U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Performance evaluation of a single core

CPD project 1

Licenciatura em Engenharia Informática e Computação

Professor: João Resende

Turma 12 - Grupo 19

Manuel Maria Faria de Sousa e Silva - up202108874

Maria Leonor Ribeiro Laranjeira - up202004453

Index

1. Problem Description

In this report, we study the effects of processor performance on the memory hierarchy when handling large data volumes. For this experiment, we will be using the AMD Ryzen 5 5500 CPU, 3.60 GHz. We implemented and evaluated the performance of three distinct matrix multiplication algorithms using two programming languages, C++ and Java. The primary goal was to understand how large-scale data impacts processor performance. To facilitate the evaluation, we utilize the Performance Application Programming Interface (PAPI) to collect key performance indicators of the program execution.

2. Algorithms

2.1. Naive Matrix Multiplication

This is the standard approach to matrix multiplication, where each row of the first matrix is multiplied by each column of the second matrix. This algorithm iterates through the rows of the first matrix. For each row, the columns of the second matrix are traversed and the sum of the products of the elements of the row and column are performed.

```
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;
        }
}
```

2.2. Line Matrix Multiplication

Improving the algorithm above, the elements of the first matrix and the corresponding elements of the second matrix are sequentially accessed. In this version, the data cache misses are reduced and the memory access is optimized due to the improvement of spatial and temporal locality.

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

2.3. Block Matrix Multiplication

This algorithm has a different approach from the others by partitioning the matrices into smaller blocks of uniform size. Then, for each block, it applies the line multiplication algorithm.

Unlike the previous algorithm, where a line exceeding the cache capacity could lead to cache misses, this approach mitigates such issues. By focusing on smaller blocks of the matrix, at a time, the data is more likely to reside within the cache during the computation, improving the execution speed. This algorithm improves both spatial and temporal locality. Spatial because when a data element is requested their neighbors will also be and temporal because when a data element is requested it has high probability to be requested in a short period of time.

```
for (int i0 = 0; i0 < m_ar; i0 += bkSize)
 for (int j0 = 0; j0 < m_ar; j0 += bkSize)
    for (int x = 0; x < m_ar; x++)
      for (int j = j0; j < min(j0 + bkSize, m_ar); j++)
        for (int i = i0; i < min(i0 + bkSize, m_ar); i++)
          phc[x*m_ar + i] += pha[x*m_ar + j] * phb[j*m_br + i];
```

2.4. Outer Loop Parallel Line Matrix Multiplication

This multi-threading approach to the Line Matrix Multiplication algorithm, contrary to all the others, which are single-threaded, allows for the task's execution to be divided by every logic processor instead of a single one, executing a thread for each processor (12 in case), making the whole process more time efficient. This algorithm parallelizes the computation across rows of the result matrix, meaning that the work of multiplying matrix pha by matrix phb and storing the result in matrix phc is distributed among the threads. Each thread is responsible for a subset of the rows of the resulting matrix phc.

```
#pragma omp parallel for private(i, j, k) shared(pha, phb, phc)
   for(i=0; i<m_ar; i++)
       for( j=0; j<m_ar; j++ )
       for( k=0; k<m_br; k++)
              phc[i*m_ar + k] += pha[i*m_ar + j] * phb[j*m_br + k];
```

2.5. Outer and Inner Loop Parallel Line Matrix Multiplication

This second multi-threaded implementation of the Line Matrix Multiplication algorithm adds another level of parallelism to the previous algorithm, by also parallelizing the innermost loop. This means that while the first approach parallelizes the computation across rows of the result matrix, this second approach attempts to parallelize both across rows and within the elements of each row. This introduces nested parallel regions, which can be less efficient than a single level of parallelism. The overhead of managing multiple parallel regions and synchronizing between them might also offset the gains from parallel execution, especially for smaller matrices or less complex operations.

```
#pragma omp parallel for private(i, j, k) shared(pha, phb, phc)
   for(i=0; i<m_ar; i++)
       for( j=0; j<m_ar; j++ )
```

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

3. Performance Metrics

To evaluate the performance of the previous algorithms, we used the metrics of execution time and L1 and L2 data cache misses. For hardware metric evaluation, we used the Performance Application Programming Interface (PAPI), a tool chosen because cache misses provide a clear indicator of cache usage efficiency by the algorithm, impacting its performance. Therefore, a lower number of cache misses signifies better cache utilization and, consequently, improvement in algorithm performance.

