FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

COMPUTAÇÃO PARALELA E DISTRIBUÍDA

CPD Project 1 - Performance evaluation

António Henrique Martinz Azevedo (202108689)
José António Pereira Martins (202108794)
Tomás Eiras Silva Martins (202108776)

March 15, 2024

Contents

1 Problem Description

This project goal is to analyse the impact of memory hierarchy on processor performance during matrix multiplication. We aim to understand how data access patterns influence computational efficiency through various implementations and performance measurements across different input sizes.

2 Algorithms

In order to understand the influence of different algorithms on processor performance during matrix multiplication, we implemented three algorithms that vary primarily in their memory access patterns. To analyze the impact of the programming languages on the performance, we implemented the algorithms in **C++** and **Rust**. This selection allows us to compare two widely used programming languages with distinct memory management approaches. Both languages offer high levels of control over memory management, making them suitable for investigating how memory access patterns affect performance.

- **Basic matrix multiplication algorithm** - implemented in C++ and Rust
- **Line matrix multiplication algorithm** - implemented in C++ and Rust
- **Block matrix multiplication algorithm** - implemented in C++

2.1 Basic matrix multiplication algorithm

This algorithm implements a basic matrix multiplication approach, where each element in the resulting matrix is computed by multiplying one row of the first matrix by one column of the second. As a result, the algorithm demonstrates a computational time complexity of $O(n^3)$ and a space complexity of $O(n^2)$, owing to its composition of three nested loops. Here, n represents the size of the input matrices.

The resulting matrix can be calculated using the following equation:

$$C[i][j] = \sum_{k=0}^{n-1} A[i][k] \cdot B[k][j] \tag{1}$$

where C is the resulting matrix, A and B are the input matrices, and n is the size of the matrices.

2.2 Line matrix multiplication algorithm

Instead of directly multiplying a row of the first matrix with a column of the second, this algorithm computes the product of a row of the first matrix with each row of the second. Despite maintaining the same time and space complexity as the algorithm before, this algorithm's efficient cache usage often reduce cache misses and improves execution time, particularly for large matrices.

2.3 Block matrix multiplication algorithm

The Block Matrix Multiplication algorithm is designed to further optimize memory access patterns by dividing the matrices into smaller blocks. This strategy enhances cache utilization and reduces cache misses compared to the Line Matrix Multiplication approach. The algorithm operates by multiplying submatrices within these blocks, thereby minimizing the number of accesses to slower memory and exploiting spatial locality. Although sharing identical space and time complexities with the preceding algorithms, block matrix multiplication excels in performance, particularly with large matrices. This superiority derives from its capability to partition matrices into smaller blocks, effectively decreasing the incidence of cache misses.

3 Performance Metrics

3.1 Metrics

To assess the performance of the C++ implementations, we utilized the PAPI (Performance API), providing us with comprehensive hardware performance monitoring. Our evaluation comprised the following key metrics:

- **Execution Time** - This metric measures the time it takes for the program to execute matrix multiplication algorithm. It serves as a fundamental benchmark for computing overall performance and facilitating comparative analysis.
- **L1 and L2 Cache Misses** - These metrics quantify the number of cache misses occurring in the L1 and L2 caches, respectively. A cache miss arises when the requested data is not found in the cache and must be retrieved from the next level of memory. This metric provides valuable insights into the memory access patterns of the program, as frequent cache misses can lead to increased execution times, which means that minimizing cache misses is crucial for optimizing program performance!
- **GFlops** - This metric, derived from "**G**iga **Fl**oating-point **O**perations **p**er **S**econd", provides a measure of a computer's performance in executing floating-point arithmetic operations. It quantifies the rate at which a system can perform such operations and is calculated using the following formula:

$$GFlops = \frac{Number\ of\ Floating\ Point\ Operations}{Execution\ Time\ in\ Seconds} \times 10^{-9} \tag{2}$$

