we do not need to calculate the index, retrieve the value and store it again every cycle.

Let n = MATRIX_SIZE, then the space complexity of this algorithm is $O(n^2)$ as it is using three matrices of size n^2 to store the input and output. The auxiliary variable is constant in terms of space.

The time complexity of this algorithm is $O(n^3)$ because it has three nested loops, where each loop iterates over the size of n. Therefore, the total number of operations performed by the algorithm is $3n^3$.

2. Simple Matrix Multiplication (Version 2)

The **second** algorithm is very similar, except that the two most inner cycles are swapped. This makes it so that instead of calculating the value of a cell in matrix C directly, we instead iterate over the lines of matrix A, and then over the lines of matrix B, multiplying one value from column *k* in matrix A by every element of the corresponding row *k* in matrix B, accumulating results in matrix C.

```
FOR i = 1 to MATRIX_SIZE
      FOR k = 1 to MATRIX_SIZE
            FOR j = 1 to MATRIX_SIZE
                  MATRIX_C[i·MATRIX_SIZE+j]
                        +=
                  MATRIX_A[i·MATRIX_SIZE+k]
                        *
                  MATRIX_B[k·MATRIX_SIZE + j]

            END FOR
      END FOR
END FOR
```

Again, as a small optimization, the value MATRIX_A[i · MATRIX_SIZE + k] can be stored in a temporary variable for every *i* and *k* from the first two cycles, to avoid accessing the memory of the matrix and calculating the index of the element every iteration of the innermost cycle. With this algorithm, the order of the loops affects the efficiency of the algorithm, and it may perform better than the first algorithm due to cache locality. In this algorithm, the innermost cycle is iterating over contiguous memory locations, which can take advantage of caching, whereas, in the first algorithm, the innermost cycle is not iterating over contiguous memory locations, which may result in cache misses and slower performance.

The space complexity of this algorithm is also $O(n^2)$ and the time complexity is also $O(n^3)$ for the exact same reasons as the first algorithm.

3. Block Matrix Multiplication

The **third** algorithm is an example of matrix block multiplication. Matrices A and B are divided into blocks of size BLOCK_SIZE and then both matrices are multiplied as if each sub block represented a single number as in the first algorithm, that is, the direct scalar product between every row *i* and column *j*. However, as the sub blocks are matrices, when multiplying with other sub blocks, the second algorithm is applied, resulting in a sub matrix of the same size (because they are square and of the same size) corresponding to the block *i, j* of the result matrix C.

```
FOR i = 1 to MATRIX_SIZE step by BLOCK_SIZE
      FOR j = 1 to MATRIX_SIZE step by BLOCK_SIZE
            FOR k = 1 to MATRIX_SIZE step by BLOCK_SIZE
                  FOR ii = i to min(i+BLOCK_SIZE, MATRIX_SIZE)
                        FOR kk = k to min(k+BLOCK_SIZE, MATRIX_SIZE)
                              FOR jj = j to min(j+BLOCK_SIZE, MATRIX_SIZE)
                                    MATRIX_C[ii·MATRIX_SIZE+jj] +=
                                    MATRIX_A[ii·MATRIX_SIZE+kk]
                                          *
                                    MATRIX_B[kk·MATRIX_SIZE + jj]
            END FOR
      END FOR
END FOR
```

Let n = MATRIX_SIZE and b = BLOCK_SIZE:

The space complexity of this algorithm is $O(n^2)$ as it is using three matrices of size n^2.

The time complexity can be expressed as $O((n/b)^3 \cdot b^3) = O(n^3)$. The outermost cycle iterates over the n rows of the matrices in steps of b, and the inner cycle iterates over the columns of the matrices in steps of b. The loops with indices *ii*, *kk*, and *jj* iterate over the smaller blocks, and each of these loops iterates over the size of b. Therefore, the total number of blocks is $(n/b)^2$ and for each block there are n / b blocks to multiply with, and each multiplication performs b^3 operations. Hence, the overall time complexity is $O((n/b)^3 \cdot b^3)$.

