Faculdade de Engenharia da Universidade do Porto

CPD first project

Performance evaluation of a single core and a multi-core implementation

Licenciatura em Engenharia Informática e Computação

Turma 8 - Grupo 17

Lara Inês Alves Cunha up202108876

Pedro Henrique Pessôa Camargo up202102365

Index

1. Problem Description

The objective of this project is to assess the impact of memory hierarchy on processor performance when handling large datasets. To achieve this, we analyze different implementations of matrix multiplication, using both single-core and multi-core approaches. The study involves measuring execution time and performance metrics using the Performance API (PAPI), comparing different algorithmic strategies, and implementing parallel versions with OpenMP. The results will help assess how memory access patterns and computational strategies influence efficiency and scalability.

2. Algorithms explanation

Matrix multiplication is a fundamental operation in numerous computational applications, and its performance is heavily influenced by how data is accessed and processed. In this study, we explore different implementations of matrix multiplication, focusing on how memory hierarchy and computational strategies impact efficiency. We begin with a straightforward row-column multiplication approach, followed by an alternative method that modifies the access pattern to improve memory usage. Additionally, we implement a block-oriented version to enhance data locality and optimize cache utilization. Finally, we extend our analysis to parallel implementations using OpenMP, evaluating how multi-core execution improves performance. Each of these approaches is analyzed in terms of execution time, computational efficiency, and scalability across different matrix sizes.

2.1 Matrix Multiplication - Algorithm OnMult

The matrix multiplication algorithm follows the standard row-column approach, where each element of the resulting matrix is computed as the sum of the products of corresponding elements from a row of the first matrix and a column of the second matrix. This method iterates through three nested loops: the outer loop selects a row from the first matrix, the middle loop selects a column from the second matrix, and the inner loop performs the element-wise multiplication and accumulation. Given that each element of the result matrix requires iterating over an entire row and column, the algorithm has a time complexity of $O(n^3)$, making it computationally expensive for large matrices.

C/C++
```c
for (i = 0; i < m_ar; i++)
  for (j = 0; j < m_br; j++)
    for (k = 0; k < m_ar; k++)
      phc[i * m_ar + j] += pha[i * m_ar + k] * phb[k * m_br + j];
```

2.2 Line Matrix Multiplication - Algorithm OnMultLine

The line matrix multiplication algorithm modifies the traditional approach by changing the order of operations to improve memory access patterns. Instead of iterating over columns in the innermost loop, this method first iterates over the rows and then processes each element in a row before moving to the next. This reordering improves cache locality, as elements of the first matrix are accessed in a row-major order, reducing the number of cache misses.

Despite this optimization, the algorithm maintains the same time complexity of $O(n^3)$, as it still requires iterating through three nested loops.

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

2.3 Block Matrix Multiplication - Algorithm OnMultBlock

The block matrix multiplication algorithm optimizes performance by dividing the matrices into smaller submatrices or blocks, which are then multiplied independently before being aggregated into the final result. This approach significantly enhances cache utilization, as smaller blocks fit better within cache memory, reducing the need for frequent memory accesses. By keeping active data in the L1 and L2 caches, block multiplication minimizes latency and improves efficiency, particularly for large matrices. Although it still has an $O(n^3)$ time complexity, the improved spatial and temporal locality leads to faster execution times.

```
C/C++
    for (int blockA = 0; blockA < numBlocksA; blockA++)
      for (int blockC = 0; blockC < numBlocksC; blockC++)
        for (int blockB = 0; blockB < numBlocksB; blockB++) {
          int actualBkA = min(bkSize, m_ar - blockA * bkSize);
          int actualBkB = min(bkSize, m_br - blockB * bkSize);
          int actualBkC = min(bkSize, m_cr - blockC * bkSize);

            for (int i = 0; i < actualBkA; i++)
              for (int k = 0; k < actualBkC; k++)
                for (int j = 0; j < actualBkB; j++)
                  phc[(i+blockA*bkSize)*m_cr+blockB*bkSize+j] +=
                    pha[(blockA*bkSize+i)*m_br+blockC*bkSize+k] *
                    phb[(k+blockC*bkSize)*m_cr+blockB*bkSize+j];
        }
```

2.4 Parallel Row-Based Matrix Multiplication - Algorithm OnMultLineParallelV1

The first parallel implementation OnMultLineParallelV1 follows the line-based multiplication approach, but with the addition of OpenMP to parallelize the outermost loop.

