1st CPD Project

Informatics and Computing Engineering, FEUP

3LEIC10, Group 11

Adriano Machado, up202105352@up.pt.
André Rodrigues, up202108721@up.pt.
Daniel Dória, up202108808@up.pt.

1 Problem Description

This project aims to analyze the impact on CPU performance during the processing of extensive datasets using matrix multiplication. Divided into two parts, the study will evaluate performance metrics in both single and multi-core environments. By implementing diverse algorithms in C++ and Julia for matrix multiplication and leveraging the PAPI API to gather relevant performance data directly linked to CPU activity, we aim to provide valuable insights into the effectiveness of different techniques on processor performance.

2 Algorithms Explanation

In the initial phase of this project, three distinct matrix multiplication algorithms were employed:

1. Column Matrix Multiplication
2. Line Matrix Multiplication
3. Block Matrix Multiplication

For both the **Column Multiplication** and **Line Multiplication** algorithms, a comparative analysis between their C++ implementations and those in another programming language was required. Consequently, we chose Julia, renowned for its suitability in numerical computation and high-performance computing tasks.

In the second phase, we were tasked with developing two parallel versions of the **Line Multiplication** algorithm.

2.1 Single-Core Algorithm Variants

2.1.1 Column Multiplication

This algorithm closely resembles the conventional manual matrix multiplication method. It involves multiplying each element of one matrix row by each corresponding element of the other matrix column. The initial C++ implementation was provided by the teaching staff, and we subsequently implemented it in Julia.

```
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;
    }
}
```

2.1.2 Line Multiplication

The Line Multiplication algorithm differs from Column Multiplication in the order of its nested for loops. Despite this seemingly minor distinction, it offers notable advantages in terms of efficiency, particularly by mitigating cache misses as we shall explore.

```
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];
        }
    }
}
```

2.1.3 Block Multiplication

The **Block Multiplication** algorithm breaks down matrices into smaller blocks for multiplication. The algorithm involves six nested for loops. The three outer loops handle the selection of submatrices for the computation, while the three inner loops execute the multiplication operations within these chosen submatrices.

This method yields notable performance advantages, particularly evident when dealing with sizable datasets. Breaking matrices into blocks mitigates cache inefficiencies. Large matrices often exceed cache capacity, resulting in frequent cache misses and consequent slowdowns in computation. By utilizing smaller blocks, this algorithm optimizes cache utilization, thereby minimizing cache misses and enhancing overall performance.

```
for (int rbs = 0; rbs < m_ar; rbs += bkSize){
    for (int cbs = 0; cbs < m_ar; cbs += bkSize){
        for (int ibs = 0; ibs < m_ar; ibs += bkSize){
            for (int r = rbs; r < rbs + bkSize; r++){
                for (int c = cbs; c < cbs + bkSize; c++){
                    for (int i = ibs; i<ibs + bkSize; i++){
    phc[r*m_ar + i] += pha[r*m_ar + c] * phb[c*m_br + i];
                    }
                }
            }
        }
    }
}
```

2.2 Multi-Core Algorithm Variants

2.2.1 First Implementation

The first parallel implementation employs OpenMP directives for straightforward parallelization. Using the **omp parallel for** directive, the outer loop is parallelized, distributing loop iterations among multiple threads. This allows concurrent computation of matrix multiplication within the nested loops, enhancing performance.

```
int n_threads = omp_get_max_threads();

#pragma omp parallel for num_threads(n_threads) private(j,k)
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];
        }
    }
}
```