- **Speedup** - This metric quantifies the amount of improvement gained by using a particular parallelization technique compared to a baseline implementation. It is calculated as the ratio of the execution time of the baseline implementation to the execution time of the optimized or parallelized version. The formula for speedup is as follows:

$$Speedup = \frac{Execution\ Time\ of\ Baseline\ Implementation}{Execution\ Time\ of\ Optimized\ Implementation} \tag{3}$$

- **Efficiency** - Efficiency measures how effectively a parallelization technique utilizes the available computational resources, such as CPU cores. It is calculated as the ratio of the speedup to the number of cores used. The formula for efficiency is as follows:

$$Efficiency = \frac{Speedup}{Number\ of\ Cores} \tag{4}$$

3.2 Hardware used

- **Processor** - Intel Core i7-10700 CPU @ 2.90GHz 8-core (16 threads)
- **L1 Cache** - 512 KB
- **L2 Cache** - 2 MB

4 Results and Analysis

We ran the algorithms three times and recorded the values of the chosen metrics. This section analyzes and compares the performance of the algorithms based on the average values from these three runs.

4.1 Basic multiplication and Line multiplication comparison

Both **C++** and **Rust** are performance-oriented programming languages that provide developers with direct control over memory management and system interactions, facilitating the creation of highly optimized code. This is apparent when comparing the execution times of basic and line multiplication algorithms, where both C++ and Rust implementations demonstrate **similar performance**, as depicted in the *Graph1*.

Comparing the execution times of both algorithms, it's clear that the line multiplication algorithm **surpasses** the basic multiplication algorithm in terms of speed - *Graph1*. This can also be evident in the *Graph2*, where the line multiplication algorithm demonstrates **fewer L1 and L2 cache misses** compared to its basic counterpart, resulting in a reduced execution time. The line multiplication algorithm's cache efficiency excels from its **sequential and contiguous** memory access. This pattern optimally utilizes cache by accessing adjacent data, aligning well with cache loading mechanisms. In contrast, the basic algorithm's non-sequential access leads to **increased cache misses** and consequently lower FLOPS, impeding performance. This can be observed in the *Graph3* as the line multiplication algorithm consistently maintains higher FLOPS across varying matrix dimensions, indicative of its **efficient access to cache**. While there may be a slight decline in FLOPS as matrix dimensions increase, the line multiplication algorithm exhibits greater stability in its performance compared to the basic algorithm as predicted.

4.2 Block multiplication and Line multiplication comparison

During the testing of the block multiplication algorithm, it became evident that this approach **outpaces** both the line multiplication algorithm and the basic multiplication algorithm, as anticipated. This superiority is clearly seen in the *Graph5*, where the block multiplication method showcases a substantial **decrease in execution time** compared to the line multiplication algorithm. Furthermore, as shown by the *Graph4*, while there may be slight fluctuations in execution time, it's notable that different block sizes exhibit similar performance, although using blocks of size 512 appears to give optimal results.

As previously discussed, faster execution times often correlate with fewer cache misses and higher FLOPS, implying that the block multiplication algorithm would **likely outperform** the line multiplication algorithm in these metrics. This expectation is partially supported by the data in the *Graph6*, where the block multiplication algorithm exhibits **lower L1 cache misses** compared to the line multiplication algorithm. However, upon examining the *Graph7*, it becomes apparent that although the difference is not that much, the line multiplication algorithm demonstrates **better performance in terms of L2 cache misses**. This observation contrasts with the initial hypothesis. To understand this apparent contradiction, it's crucial to recognize the **significant speed disparity** between **L1 and L2 caches**. The block multiplication algorithm's strategy of breaking matrices into smaller blocks enhances spatial locality, reducing L1 cache misses by facilitating **more efficient data access and reuse**. Consequently, data required for calculations is **often readily available in the faster L1 cache**, minimizing the time required to fetch data from slower memory. However, due to the algorithm's utilization of larger amounts of data overall, it may exceed the capacity of the L2 cache, resulting in more misses at this level. Despite these increased L2 cache misses, the block multiplication algorithm remains faster overall due to its efficient use of the L1 cache.

