Faculty of Engineering of the University of Porto

Performance evaluation of a single core

Parallel and Distributed Computing

Project 1

Students:

- André Sousa (up202005277@edu.fe.up.pt)
- Gonçalo Pinto (up202004907@edu.fe.up.pt)
- Pedro Fonseca (up202008307@edu.fe.up.pt)

Index

Problem description and algorithms explanation

This project aims to understand how large amounts of data impacts the performance of the processor's memory. This study will focus on matrix multiplication.

There were three distinct methods for matrix multiplication, namely **column multiplication**, **line multiplication** and **block multiplication**.

Column multiplication is the most naive solution where all the elements from the line **n** are multiplied by the elements of the column **m** in order to get value **V(n, m)**.

```
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;
    }
}
```

Image 1 - main loop (**column multiplication**)

Line multiplication follows a similar structure as the previous method, but instead of computing the result in a single step, it builds up the value **V(n, m)** progressively by computing and adding each product of corresponding elements as it moves along the **row**.

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

Image 2 - main loop (**line multiplication**)

In **block multiplication**, the matrix is divided into blocks and each block is processed separately using the **line multiplication** method to calculate their respective values. Finally, the results are combined to obtain the final product.

```
for (n_i=0; n_i < n_max; n_i++) {

    for (n_j=0; n_j < n_max; n_j++) {

        for (n_k=0; n_k < n_max; n_k++)

        for(i = n_i * bkSize; i < (n_i* bkSize + bkSize); i++) {
            for( k =n_k * bkSize; k < (n_k* bkSize + bkSize); k++) {
                for( j = n_j * bkSize; j < (n_j * bkSize + bkSize); j++) {
                        phc[i*m_ar+j] += pha[i*m_ar+k] * phb[k*m_br+j];
                    }
            }
    }

    }

}
```

Image 3 - main loop (**block multiplication**)

Performance metrics

During the testing phase, we collected data such as the **execution time**, as well as the **number of cache misses** that occurred in both the **L1** and **L2** data caches.

For the **column multiplication** process, both the execution time and cache misses will be at their highest because the program stores matrix information that will not be used in the next iterations.

As for the **line multiplication**, it is expected that there will be a significant impact on those factors because cache access will occur more frequently because the matrix is iterated line-by-line rather than column-by-column.

In **block multiplication**, the metrics should vary with the block size, but it is expected that if the block size is equal to the matrix dimensions, the result should be similar to the **line multiplication method**.

Results and analysis

Despite sharing a time complexity of O(n^3), our study yielded a range of outcomes due to the differing approaches employed. As expected, column multiplication proved to be the least effective approach to matrix multiplication, as shown in **image 4** and **image 5**. This is due to the behavior of memory operations, wherein the CPU must retrieve the required data from main memory and store it in cache memory. However, instead of storing only the required element, the CPU stores an entire block of contiguous matrix elements in anticipation of future computations. This leads to a performance slowdown, as the subsequent iteration requires a different non-contiguous element from the same matrix, requiring the CPU to perform the same time-consuming process of retrieving and storing the data.

To address this issue, we modified the algorithm to perform row multiplications instead of column multiplications. This approach is more "cache-friendly" as the CPU retrieves a block of elements in one iteration and stores them in its cache memory, which is used in subsequent iterations. Furthermore, we implemented a block multiplication approach that divides each matrix into blocks of X elements and uses the row multiplication technique to perform calculations. This approach proved to be superior to the initial column multiplication approach but was still outperformed by the row multiplication methodology, as seen in **image 6**, **image 7**, **image 8**, **image 9** and **image 10**. It is worth noting that in block multiplication with matrices of 8192 elements with block size of 256 bytes we got a value that is outside of what was expected. As seen on **image 8**, there is a sudden increase in time, instead of a steady decrease as seen in every measurement within each matrix size. We believe this is due to the hardware limitation of the L1 cache. The matrix is big enough to be at the tipping point of worsening the performance. After its sudden increase, the values for the other measurements go back to what is expected.

