U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Performance evaluation of a single core

Parallel and Distributed Computing - Project 1
March, 2024

Fábio Rocha up202005478

José Guedes up202007651

José Isidro up202006485

Contents Table

1. Problem description and algorithms explanation

In this project, we embark on a comprehensive exploration into the relationship between memory hierarchy and processor performance, particularly in the context of accessing large amounts of data. Our focal point lies in scrutinizing the impact of memory management and algorithm optimizations on enhancing the efficiency of matrix multiplication tasks, using both C++ and Java programming languages.The performance indicators were gathered using a Performance API (PAPI).

Our study offers valuable insights into enhancing matrix multiplication performance through memory management and algorithm optimization, applicable to both C++ and Java programming languages. It focused on analyzing the effectiveness of memory management, which includes cache optimization, and algorithm optimizations in improving matrix multiplication performance.

1.1. The Matrix Multiplication Algorithm

This algorithm performs matrix multiplication for square matrices of size ‘matrixSize. The algorithm is composed of three nested for loops. For the multiplication of matrix A with matrix B, resulting in matrix C (C = A × B):

- The first *for loop* represents the line for A and C
- The second *for loop* represents the column for B and C
- The third *for loop* represents the line for B and the column for A

Its time and space complexities are O(n^3) and O(n^2), respectively, where n is the dimension of the matrices involved.

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

Figure 1: Regular Multiplication Algorithm

1.2. The Matrix Line Multiplication Algorithm

This algorithm is also performing matrix multiplication for square matrices of size ‘matrixSize’, however, it changes the order of the loops compared to the previous algorithm. The order of nested loops affects the way elements of the matrices are accessed during multiplication.

For the multiplication of matrix A with matrix B, resulting in matrix C (C = A × B):

- The first *for loop* represents the line for A and C

- The second *for loop* represents the line for B and the column for A
- The ethird *for loop* represents the column for B and C

This can have implications for cache performance and overall efficiency, but in terms of big O complexity, both algorithms have the same time and space complexities, are O(n^3) and O(n^2), respectively.

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

Figure 2: Line Multiplication Algorithm

1.3. The Matrix Block Multiplication Algorithm

This algorithm is a variation of the previous ones and it applies a technique called "block-based matrix multiplication". It's an optimization aimed at improving cache utilization, reducing cache misses, and enhancing overall performance, especially for large matrices.

This approach breaks down the matrices into smaller blocks with a specified size, referred to as 'blockSize', and executes matrix multiplication on these segmented blocks. Rather than accessing singular elements within the matrices, it computes operations on entire blocks of elements. This strategy enhances cache performance by leveraging the likelihood of contiguous memory allocation for the elements within these blocks.

The overall time and space complexities of this algorithm still are O(n^3), because $n^3 \div b^3 \times b^3 = n^3$, and O($n^2$), respectively, but with potentially better performance due to improved cache utilization, especially for large matrices.

```
for(ii=0; ii<matrixSize; ii+=blockSize) {
    for( kk=0; kk<matrixSize; kk+=blockSize){
        for( jj=0; jj<matrixSize; jj+=blockSize) {
            for (i = ii ; i < ii + blockSize ; i++) {
                for (k = kk ; k < kk + blockSize ; k++) {
                    for (j = jj ; j < jj + blockSize ; j++) {
                        phc[i*matrixSize+j] += pha[i*matrixSize+k] * phb[k*matrixSize+j];
                    }
                }
            }
        }
    }
}
```

Figure 3: Block Multiplication Algorithm

2. Performance metrics

2.1. PAPI Events

As previously mentioned, the Performance API (PAPI) was used to collect data directly from the CPU. The PAPI project specifies a standard application programming interface (API) for accessing hardware performance counters available on most modern microprocessors. These counters exist as a small set of registers that count Events, occurrences of specific signals related to the processor's function. PAPI offers access to a wide range of events, but for the context of this project, we were interested in two Data Cache Misses, them being:

- **PAPI_L1 DCM** - Level 1 Data Cache Misses
- **PAPI_L2 DCM** - Level 2 Data Cache Misses

2.2. Time

For every test, we also kept the time the CPU took to perform the multiplication.

2.3. Hardware

For testing, we used the FEUP's computers which are equipped with an Intel Core i7-9700 CPU, with a total of 8 cores and a cache of 12MB.

