Computação Paralela e Distribuída

Trabalho 1

21 de março de 2025

Tomás Oliveira - up202208415

Diogo Ferreira - up202205295

Álvaro Torres - up202208954

Index

1. Problem description

In this project, we aim to analyze the performance impact of different matrix multiplication algorithms on processor performance. The study will be conducted by implementing and evaluating various matrix multiplication techniques using both single-core and multi-core processing, leveraging the Performance API (PAPI) to collect execution metrics.

The key objectives of this project include:

1. **Single-Core Performance Evaluation:**
    - Implementing a basic matrix multiplication algorithm in C/C++ and another programming language of choice (e.g., Java, C#, Fortran) without using external libraries. Our choice was to use C#.
    - Measuring execution time for different matrix sizes, ranging from 600x600 to 3000x3000, with increments of 400.
    - Implementing an alternative matrix multiplication algorithm that optimizes memory access patterns by modifying the computation order.
    - Extending the performance evaluation to larger matrix sizes (4096x4096 to 10240x10240 with increments of 2048) for the C/C++ version.
    - Developing a block-oriented matrix multiplication algorithm in C/C++ that divides matrices into smaller blocks to optimize cache usage.

2. **Multi-Core Performance Evaluation:**
    - Implementing parallel versions of the two matrix multiplication algorithms using OpenMP.
    - Analyzing the performance improvement achieved through parallelization by evaluating key metrics such as Mega Floating Point Operations per Second (MFLOPS), speedup, and efficiency.
    - Comparing different OpenMP parallelization strategies, including parallelizing the outermost loop and parallelizing inner loops with #pragma omp parallel for.

2. Algorithms explanations

OnMult(int m_ar, int m_br) in C++

The OnMult(int m_ar, int m_br) function in C++ implements standard matrix multiplication using a straightforward row-by-column approach.

1. Memory Allocation and Initialization

```C/C++
double *matrixA, *matrixB, *matrixC;

matrixA = (double *)malloc((m_ar * m_ar) * sizeof(double));

matrixB = (double *)malloc((m_ar * m_ar) * sizeof(double));

matrixC = (double *)malloc((m_ar * m_ar) * sizeof(double));

initialize_matrices(matrixA, matrixB, matrixC, m_ar, m_br);
```

Three matrices are dynamically allocated as contiguous 1D arrays:

- matrixA: First matrix (dimensions m_ar × m_ar)
- matrixB: Second matrix (dimensions m_ar × m_ar)
- matrixC: Result matrix (dimensions m_ar × m_br)

The initialize_matrices() function initializes:

- matrixA with all elements set to 1.0
- matrixB with values where each element equals its row index + 1
- matrixC with all elements set to 0.0

2. Performance Measurement

```C/C++
double start_time = omp_get_wtime();

// Matrix multiplication code

double elapsed = omp_get_wtime() - start_time;
```

- Uses OpenMP's high-precision timer omp_get_wtime() to measure execution time
- Time is measured in seconds (not milliseconds like in C#)

3. Matrix Multiplication Algorithm (Triple Nested Loop)

```C/C++
for (int i = 0; i < m_ar; i++) {

      for (int j = 0; j < m_br; j++) {

      double temp = 0;

      for (int k = 0; k < m_ar; k++) {

            temp += matrixA[i * m_ar + k] * matrixB[k * m_br +
j];

      }

      matrixC[i * m_ar + j] = temp;

      }

}
```

The matrix multiplication follows the mathematical formula: $C[i,j] = \Sigma(A[i,k] \times B[k,j])$ for k=0 to m_ar-1

- Outer loop (i): Iterates through each row of matrix A
- Middle loop (j): Iterates through each column of matrix B
- Inner loop (k): Computes the dot product between row i of A and column j of B

The computation uses linear indexing i * m_ar + k to access the 2D matrix elements stored in a 1D array

4. Memory Management and Result

```C/C++
clean_matrices(matrixA, matrixB, matrixC);

return elapsed;
```

Explicitly frees the dynamically allocated memory through the clean_matrices() function
Returns the elapsed time in seconds as a double value

OnMult(int m_ar, int m_br) in C#

The provided C# function, OnMult(int m_ar, int m_br), performs matrix multiplication using a straightforward, row-by-column approach. Below is a breakdown of its functionality:

1. Variable Initialization

- Stopwatch stopwatch = new Stopwatch();
    - This initializes a Stopwatch instance to measure execution time.
- double[] pha = new double[m_ar * m_ar];
    - This array represents the first matrix (A), stored in a 1D array instead of a 2D array to optimize memory access.
- double[] phb = new double[m_ar * m_ar];
    - This array represents the second matrix (B), also stored in 1D.
- double[] phc = new double[m_ar * m_ar];
    - This array stores the result of the matrix multiplication (C).

2. Matrix Initialization

The first matrix pha is initialized with all elements set to 1.0. The second matrix phb is initialized such that phb[i][j] = (i + 1), meaning each row increases sequentially.

3. Matrix Multiplication (Triple Nested Loop)

The matrix multiplication follows the standard approach:

C[i][j]=∑k=0n−1A[i][k]×B[k][j]C[i][j] = \sum_{k=0}^{n-1} A[i][k] \times B[k][j]

- Outer Loop (i): Iterates over rows of the first matrix.
- Middle Loop (j): Iterates over columns of the second matrix.
- Inner Loop (k): Computes the dot product of the row from pha and column from phb.

Each element of phc is computed as:

```C/C++
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;
```

This performs a row-wise traversal of pha and a column-wise traversal of phb, storing the result in phc.

4. Performance Measurement

- stopwatch.Start(); begins tracking the execution time before matrix multiplication starts.
- stopwatch.Stop(); stops the timer after the computation is complete.
- The total execution time is stored in stopwatch.ElapsedMilliseconds.

Key Observations

1. **1D Array Representation**
    - Instead of using double[,] (a 2D array), matrices are stored in a flattened 1D array to enhance cache locality and reduce pointer dereferencing overhead.
    - pha[i * m_ar + j] simulates access to a 2D matrix stored in row-major order.

2. **Computational Complexity**
    - The time complexity of this algorithm is $O(n^3)$ since it uses three nested loops.

3. **Performance Considerations**
    - Memory Access Pattern: The row-major order helps reduce cache misses.
    - Potential Optimization: Using loop unrolling, blocking, or parallelization (e.g., OpenMP in C++ or Parallel.For in C#) could improve performance.

This C# implementation mirrors the logic of the C++ algorithm, allowing for a fair performance comparison between the two languages when testing different matrix sizes.

OnMultLine(int m_ar, int m_br) in C++ and C#

The OnMultLine function in C++ performs matrix multiplication using a row-wise approach. Key aspects include:

- **Memory Allocation:** The matrices are dynamically allocated using malloc.
- **Initialization:** The function initialize_matrices initializes the matrices before multiplication.
- **Computation:** The outer loop iterates over rows, while an intermediate loop fetches an element from matrixA, optimizing memory access. The innermost loop computes the corresponding product and accumulates the results in matrixC.
- **Performance Measurement:** Execution time is measured using clock() and reported in seconds.
- **Output Verification:** The function prints the first 10 elements of the resulting matrix to verify correctness.
- **Memory Cleanup:** Dynamically allocated memory is freed using clean_matrices.