In our study, we tested the three aforementioned approaches in C/C++ but we also implemented the first two in Python. However, we were forced to adopt a different secondary programming language due to the fact that Python is a interpreted language with dynamic type checking which makes this language a lot slower than compiled languages and that is why we weren't able to make all the measurements with Python because of time limitations.

Since we wanted to test these approaches on an interpreted language (to compare a compiled language with an interpreted one) and Python is not a viable option, we chose to implement them in JavaScript, as shown in **image 4** and **image 6**. JavaScript is much more efficient than Python because it is designed to run in web browsers, uses just-in-time compilation, and has a focus on performance, whereas Python is an interpreted language with a dynamic type system and more general-purpose design. The obtained results were that C/C++ is still much more efficient than JavaScript (or any other interpreted language).

Conclusions

In conclusion, this study investigated the impact of large amounts of data on processor memory performance through the analysis of three different algorithms for matrix multiplication. The results indicate that the column multiplication method is the least efficient, due to the nature of memory operations and the CPU's tendency to store entire blocks of contiguous matrix elements in anticipation of future computations, resulting in slower performance. The line multiplication method proved to be more "cache-friendly" as the CPU retrieves a block of elements in one iteration and stores them in its cache memory, which is used in subsequent iterations. Finally, the block multiplication approach, which divides each matrix into blocks of X elements and uses the row multiplication technique to perform calculations, was found to be superior to the initial column multiplication approach but was still outperformed by the row multiplication methodology.

The study also highlights the significant difference in performance between compiled languages like C/C++ and interpreted languages like JavaScript, with C/C++ still proving to be much more efficient. Overall, this study provides valuable insights into the performance impact of different matrix multiplication algorithms and demonstrates the importance of considering memory performance when dealing with large amounts of data.

Annexes

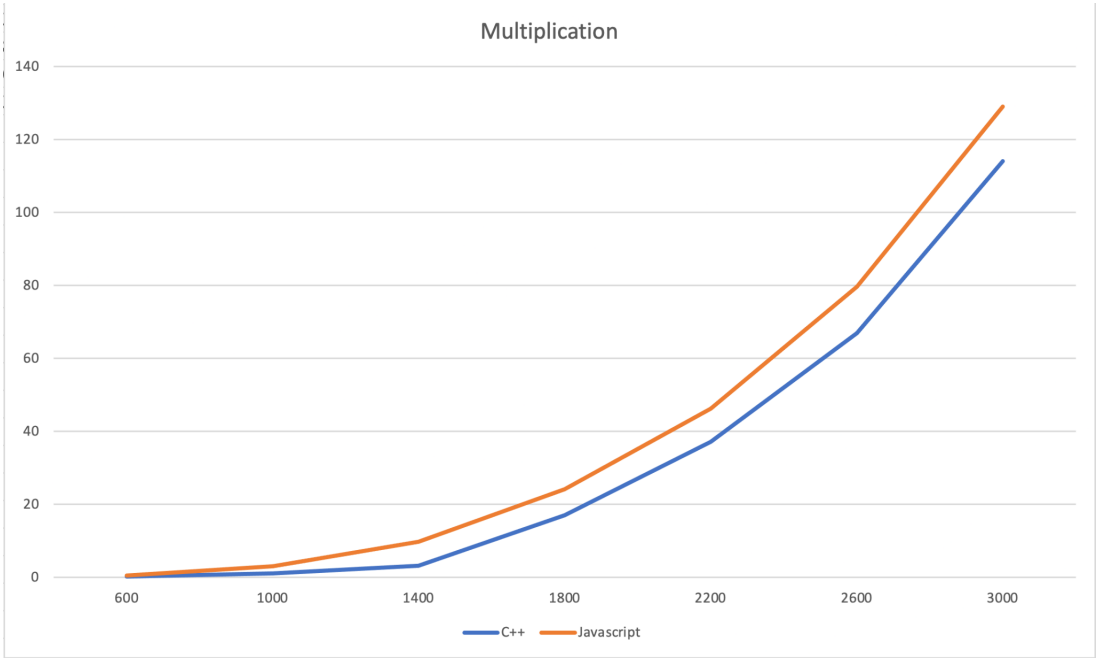

Image 4 - **column multiplication** (execution time)