When comparing the FLOPS of the block multiplication algorithm to the line multiplication algorithm, we anticipated **superior performance from the block algorithm**. Additionally, we expected that varying block sizes would demonstrate consistency in FLOPS across different matrices sizes, with a block size of 512 being the most effective, followed by 256 and 128. These expectations were **largely verified** by the data presented in the *Graph8*, except for a **notable anomaly**: when using block sizes of 256 and 512 on a matrix with dimensions of 8192, there was a **significant decline in FLOPS**, contrasting with the consistent performance observed across other matrix dimensions. Despite in-depth analysis, we couldn't identify a noticeable pattern in the data to explain this phenomenon, particularly given that even on matrices with higher dimensions, the FLOPS remained consistent.

4.3 Line multiplication algorithm - Parallel versions

This section explores the performance gains achieved through two distinct approaches for parallelizing a nested loop structure using OpenMP's **#pragma omp parallel for** instruction.

Listing 1: Parallel 1

```
#pragma omp parallel for
for (int i=0; i<n; i++) {
    for (int k=0; k<n; k++) {
        for (int j=0; j<n; j++) {
            // computations here
        }
    }
}
```

- Distributes iterations of the **outer loop (i)** among multiple threads for concurrent execution
- **The inner loops (k and j)** are executed sequentially within each thread

Listing 2: Parallel 2

```
#pragma omp parallel
for (int i=0; i<n; i++) {
    for (int k=0; k<n; k++) {
        #pragma omp for
        for (int j=0; j<n; j++) {
            // computations here
        }
    }
}
```

- Distributes iterations of the **innermost loop (j)** among multiple threads for concurrent execution
- **The outer loops (i and k)** are executed sequentially within each thread

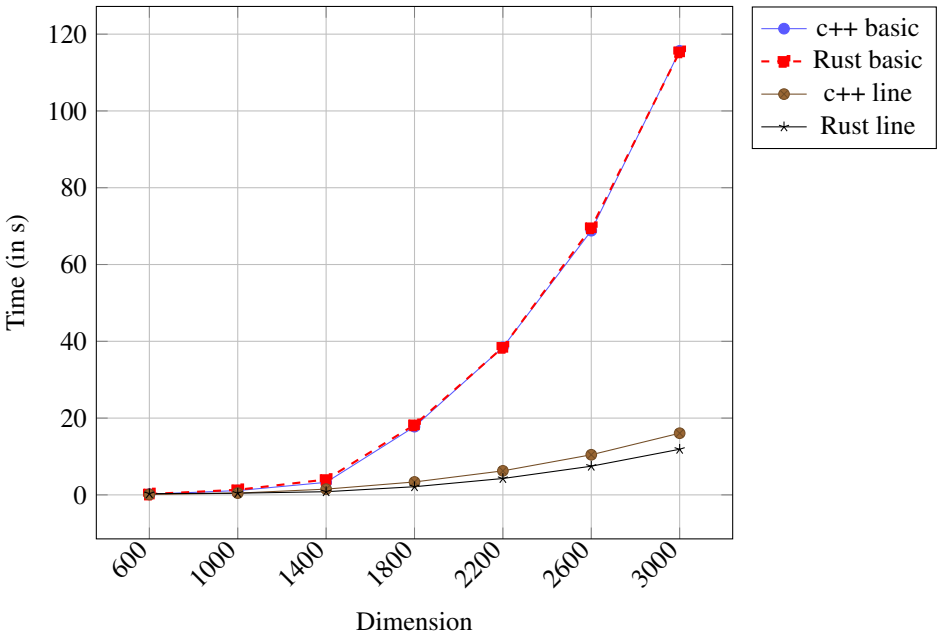
While both parallelization strategies managed to surpass the performance of the sequential version, it was evident that the first strategy outperformed the second by approximately 4.75 times, and exceeded the sequential version by a factor of 5.6, as shown in the *Graph9*. Despite this significant gap, the second strategy still demonstrated a slight improvement over the sequential version, achieving a speedup of 1.18.