3. Results and analysis

3.1. Matrixes from 600x600 to 3000x3000

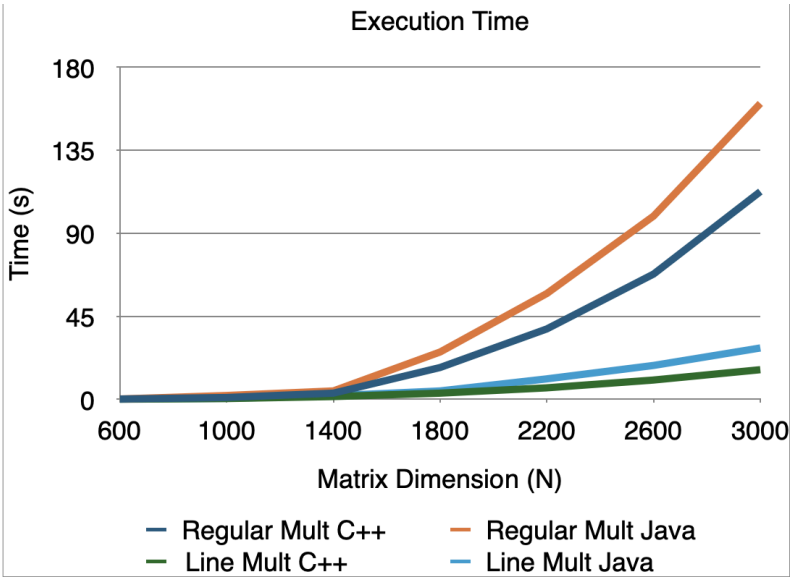

Figure 4: Execution Time for Regular and Line Multiplication

Matrix multiplication performance comparison between Java and C++ languages reveals intriguing insights into their efficiency and optimization capabilities. Initially, focusing on the conventional approach, both implementations were tested across various matrix sizes, ranging from 600x600 to 3000x3000. Notably, C++demonstrated a marginal advantage over Java.

However, delving deeper into optimization techniques, the line multiplication method emerges as a compelling alternative.This approach mitigates cache misses, minimizes instruction overhead, and accelerates execution times. This optimization capitalizes on cache line loading, where fetching neighboring data concurrently reduces cache misses, crucial for faster processing.

C++ excels in performance due to its unique capability of granting developers direct access to hardware resources and system-level programming. This feature empowers programmers to finely adjust their applications for specific hardware configurations, thereby optimizing performance by reducing unnecessary overhead. Moreover, C++ provides developers with the ability to directly manage cache memory, offering them greater control over cache utilization and minimizing cache misses. Given that fetching data from main memory is notably slower than accessing cached data, efficiently managing cache usage becomes pivotal in enhancing program performance. In contrast, Java lacks the same level of control over cache management, potentially leading to less efficient cache utilization and a higher incidence of cache misses.