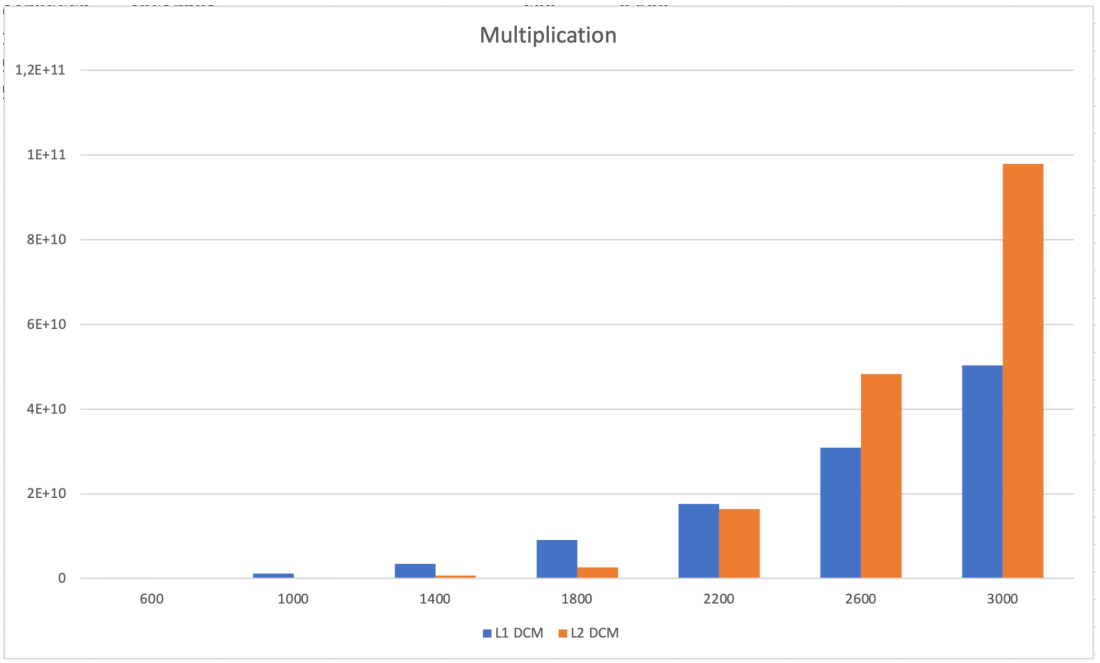

Image 5 - **column multiplication** (L1 and L2 misses)