As highlighted in the *metrics section*, the efficiency of parallelization strategies can be quantified using the *efficiency equation*. As expected, the strategy that achieved the highest speedup also exhibited the highest efficiency - *Graph11*. This efficiency metric serves as a validation of the achieved FLOPS by the parallelized versions, with the most efficient strategy naturally capable of delivering higher FLOPS, as illustrated in the *Graph10*.

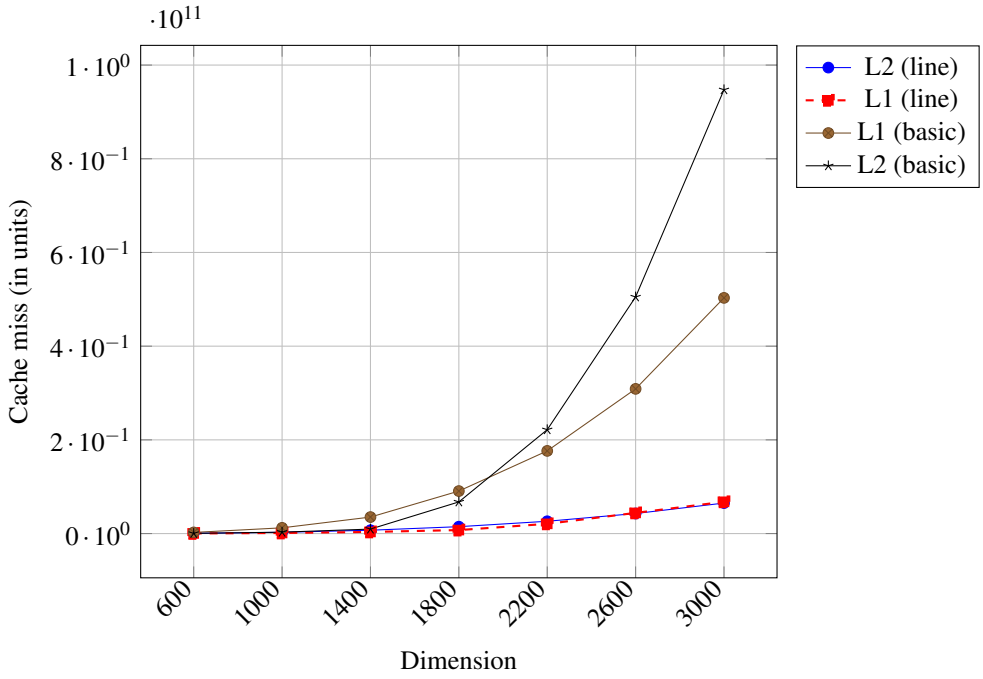
5 Conclusion

6 Annexes

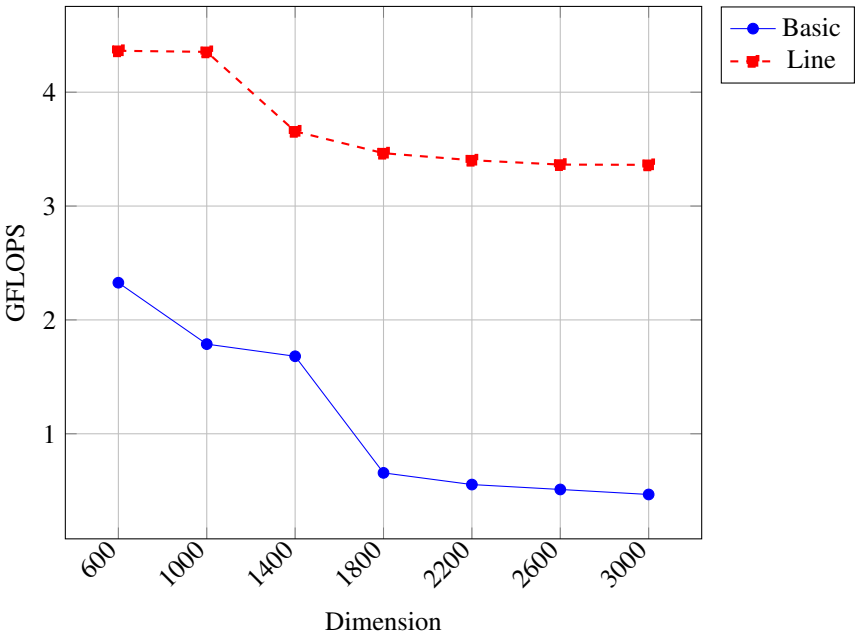
Graph 1: Execution Time: basic vs line (Rust and C++)