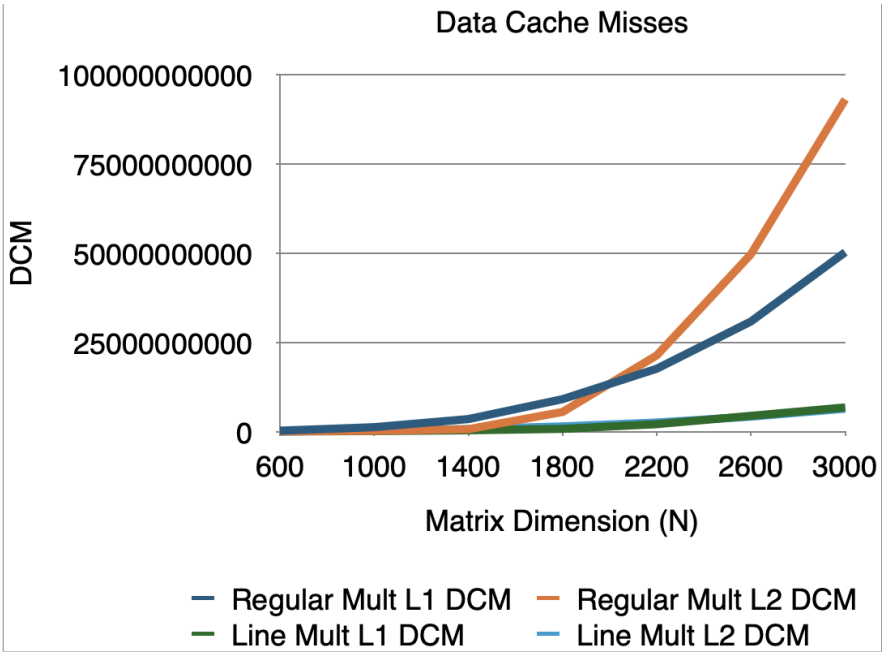

Figure 5: DCM for Regular and Line Multiplication

Examining the relationship between Data Cache Misses (DCMs) and GigaFLOPS (Gflops), a notable trend unfolds. As matrix size increases, DCMs surge, particularly L2 DCM surpassing L1 DCM. This uptick signifies heightened data fetching from slower memory subsystems, consequently impeding flops, as depicted in the accompanying graph.

Ultimately, while C++ holds a slight edge in conventional Matrix Multiplication, the efficacy of optimization techniques such as row multiplication underscores the nuanced interplay between language choice, algorithm design, and hardware considerations in achieving optimal performance.

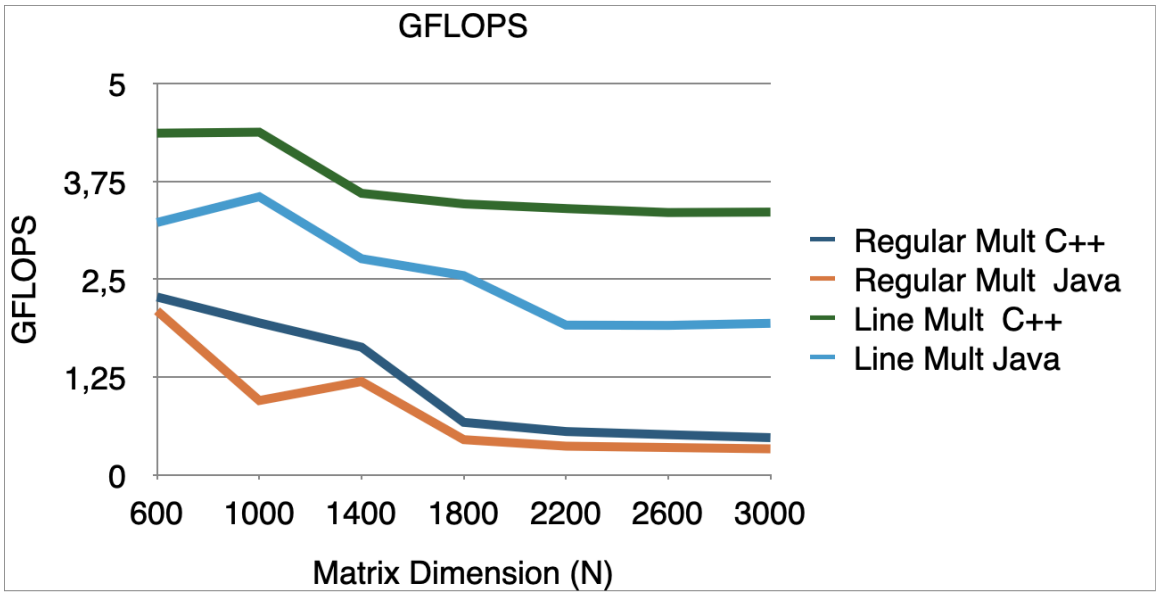

Figure 6: Floating-point Operations for Regular and Line Multiplication (GFLOPS = FLOPS *10e-9)

Although the second algorithm performs a higher number of floating-point operations per second, it exhibits fewer cache misses in both L1 and L2 caches. Additionally, there is a noticeable trend of the regular algorithm becoming slower as the dimension increases. This is attributed to its increasing need to access higher-level memories to fetch data. In contrast, the line algorithm effectively maintains a stale line, which correlates with the amount of available data in the cache and the sequential floating-point operations it executes.