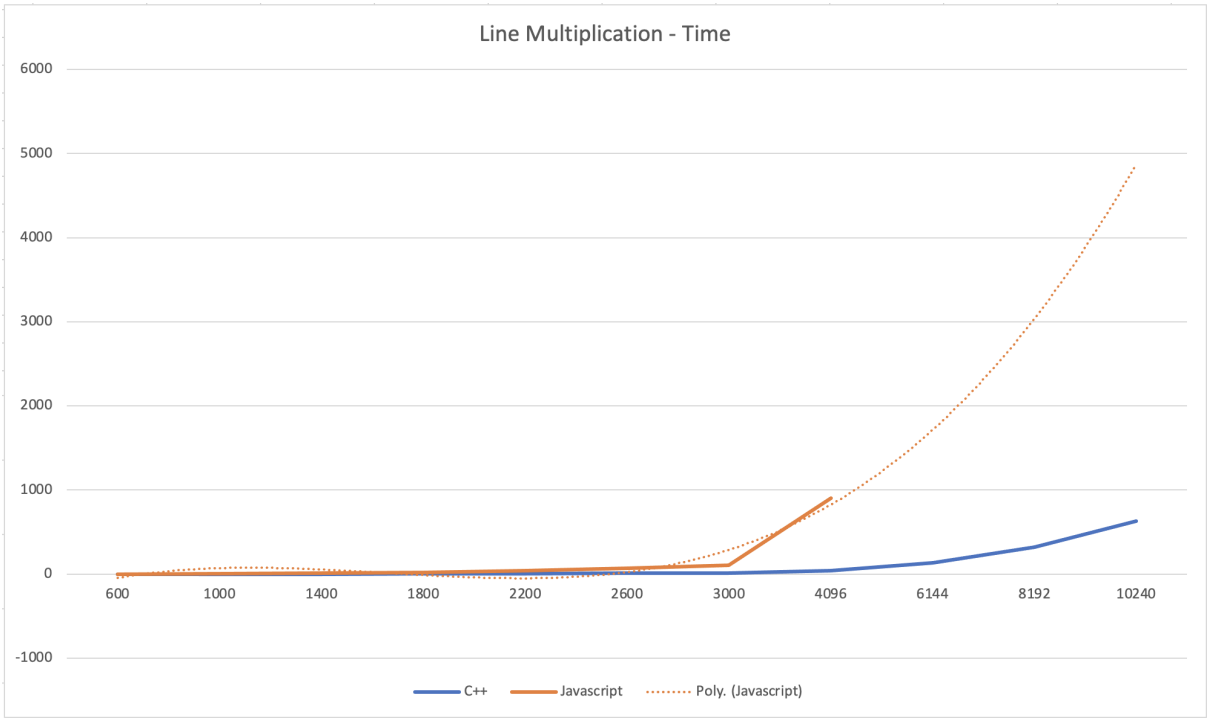

Image 6 - **line multiplication** (execution time)

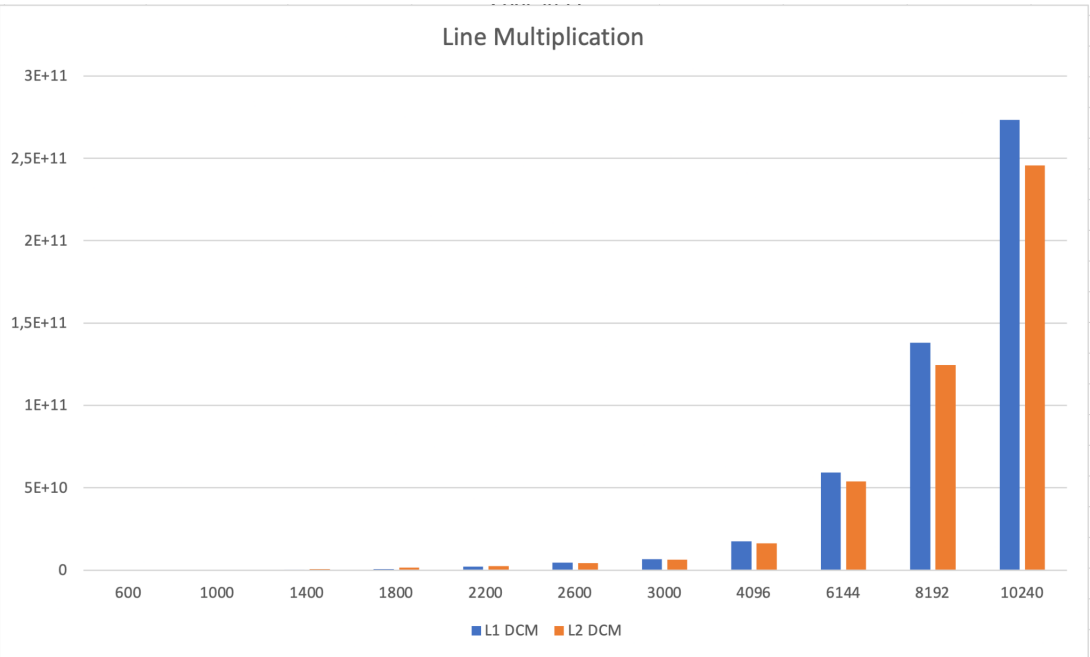

Image 7 - **line multiplication** (L1 and L2 misses)