The OnMultLine function in C# mirrors the C++ implementation with similar logic:

- **Memory Allocation:** Uses one-dimensional arrays (pha, phb, and phc) to store matrices efficiently.
- **Initialization:** Matrix pha is initialized to 1.0, while phb is assigned values based on row indices.
- **Computation:** Uses a row-wise approach with an intermediate variable r to reduce redundant memory accesses.
- **Performance Measurement:** The Stopwatch class is used to measure execution time.
- **Output:** Calls PrintOrWriteResults to log the results.

OnMultBlock(int m_ar, int m_br, int bkSize) in C++ and C#

The OnMultBlock function introduces blocking to optimize cache performance.

- **Matrix Initialization:** Allocates and initializes matrices dynamically.
- **Multiplication Logic:**
    - Iterates over matrices in blocks of size bkSize.
    - Each block is processed independently to improve cache locality.
    - min() ensures boundary conditions are handled correctly.
- **Time Measurement:** Uses clock().
- **Output:** Displays execution time for different block sizes.

The OnMultBlock function in C# follows the same block-based approach as the one implemented in C++.

- **Matrix Initialization:** Uses one-dimensional arrays with linear indexing for better cache behavior.
- **Multiplication Logic:**
    - Iterates over matrix blocks of size bkSize × bkSize.
    - Computes sub-blocks independently, improving cache efficiency.
    - Uses Math.Min() to handle boundary conditions at matrix edges.
    - Implements line-oriented multiplication within each block.
- **Time Measurement**: Uses Stopwatch for precise performance timing.
- **Output:** Execution time is printed using PrintOrWriteResults, including the block size in the algorithm name for analysis.

OnMultLineExtParallel(int m_ar, int m_br) in C++

The OnMultLineExtParallel function implements matrix multiplication using a line-oriented approach with external parallelization.

1. Memory Allocation and Initialization

```C/C++
double *matrixA, *matrixB, *matrixC;
matrixA = (double *)malloc((m_ar * m_ar) * sizeof(double));
matrixB = (double *)malloc((m_ar * m_ar) * sizeof(double));
matrixC = (double *)malloc((m_ar * m_ar) * sizeof(double));
initialize_matrices(matrixA, matrixB, matrixC, m_ar, m_br);
```

Three matrices are dynamically allocated as contiguous 1D arrays, following the same pattern as the standard implementation. The matrices are initialized with the same values as in the standard algorithm.

2. Performance Measurement

```C/C++
double start_time = omp_get_wtime();
// Parallelized matrix multiplication code
double elapsed = omp_get_wtime() - start_time;
```

The execution time is measured using OpenMP's high-precision timer, capturing only the time spent in the actual computation.

3. Parallelized Matrix Multiplication Algorithm

```C/C++
#pragma omp parallel for
for (int i = 0; i < m_ar; i++) {
    for (int k = 0; k < m_ar; k++) {
        double temp = matrixA[i * m_ar + k];
        for (int j = 0; j < m_br; j++) {
            matrixC[i * m_ar + j] += temp * matrixB[k * m_br +
j];
        }
    }
}
```

The algorithm uses a line-oriented approach (i-k-j loop order) with external parallelization:

- The #pragma omp parallel for directive parallelizes the outermost loop (i-loop)
- Each thread processes different rows of the result matrix independently
- Threads work on separate portions of matrixC with no data dependencies between them
- This approach has minimal synchronization overhead as threads operate on independent data

4. Memory Management and Result

```C/C++
clean_matrices(matrixA, matrixB, matrixC);
return elapsed;
```

The allocated memory is freed, and the elapsed time is returned as the function result.

OnMultLineIntParallel(int m_ar, int m_br) in C++

The OnMultLineIntParallel function also implements matrix multiplication using a line-oriented approach but with internal parallelization.

1. Memory Allocation and Initialization

```C/C++
double *matrixA, *matrixB, *matrixC;
matrixA = (double *)malloc((m_ar * m_ar) * sizeof(double));
matrixB = (double *)malloc((m_ar * m_ar) * sizeof(double));
matrixC = (double *)malloc((m_ar * m_ar) * sizeof(double));
initialize_matrices(matrixA, matrixB, matrixC, m_ar, m_br);
```

The memory allocation and initialization are identical to the other implementations.

2. Performance Measurement

```C/C++
double start_time = omp_get_wtime();
// Internally parallelized matrix multiplication code
double elapsed = omp_get_wtime() - start_time;
```

Execution time is measured in the same way as the other implementations.

3. Internally Parallelized Matrix Multiplication Algorithm

```C/C++
#pragma omp parallel
{
    for (int i = 0; i < m_ar; i++) {
        for (int k = 0; k < m_ar; k++) {
            double temp = matrixA[i * m_ar + k];
            #pragma omp for
            for (int j = 0; j < m_br; j++) {
                matrixC[i * m_ar + j] += temp * matrixB[k *
m_br + j];
            }
        }
    }
}
```

The algorithm employs a line-oriented approach with internal parallelization:

- A parallel region is created once with `#pragma omp parallel`
- All threads execute the same i and k iterations together
- The innermost loop (j-loop) is parallelized with `#pragma omp for`
- For each (i,k) pair, threads divide the j-loop iterations among themselves
- An implicit barrier occurs at the end of each j-loop, synchronizing all threads
- The temp variable is properly shared among threads as it's calculated before the parallel j-loop

4. Memory Management and Result

```C/C++
clean_matrices(matrixA, matrixB, matrixC);
return elapsed;
```

As with the other implementations, allocated memory is freed, and the elapsed time is returned.

3. Performance metrics

When diving into matrix multiplication performance, we needed to select metrics that would give us a complete picture of how these algorithms behave in real-world conditions. Time measurements are the core of our analysis, but they only allow for a partial analysis. To truly understand what's happening, we expanded our investigation to include several other metrics.

Execution time serves as our primary standard. We measured this in seconds using OpenMP's timing functions in C++ and the Stopwatch class in C#. This straightforward metric lets us make immediate comparisons between different implementations.

We incorporated MFLOPS (Mega Floating Point Operations per Second) into our analysis to also showcase the computational workload being performed. For matrix multiplication, we know each N×N operation requires roughly 2×N^3 floating-point operations. By dividing this by execution time, we get a normalized measure of computational efficiency that remains meaningful across different matrix sizes. This helps us understand whether an algorithm is truly more efficient or just dealing with a smaller workload.

The relationship between algorithms and memory hierarchy emerged as a crucial factor in our analysis, so we collected data about cache misses using PAPI. Each time the processor requests data not found in L1 or L2 cache, it causes significant delays while fetching from slower memory levels. In the computer we tested the algorithms in, that had an Intel Core i7-9700 CPU with 64KB L1 cache per core (32KB for instructions + 32KB for data) and 256KB L2 cache per core, these delays can significantly impact performance.

For our parallel implementations, we used two more metrics. Speedup tells us how much faster the parallel version runs compared to the sequential one. Efficiency reveals how we're using the additional computing resources. These metrics helped us identify when adding more cores stops being beneficial and why certain parallelization approaches work better than others.

In summary, we analysed execution time for both C++ and C# implementations and MFLOPS, speedup, efficiency and L1 and L2 cache misses for the C++ version.

Together, these metrics allowed us to build a comprehensive understanding of matrix multiplication performance across the different algorithms implementations in C++ and C#.

4. Results and analysis

Single-Core Performance Analysis

Language Performance Difference: C++ vs C#

Our tests revealed a consistent and significant performance difference between C++ and C# implementations across all algorithms and matrix sizes. C++ outperformed C# in every test we ran.