In addition, the algorithms were implemented in Java and C++ with the aim of observing the effect of the choice of programming languages on the performance. For both languages and each algorithm, the machine's capability was estimated based on FLOPS calculations:

$$FLOPS = \frac{2 \times (matrix\ size)^3}{execution\ time}$$

In order to showcase the difference in Parallel Algorithms compared to their Sequential implementations we calculate the speedup and efficiency:

$$Speedup = \frac{Tsequential}{Tparallel} \qquad Efficiency = \frac{Speedup}{nº\ logical\ processors} = \frac{Speedup}{12}$$

To ensure the consistency of our data, the same computer was used for all measurements and these were repeated five times to ensure reliability. Below, we detail the hardware specifications that were employed for these evaluations:

CPU	AMD Ryzen 5 5500, 3.60 GHz

Cache	Total Size	Line Size	Nº of Lines	Associativity
L1 Data Cache	32 KB	64 bytes	512	8
L1 Instruction Cache	32 KB	64 bytes	512	8
L2 Unified Cache	512 KB	64 bytes	8192	8
L3 Unified Cache	16 MB	64 bytes	262 144	16

Figure 1 - Information on the Hardware Used

4. Results and analysis

After gathering performance data, we started a comparative analysis of execution times, cache miss rates, and FLOPS across the three algorithms we implemented. The investigation also extended to the impact of block size on the performance of the third algorithm. Additionally, we analyze the effect of the selected programming languages, C++ and Java, in the initial two algorithms. The full results of our study are available in the Annexes.

4.1. Time comparison between algorithms

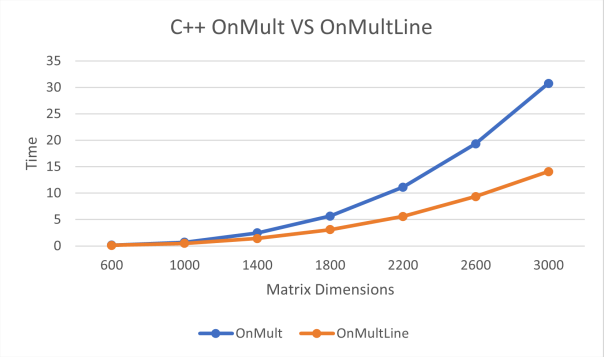

Figure 2 - Comparison graphic between Naive Matrix Multiplication and Line Matrix Multiplication algorithm in C++

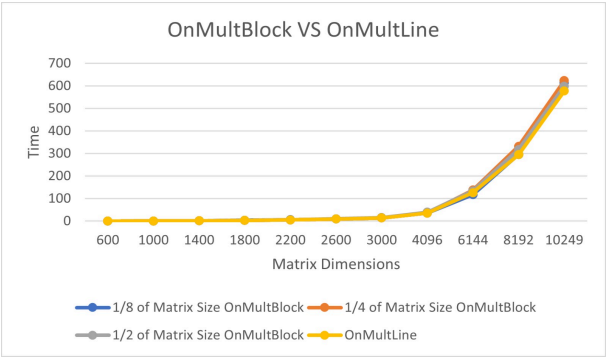

Figure 3 - Comparison graphic between Line Matrix Multiplication and Block Matrix Multiplication algorithm in C++

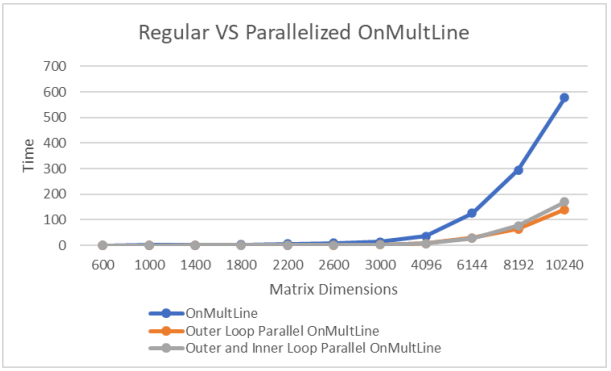

Figure 4 - Comparison graphic between Regular Line Matrix Multiplication and the Parallelized Line Matrix Multiplication algorithms

To compare the different matrix multiplication algorithms, we evaluated their execution times for various matrix sizes, using the C++ implementation. Our goal was to identify the most effective algorithm for matrix multiplication.

As predicted, the Naive Matrix Multiplication algorithm was the slowest performance. However, when comparing the Line Matrix Multiplication with the Block Matrix Multiplication, we observed that the difference between the two algorithms is almost insignificant, even though they both showed a significant time improvement from the Naive Matrix Multiplication.