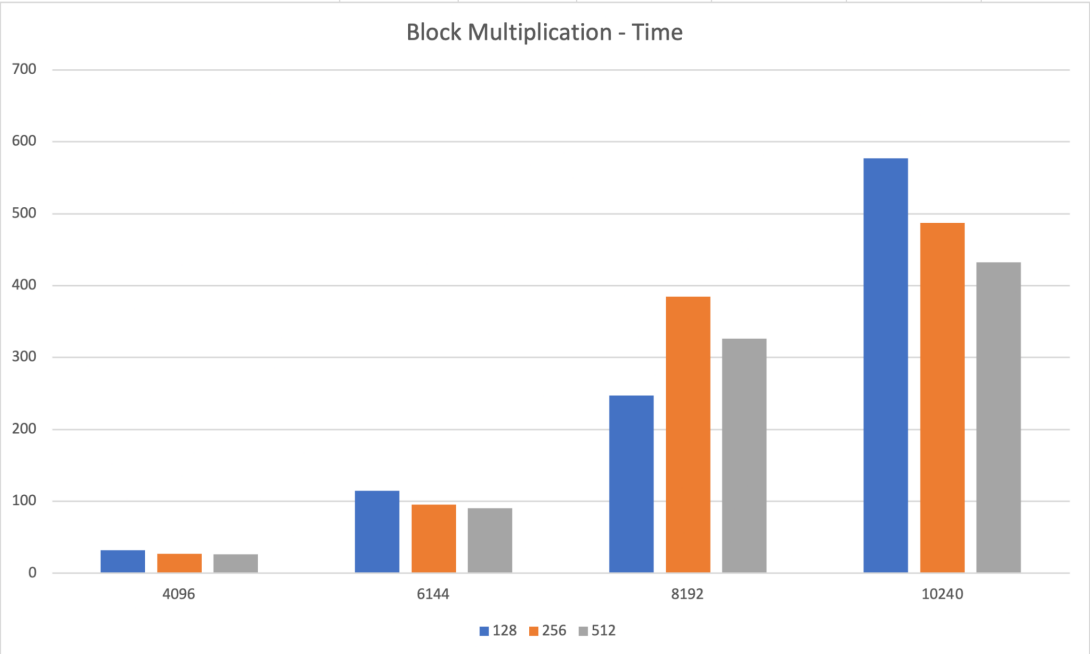

Image 8 - **block multiplication** (execution time)