This means that each thread processes different rows of the result matrix independently. The key benefit of this approach is that each thread operates on distinct memory regions, reducing synchronization overhead. However, performance gains are limited by memory bandwidth constraints when multiple threads access matrix B simultaneously. For smaller matrices, Parallel V1 may be preferable due to its simplicity and reduced synchronization overhead.

```C/C++
    #pragma omp parallel for private(k, j)
    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];
```

2.5 Parallel Nested Loop Matrix Multiplication - Algorithm OnMultLineParallelV2

A second parallel approach OnMultLineParallelV2 modifies the parallelization strategy by introducing nested OpenMP parallelization. Here, OpenMP parallelizes both the outermost loop and the innermost loop, balancing the workload between multiple threads at different loop levels. This approach allows better memory utilization by controlling when and how threads access matrix B. Parallel V2 is more efficient when processing large matrices, as it reduces memory contention and improves cache locality.

```C/C++
#pragma omp parallel private(i, k)
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

In the performance metrics section of the report, we will analyze the computational efficiency of our matrix multiplication implementations based on relevant performance indicators. Given our system's hardware specifications, an Intel Core i7-9700 CPU with 8 cores, 8 threads, and a maximum clock speed of 4.7 GHz we will consider key metrics such as execution time GFLOPS, speedup and efficiency.

Furthermore, we will leverage the PAPI (Performance API) to collect hardware performance counters, including cache misses, memory accesses, and floating-point operations. Given that our CPU has a multi-level cache hierarchy (256 KB L1, 2 MB L2, and 12 MB L3), we will specifically track cache utilization and memory bandwidth to understand the impact of memory access patterns on performance.