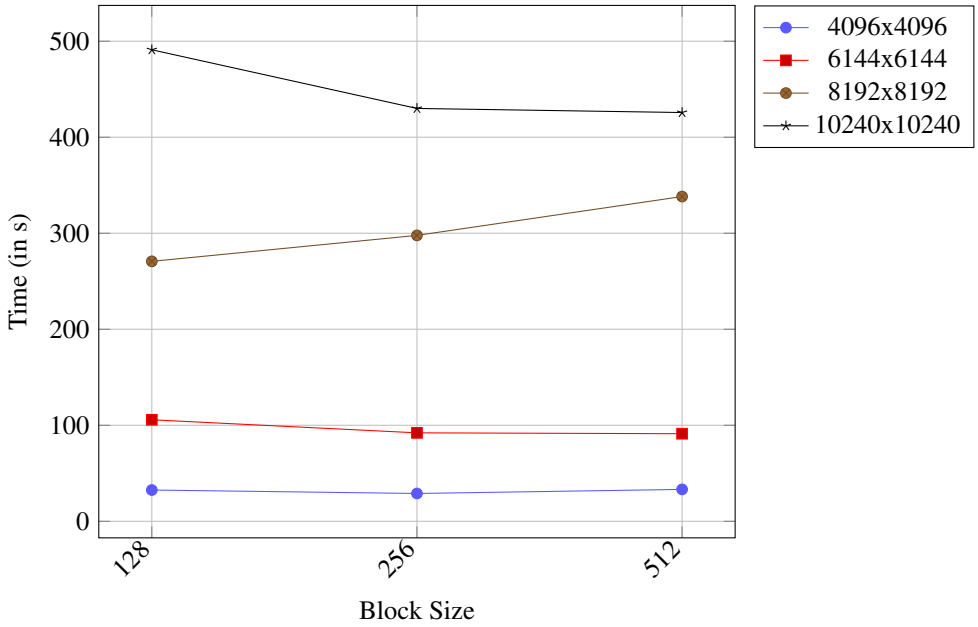
Graph 2: Cache Misses: line vs basic (C++)



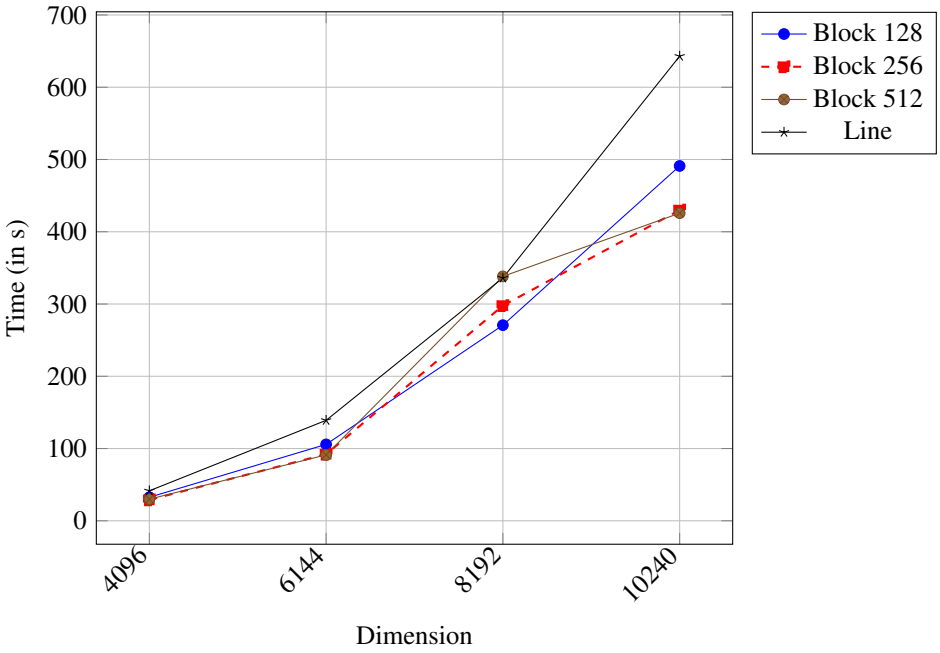
Graph 3: GFLOPS Comparison: Line vs Basic (c++)



