U. PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Parallel and Distributed Computing

1st Project Report

Luna Gomes Cunha
Marta Gomes Ferreira Martins
Tiago Daniel Martins dos Santos

Turma 5 - Grupo 15

L.EIC
Faculdade de Engenharia da Universidade do Porto
March 21, 2025

Contents

Problem Description

The goal of this project is to evaluate the impact of memory hierarchy on processor performance when performing large-scale matrix multiplications. We implemented and analyzed three different matrix multiplication algorithms:

1. **Standard multiplication** (row by column)
2. **Line-by-line multiplication** (optimized row-based approach)
3. **Block multiplication** (cache-efficient optimization)

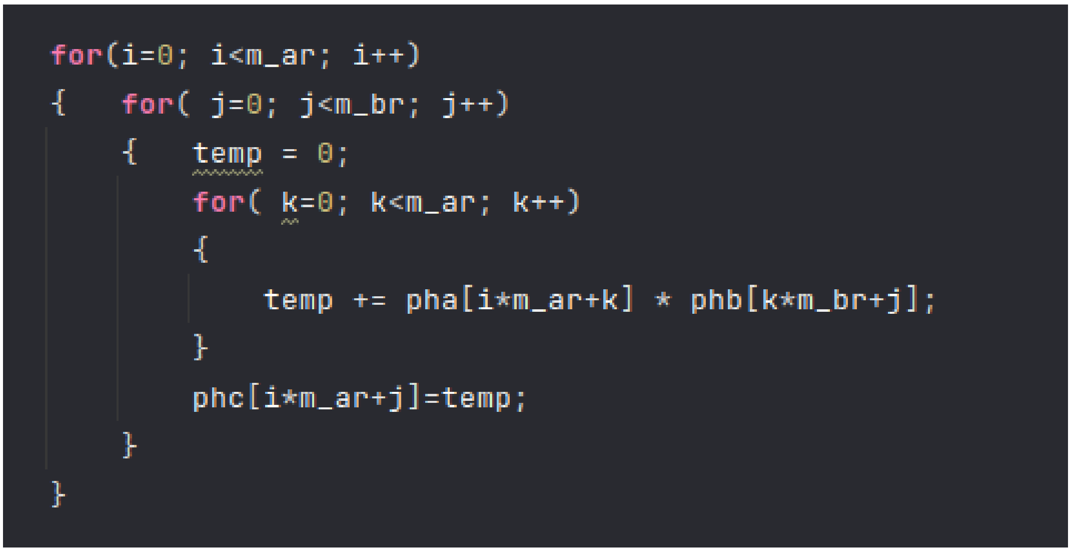
Algorithms Explanation

These 3 algorithms were implemented in C++ and in Python as requested, we only implemented standard multiplication and line-by-line multiplication in both languages and block multiplication only in c++, comparing execution times for different matrix sizes and block sizes. Additionally, the Performance API (PAPI) was used to measure cache misses (L1 DCM, L2 DCM and L3 TCM), helping us understand the effect of cache optimization.

2.1 Standard Matrix Multiplication (Row x Column)

This is the basic algorithm where each element of the resulting matrix is computed by summing the dot product of a row of Matrix A with a column of Matrix B.

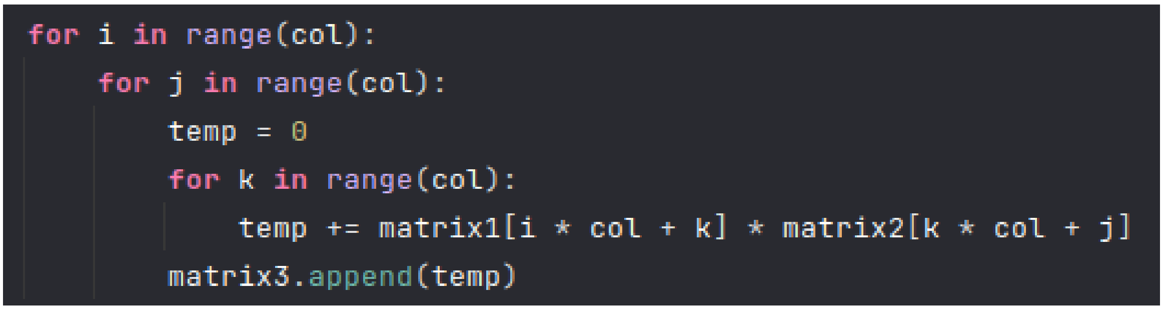
C++ Implementation (`OnMult`)



```
for(i=0; i<m_ar; i++)
{   for( j=0; j<m_br; j++)
    {   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;
    }
}
```