For the block-oriented implementation, we will compare different block sizes (128, 256, 512) and observe their effect on performance, particularly in terms of cache locality and memory latency. Overall, these metrics will provide insight into the efficiency of each implementation and the impact of memory hierarchy and parallelization strategies on computational performance.

4. Results and Analysis

4.1 Line-by-line and matrix multiplication in Python and C++

The graphs illustrate the performance of matrix multiplication implementations in Python and C++, using both the traditional multiplication method (OnMult) and the line-by-line optimized version (OnMultLine). Performance was evaluated based on execution time and floating-point operations per second (MFLOPS).

In the Matrix Dimension x Time graph, it is evident that the C++ implementations consistently outperform their Python counterparts across all tested matrix dimensions. This highlights the computational efficiency and memory access advantages of C++. Among the Python implementations, the OnMultLine approach is slightly faster than OnMult, showing that reorganizing memory access patterns benefits execution, even in an interpreted environment. For C++, the performance difference between OnMult and OnMultLine is less pronounced, likely due to the compiler’s ability to optimize memory usage more effectively.

In the MFLOPS x Matrix Dimension graph, the C++ implementations also demonstrate significantly higher performance compared to Python. For larger matrix dimensions, the MFLOPS rate of the OnMultLine implementation in C++ is approximately double that of the OnMult version in Python, underscoring the impact of low-level optimizations in maximizing hardware utilization. However, the MFLOPS rates of all implementations tend to decline as matrix dimensions grow, possibly due to memory bandwidth limitations and increased cache access costs at higher levels of the memory hierarchy.

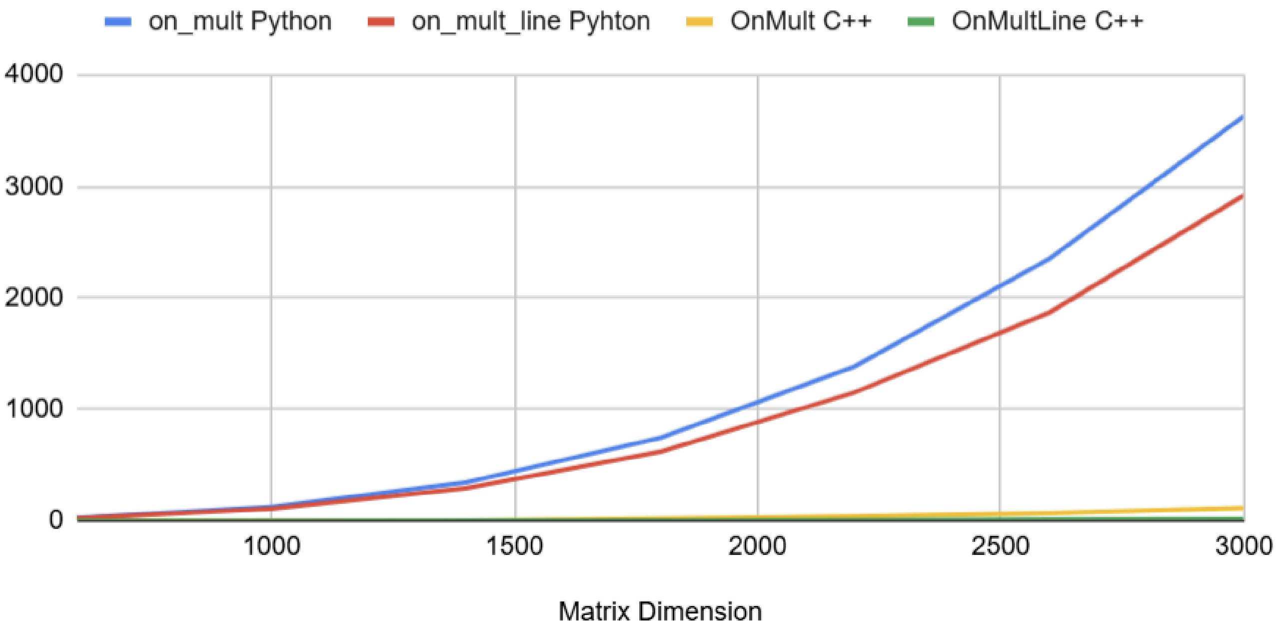

Fig 1. Matrix Dimension x Time graph

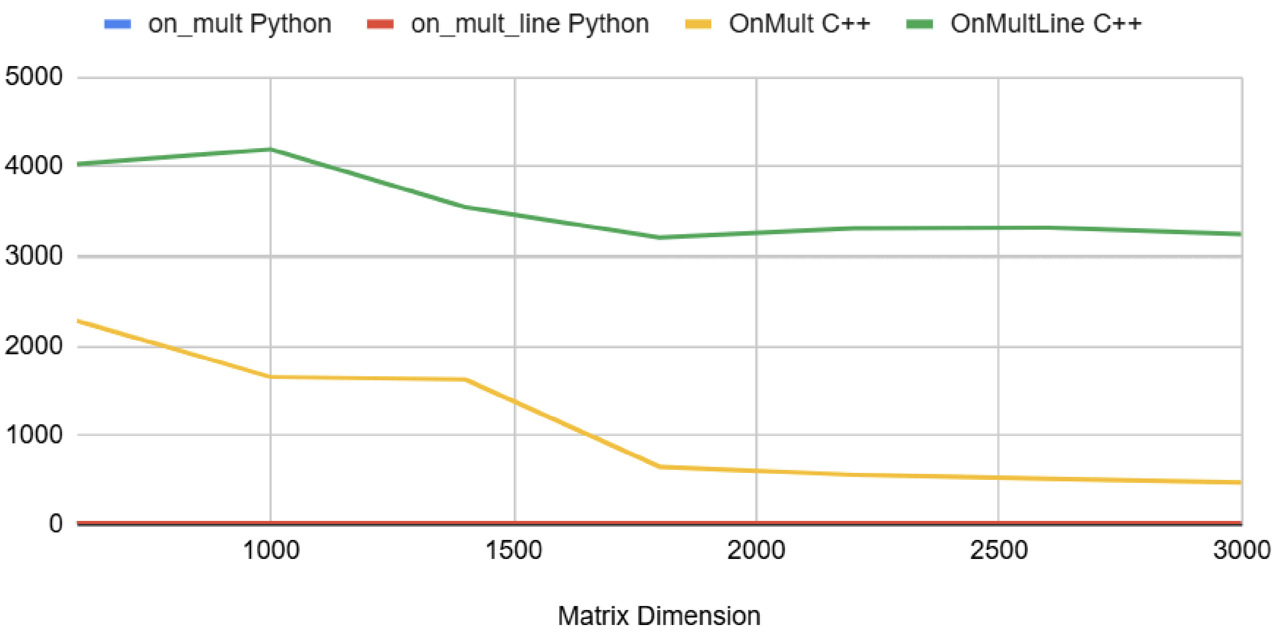

Fig 2. Matrix Dimension x MFlops graph

4.2 Block Matrix Multiplication with different block sizes