Looking at the execution time comparison in "execution_time_comparison.png" (Figure 5), the gap becomes increasingly apparent as matrix dimensions grow. We can quantify this difference more precisely from "relative_performance.png" (Figure 14), which shows that C# runs 1.5 to 3.1 times slower than C++ for the standard algorithm. Perhaps more striking is that for the line algorithm, this performance penalty jumps to 4.3 to 5.8 times slower. When we tested the block algorithms with large matrices, C# consistently performed about 4 times slower than C++, regardless of the block size or matrix dimensions.

Several factors can explain these significant differences. First of all, there's the fundamental difference in execution models. C++ code compiles directly to native machine code that runs on the processor, while C# operates within the Common Language Runtime (CLR), using Just-In-Time compilation that introduces unavoidable overhead. This extra layer between the code and hardware creates performance penalties that are difficult to eliminate.

Memory management also plays a crucial role. Our C++ implementation uses direct memory allocation with manual management, giving us precise control over memory layout and access patterns. In contrast, C# relies on garbage collection and adds layers of memory indirection that impact performance.

The way arrays are implemented and accessed differs between the languages as well. Although both our C++ and C# code use similar 1D array representations with manual indexing calculations, the underlying memory handling by the CLR introduces additional overhead that isn't present in the C++ version.

Finally, compiler optimizations make a difference. We compiled our C++ code with the -O2 optimization flag, giving us optimizations that the C# JIT compiler might not match. These optimizations can significantly improve loop execution and memory access patterns.

We noticed the performance difference narrowing slightly with larger matrices. This suggests that as problem sizes increase, memory access patterns and cache behavior become the dominant performance factors, reducing the impact of language-specific overhead.

Algorithm Efficiency

Beyond language differences, we found big efficiency variations between different multiplication algorithms, with consistent patterns emerging across both C++ and C# implementations.

The line algorithm consistently outperformed the standard algorithm by a substantial margin. In "execution_time_comparison.png" (Figure 5), we can see this gap becoming bigger as matrix dimensions increase. By the time we reach 3000×3000 matrices, the difference becomes noticeable.

This performance difference comes directly from how these algorithms interact with the memory hierarchy. The standard algorithm in C++ (using i-j-k loop ordering) accesses the second matrix in a column-wise way, which is inefficient in languages that store matrices in row-major order. This creates frequent cache misses as the processor repeatedly jumps across distant memory locations.

In contrast, the C++ line algorithm (using i-k-j ordering) accesses both matrices in a row-wise pattern, taking full advantage of spatial locality and cache line utilization. "l1_cache_misses.png" (Figure 1) provides evidence for this explanation, showing dramatically fewer L1 cache misses for the line algorithm compared to the standard approach. Considering our Intel i7-9700's relatively small 64KB L1 cache per core, this access pattern optimization becomes particularly important.

The computational efficiency difference is equally striking. Looking at "mflops_comparison.png" (Figure 13), the line algorithm achieves between 3000-4300 MFLOPS, while the standard algorithm struggles to reach even half that performance, ranging from 500-2100 MFLOPS. This efficiency difference directly translates to the execution time differences we observed.

For large matrices (4096×4096 to 10240x10240), we implemented block-oriented multiplication with different block sizes (128 to 512). The impact of block size on performance revealed interesting patterns that vary with matrix dimensions. With 4096×4096 matrices, as shown in "block_comparison_4096.png" (Figure 11), block sizes 256 and 512 performed similarly in C++, with block size 128 trailing slightly behind. For 8192×8192 matrices, depicted in "block_comparison_8192.png" (Figure 2), block size 128 emerged as the clear winner in C++. For 10240×10240 matrices ("block_comparison_10240.png", Figure 10) block size 512 was the winner with a very slight advantage over the others..

These performance differences have a direct connection with cache behavior. "l2_cache_misses.png" (Figure 12) shows that Block 128 produces significantly more L2 cache misses than the alternatives, while Block 512 generates fewer misses than Block 256 at large matrix sizes. All block algorithms, however, substantially outperform the standard algorithm in terms of cache efficiency. When examining these results, it's important to consider that our processor has a 256KB L2 cache per core and a shared 12MB L3 cache.

The superior performance of larger block sizes reflects a careful balance between two factors: blocks must be small enough to fit comfortably in cache, but large enough to minimize the overhead of block management. For our specific hardware configuration, a block size of 512 balanced these factors more effectively at large matrix dimensions.

Memory Hierarchy Effects

Throughout our analysis, the impact of memory hierarchy emerged as perhaps the single most significant factor influencing matrix multiplication performance, particularly for large matrices.

Cache miss patterns reveal an interesting observation when viewed alongside matrix dimensions. In "l1_cache_misses.png" (Figure 1) and "l2_cache_misses.png" (Figure 12), we see that cache misses don't increase linearly with matrix size – rather exponentially. The standard algorithm shows an especially steep increase in L1 cache misses, while the line algorithm maintains relatively low miss rates even as matrices grow. This explains why the performance difference between these algorithms gets bigger with larger matrices.

The relationship between block size and cache performance reveals subtle but important trade-offs. Smaller blocks (128) theoretically fit better in L1 cache (64KB per core in our system), but they cause higher overall miss rates due to more frequent transitions between blocks. Larger blocks (512) exceed typical L1 cache sizes but reduce the total number of block transitions, resulting in fewer overall cache misses. This suggests they make better use of our processor's 256KB L2 cache and 12MB shared L3 cache. "l2_cache_misses.png" (Figure 12) confirms this pattern, showing Block 128 with higher L2 cache misses compared to Block 512, especially at larger matrix sizes.

While algorithms with fewer cache misses generally perform better, the correlation isn't perfectly linear. In "l2_cache_misses.png" (Figure 12) and "mflops_comparison.png" (Figure 13), we see the line algorithm with the fewest cache misses and highest MFLOPS, and Block 512 with fewer cache misses and better performance than Block 128. However, the performance difference between Block 256 and Block 512 is less pronounced than their cache miss difference would suggest.

Multi-Core Performance Analysis

Parallel Scaling

Our parallel implementations of the line algorithm revealed insights into how matrix multiplication scales across multiple cores of our Intel i7-9700 CPU. The results weren't as straightforward as simply adding more processors and getting proportional speedup.

"speedup_comparison.png" (Figure 9) shows that external parallelization (parallelizing the outermost loop) achieved speedups between 3.9x and 5.9x, while internal parallelization (parallelizing the innermost loop) managed only modest improvements, ranging from 0.5x to 1.3x. For external parallelization, we observed that speedup generally decreased as matrix size increased, dropping from approximately 5.8x with 2000×2000 matrices to about 4.1x with 10240×10240 matrices.

This scaling behavior reveals important characteristics of parallel matrix multiplication. With smaller matrices, external parallelization achieves near-linear speedup, approaching the

theoretical maximum based on the 8 cores of our processor. However, as matrices grow larger, parallel efficiency declines steadily, dropping from approximately 0.7 to 0.5 for external parallelization, as shown in "parallel_efficiency.png" (Figure 3). By the time we reach very large matrices (8192+), speedup stabilizes around 4x, suggesting that memory bandwidth rather than computational capacity becomes the limiting factor.

External vs. Internal Parallelism

The performance difference between our two parallelization approaches was very strong. External parallelization (making each thread handle different rows of the result matrix) drastically outperformed internal parallelization (having threads collaborate on calculating individual elements).