(Image 1 - c++ standard multiplication implementation on matrixproduct.cpp)

Python Implementation (`matrixCalc`)



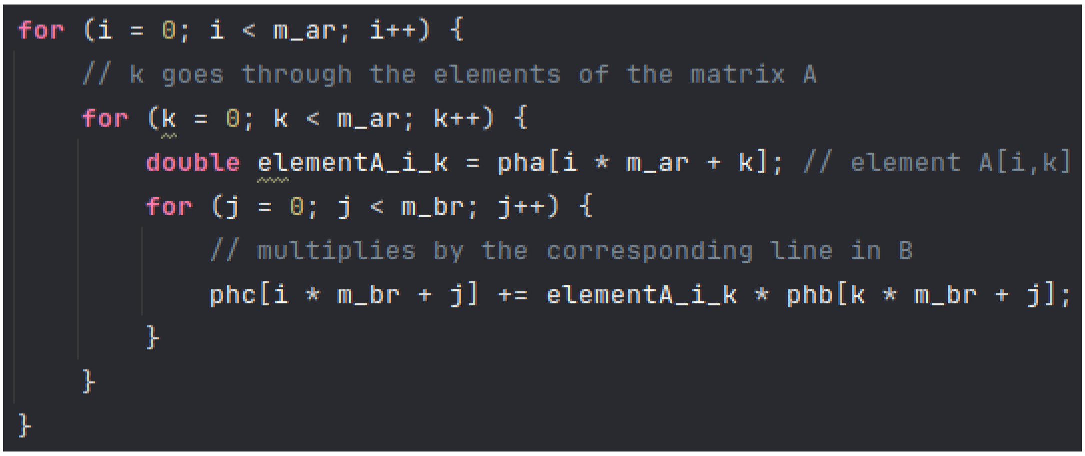
```python
for i in range(col):
    for j in range(col):
        temp = 0
        for k in range(col):
            temp += matrix1[i * col + k] * matrix2[k * col + j]
        matrix3.append(temp)
```

(Image 2 - python standard multiplication implementation on matrixproduct.py)

2.2 Line-by-Line Multiplication

Instead of computing each element independently, this method reduces redundant memory accesses by reusing values in cache more effectively.

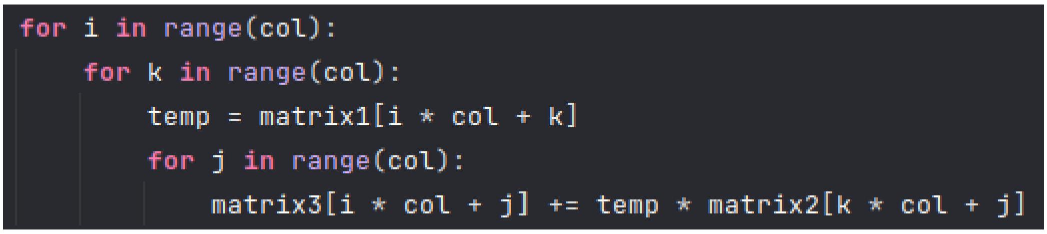
C++ Implementation (`OnMultLine`)



```cpp
for (i = 0; i < m_ar; i++) {
    // k goes through the elements of the matrix A
    for (k = 0; k < m_ar; k++) {
        double elementA_i_k = pha[i * m_ar + k]; // element A[i,k]
        for (j = 0; j < m_br; j++) {
            // multiplies by the corresponding line in B
            phc[i * m_br + j] += elementA_i_k * phb[k * m_br + j];
        }
    }
}
```