The graphs compare the performance of block matrix multiplication with different block sizes (128, 256, and 512), analyzing both execution time and MFLOPS across varying matrix dimensions.

In the Time x Dimension graph, the block size of 128 consistently achieves the lowest execution times for most matrix dimensions. This suggests that smaller blocks improve cache utilization and reduce memory access latency. For smaller matrices (dimensions below 7000), the execution time differences between block sizes are minimal, but as the matrix dimension grows, larger blocks begin to show performance degradation. This is likely due to poorer cache locality when blocks become too large to fit efficiently in the L1 or L2 cache.

The MFLOPS x Matrix Dimension graph further confirms these observations. The performance peaks at a matrix dimension of approximately 6000 for all block sizes, with the block size of 128 achieving the most stable MFLOPS rates across all dimensions. Larger blocks (256 and 512) show more variability in performance, with a noticeable dip for larger matrix dimensions. This behavior can be attributed to increased cache misses and memory contention, as larger blocks require more memory bandwidth.

These results highlight the trade-offs associated with block size selection in matrix multiplication. While larger blocks may reduce the number of iterations, smaller blocks tend to optimize cache performance, especially for larger datasets.

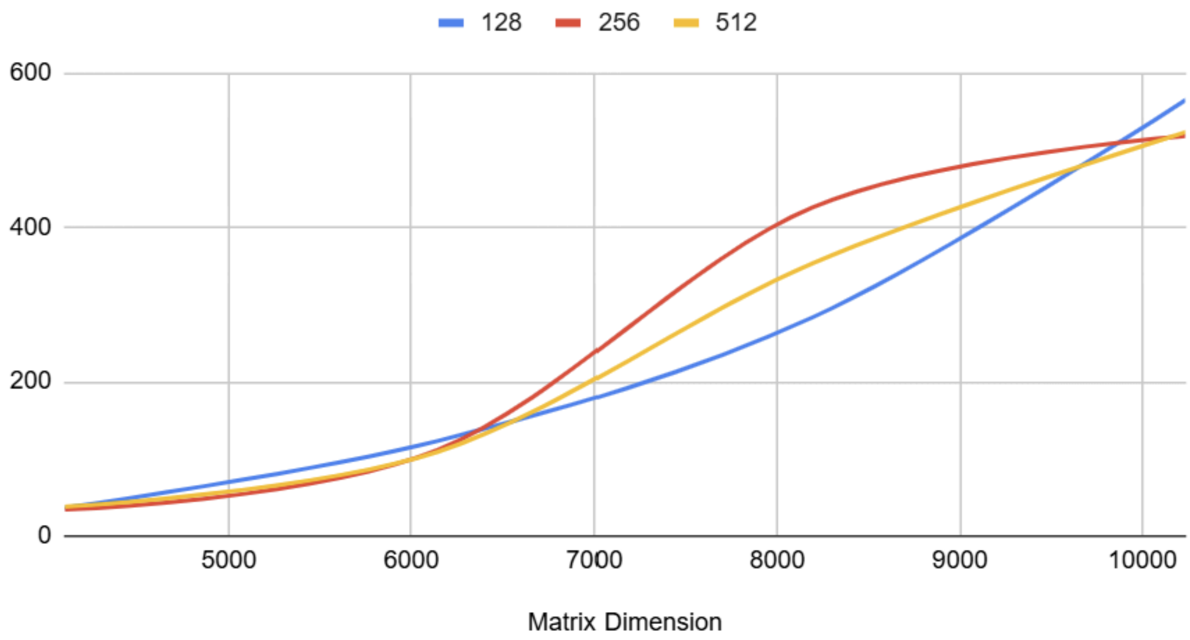

Fig 3. Matrix Dimension x Time graph

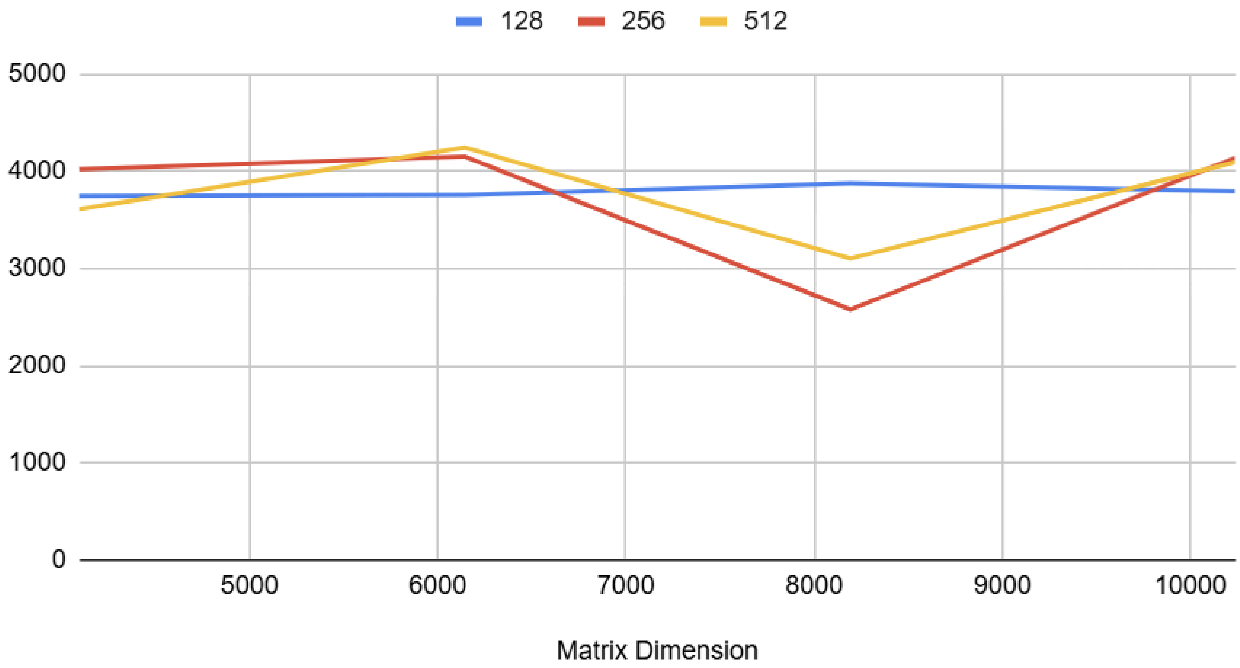

Fig 4. Matrix Dimension x MFlops graph

4.3 Line-by-line single and multi-core

The graphs analyze the performance of matrix multiplication using line-by-line processing in both single-core and multi-core implementations (Parallel V1 and Parallel V2). The comparison evaluates execution time and MFLOPS across varying matrix dimensions.