Additionally, we were assigned the responsibility of evaluating the performance of the second algorithm's C++ implementation with larger matrix sizes. The testing encompassed matrix sizes ranging from 4096 to 10240. The outcomes of these assessments are visually represented in the subsequent figures:

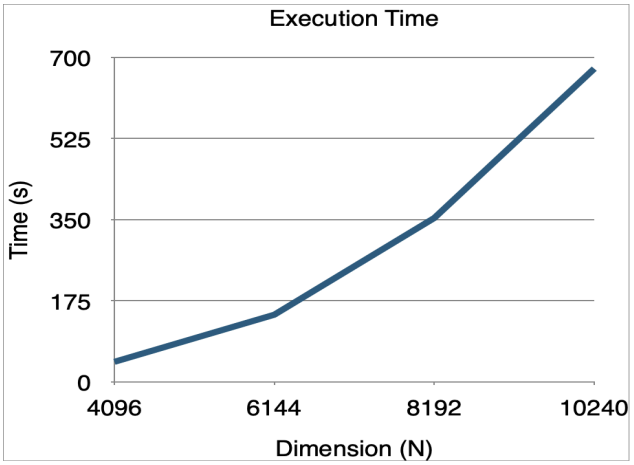

Figure 7: Execution Time for Line Multiplication

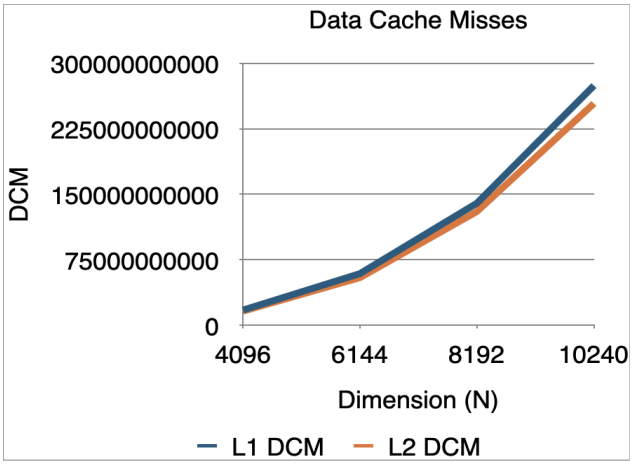

Figure 8: DCM for Line Multiplication

3.2. Matrixes from 4096x4096 to 10240x10240

The comparison between the Line Multiplication and the Block Multiplication in terms of execution time reveals that the second algorithm is more efficient independently of matrix and block size.

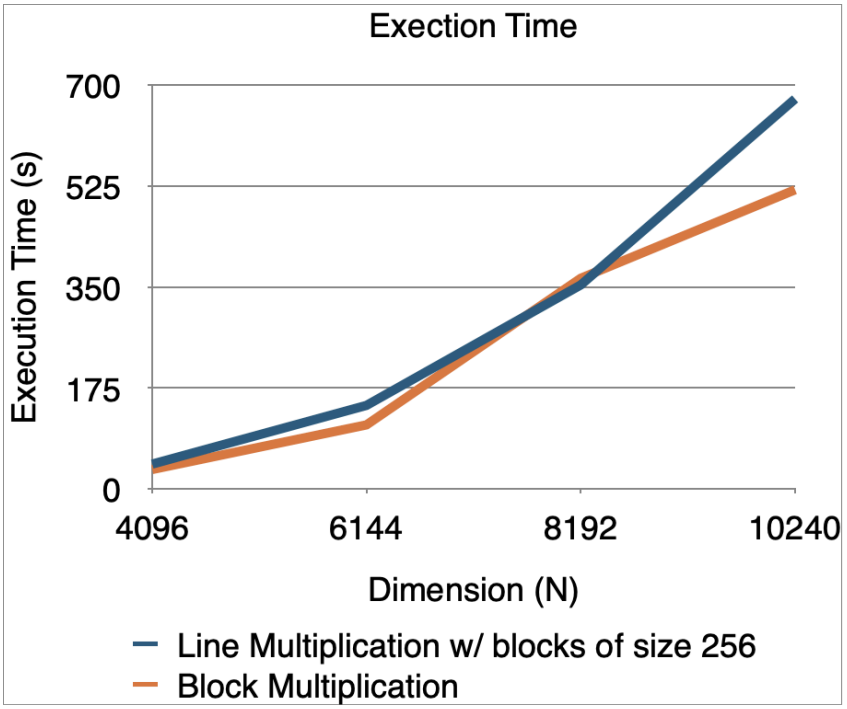

Figure 9: Execution Time for Line and Block Multiplication (of size 256)

Comparing different block sizes, we observed that the execution time decreases incrementally from sizes 32 to 256, and then increases from 256 to 2048. Therefore, we conclude that the block size of 256 seems to be the ideal size for this configuration.