(Image 3 - c++ line-by-line multiplication implementation on matrixproduct.cpp)

Python Implementation (`matrixLine`)



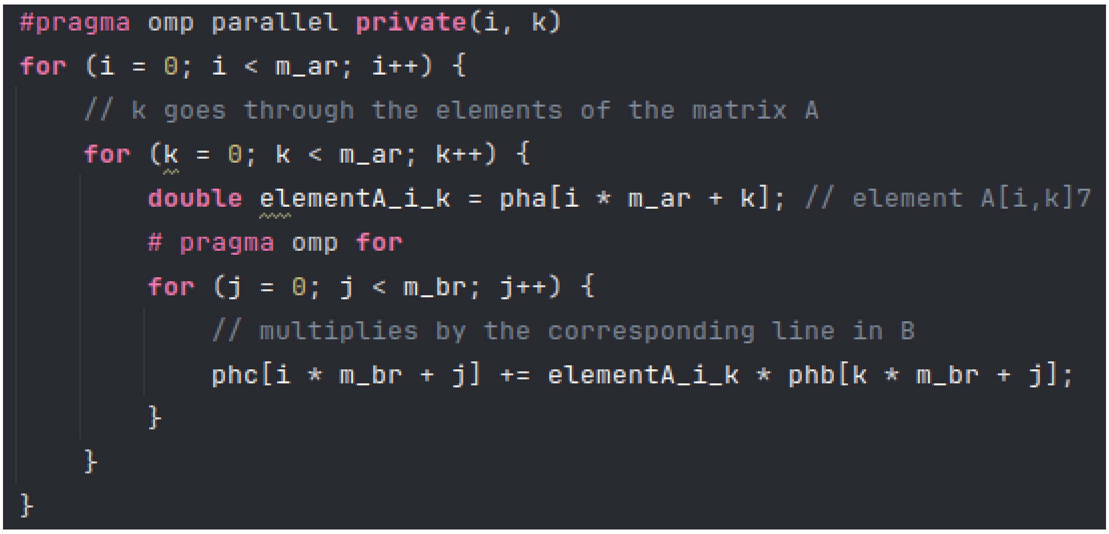
```python
for i in range(col):
    for k in range(col):
        temp = matrix1[i * col + k]
        for j in range(col):
            matrix3[i * col + j] += temp * matrix2[k * col + j]
```

(Image 4 - python line-by-line multiplication implementation on matrixproduct.py)

2.3 Parallel Multiplication

The following two functions implement parallel matrix multiplication using OpenMP, optimizing cache efficiency by iterating line by line rather than row by column.

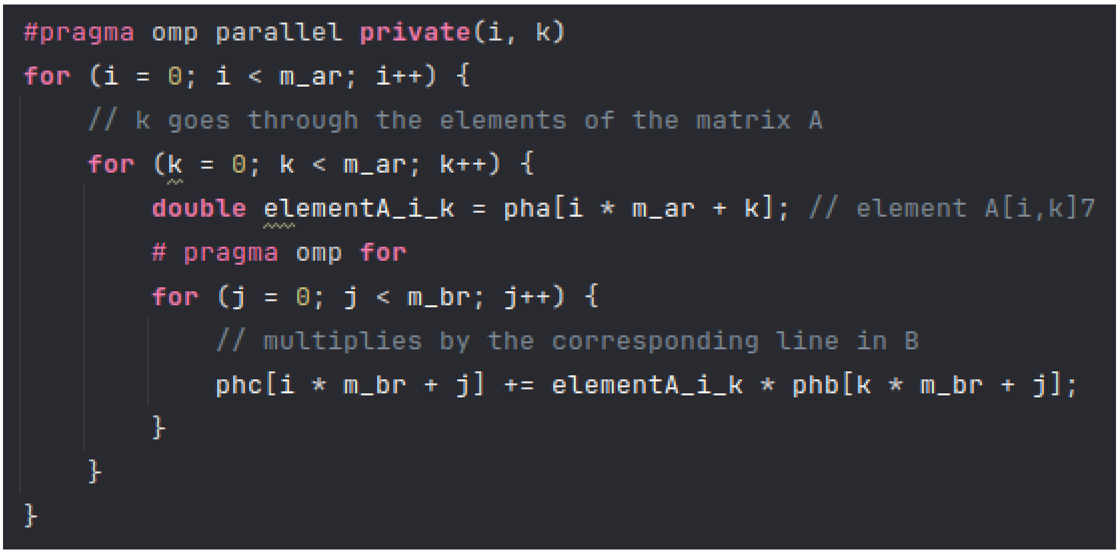
C++ Implementation (`OnMultLineParallel1`)



```cpp
#pragma omp parallel private(i, k)
for (i = 0; i < m_ar; i++) {
    // k goes through the elements of the matrix A
    for (k = 0; k < m_ar; k++) {
        double elementA_i_k = pha[i * m_ar + k]; // element A[i,k]7
        # pragma omp for
        for (j = 0; j < m_br; j++) {
            // multiplies by the corresponding line in B
            phc[i * m_br + j] += elementA_i_k * phb[k * m_br + j];
        }
    }
}
```

