Faculty of Engineering of the University of Porto

CPD Report: Assignment 1

Juliana Durán (202411311)
Karlo Jagar (202411506)
Korina Jurić (202411535)

March 2025

0. Introduction

This project looks at how a computer's memory system affects performance when working with large amounts of data. We'll use matrix multiplication as our test case since it requires accessing lots of data in different patterns. To measure how well our programs perform, we'll use the Performance API (PAPI) to collect important statistics during execution. These measurements will help us understand the differences between various matrix multiplication methods and how they interact with the computer's memory system when handling matrices of different sizes.

1. Problem Description

1.1 Performance evaluation of a single core

To evaluate performance, we implement and compare three different matrix multiplication strategies across two programming languages. Our goal is to determine which approach achieves the best execution time.

The three different matrix multiplication strategies implemented and tested were:

1. **Standard**: each element in the result matrix is computed by iterating through rows of the first matrix and columns of the second matrix.
2. **Line**: each element of the first matrix is multiplied with the corresponding row of the second matrix.
3. **Block-Oriented**: the matrices are divided into smaller sub-blocks, and multiplication is performed block by block.

Each of these approaches is implemented in **C++** and **Python** to compare execution times and determine which method and language combination is the most efficient.

1.2 Performance evaluation of a multi-core implementation

To evaluate performance, we implement and compare two different parallelization strategies using C++ with OpenMP. Our goal is to determine which approach achieves the best execution time.

The two different parallelization strategies implemented and tested were:

1. **Parallelizing the Outer Loop**: In this method, the outermost loop is divided among multiple threads, allowing each thread to process a different row of the matrix simultaneously.
2. **Parallelizing the Inner Loop**: In this method, the outer loops run sequentially while the innermost loop is distributed among multiple threads.

2. Algorithm explanation

Standard Matrix Multiplication directly implements the mathematical definition by using three nested loops to compute each element of the result matrix as the dot product of a row from the first matrix and a column from the second matrix. This naive approach is straightforward but often inefficient for large matrices due to poor cache utilization.

Line Matrix Multiplication processes the computation by rows, where for each row of the first matrix, it computes the products with all relevant elements from the second matrix before moving to the next row. This approach improves locality of reference by reusing row data that's already been loaded into cache.

Block Matrix Multiplication divides the input matrices into smaller sub-matrices or blocks, then performs multiplication on these blocks. This technique significantly improves cache performance by ensuring that blocks fit entirely within cache memory, reducing the number of expensive memory accesses and resulting in better overall performance for large matrices.

2.1 Python Codes

2.1.1 Standard Matrix Multiplication

```python
for i in range(m_ar): # For each row in the matrix A
    for j in range(m_br): # For each column in the matrix B
        temp = 0.0  # Temporary value for the result matrix cell (i, j)
        for k in range(m_ar): # For each element in the row/column pair of the two matrices
            temp += pha[i][k] * phb[k][j] # Multiply and add to the temporary value
        phc[i][j] = temp # Assign the temporary value to the result matrix cell (i, j)
```

2.1.2 Line Matrix Multiplication

```python
for i in range(m_ar): # For each row in the matrix A
    for k in range(m_ar): # For each element in the row of the matrix A
        temp = pha[i][k] # Store the element in a temporary variable
        for j in range(m_br): # For each column in the matrix B
            # Multiply the element with the corresponding element in the matrix B and
            # add to the result matrix cell (i, j)
            phc[i][j] += temp * phb[k][j]
```

2.2 C++ Codes

2.2.1 Standard Matrix Multiplication

```cpp
for(i = 0; i < m_ar; i++) { // For each row in the matrix A
    for(j = 0; j < m_br; j++) { //  For each column in the matrix B
        temp = 0; // Temporary value for the result matrix cell (i, j)
        for(k = 0; k < m_ar; k++) { // For each element in the row/column pair of the two matrices
            temp += pha[i * m_ar + k] * phb[k * m_br + j]; // Multiply and add to the temporary value
        }
        phc[i * m_ar + j] = temp; // Assign the temporary value to the result matrix cell (i, j)
    }
}
```

2.2.2 Line Matrix Multiplication

```
for(i = 0; i < m_ar; i++) { // For each row in the matrix A
    for(k = 0; k < m_ar; k++) { // For each element in the row of the matrix A
        double temp = pha[i * m_ar + k]; // Store the element in a temporary variable
        for(j = 0; j < m_br; j++) // For each column in the matrix B
        // Multiply the element with the corresponding element in the matrix B and add to the result matrix cell (i, j)
        phc[i * m_ar + j] += temp * phb[k * m_br + j];
    }
}
```