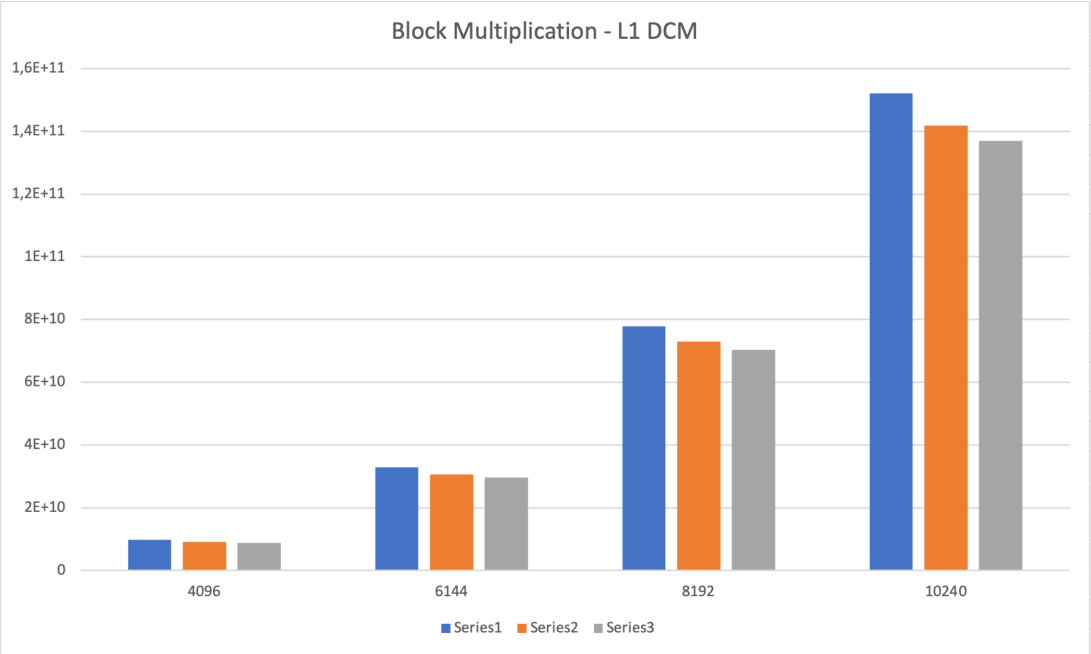

Image 9 - **block multiplication** (L1 cache misses)

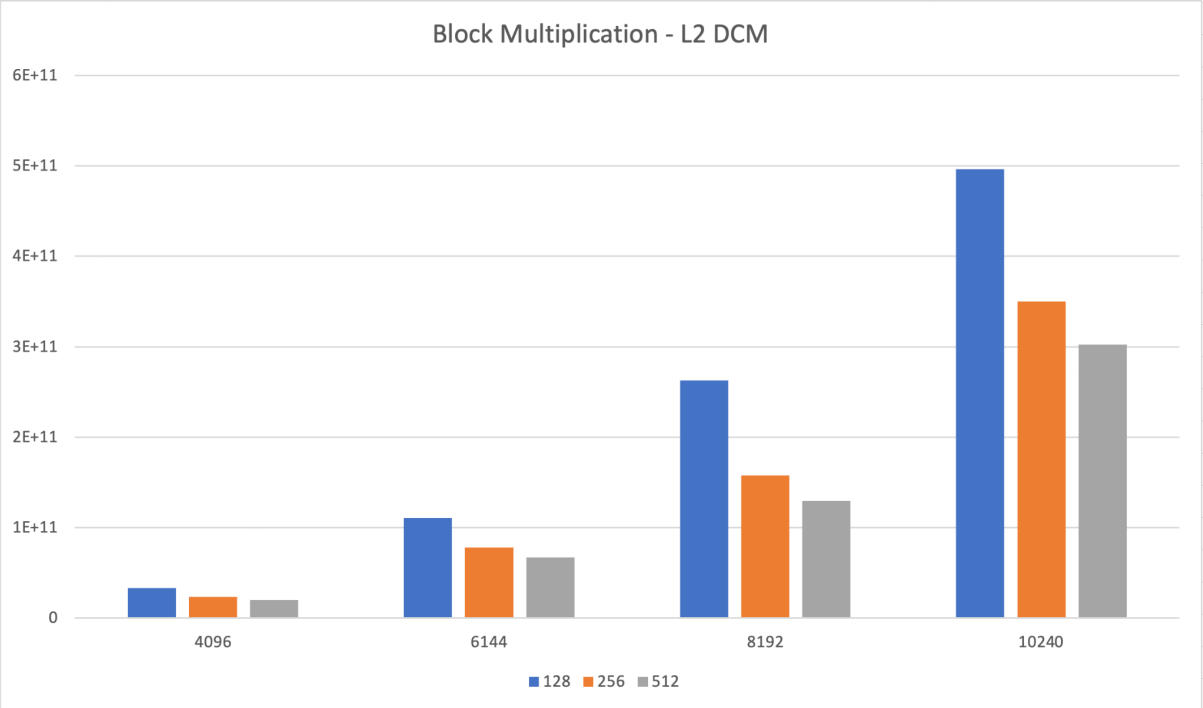

Image 10 - **block multiplication** (L2 cache misses)