(Image 5 - c++ line-by-line parallel1 multiplication implementation on matrixproduct.cpp)

C++ Implementation (`OnMultLineParallel2`)



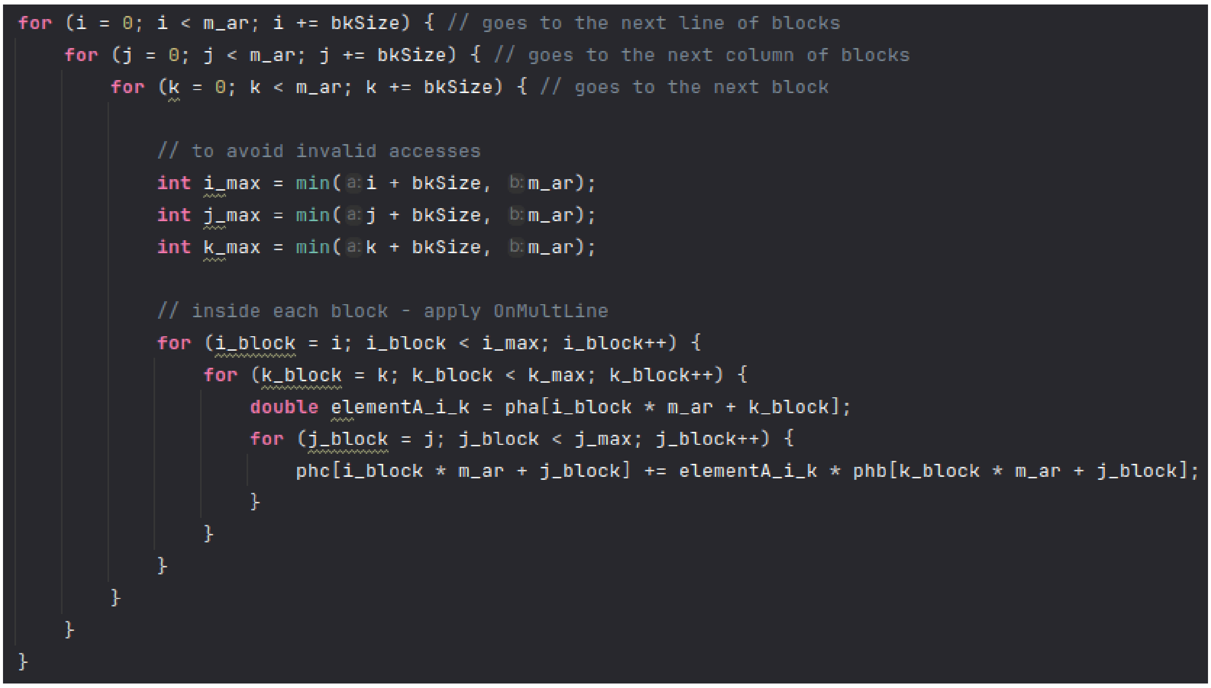
```cpp
#pragma omp parallel private(i, k)
for (i = 0; i < m_ar; i++) {
    // k goes through the elements of the matrix A
    for (k = 0; k < m_ar; k++) {
        double elementA_i_k = pha[i * m_ar + k]; // element A[i,k]7
        # pragma omp for
        for (j = 0; j < m_br; j++) {
            // multiplies by the corresponding line in B
            phc[i * m_br + j] += elementA_i_k * phb[k * m_br + j];
        }
    }
}
```