In the Time x Matrix Dimension graph, the single-core line-by-line implementation demonstrates the highest execution times, with performance degrading significantly as the matrix dimension increases. This result is expected due to the absence of parallelization, leading to a bottleneck in computation. The Parallel V1 implementation shows a substantial improvement, achieving lower execution times across all dimensions due to multi-core utilization. Parallel V2 outperforms both methods, with the lowest execution times, highlighting the benefits of an optimized multi-core approach.

The MFLOPS x Matrix Dimension graph reinforces these findings. The line-by-line implementation maintains a relatively low and stable MFLOPS rate, reflecting its inefficiency. Parallel V1 achieves better performance but exhibits some decline in MFLOPS as the matrix size increases, potentially due to overhead in thread synchronization or cache contention. In contrast, Parallel V2 consistently achieves the highest MFLOPS, showing superior scalability and efficiency, with a gradual decline attributed to memory bandwidth limitations for larger matrices.

These results demonstrate the clear advantages of parallel processing over single-core methods, with Parallel V2 emerging as the most efficient approach for large-scale matrix computations.

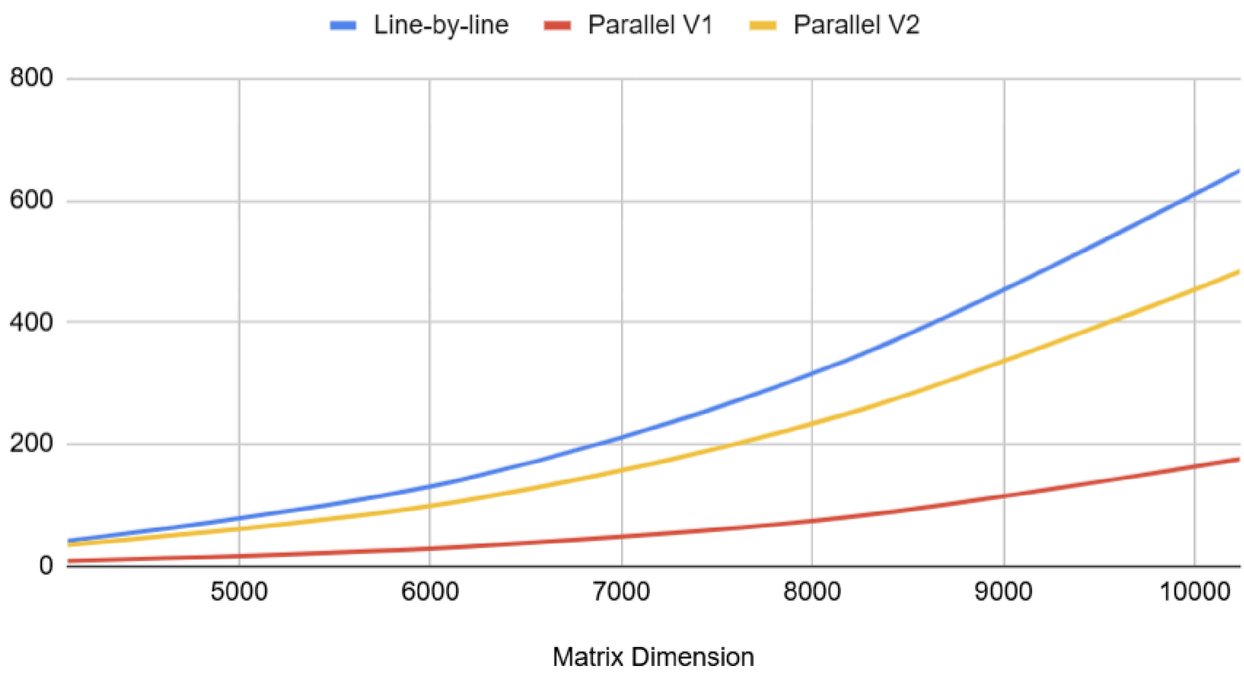

Fig 5. Matrix Dimension x Time graph

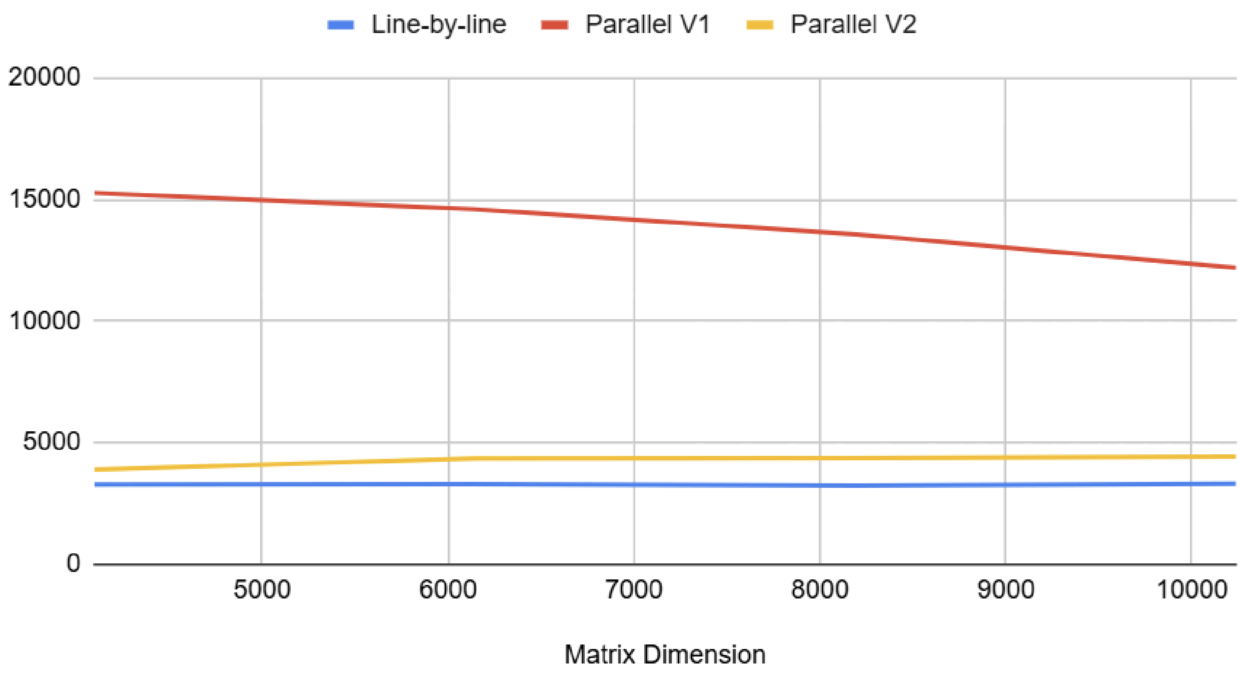

Fig 6. Matrix Dimension x MFlops graph

4.4 Line-by-line efficiency and speedup

This section evaluates the speedup and efficiency of line-by-line matrix multiplication when comparing the outer loop and inner loop implementations.

The speedup graph illustrates the performance gains of the inner and outer loop parallelization relative to the baseline. As the matrix dimension increases, the outer loop consistently achieves higher speedup values compared to the inner loop. For smaller matrices, the speedup for both implementations is closer, but as the matrix dimension grows, the outer loop diverges, showing a more pronounced performance improvement. The higher speedup for the outer loop can be attributed to its better ability to parallelize and distribute workloads effectively across cores, reducing overall computation time.