2.2.2 Second Implementation

The second implementation of the parallel matrix multiplication algorithm presents a significant departure from the first approach. While both methods utilize OpenMP directives for parallelization, the second variant introduces nested parallelism. In this approach, a parallel region is established, within which the innermost loop responsible for matrix multiplication is targeted by a **omp parallel for** directive.

```
int n_threads = omp_get_max_threads();

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

3 Performance metrics

To ensure consistency and reliability in our algorithm evaluations, we conducted all tests on a single machine, minimizing external influences on the results. Additionally, the machine remained connected to a power outlet throughout testing to prevent interruptions caused by CPU battery management. Below are the specifications of the machine used for our evaluations:

To assess the performance of our C++ algorithms, we leveraged the **PAPI API**, providing insights into CPU metrics and utilization of CPU cache memory during the execution. In addition to measuring the algorithm's execution time, we monitored the occurrence of cache misses at both L1 and L2 cache levels. Furthermore, for variants of the Multi-Core Algorithm, we computed MFlops, speedup, and efficiency metrics.

Component	Specification
Operating System	Ubuntu 5.15.146.1-microsoft-standard-WSL2
Processor	Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
RAM	12.0 GB (11.9 GB usable)
L1d Cache	192 KiB (6 instances)
L1i Cache	192 KiB (6 instances)
L2 Cache	1.5 MiB (6 instances)
L3 Cache	12 MiB (1 instance)

Table 1: Machine Specifications

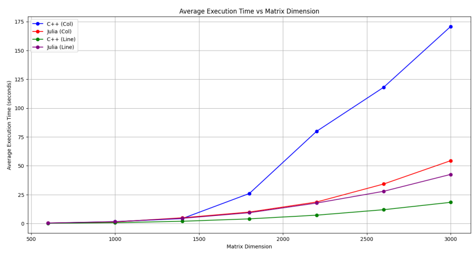
4 Results and Analysis

4.1 Column and Line matrix multiplication algorithms

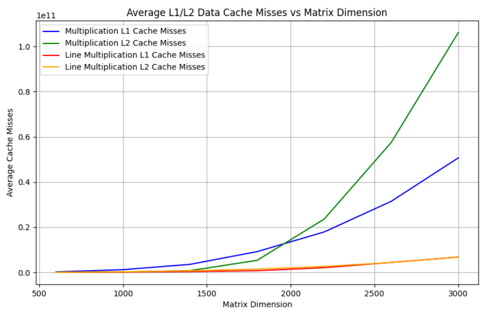
Figure (a) demonstrates that line matrix multiplication consistently outperforms column matrix multiplication in terms of execution time. This superiority can be attributed, at least in part, to the enhanced cache locality inherent in the line multiplication algorithm. By optimizing data reuse within the CPU cache, the line multiplication approach effectively reduces the occurrence of cache misses, resulting in notable performance enhancements.

The comparison between C++ and Julia implementations reveals diverse performance disparities across the matrix multiplication algorithms. While Julia demonstrates an execution time advantage for the line matrix multiplication, possibly due to its proficiency in numerical computation and Just-In-Time (JIT) compilation, differences emerge concerning the column algorithm. Despite both languages exhibiting reductions in execution time, C++ demonstrates a more substantial decrease, resulting in shorter execution times for the column algorithm compared to Julia's implementation.

Furthermore, Figure (b) reinforces the performance advantage of line matrix multiplication over column multiplication. As matrix dimensions increase, both algorithms experience a rise in L1 and L2 cache misses due to growing data demands exceeding cache capacity. However, line multiplication consistently exhibits lower cache miss counts compared to column multiplication for both cache levels.