(Image 6 - c++ line-by-line parallel2 multiplication implementation on matrixproduct.cpp)

2.4 Block Multiplication (Cache Optimization)

This method divides matrices into sub-blocks and performs partial multiplications, leveraging spatial locality to reduce cache misses. This method is particularly useful for large matrices, where cache efficiency significantly affects performance. It is only implemented in C++ as requested.

C++ Implementation (`OnMultBlock`)



```
for (i = 0; i < m_ar; i += bkSize) { // goes to the next line of blocks
    for (j = 0; j < m_ar; j += bkSize) { // goes to the next column of blocks
        for (k = 0; k < m_ar; k += bkSize) { // goes to the next block

            // to avoid invalid accesses
            int i_max = min(i + bkSize, m_ar);
            int j_max = min(j + bkSize, m_ar);
            int k_max = min(k + bkSize, m_ar);

            // inside each block - apply OnMultLine
            for (i_block = i; i_block < i_max; i_block++) {
                for (k_block = k; k_block < k_max; k_block++) {
                    double elementA_i_k = pha[i_block * m_ar + k_block];
                    for (j_block = j; j_block < j_max; j_block++) {
                        phc[i_block * m_ar + j_block] += elementA_i_k * phb[k_block * m_ar + j_block];
                    }
                }
            }
        }
    }
}
```

(Image 7 - c++ block multiplication implementation on matrixproduct.cpp)

3. Performance Metrics

To evaluate the performance of the algorithms implemented in this project, we used the Performance API (PAPI), which provides access to various low-level CPU performance metrics. The key parameters analyzed include the execution time of the algorithms and the cache misses at different levels (L1, L2, and L3). On parallel versions we analized MFlops, execution time and cache misses. Since cache miss operations introduce considerable processing overhead, their variation is a critical indicator of the efficiency of our implementations. These values are measured directly from the hardware counters provided by PAPI, ensuring accuracy and reliability in our performance analysis.

Each test was performed by launching a new process for every measurement, rather than executing multiple iterations within a single process. This approach prevents memory allocation artifacts and ensures that each measurement starts with a clean execution state.

For the C++ implementations, we compiled the program using the -O2 optimization flag, which balances compilation time with runtime efficiency.

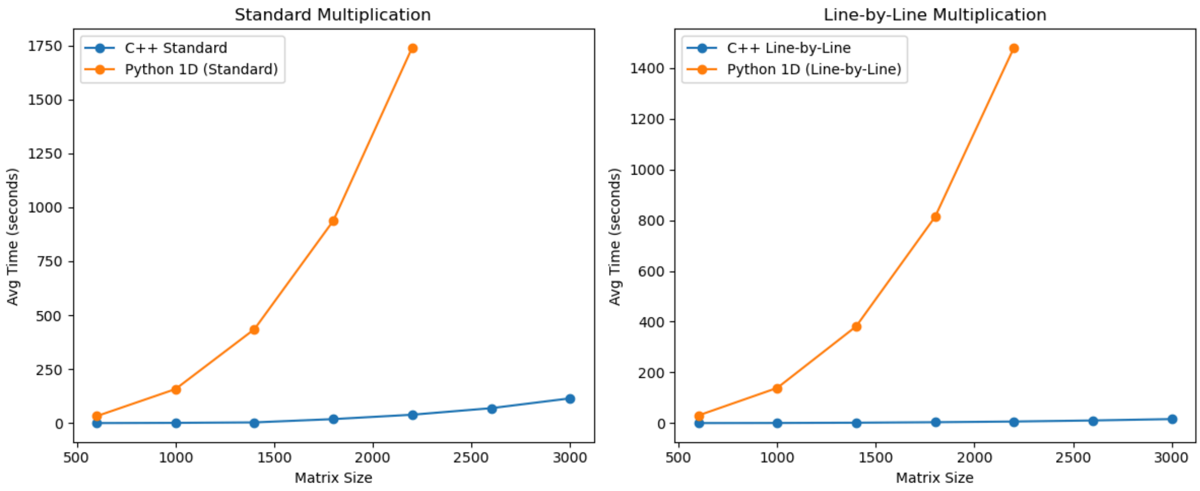
Furthermore, we conducted a comparative analysis of execution times between the C++ implementations and an equivalent implementation in Python. Since Python does not natively support low-level memory optimizations, this comparison provides insights into the efficiency of compiled languages versus interpreted languages in computationally intensive tasks such as matrix multiplication.