Finally, and once again according to the theoretical predictions, the Parallel Computing implementations of the Line Matrix Multiplication algorithm proved to be the fastest experiences of all.

Between the two parallel algorithms, the Outer Loop Parallelized Line Multiplication algorithm proved to be consistently slightly less time consuming compared to the Outer and Inner Loop Parallelized Line Multiplication algorithm, as predicted.

4.2. Comparison of Block Matrix Multiplication algorithm with different block sizes

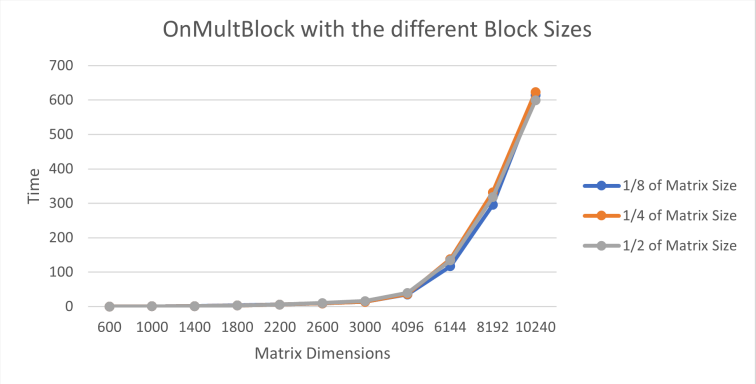

Figure 5 - Comparison graphic for different block sizes in the Block Matrix Multiplication algorithm

To analyze the effect of the block sizes on the efficiency of the Block Matrix Multiplication algorithm, we compared the multiplication of different matrix dimensions from 600 to 10240, using three distinct block sizes — ½, ¼ and ⅛ of the corresponding size of the block.

The time differences observed in our findings were minimal, leading to an inconclusive determination of block size impact on algorithm efficiency. Consequently, the analysis did not provide a definitive insight into the effect of block size variations on execution times.

4.3. Comparison between the C++ and Java algorithm implementations

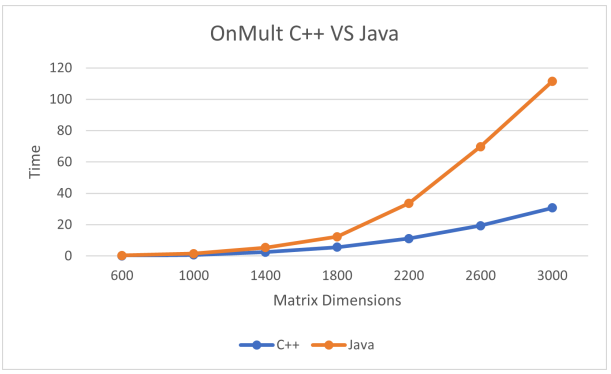

Figure 6 - Comparison graphic between C++ and Java in the Naive Matrix Multiplication algorithm

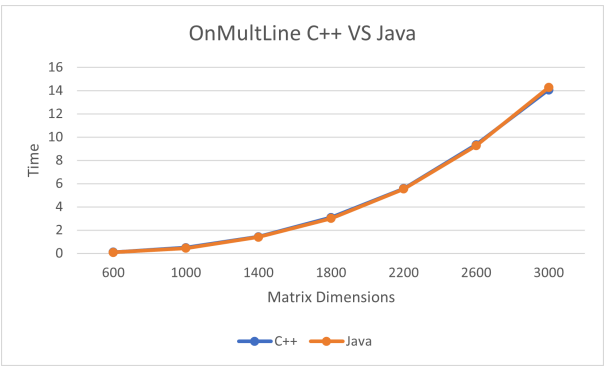

Figure 7 - Comparison graphic between C++ and Java in the Line Matrix Multiplication algorithm

To compare the performance of Java and C++ implementations, we did a side-by-side execution time comparison of algorithms developed in both languages.

The results from our analysis revealed that the C++ in the Naive Matrix Multiplication algorithm had a significant improvement in performance compared to Java. This outcome can be attributed to the C++ more efficient handling of lower-level operations. However, when analyzing the results of the Line Matrix Multiplication algorithm, we observe that the performance difference between both languages is imperceptible.

From these observations, it can be concluded that, overall, the C++ version has a better performance than the other language.

