2023/2024

Report : Performance evaluation of single-core and multi-core implementations

Grupo 17 – Turma 3

Licenciatura em Engenharia Informática e Computação

António Marujo Rama	up202108801@fe.up.pt
João Mendes Silva Belchior	up202108777@fe.up.pt
José Francisco Reis Pedreiras Neves Veiga	up202108753@fe.up.pt

Index

0. Introduction

This report is dedicated to an in-depth analysis of key performance indicators via the application of various matrix multiplication algorithms. By analyzing a selection of these algorithms, we intend to reveal insights into CPU utilization and execution speed, among other metrics.

1. Methodology

1.1 - Testing Procedure

We implemented and tested three unique algorithms on a single-core implementation before selecting one for further evaluation on two diverse multi-core configurations. To ensure a comprehensive assessment, we experimented with varying matrix sizes for each algorithm, conducting each test three times to improve the accuracy and reliability of the results. These experiments were performed on a designated computer, specifically the university's system (with an 8 core CPU), under consistent conditions. To evaluate performance, we gathered a variety of metrics, including execution time, L1 cache misses and L2 cache misses. These last two were measured using PAPI (Performance API).

1.2 - Standard Matrix Multiplication (single-core C++/Python)

This algorithm follows the conventional approach of iterating through each row of the first matrix and each column of the second matrix to compute the dot products for the elements of the result matrix. Memory allocation is done linearly for matrices `pha`, `phb`, and `phc`, with initialization ensuring `pha` is filled with 1.0s and `phb` with incrementing values starting from 1.0. The multiplication process is straightforward but can be computationally intensive due to the triple nested loop structure, which has a time complexity of **$O(n^3)$**.

```
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;
    }
}
```

1.3 - Line Matrix Multiplication (single-core in C++/Python)

This method optimizes the standard approach by reordering the loops to access memory in a more cache-friendly manner. It initializes the matrices in the same way but changes the multiplication logic to iterate line by line. This allows for better utilization of cache lines, potentially reducing cache misses and improving overall execution time. The algorithm still follows an **$O(n^3)$** time complexity but aims to enhance performance through improved memory access.

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

1.4 - Block Matrix Multiplication (single-core in C++)

In this method we try an even different approach, by further breaking down the matrices into smaller blocks and performing multiplications on these blocks. This strategy reduces cache misses and improves data reuse. This algorithm divides each matrix into submatrices of size b×b, where b is the block size. Multiplications are then performed on these blocks rather than individual elements.

The time complexity remains **$O(n^3)$**, but the practical execution time is notably improved due to the reduction in cache misses and enhanced data locality.

```
for (i0 = 0; i0 < m_ar; i0 += bkSize) {
  for (k0 = 0; k0 < m_br; k0 += bkSize) {
    for (j0 = 0; j0 < m_ar; j0 += bkSize) {
      for (i = i0; i < min(i0 + bkSize, m_ar); i++) {
        for (k = k0; k < min(k0 + bkSize, m_ar); k++) {
          double temp = pha[i*m_ar + k];
            for  (j = j0; j < min(j0 + bkSize, m_br); j++) {
            phc[i*m_ar + j] += temp * phb[k*m_br + j];
            }
          }
        }
      }
    }
  }
```

1.5 - Line Matrix Multiplication (multi-core in C++)

We built upon the foundational Line Matrix Multiplication algorithm showcased earlier, creating two distinct multi-core optimized techniques.

The first algorithm employs OpenMP to distribute the multiplication workload across multiple threads, with each thread handling a subset of the rows in the resulting matrix. This strategy allows for concurrent execution of the outer and middle loops, with the innermost loop performing the actual multiplication and addition. For clarity in this report, we will refer to this algorithm as '*multi-core v1*'.

```
#pragma omp parallel for private (k, j)
for (i = 0; i < m_ar; i++) {
```

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

The second algorithm takes a slightly different approach by explicitly declaring a parallel region with `#pragma omp parallel` and then using `#pragma omp for` within the innermost loop. This approach suggests a parallelism where the distribution of work focuses more on the inner multiplication operations across different threads. We will be referring to this algorithm as 'multi-core v2'.