4. Results and Analysis

In this section, we analyze the performance of different matrix multiplication approaches implemented in C++ and Python, focusing on execution time, computational efficiency, and memory hierarchy utilization.

1. C++ vs. Python:

We compare the execution time of the standard and line-by-line multiplication implementations across different matrix sizes. This comparison highlights the efficiency differences between a compiled language (C++) and an interpreted language (Python) in computationally expensive operations.



(Image 8 - c++ vs python multiplications)

Key observations:

- C++ is Significantly Faster. Across all tested sizes, C++ outperforms Python by a large margin

Line-by-Line vs. Standard

- Line-by-Line in C++ shows lower times than Standard in C++ for these sizes, suggesting that how data is accessed row-by-row (or column-by-column) can improve cache usage or reduce overhead in the loop structure.
- Line-by-Line in Python also improves on Python's Standard approach at smaller sizes, but the overhead of the Python interpreter still dominates for large matrices.

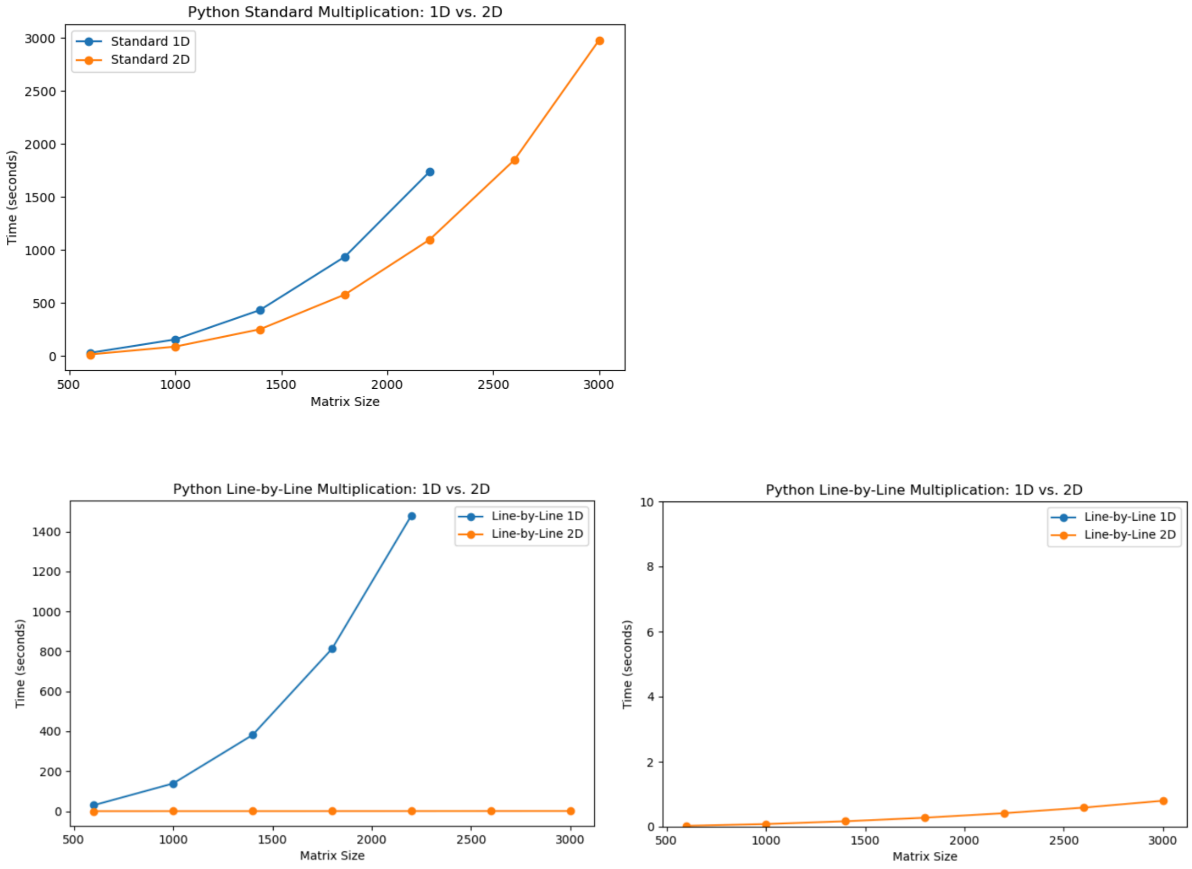
Compiled vs. Interpreted

- These results illustrate the fundamental efficiency advantage of a compiled language. C++ benefits from compiler optimizations (e.g., loop unrolling, inlining) and direct memory access.
- Python, being interpreted, pays a higher overhead per loop iteration—particularly in pure-Python list-based multiplications.