External parallelization achieved 3-6x speedup over sequential execution, while internal parallelization barely improved performance, achieving only 0.5-1.3x speedup. The MFLOPS comparison in "parallel_mflops.png" (Figure 4) makes this difference even clearer, with external parallelization reaching 13,000-26,000 MFLOPS compared to just 2,000-4,500 MFLOPS for internal parallelization. This can be explained by the fact that internal parallelization requires synchronization after each iteration of the middle loop, creating substantial thread management overhead that often outweighs the benefits of parallelism.

External parallelization, by contrast, allows each thread to work on completely independent portions of the result matrix without interruption, maximizing thread utilization, minimizing context switching as well as the competition of threads for access to the same matrix regions.

The efficiency data in "parallel_efficiency.png" (Figure 3) quantifies this difference, showing external parallelization reaching up to 0.7 efficiency while internal parallelization barely exceeds 0.15 efficiency.

Performance Bottlenecks

As matrix sizes increased, we observed clear signs of system bottlenecks limiting parallel performance. Beyond 6144×6144, speedup for external parallelization plateaued around 4x despite having 8 cores available.

We see evidence of this bottleneck in several places: declining MFLOPS for external parallelization with larger matrices ("parallel_mflops.png", Figure 4), stabilizing speedup despite increasing computational requirements ("speedup_comparison.png", Figure 9), and significant increases in cache misses with matrix size ("parallel_l2_misses.png", Figure 7 and "parallel_l1_misses.png", Figure 8).

Cache behavior significantly influences parallel efficiency as well. With smaller matrices, data fits better in combined CPU caches (our system has a total of 512KB L1 cache and 2MB L2 cache across all cores, plus 12MB shared L3), allowing near-linear speedup. As matrix dimensions grow, cache misses increase dramatically, forcing threads to wait for memory access and reducing parallel efficiency. L2 cache misses grow from almost zero at small matrix sizes to hundreds of millions at large sizes.

For internal parallelization, the low efficiency compared to external parallelization (0.15 vs. 0.7) highlights the severe impact of synchronization overhead. Each synchronization point requires thread coordination, cache coherence and potential context switching. These overheads accumulate significantly in the internal approach due to frequent synchronization after each middle loop iteration.

5. Conclusions

Our analysis points to a clear conclusion: optimal performance for matrix multiplication depends on memory access patterns that maximize cache utilization (line algorithm), parallelization strategies that minimize synchronization (external parallelization), and block sizes that balance cache utilization with computational efficiency (larger blocks for very large matrices).

Different algorithms are optimal in different scenarios. The standard algorithm, while simple, consistently underperforms due to its inefficient memory access patterns. The line algorithm emerges as the clear winner for most single-threaded applications, delivering 2-4x faster execution across all matrix sizes by better aligning with the computer's memory organization. For very large matrices, block algorithms prove most effective, with performance improving as block size increases up to 512 on the processor we used.

Regarding parallelism, external parallelization of the line algorithm delivers the best overall performance, with speedups of 4-6x over sequential execution.

Matrix multiplication remains a fundamental operation in scientific computing, and our findings demonstrate that achieving peak performance requires an understanding of algorithm structure, memory access patterns, language implementation details, and hardware characteristics.