The efficiency graph provides insights into how effectively the available computational resources are utilized. Similar to the speedup trend, the outer loop demonstrates higher efficiency across all matrix dimensions. However, efficiency decreases as the matrix dimension increases, for both the outer and inner loops. This decline is expected and is caused by parallel overheads such as thread synchronization and memory access contention, which become more significant as the workload scales. The inner loop, despite achieving lower efficiency, maintains a relatively stable trend for larger matrix dimensions, indicating that its performance is less sensitive to scaling compared to the outer loop.

The results confirm that the outer loop parallelization is more effective than the inner loop in achieving both higher speedup and better resource utilization.

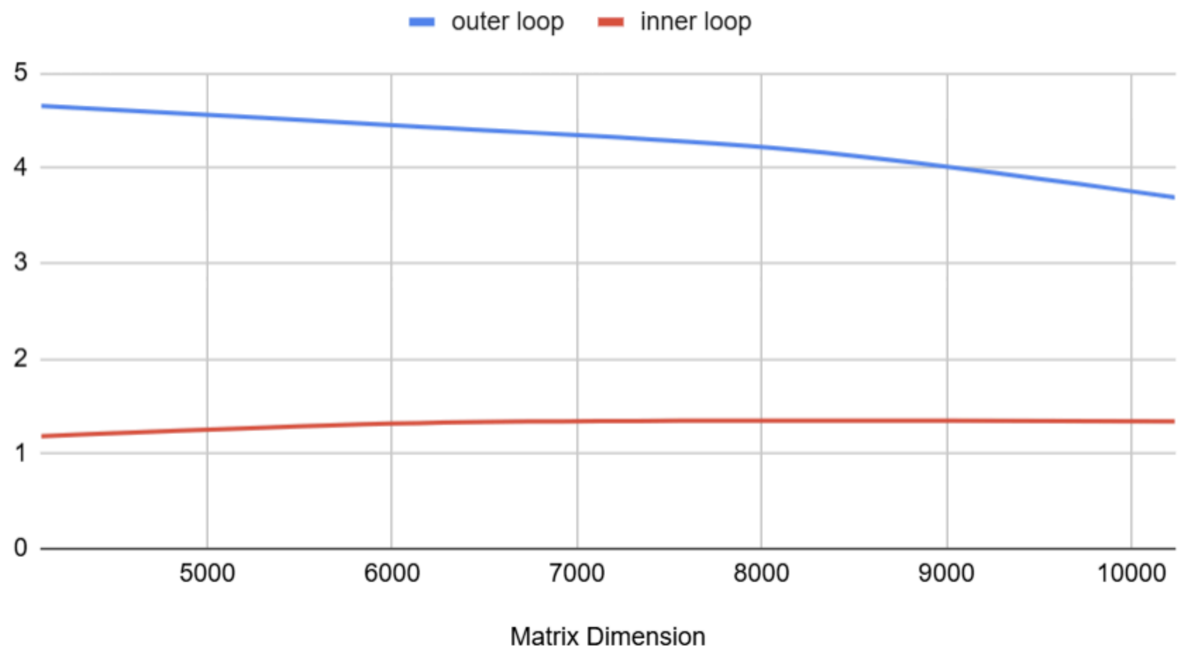

Fig 7. Parallelized outer and inner loop speedup graph

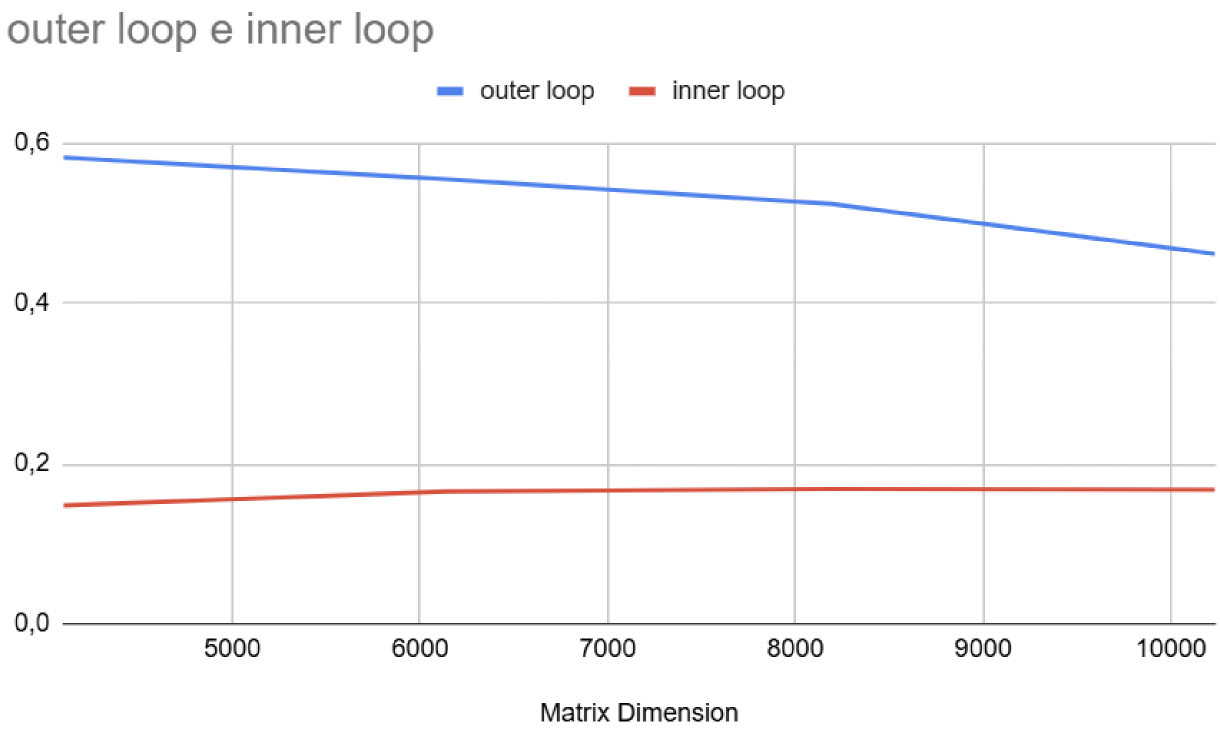

Fig 8. Parallelized outer and inner loop efficiency graph

5. Conclusions

The results confirm that the outer loop parallelization is more effective than the inner loop in achieving both higher speedup and better resource utilization. Nevertheless, the drop in efficiency with larger matrices highlights the potential for further optimization, such as minimizing parallelization overhead or improving memory access patterns.

This report explored the performance evaluation of different implementations of matrix multiplication, focusing on the impact of memory hierarchy and parallel processing on computational efficiency. The project analyzed traditional, line-by-line, and block-based matrix multiplication algorithms, as well as parallelized implementations using OpenMP, measuring their performance on a single-core and multi-core system.