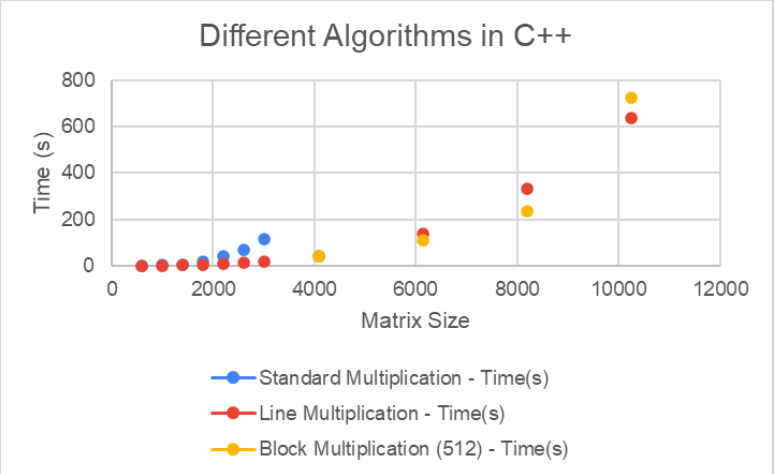
```
#pragma omp parallel
for (int i = 0; i < m_ar; i++) {
    for (int k = 0; k < m_ar; k++) {
        double temp = pha[i * m_ar + k];
        #pragma omp for
        for (int j = 0; j < m_br; j++) {
            phc[i * m_ar + j] += temp * phb[k * m_br + j];
        }
    }
}
```

2. Results

2.1 - Execution time comparisons

2.1.1 - Performance of the different algorithms in C++

The standard algorithm exhibited the slowest performance. Block multiplication generally outperformed line multiplication.



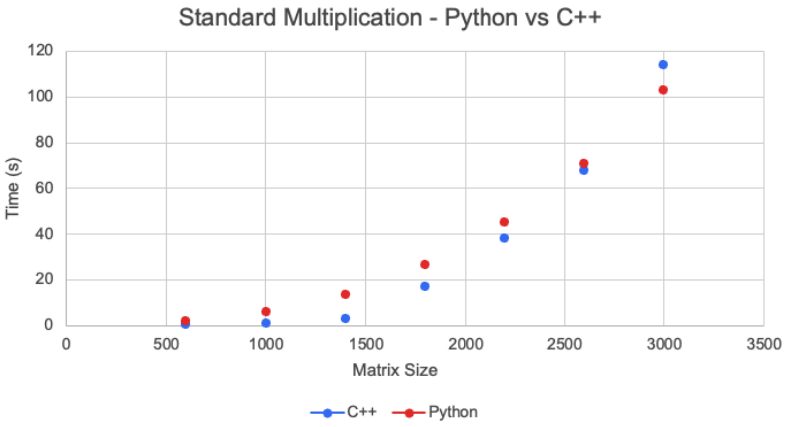
However, for a matrix size of 10240, line multiplication showed superior performance. Overall, the choice between line and block multiplication depends on factors like matrix size and hardware configuration.

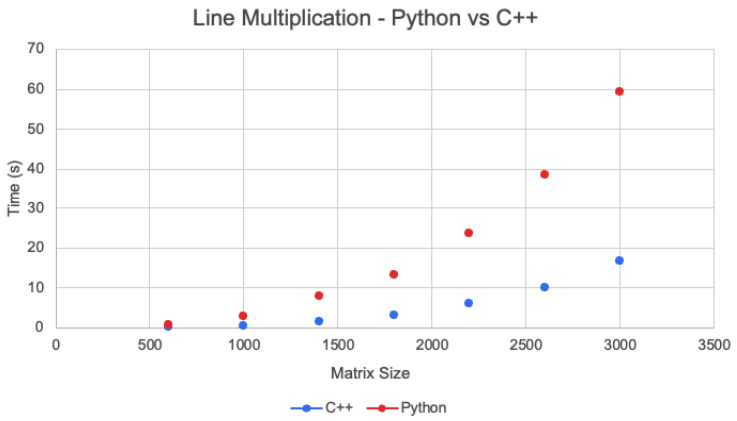
2.1.2 - C++ vs Python

For the two algorithms implemented in Python, we solely measured the execution time. Let's examine how this compares to the execution times observed in the C++ implementations.