6. Annexes

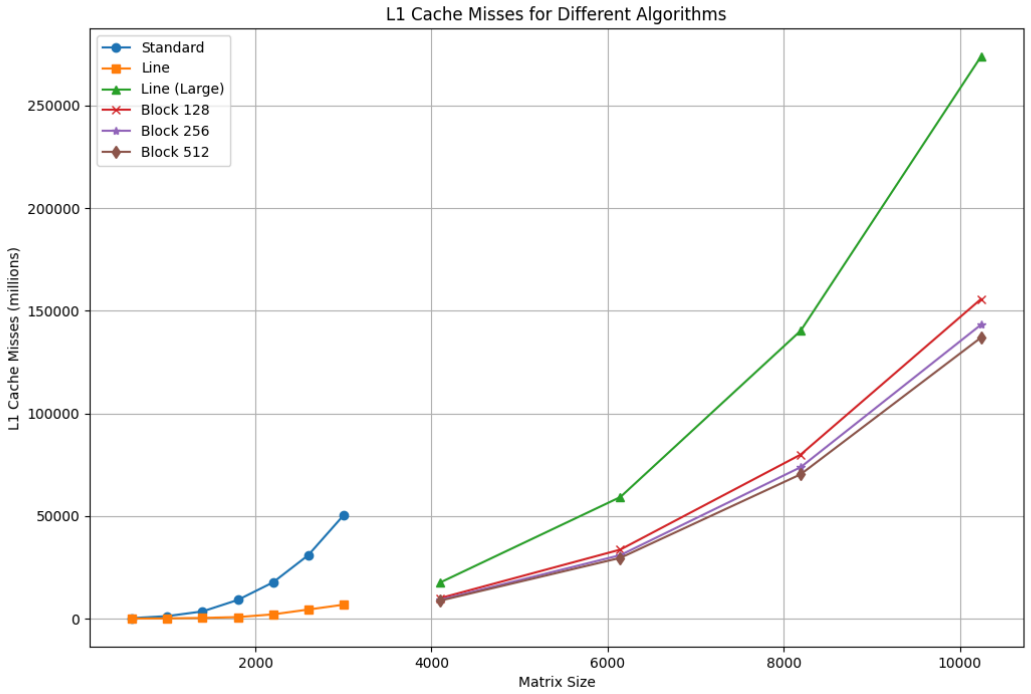

Figure 1 - L1 cache misses

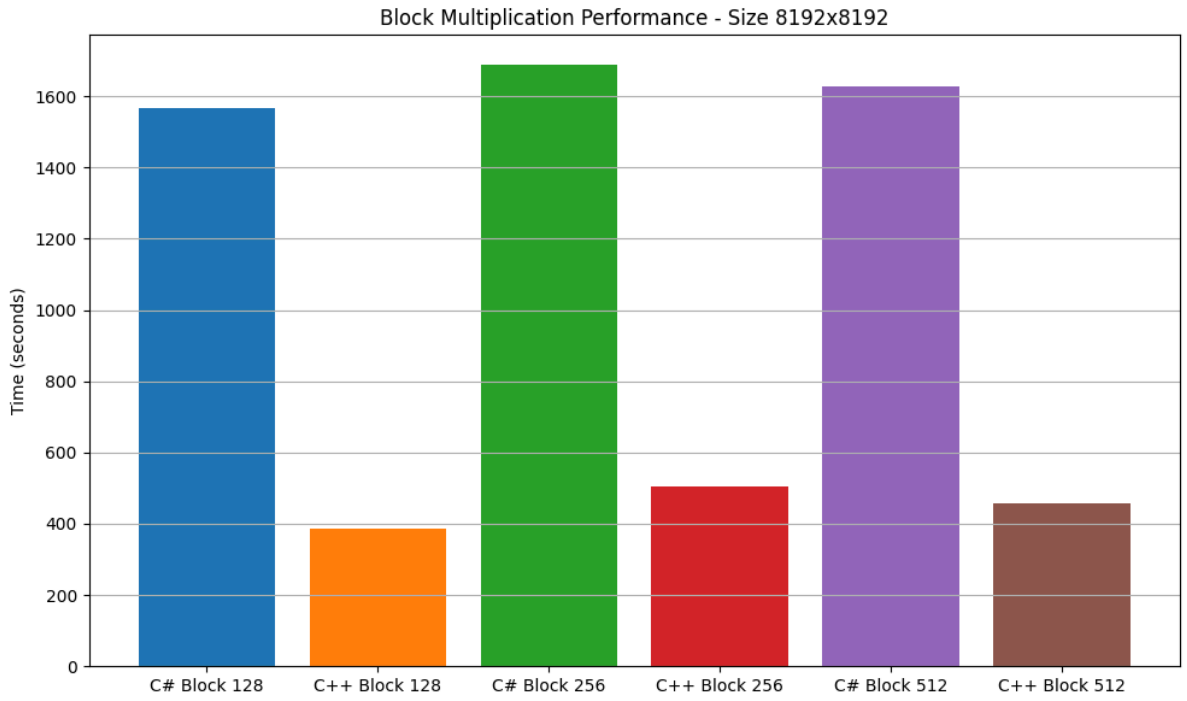

Figure 2 - Block comparison 8192

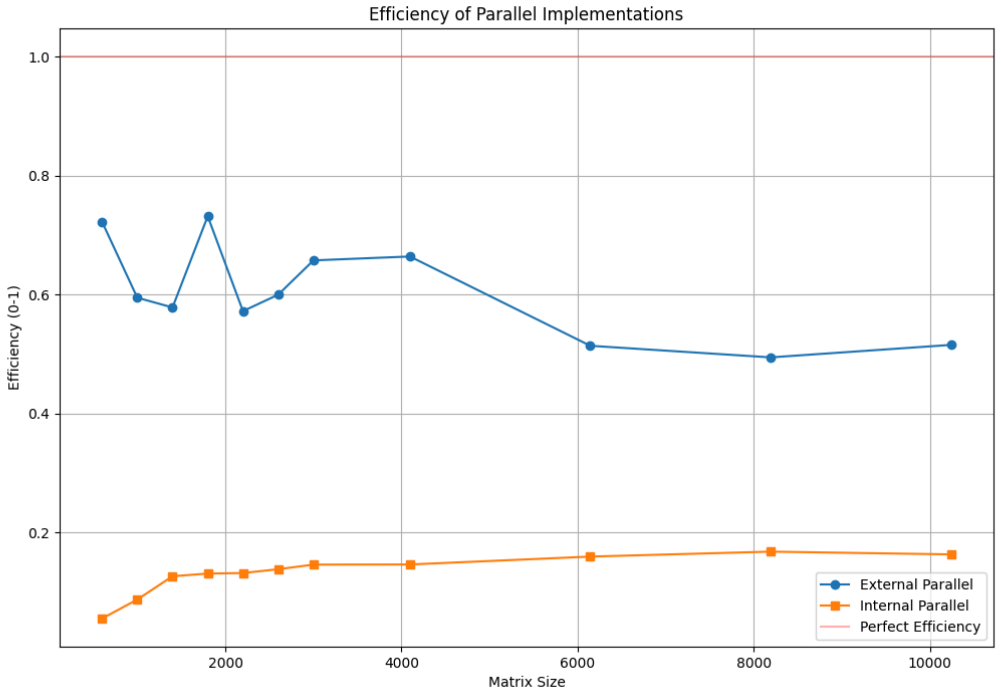

Figure 3 - Parallel efficiency

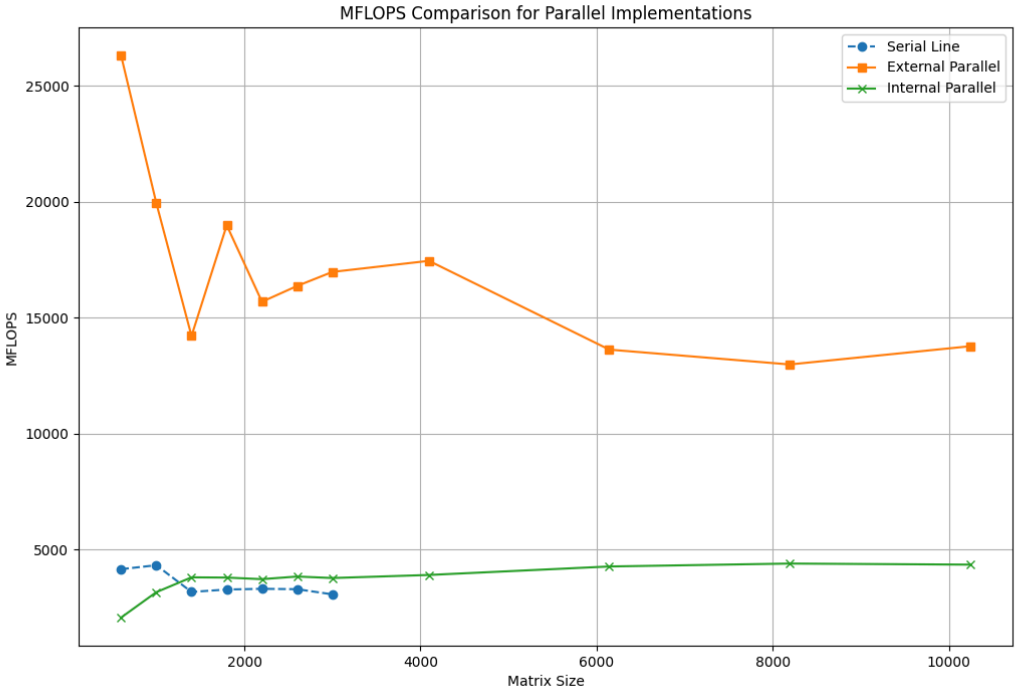