(a) Average execution time comparison of column and line matrix multiplication algorithms



(b) Average number of cache misses related to matrix size

4.2 Line and Block matrix multiplication algorithms

In general, block matrix multiplication can be faster than traditional line-by-line matrix multiplication due to improved cache utilization and reduced memory access overhead. This improvement is especially noticeable on modern computer architectures with hierarchical memory systems like CPUs and GPUs.

If we take as an example the measurements made in the chart below (Figure 3b), regardless of the size of the blocks, the block multiplication is always faster than the line multiplication, thus confirming the statement of the previous paragraph.

Block matrix multiplication can be faster than Line matrix multiplication mainly due to reduced memory access overhead and cache utilization. When performing matrix multiplications, we are repeatedly accessing elements of the matrices. By using blocks, we can fit more elements of the matrices into cache at once reducing the number of times needed to fetch data from main memory and also we minimize the number of times needed to access the main memory, as we're performing operations on smaller subsets of the matrices.

Overall, while implementing **Block Matrix Multiplication** requires more complex code compared to **Line Matrix Multiplication**, it results in significant performance improvements, especially for large matrices.

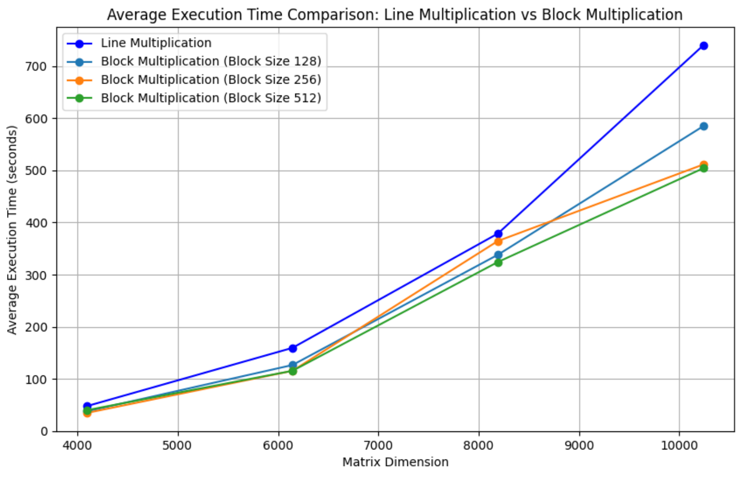


Fig. 2: Line and block matrix multiplication execution time comparison

4.3 Parallel Line Matrix Multiplication Approaches

Using **OpenMP**, we implemented two different parallelization techniques to assess whether they could improve the performance of our line matrix multiplication algorithm. The resulting average execution times, speedup, and efficiency were as follows:

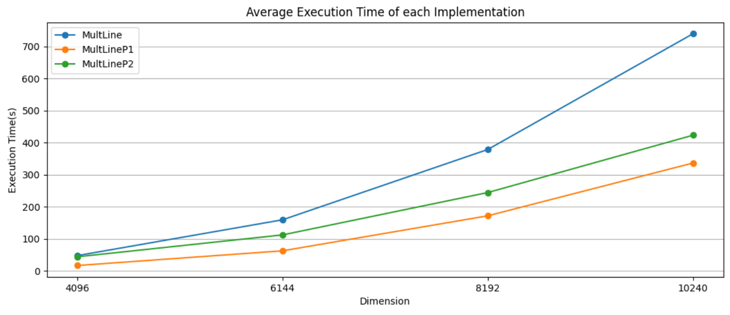
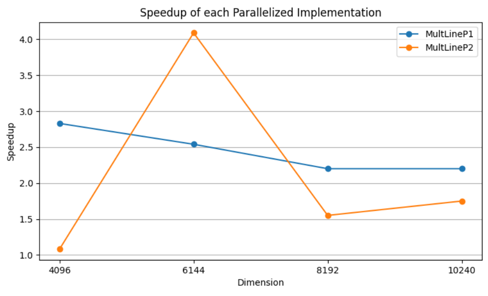
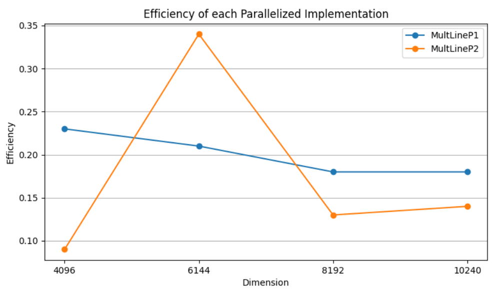


Fig. 3: Parallelized and non Parallelized execution time comparison



(a) Speedup comparison between Parallelized algorithms



(b) Efficiency comparison between Parallelized algorithms

Upon analysis of this data, it becomes evident that the employment of parallelization techniques always returned better results than the plain algorithm, no matter the implementation.

5 Conclusions

This project has improved our knowledge of how crucial memory management is to increasing program efficiency. Through careful analysis of memory management strategies in conjunction with the hardware they run on, sequential programs can achieve significant improvements even in the absence of parallel computing.

Furthermore, the programming language selection becomes crucial for putting computational solutions into practice. Apart from the inherent characteristics of each language that make it easier to apply specific ideas, it is critical to evaluate the degree to which the language can carry out recommended activities at different levels of complexity.