In the case of the standard matrix multiplication algorithm, the C++ implementations showed superior performance for smaller matrix sizes. However, as the size of the matrices increased, the performance gap narrowed, with Python eventually outperforming C++ for a matrix size of 3000.

For the line matrix multiplication algorithm, C++ consistently surpassed Python in performance across all matrix sizes.

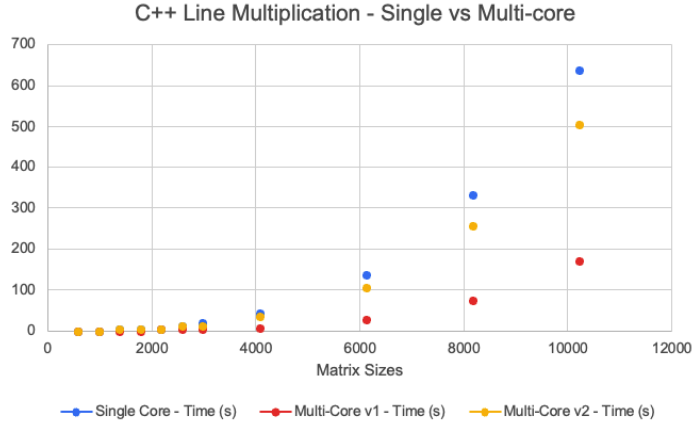


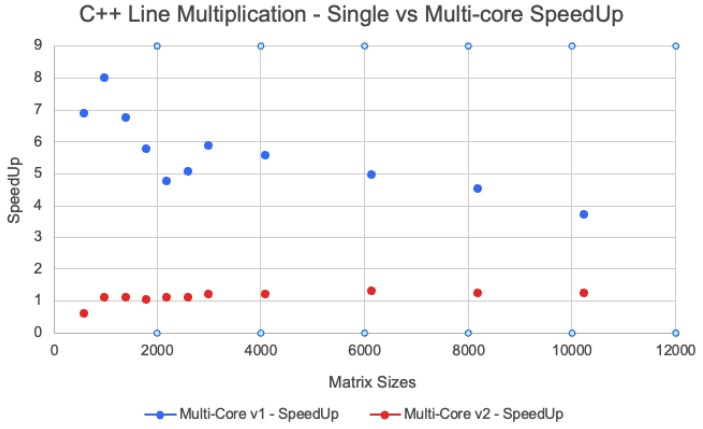


2.2 - Multi-core performance results in C++

As it can be seen in the graph below, the multi-core v1 implementation for line multiplication substantially boosted the operation's efficiency, whereas the implementation of multi-core v2 resulted in a small performance gain.

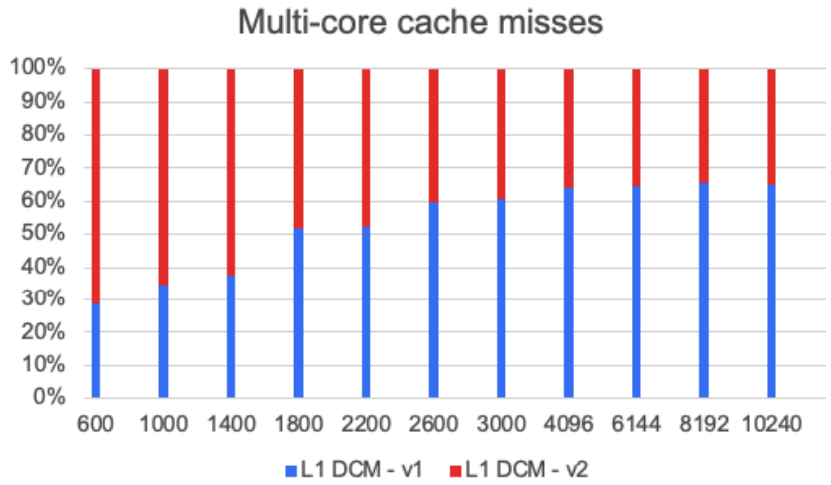
To more accurately assess the magnitude of performance enhancements, we'll analyze the speedup metrics for each multi-core approach. For the v1 version, an initial speedup near 7 times was observed for smaller matrices, which gradually diminished with increasing matrix sizes, ending at approximately a 4 times improvement. In contrast, the v2 version's speedup was negligible, staying at around 1.1 for all matrix sizes.