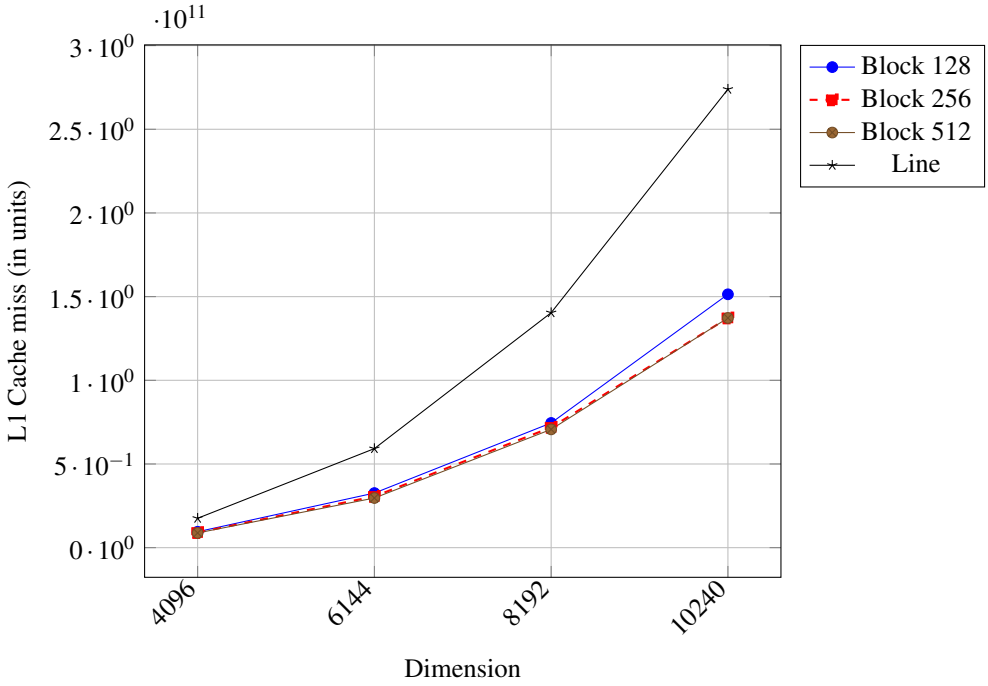
Graph 4: Execution Time: block size relationship (C++)