Figure 4 - Parallel MFLOPS

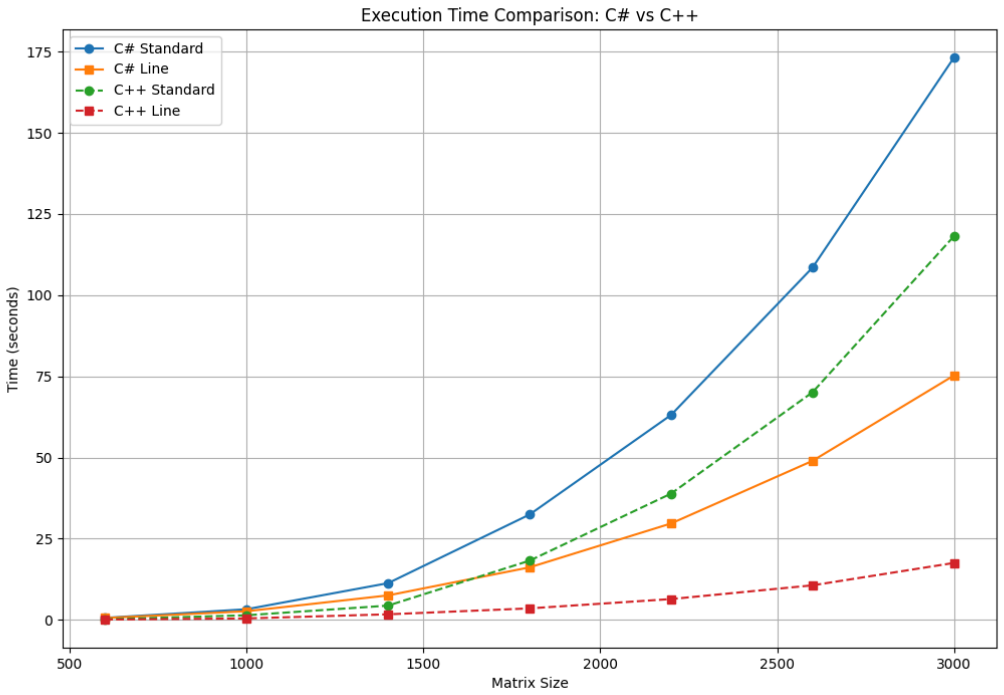

Figure 5 - Execution time comparison

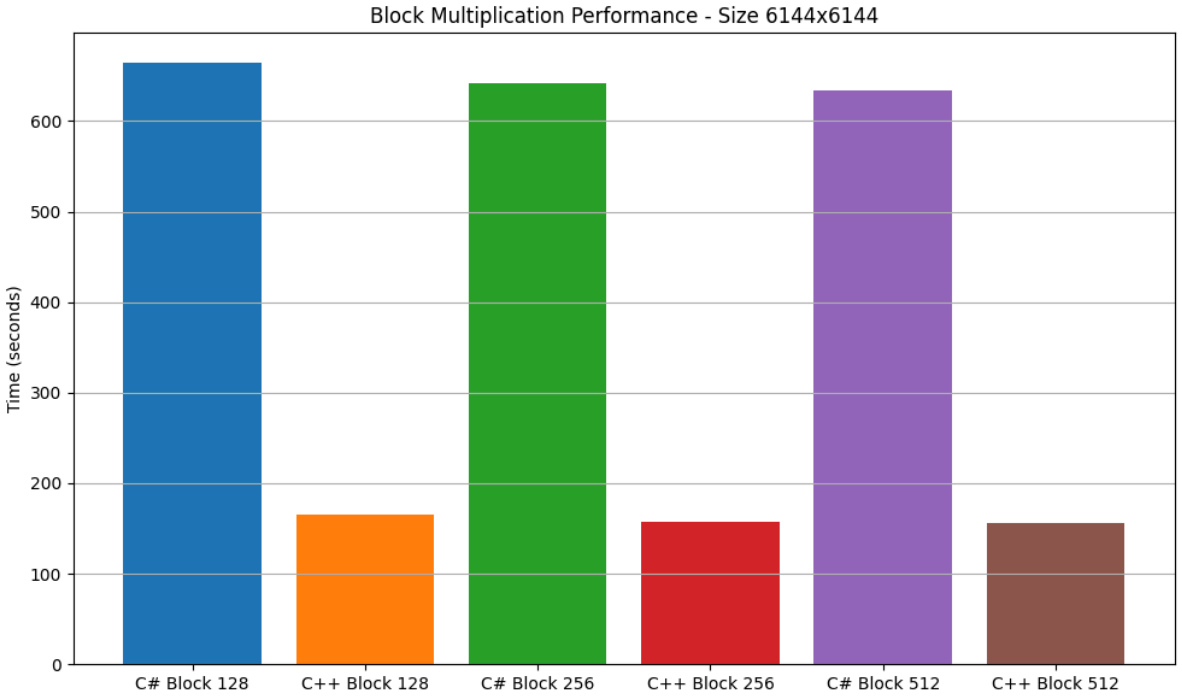

Figure 6 - Block comparison 6144

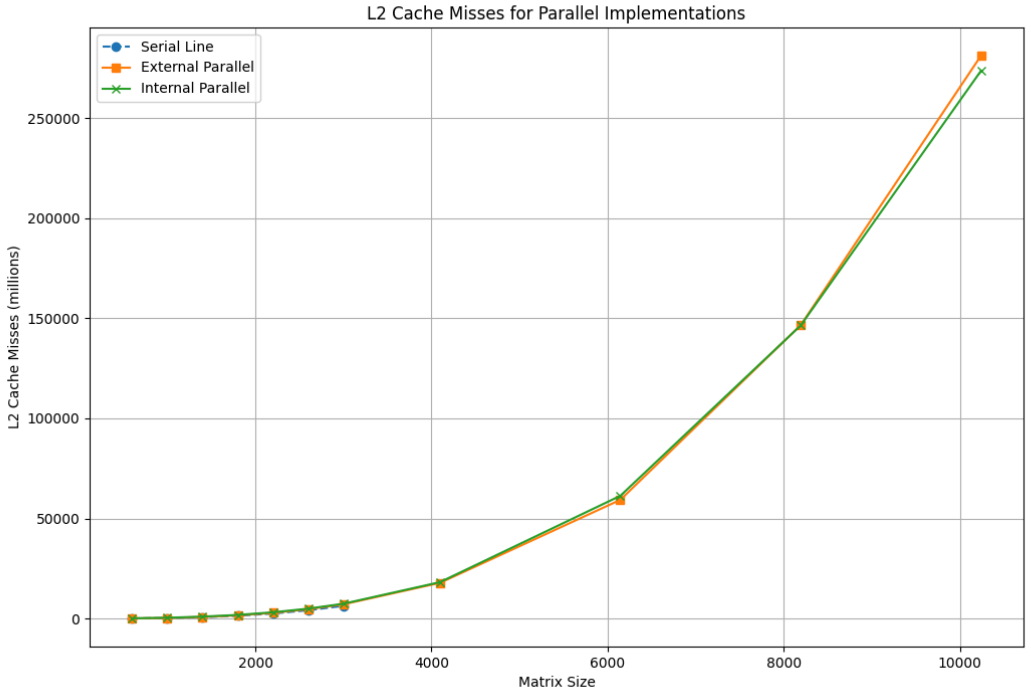

Figure 7 - Parallel L2 cache misses