2.3 - Cache miss results

The comparison between 'multi-core v1' and 'multi-core v2' highlights the difference in cache misses, with 'multi-core v1' consistently registering fewer cache misses than 'multi-core v2', as shown in the chart below.



3. Discussion

3.1 - Execution time analysis

3.1.1 - Analysis of performance for different algorithms in C++

The **standard matrix multiplication algorithm** exhibits the slowest execution times due to its high computational intensity and consequent cache inefficiencies. In contrast, the **line matrix multiplication algorithm**, which optimizes memory access patterns, demonstrates improved performance, particularly for mid-sized matrices. However, the **block matrix multiplication algorithm**, by partitioning matrices into smaller blocks, surpasses both standard and line methods, leveraging superior cache utilization and reduced memory overhead. While **block matrix multiplication** generally outperforms others for mid-sized matrices, the line multiplication algorithm exhibits superior performance for larger matrices, typically beyond 8192 to 10240. The selection of the most suitable algorithm, should consider application requirements and hardware constraints to achieve optimal performance.

3.1.2 - C++ vs Python Analysis

The comparative analysis between Python and C++ in executing standard and line matrix multiplication algorithms underscores fundamental differences in their performance characteristics, especially in relation to the size of the data sets.

For **standard matrix multiplication**, C++ showcases its strength with smaller matrices due to its efficient memory management and compilation advantages. However, as the matrix size increases, Python begins to get closer to C++ for sizes around 3000, likely due to Python's utilization of optimized numerical libraries like NumPy, which leverage

underlying optimized C or Fortran code. This suggests that while C++ is generally more efficient for tasks involving smaller datasets or where low-level optimizations are critical, Python's high-level abstractions and library ecosystem make it more adept at handling larger-scale computations efficiently.

In contrast, the consistent performance superiority of C++ over Python in **line matrix multiplication** across all sizes highlights C++'s performance in algorithmically intensive computations. This emphasizes the importance of considering the specific computational demands and data scales when choosing between Python and C++, highlighting the importance of leveraging each language's strengths based on the context of the application.

3.2 - Multi-core performance analysis

The first version of the algorithm demonstrates greater efficiency largely due to the optimal placement and scope of the OpenMP parallelization directive. By specifying `#pragma omp parallel for` before the outermost loop and privatizing the k and j variables, it ensures that each thread receives a distinct row of the matrix to process. This setup minimizes thread creation overhead and maximizes data locality, as each thread works on a continuous section of memory, leading to better cache utilization.

In contrast, the second version places the `#pragma omp parallel` directive outside of all loops and only parallelizes the innermost loop with `#pragma omp for`. This approach results in a high overhead from frequently creating and destroying threads for each execution of the inner loop, which occurs many times. Additionally, this causes poor cache performance, as different threads may access widely separated memory locations in rapid succession.

These factors account for the performance discrepancies observed in the results.

3.3 - Cache miss analysis

The consistent reduction in cache misses by 'multi-core v1' suggests a more efficient memory access pattern and utilization of cache hierarchies. This efficiency is due to the parallelization strategy and data distribution among threads, and aligns with the previously discussed improvements in execution time, further reinforcing the connection between optimized cache behavior and improved computational performance.

4. Conclusion

This report has established a clear understanding of the performance capabilities of matrix multiplication algorithms on single and multi-core processors. The performed tests and use of PAPI for cache miss tracking revealed the superior efficiency of the 'multi-core v1' algorithm. The results confirm the critical role of parallel processing in optimizing computational tasks, emphasizing the importance of investigating such multi-processing approaches in algorithm design for getting better performance outcomes.

Appendices

The collected data, as well as extra graphs, can be viewed in this excel file.