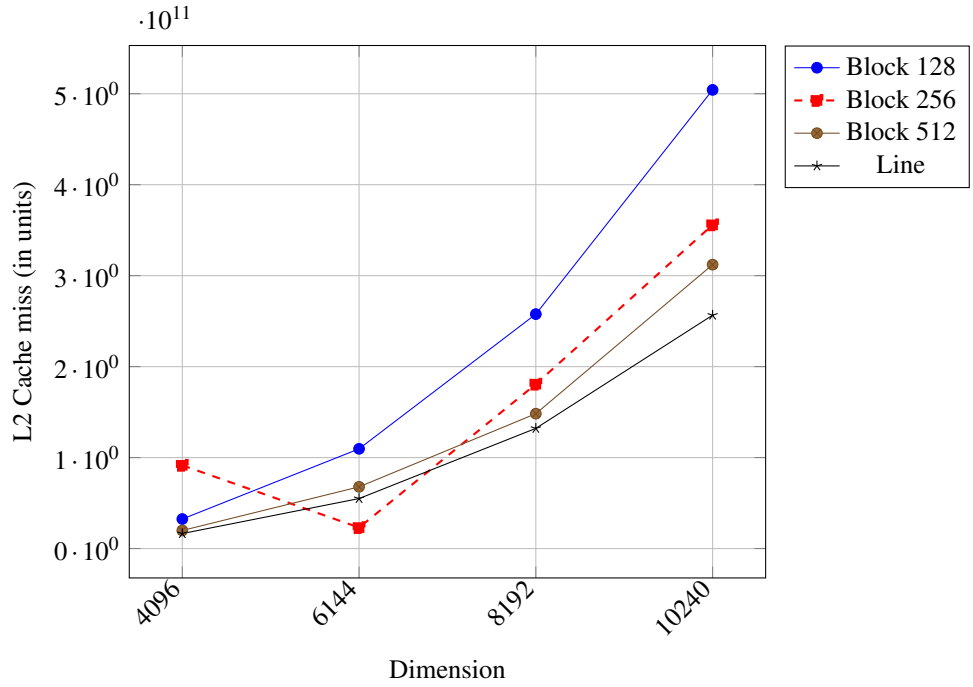
Graph 5: Execution time: Block vs Line (C++)



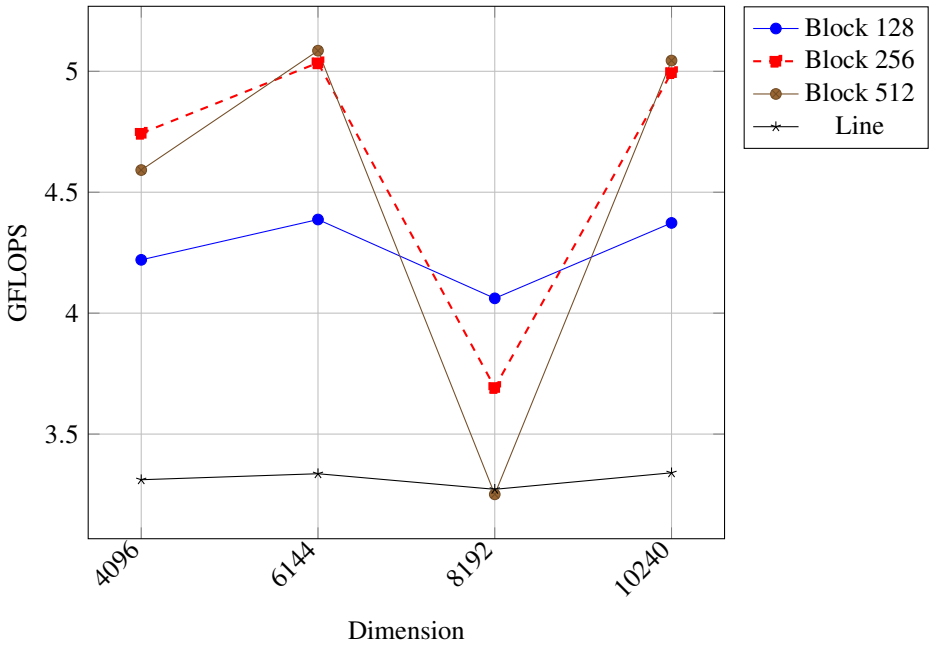
Graph 6: L1 Cache Misses: block vs line (C++)