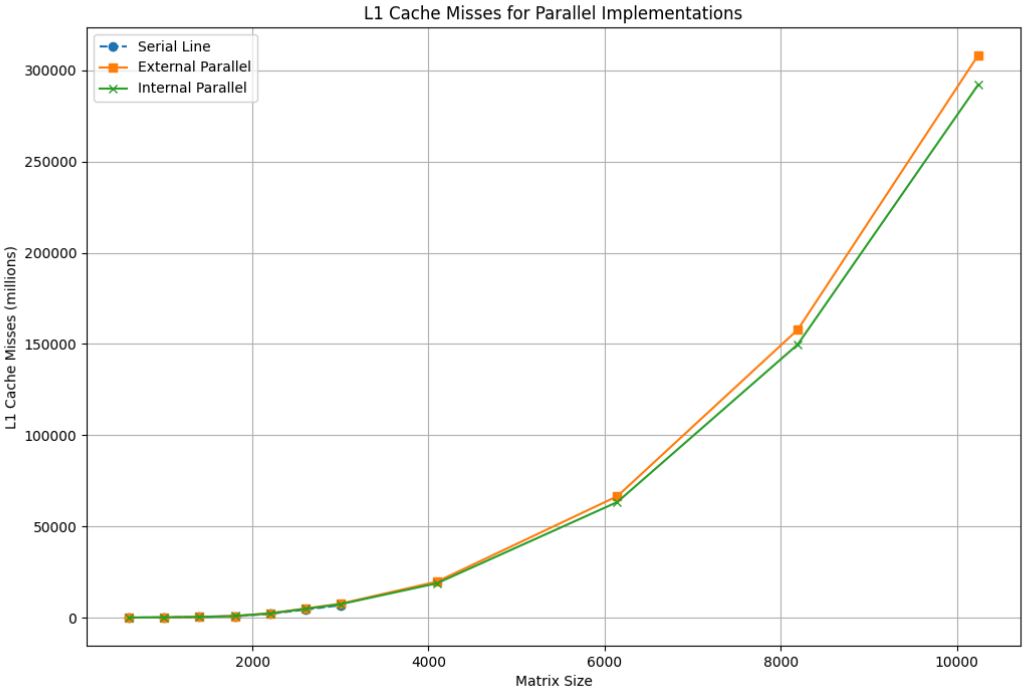

Figure 8 - Parallel L1 cache misses

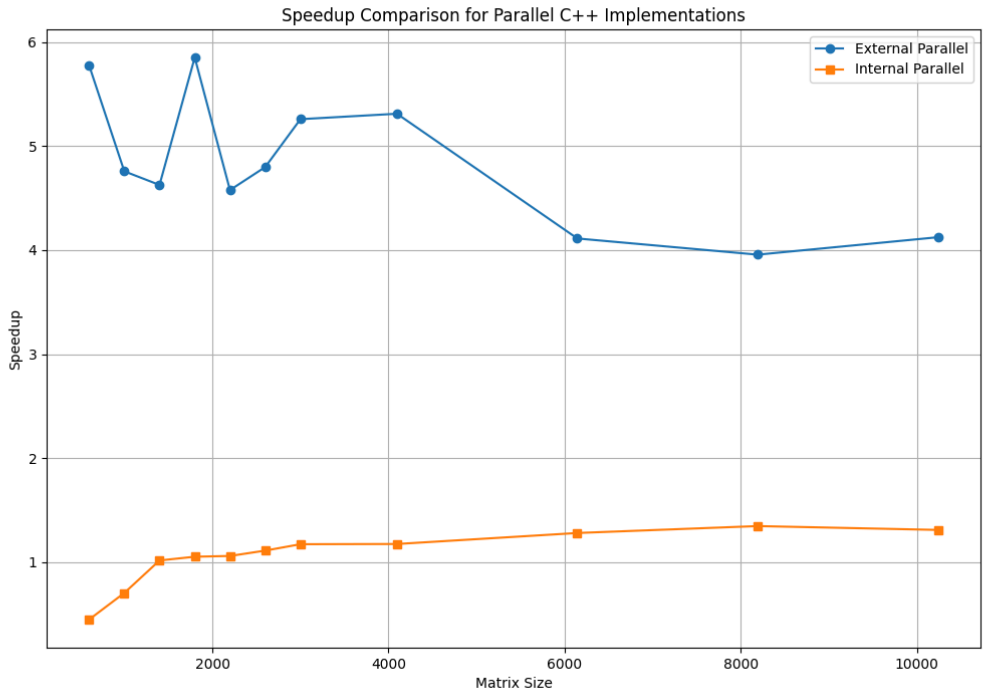

Figure 9 - Speedup comparison

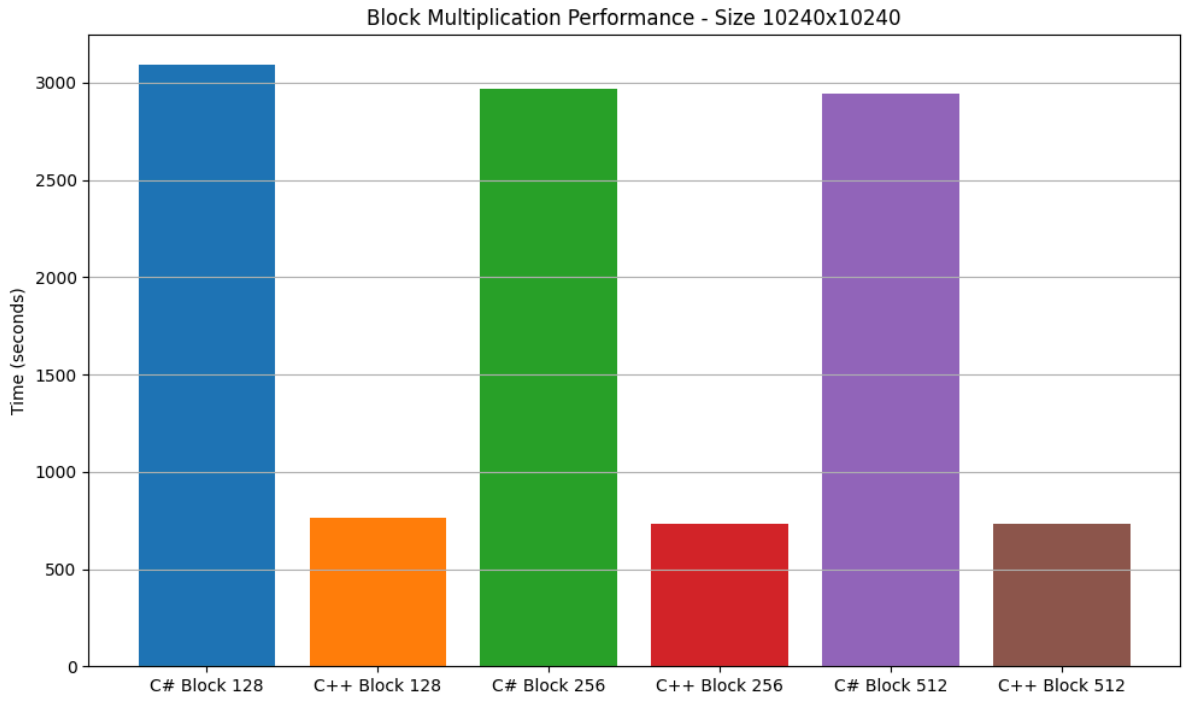

Figure 10 - Block comparison 10240

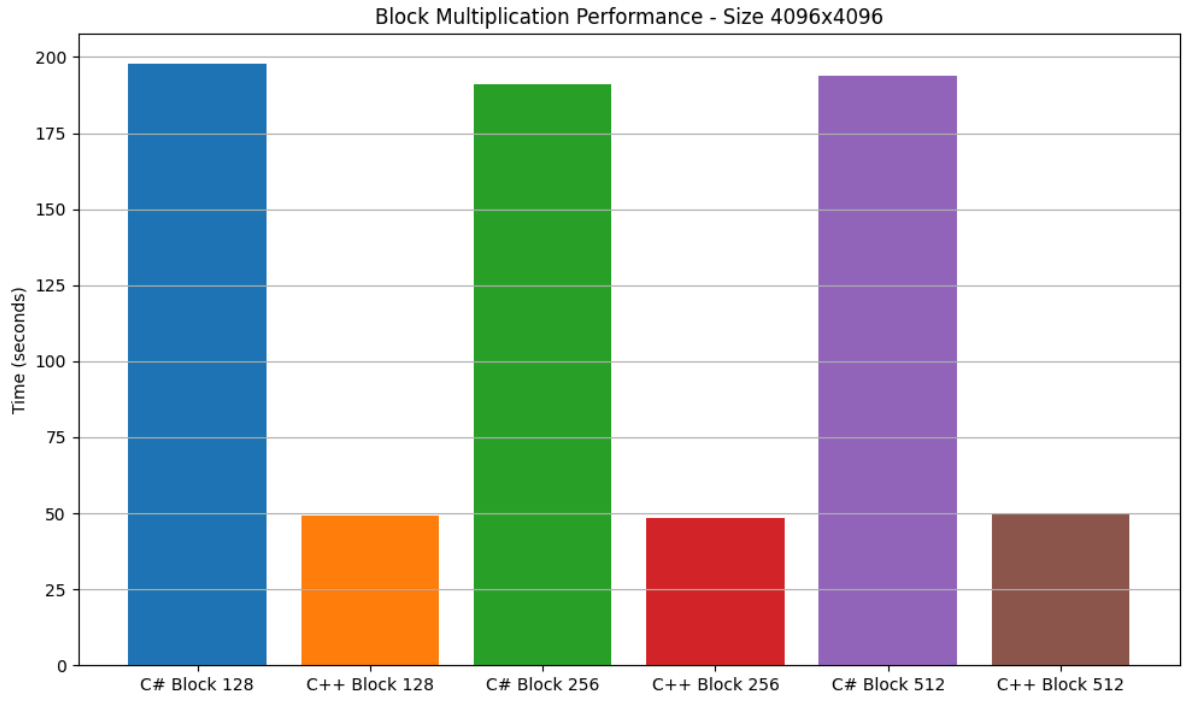