4.4. Comparison between cache performance of the algorithms

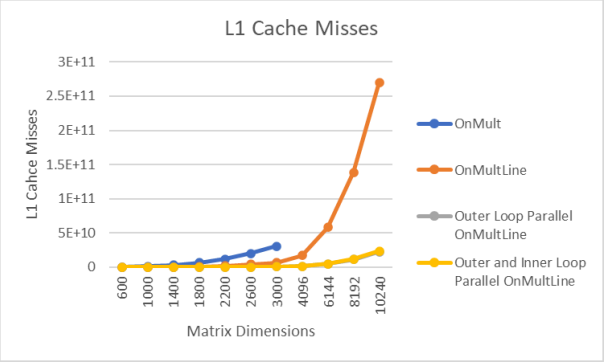

Figure 8 - L1 Cache Misses Comparison graphic between the Sequential and Parallel Matrix Multiplication algorithms

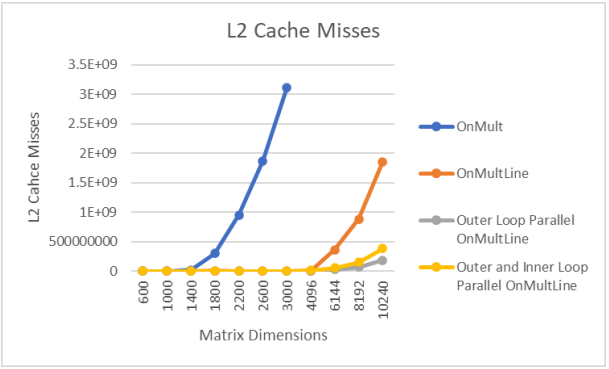

Figure 9 - L2 Cache Misses Comparison graphic between the Sequential and Parallel Matrix Multiplication algorithms

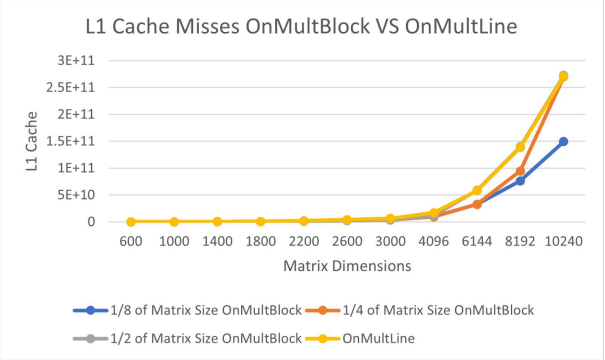

Figure 10 - L1 Cache Misses Comparison graphic between the Line Matrix Multiplication algorithm and the Block Matrix multiplication algorithm with the different block sizes

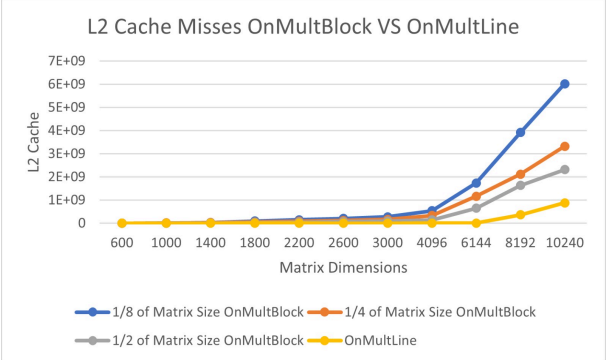

Figure 11 - L2 Cache Misses Comparison graphic between the Line Matrix Multiplication algorithm and the Block Matrix multiplication algorithm with the different block sizes

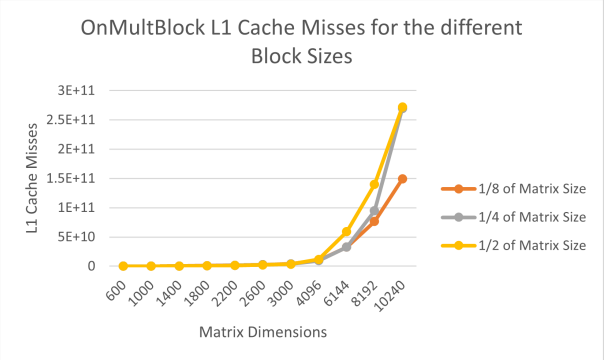

Figure 12 - L1 Cache Misses Comparison graphic between all the Block Sizes in the Block Matrix Multiplication Algorithm