The results demonstrated that optimizing memory access patterns and leveraging parallelization significantly improve execution time and computational efficiency. Among the single-core approaches, the line-by-line method provided better cache utilization, while the block-based implementation further enhanced performance by reducing memory access latency through efficient cache utilization. In the multi-core context, the parallelized nested-loop approach (Parallel V2) achieved the best results, demonstrating superior scalability and resource utilization, especially for larger matrix dimensions.

Performance metrics such as execution time, MFLOPS, speedup, and efficiency were thoroughly analyzed, highlighting the trade-offs associated with different strategies. The study also emphasized the importance of balancing block size and workload distribution to maximize cache efficiency and minimize memory contention.

In summary, this project underscores the critical role of memory hierarchy and parallelization in improving computational performance for large-scale matrix multiplication. These findings provide valuable insights for optimizing matrix operations in high-performance computing applications.

Annexes

A1. Final Results

A1.1 C++ Matrix multiplication using OnMult algorithm

Matrix Size	Time (s)	L1_DCM	L2_DCM	MFlops
600	0,188781	244500312	38810819	2288,365884
1000	1,2061	1227303906	336650837	1658,237294
1400	3,36574	3484749721	1857364975	1630,547814
1800	17,8964	9075568343	7712692332	651,7511902
2200	37,9949	17651369562	21940032169	560,4962771
2600	68,3297	30900268448	51353850381	514,4468657
3000	114,176	50300680470	96037628474	472,9540359

A1.2 Python matrix multiplication using on_mult algorithm

Matrix Size	Time (s)	MFlops
600	24,83058143	17,3979011
1000	122,7375188	16,29493589
1400	344,793283	15,91678339
1800	743,0972531	15,69646497
2200	1384,92857	15,37696633
2600	2350,853787	14,95286529
3000	3631,593434	14,8695059

A1.3 C++ Line-by-line matrix multiplication using OnMultLine algorithm

Matrix Size	Time (s)	L1_DCM	L2_DCM	MFlops
600	0,107146	27110391	58160819	4031,881731
1000	0,476708	125736223	264602943	4195,440395
1400	1,54746	346109674	701062914	3546,456774
1800	3,63997	745238950	1433118924	3204,422014
2200	6,43848	2073634102	2528463836	3307,612977
2600	10,5941	4412728876	4111928259	3318,073267
3000	16,6577	6780554430	6319101387	3241,744058
4096	41,9288	17545951093	16177362245	3277,912878