Figure 11 - Block comparison 4096

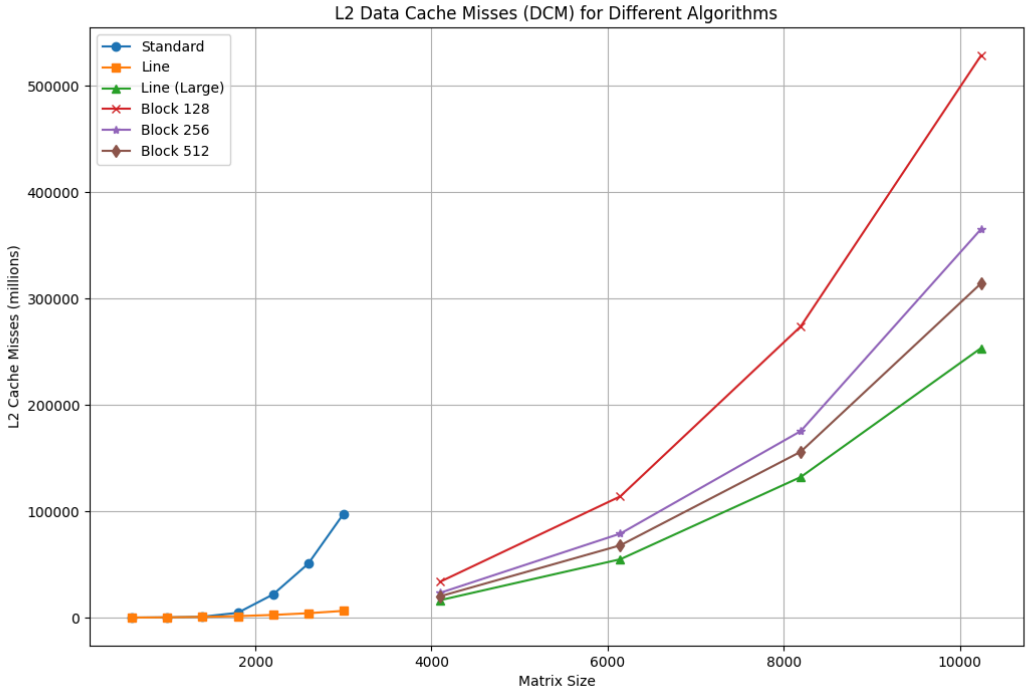

Figure 12 - L2 cache misses

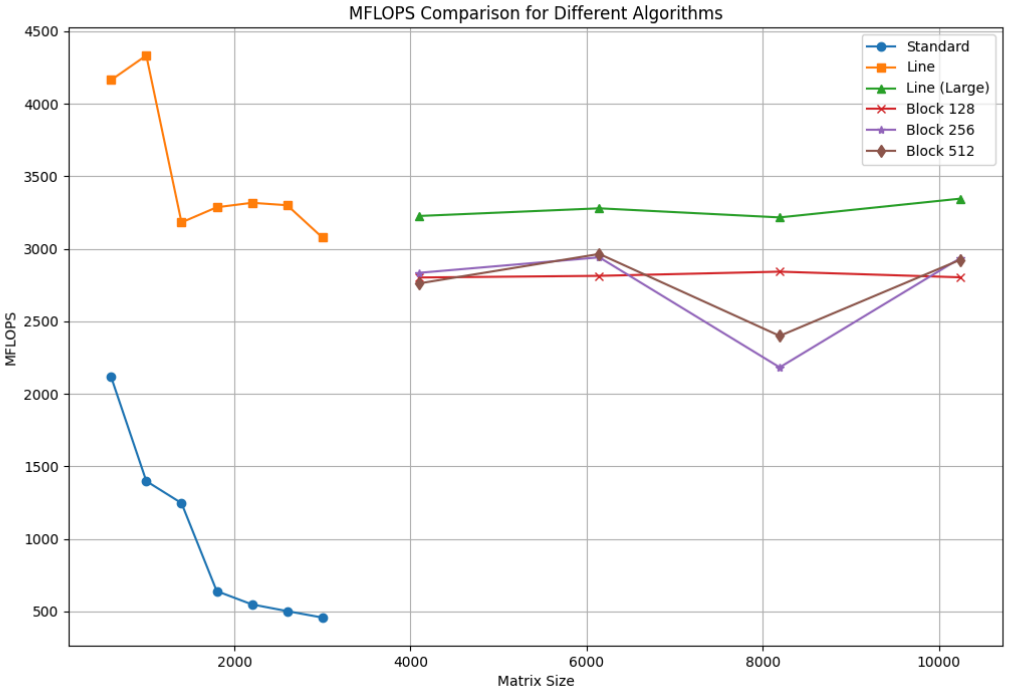

Figure 13 - MFLOPS comparison

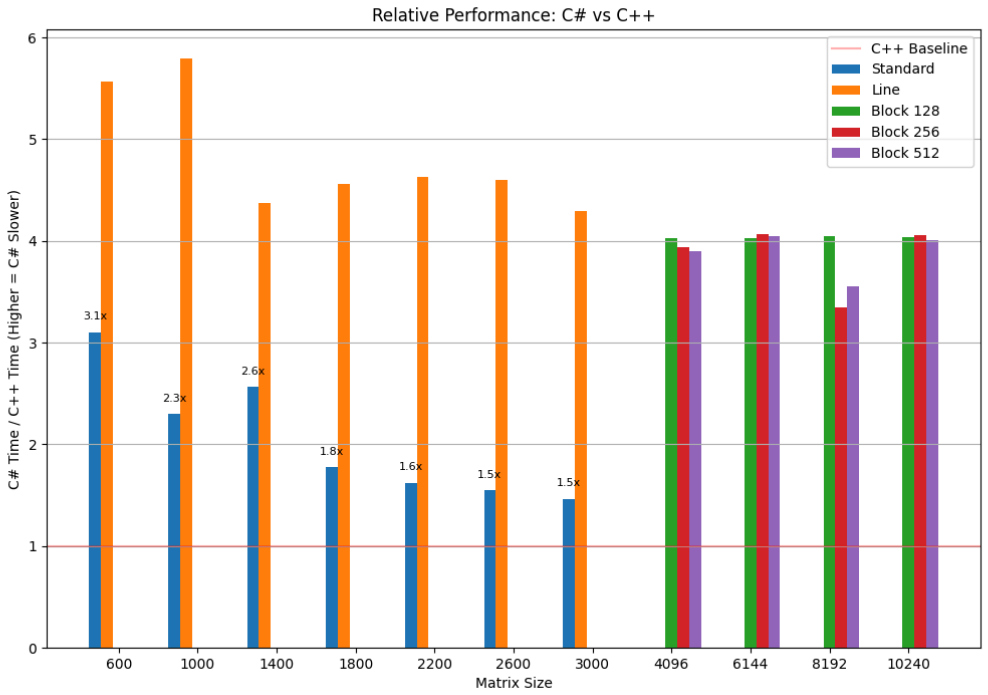

Figure 14 - Relative performance