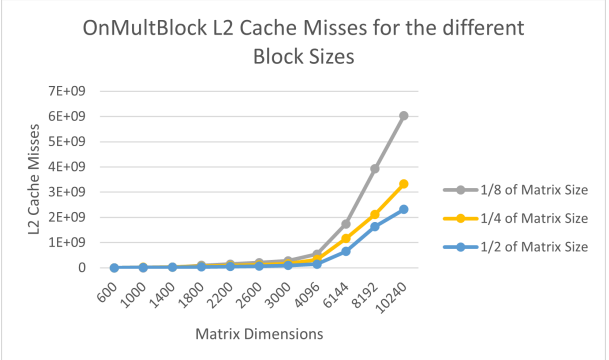

Figure 13 - L2 Cache Misses Comparison graphic between all the Block Sizes in the Block Matrix Multiplication Algorithm

A study was conducted on the effects of algorithm optimizations on cache utilization for the multiplication of matrices. This study found that the Line Matrix Multiplication algorithm significantly reduces cache misses compared to the Naive Matrix Multiplication algorithm. However, when contrasting Line and Block Matrix Multiplication, a decrease in level 1 cache misses was noted, due to an increase in level 2 cache misses.

4.5. Comparison between FLOPS

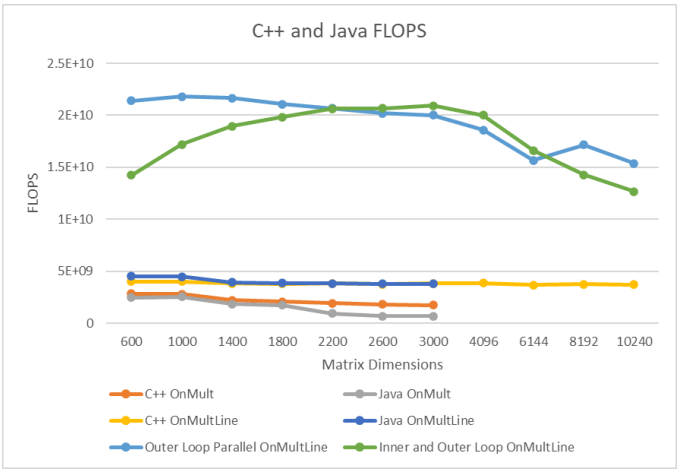

Figure 14 - Comparison graphic of FLOPS for all the Line and Naive Multiplication algorithms in Java and C++, Sequential and Parallel

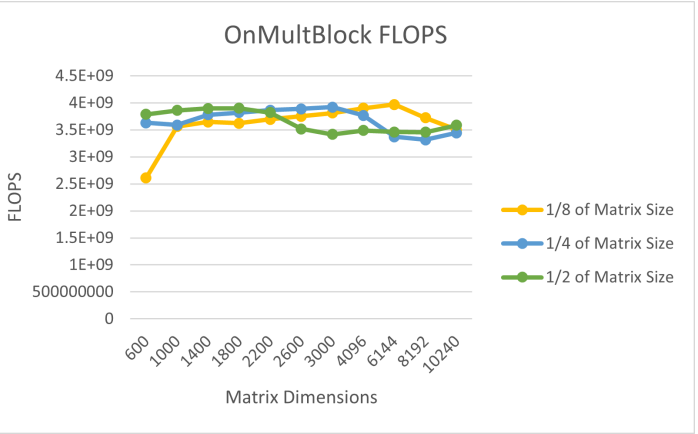

Figure 15 - Comparison graphic of FLOPS for the Block Multiplication algorithm

Calculating FLOPS offers a quantifiable metric to assess the computational speed of our algorithms. Through this calculation, it was anticipated and confirmed that, between the two programming languages used, algorithms implemented in C++ have a better performance, and between Parallel and Sequential Computing, the former presents better performance.

The outline of the speedup values of each of these algorithms and consequently their efficiency values, is similar to the one of their FLOPS. The Outer Loop Parallelized Line Multiplication shows higher efficiency for Matrix Sizes in the interval of 600 to 2200 and 6144 (exclusive) to 10240 when compared to the Inner and Outer Loop Parallelization which has shown to be more efficient from Matrix Sizes of 2200 (exclusive) to 6144.

5. Conclusions

This project has shown us the crucial role of algorithm optimization in enhancing the performance of the program. By selecting and improving our matrix multiplication algorithms, we were able to minimize cache misses and make efficient use of processor resources. This not only improved the speed of our calculations but also provided valuable insights into the effective management of cache memory.

In summary, our final results highlighted the importance of considering both spatial and temporal data locality in program design, emphasizing that choosing the right algorithms and optimization can lead to significant performance gains in computational tasks.

6. References

- https://www.cpu-world.com/CPUs/Zen/AMD-Ryzen%205%205500.html, Accessed on 15/03/2024