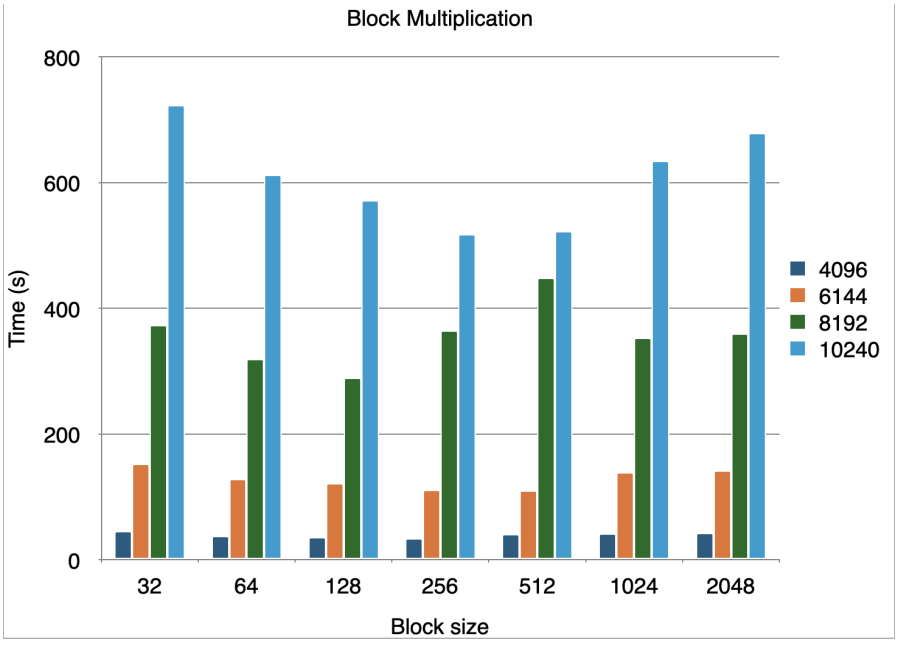

Figure 10: Execution Time for Block Multiplication with multiple block sizes

The analysis of FLOPS proved that Block Matrix Multiplication with a block size of 256x256 yielded the best result.

Figure 11: Floating-point Operations for Block Multiplication with multiple block sizes (GFLOPS = FLOPS *10e-9)

High computational power often requires advanced processors with larger and more sophisticated cache hierarchies to mitigate cache misses. Efficient caching mechanisms help minimize cache misses, thereby optimizing performance in floating-point operations.
If the cache hierarchy cannot keep pace with the processor's speed, this will lead to more frequent cache misses . Hence, striking a balance between computational power and cache efficiency is paramount for maximizing overall system performance.
As we can verify in this figure the GFLOPS increase as the cache misses decrease.

3.3. Performance evaluation of a multi-core implementation

In the first approach, we make use of OpenMP's parallelization through the #pragma omp parallel for directive, in order to parallelize the outer loop, exclusively. This means that each parallel region generated by the outer loop will sequentially execute the entire set of nested loops. On the other hand, in the second approach, we apply the #pragma omp parallel directive for the two outer loops and then the #pragma omp for directive for the most inner loop.

According to the Amdahl's law: ***Speedup*** = $\frac{1}{\frac{(1-s)}{P}+s}$,where p is the number of processors and s is the work that is sequential, therefore, (1 - s) is the work that can be parallelized.