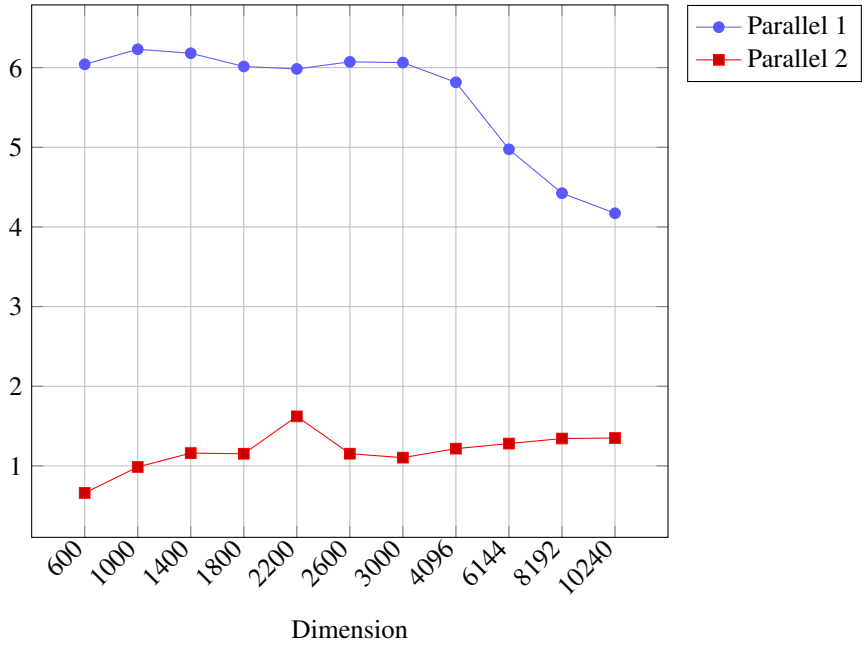
Graph 7: L2 Cache Misses: block vs line (C++)



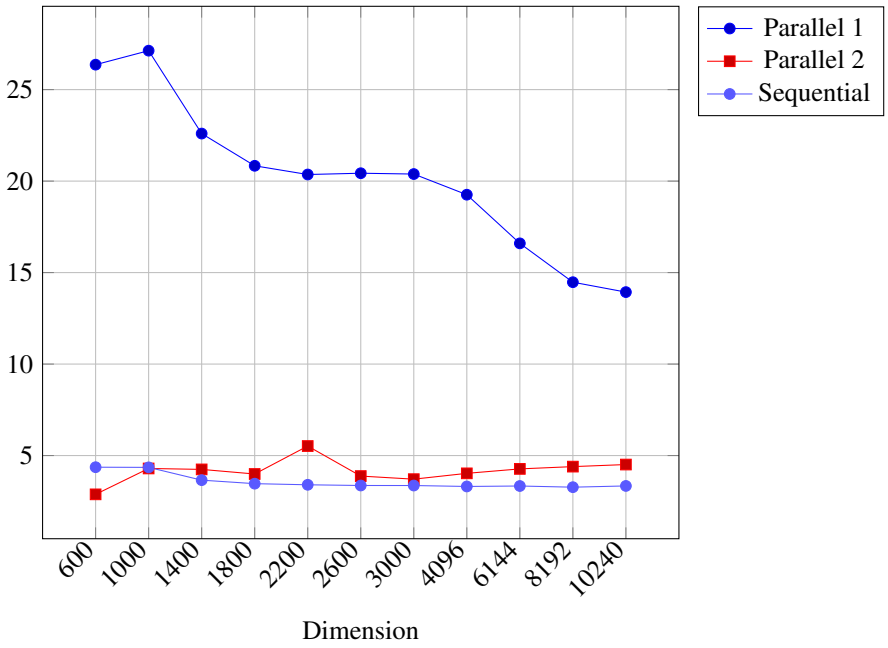
Graph 8: GFLOPS Comparison: Line vs Block (c++)



Graph 9: Speedup: line - Parallel 1 vs Parallel 2 (C++)



Graph 10: GFLOPS Comparison: line - Parallelelism vs sequential (C++)



Graph 11: Efficiency: line - Parallel 1 vs Parallel 2 (C++)