By choosing an optimal BLOCK_SIZE, which depends on the used hardware, e.g., cache sizes, we can minimize the number of cache misses and achieve better performance.

Performance Metrics

We measured the time it took to multiply matrices of varying sizes and also varying block sizes for the third algorithm. For every run of the program in C++ we also measured the **L1 Data Cache Misses** (DCM), **L2 DCM** and **L2 Direct Cache Accesses** (DCA). The first and second algorithms were tested in both C++ and Rust, with matrix sizes ranging from 600 to 3000 for the former and from 600 to 10240 in the latter. The third algorithm, tested only in C++ ran with matrix size varying from 4096 to 10240 and block sizes between 128 and 512. The times were measured 3 times and averaged. All the benchmarks were run using an Intel Core i7-10510U, and it is important to note that these were the cache modules in the system:

- L1 Data cache of 128 KiB with 4 instances;
- L1 Instructions cache of 128 KiB with 4 instances;
- L2 Cache of 1 MiB with 4 instances;
- L3 Cache of 8 MiB with 1 instance.

Results and Analysis

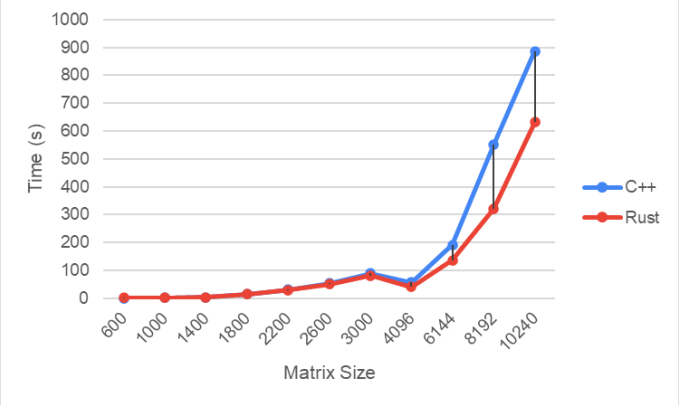

Fig 1. C++ vs Rust, runtime comparison for second and third algorithms

We observed that the execution times were generally faster in Rust (Fig 1). Despite being two relatively similar high-performance languages, this result may be because Rust has mechanisms to ensure memory accesses are safe and efficient, while C++ prioritizes low-level control.

The second algorithm was faster than the first because it takes advantage of memory access patterns and cache utilization. In the first algorithm, we access different rows of the second matrix to compute the value of a single cell in the result matrix. This requires constantly getting rows from memory and caching them in vain since we will get a cache miss when we try to get the next value which is not in the previously cached row. For example, a single row of 3000 64-bit floats is already larger than our L2 cache (3000 x 8 bytes > 4 instances of 1 MiB). However, the second algorithm takes advantage of the memory layout and traverses the second matrix in row-major order, which guarantees fewer cache misses since the values that are cached will be used sequentially.

We also observed that the second algorithm was faster than the third due to its memory access pattern. The block multiplication algorithm traverses the matrices in a blocked fashion, which can result in more cache misses and inefficient use of the cache. The third algorithm accesses sub-blocks of matrices A, B and C which may not fit entirely in cache (e.g., 128 x 128 x 8 bytes >> our cache size) resulting in more cache misses. In contrast, the second algorithm accesses only one row of each matrix at a time.