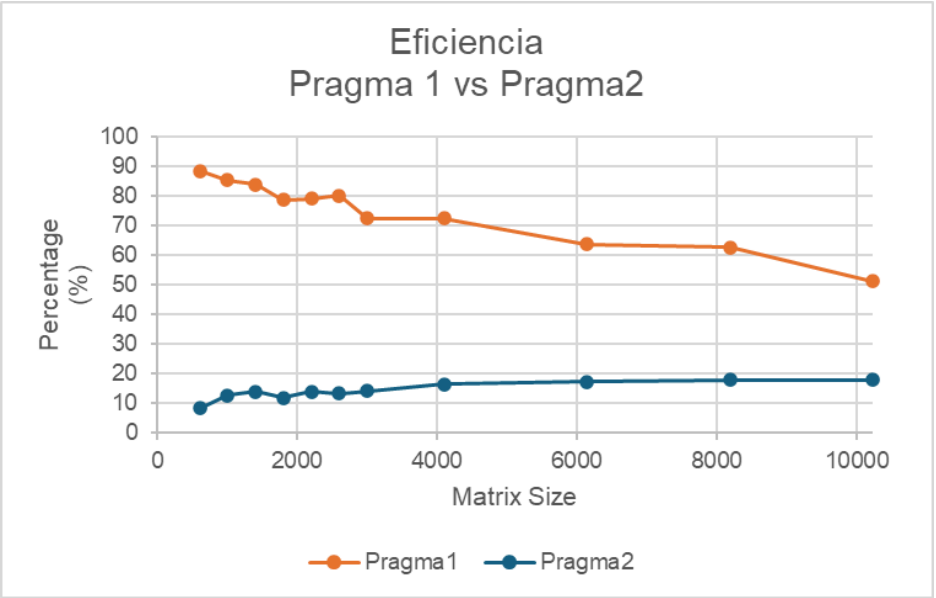

Figure 12: Efficiency for Line Multiplication w/ multi-Core

For the first approach (Pragma 1), the maximum speedup from the calculations was 7.071 for a matrix of size 600x600 using 8 processors, resulting in an efficiency of 88.39%. In the second approach, the maximum speedup was 1.421 for a matrix of size 10240x10240 and the same number of processors, resulting in an efficiency of 17.76%.

Regarding the Floating-point operations, the results observed are consistent with the efficiency of the algorithms. Although the first approach proved to be more efficient than the second, when incrementing the matrix size, the number of FLOPS decreases substantially.

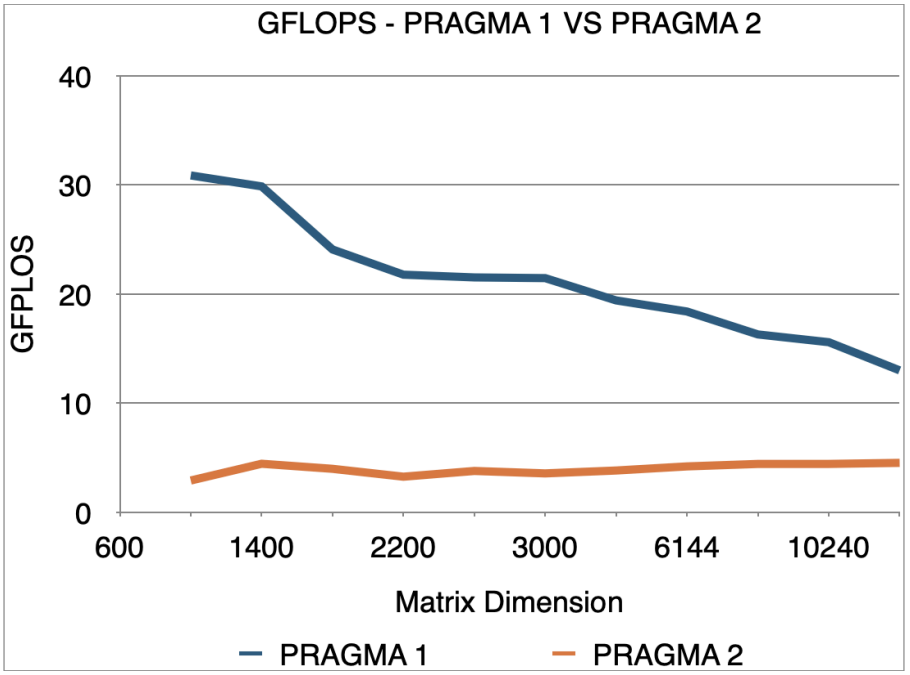

Figure 13: Floating-point Operations for Line Multiplication w/ multi-Core

4. Conclusion

In summary, this assignment successfully accomplished its objectives by testing and validating our initial hypothesis. Our investigation revealed that the compiler lacks the capability to identify certain data affinities during compilation, particularly evident in scenarios such as matrix multiplication algorithms. This highlights opportunities for enhancing code development through more efficient memory management strategies. Moreover, our examination uncovered the potential improvements promised by techniques like Line and Block Matrix Multiplication. Utilizing event data captured through PAPI, we confirmed that the optimizations primarily stem from minimizing cache misses within the matrix calculation algorithm. These findings underscore the importance of thoughtful optimization strategies in software development, particularly in computational-intensive tasks.

5. Bibliography

- PAPI Documentation
- Performance of parallel sparse matrix-matrix multiplication