2.2.3 Block Matrix Multiplication

```
for(ii = 0; ii < m_ar; ii += bkSize) // For each block row in the matrix A
    for(kk = 0; kk < m_ar; kk += bkSize) // For each block column in the matrix A
        for(jj = 0; jj < m_br; jj += bkSize) // For each block column in the matrix B
            for(i = ii; i < min(ii + bkSize, m_ar); i++) // For each row in the block row of the matrix A
                for(k = kk; k < min(kk + bkSize, m_ar); k++) { // For each element in the row of the block row of the matrix A
                    double temp = pha[i * m_ar + k]; // Store the element in a temporary variable
                    for(j = jj; j < min(jj + bkSize, m_br); j++) // For each column in the block column of the matrix B
                    // Multiply the element with the corresponding element in the matrix B and
                    // add to the result matrix cell (i, j)
                        phc[i * m_ar + j] += temp * phb[k * m_br + j];
                }
```

2.2. Outer Loop Matrix Multiplication

```
// Creates a team of threads with private copies of i,j,k,temp and shared access to matrices
#pragma omp parallel for private(i,j,k,temp) shared(pha, phb, phc)
for(i = 0; i < m_ar; i++) { // For each row in the matrix A
    for(j = 0; j < m_br; j++) { //  For each column in the matrix B
        temp = 0; // Temporary value for the result matrix cell (i, j)
        for(k = 0; k < m_ar; k++) { // For each element in the row/column pair of the two matrices
            temp += pha[i * m_ar + k] * phb[k * m_br + j]; // Multiply and add to the temporary value
        }
        phc[i * m_ar + j] = temp; // Assign the temporary value to the result matrix cell (i, j)
    }
}
```

Outer Loop Parallelization spreads the workload by assigning different rows of the result matrix to separate threads. Each thread handles complete rows on its own, which means better memory access patterns and less overhead from thread management. This method usually runs faster because threads don't need to wait for each other while doing calculations.

2.2.5 Inner Loop Matrix Multiplication

```
// Creates a team of threads with private copies of i,j,k,temp and shared access to matrices
#pragma omp parallel private(i,j,k,temp) shared(pha, phb, phc)
for(i = 0; i < m_ar; i++) { // For each row in the matrix A
    for(j = 0; j < m_br; j++) { //  For each column in the matrix B
        temp = 0; // Temporary value for the result matrix cell (i, j)
        #pragma omp for // Parallelize the inner loop
        for(k = 0; k < m_ar; k++) { // For each element in the row/column pair of the two matrices
            temp += pha[i * m_ar + k] * phb[k * m_br + j]; // Multiply and add to the temporary value
        }
        phc[i * m_ar + j] = temp; // Assign the temporary value to the result matrix cell (i, j)
    }
}
```

Inner Loop Parallelization splits up the work for calculating each single element of the result matrix among multiple threads. While this sounds good in theory, it actually creates more problems than it solves in most cases. Threads have to communicate more often, and they end up fighting over the same memory locations, which slows everything down except in some specialized cases with huge matrices.

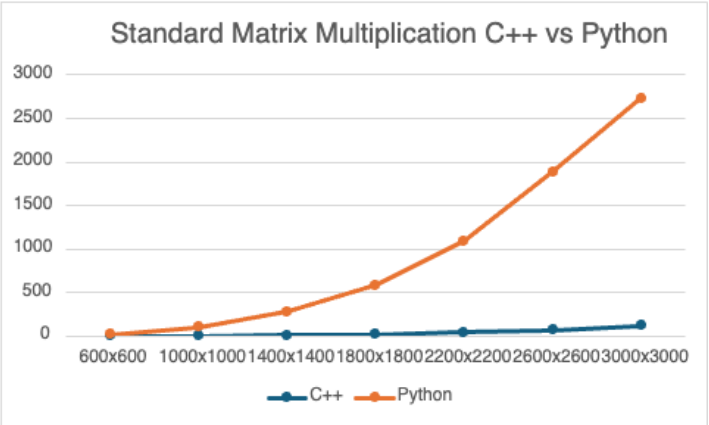
3. Performance metrics

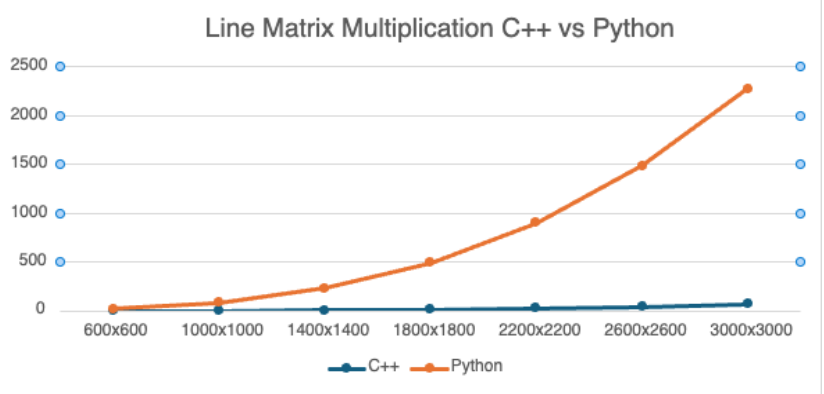
All data collected from our tests is available in the **Assigment 1 CPD Data** Excel spreadsheet, which contains comprehensive tables and visual graphs of the results. Testing was conducted on the Linux operating system using FEUP's laboratory computers.