6144	140,978	59128341615	53468296636	3290,27556
8192	339,693	140261760148	130570742111	3236,780351
10240	649,828	273635711941	254019784057	3304,69547

A1.4 Python Line-by-line matrix multiplication using on_mult_line algorithm

Matrix Size	Time (s)	MFlops
600	22,893408	18,87006076
1000	106,034663	18,86175656
1400	291,564464	18,82259561
1800	618,400345	18,86156774
2200	1152,337756	18,48069274
2600	1868,299474	18,81497077
3000	2921,151767	18,48585911

A1.5 C++ Block matrix multiplication using OnMultBlock algorithm

Matrix Size	Block Size	Time (s)	L1_DCM	L2_DCM	MFlops
4096	128	36,6616	9735141668	32651817366	3748,853118
4096	256	34,1643	9094940493	23602179164	4022,882174
4096	512	38,0352	8773289873	19572063290	3613,467353
6144	128	123,39	32794801991	110867243306	3759,271156
6144	256	111,68	30672333250	79710083754	4153,442586
6144	512	109,237	29650434253	67906202178	4246,331078
8192	128	283,504	77818911810	266945671279	3878,293173
8192	256	426,223	72562740471	163407561003	2579,662824
8192	512	353,775	70405002824	141787765486	3107,940436
10240	128	565,771	151432676821	523578605423	3795,676427
10240	256	518,629	141966399687	369463955551	4140,693343
10240	512	523,864	137121136059	312743725156	4099,315181

A1.6 C++ Parallel Line-by-line matrix multiplication using OnMultLineParallelV1 algorithm

Matrix Size	Time (s)	L1_DCM	L2_DCM	Speedup	Efficiency	MFlops
4096	8,99635	2196390795	2115912602	4,660645706	0,5825807133	15277,19058
6144	31,7511	7423988915	7102286493	4,440098138	0,5550122673	14609,14639
8192	81,0251	17470301753	16739049510	4,192441601	0,5240552002	13570,0126
10240	176,01	34299693707	31818768135	3,691994773	0,4614993466	12200,9184

A1.7 C++ Parallel Line-by-line matrix multiplication using OnMultLineParallelV2 algorithm

Matrix Size	Time (s)	L1_DCM	L2_DCM	Speedup	Efficiency	MFlops
4096	35,3538	1267963418	2537219988	1,185977179	0,1482471474	3887,529869
6144	106,308	4155295314	8368231129	1,326127855	0,1657659819	4363,326071
8192	251,386	9436094496	16742216844	1,351280501	0,1689100626	4373,798174
10240	484,018	18553724797	31059963802	1,342569904	0,1678212381	4436,784682