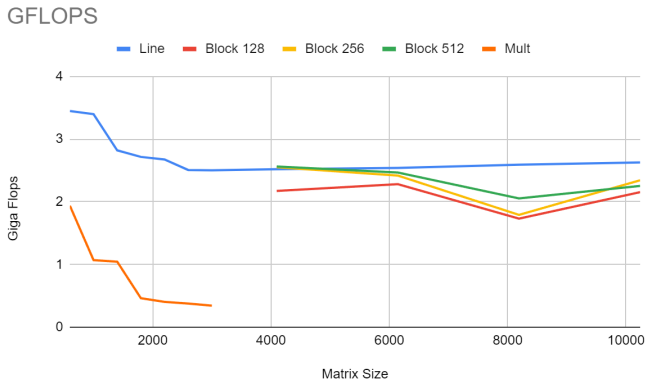

Fig 2. Gigaflops of the different algorithms

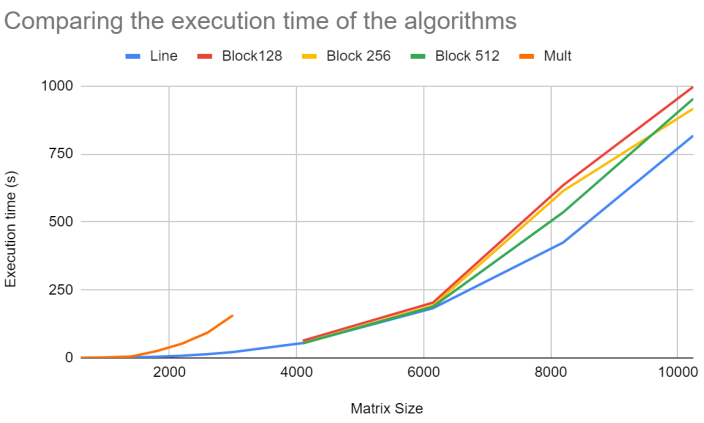

Fig 3. Comparison of the execution time of the algorithms

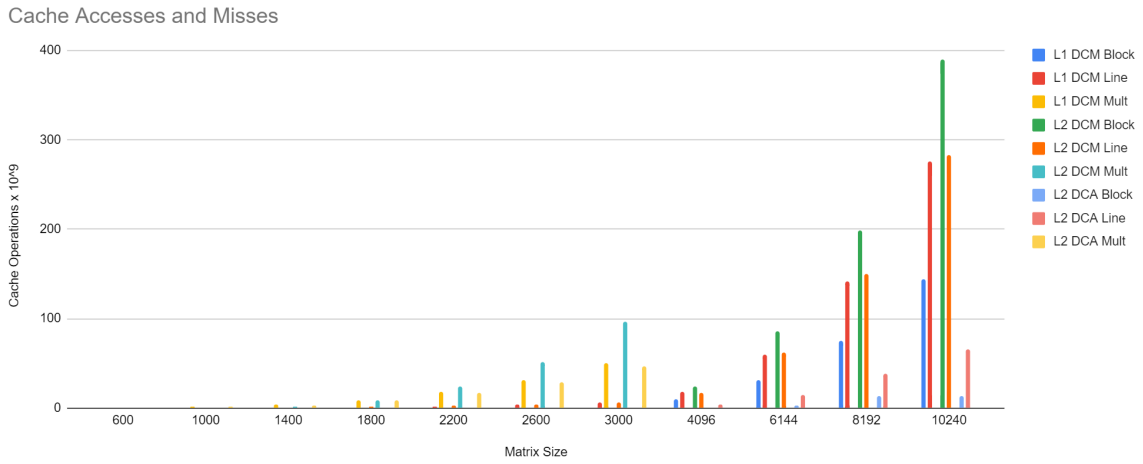

Fig 5. Cache accesses by type of cache and algorithm

We can observe the following trends: the second algorithm performs fewer L2 cache accesses and misses but more L1 data cache misses and the second algorithm also performs far less cache accesses and misses than the first algorithm.

Conclusions

In conclusion, this project gave us the chance to learn more about memory management and how it affects processor speed. We were capable of analyzing the effect of memory management strategies on program execution durations by implementing several matrix multiplication algorithms and evaluating the performance data using PAPI.

The results confirmed the idea that optimizations are crucial for enhancing memory management. We were able to conclude the advantages of having Multi Line Matrix Multiplication and Block Matrix Multiplication, which dramatically lower cache misses and enhance performance.

Overall, this project improved our knowledge of memory management and how it affects processor performance. It also provided valuable insights into how to optimize matrix multiplication algorithms.

Annexes

Data sheet