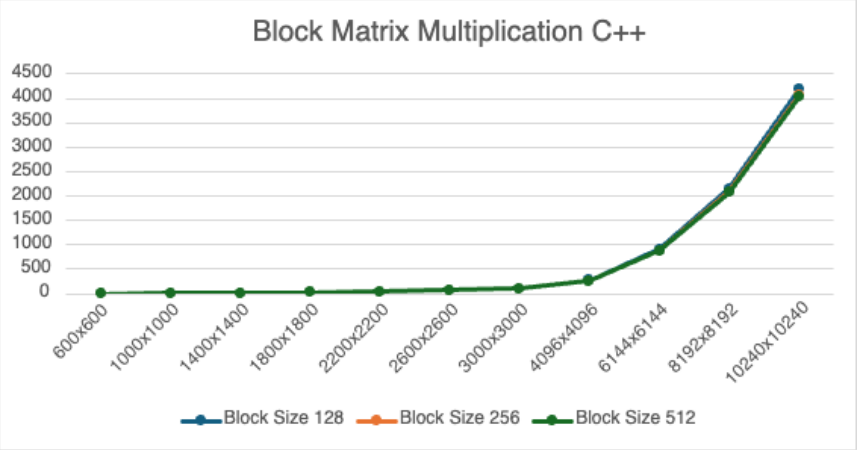
4. Results and analysis

4.1 Standard vs Line vs Block Matrix Multiplication

The graphs reveal performance differences between matrix multiplication implementations. In both Standard and Line multiplication methods, C++ dramatically outperforms Python, with Python's execution time increasing exponentially as matrix dimensions grow while C++ maintains relatively low execution times even at 3000×3000 matrices. This performance gap highlights the significant advantage of compiled languages over interpreted ones for computation-heavy tasks. The Block matrix multiplication graph shows this method scales better to extremely large matrices (up to 10240×10240), though execution time eventually spikes at the largest dimensions. Interestingly, different block sizes (128, 256, and 512) appear to have minimal impact on performance as their lines overlap almost completely. These results underscore how both programming language choice and algorithm selection critically impact matrix multiplication efficiency.





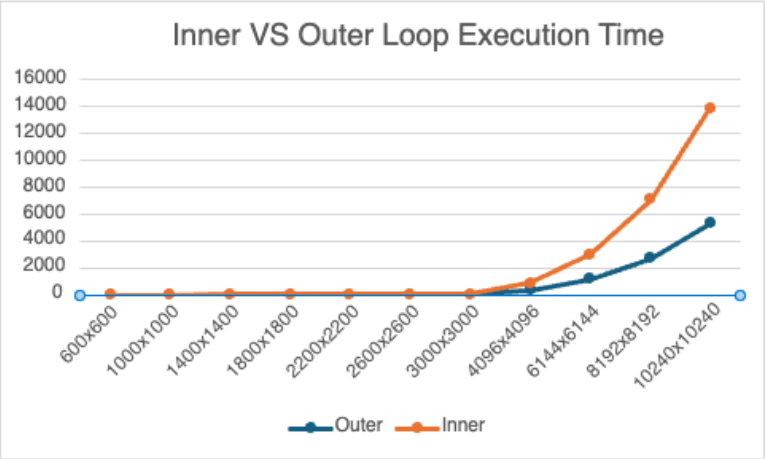


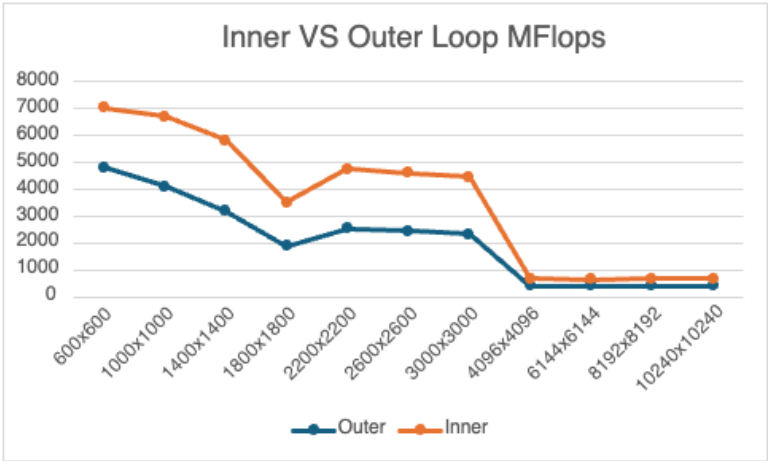
4.2 Inner Loop vs Outer Loop Parallelization

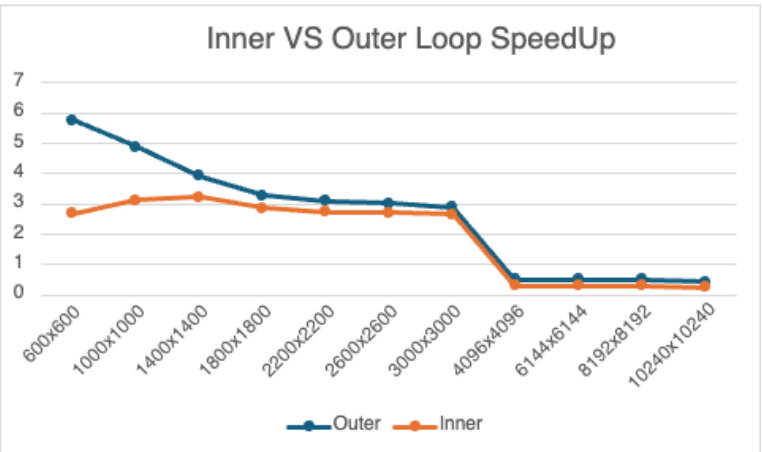
The comparison between Inner and Outer Loop parallelization approaches reveals several important performance patterns. The execution time graph shows that the Outer Loop implementation performs better than the Inner Loop approach, with the difference becoming much larger at bigger matrix sizes (8192×8192 and above). While both methods show similar execution times up to 3000×3000 matrices, the Inner Loop approach shows a sharp increase in execution time for very large matrices, reaching about 14,000s at 10240×10240 compared to about 6,000s for the Outer Loop implementation.

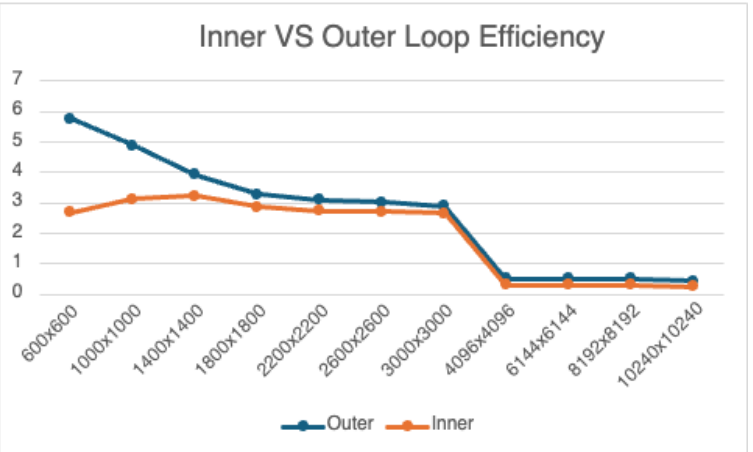
The MFlops graph shows better performance for the Inner Loop approach with smaller matrices, but both methods lose computational efficiency as matrix sizes increase, eventually performing similarly at very large sizes. The SpeedUp and Efficiency graphs show similar trends, with the Outer Loop implementation performing better with smaller matrices (reaching about 6× speedup for 600×600 matrices) but becoming similar to Inner Loop performance at larger sizes. Beyond 4096×4096, both methods show very little speedup and efficiency gains, approaching zero for the largest tested matrices.

These results show that Outer Loop parallelization works better overall, especially for small to medium-sized matrices. The Outer Loop approach is better because the threads have more substantial work to do. The work is more evenly spread among threads, which reduces overhead and improves memory access patterns, though neither method works well for extremely large matrices.









5. Conclusion

In this project, we studied how a computer's memory system affects performance when working with large amounts of data through matrix multiplication. Our tests showed several important findings. First, the choice of programming language makes a big difference, with C++ running much faster than Python because it's compiled rather than interpreted. Among the different matrix multiplication methods we tried, Block Matrix Multiplication worked best for large matrices because it uses the computer's cache memory more efficiently.

When we looked at ways to split the work across multiple processor cores, we found that Outer Loop Parallelization performed better than Inner Loop Parallelization, especially for small to medium-sized matrices. This happens because Outer Loop gives each thread more substantial work to do and spreads the workload more evenly, reducing overhead and improving memory access patterns. However, both approaches struggled with very large matrices.

Our results show how important it is to choose the right algorithm, programming language, and parallelization method when doing computation or any kind of data manipulation. The way data moves through the computer's memory hierarchy greatly affects performance, and techniques that make good use of the cache show clear advantages.