Overall, these results confirm that C++ is far more efficient for naive matrix multiplication loops, while Python's interpreted nature leads to significantly longer times, particularly at larger scales. The line-by-line approach in both languages provides some internal optimizations, but the language-level differences remain the dominant factor in performance.

2. 1D vs. 2D Matrix Representation in Python:

We evaluate the performance differences between using 1D arrays (flat lists) and 2D lists in Python for both standard multiplication and line-by-line multiplication. This analysis helps understand the impact of memory access patterns and Python's data structure overhead on execution time.



(Image 9 - python 1D vs 2D multiplications)

Line-by-Line vs. Standard (Python)

Line-by-Line consistently finishes sooner than Standard in your tests, indicating that the row-by-row approach (with carefully structured loops) reduces overhead in Python.

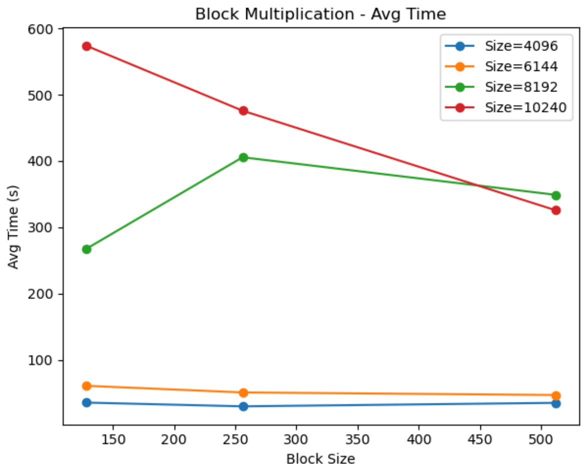
For large matrices, as the time gap between these two methods grows we see that the line-by-line logic is generally more efficient in pure Python loops.

1D vs. 2D in Python

- Line-by-Line: The 2D version outperforms the 1D version by a wide margin at larger sizes. The 1D code likely performs more arithmetic per element access, causing higher overhead in Python.
- Standard: The 1D version proves faster than the 2D version as the matrix grows. In this case, the 1D indexing appears to be less costly than repeated 2D list lookups.

3. Block Multiplication in C++:

We implemented a **block-oriented** matrix multiplication to enhance **cache locality**, testing three block sizes—**128**, **256**, and **512**—across various matrix dimensions.



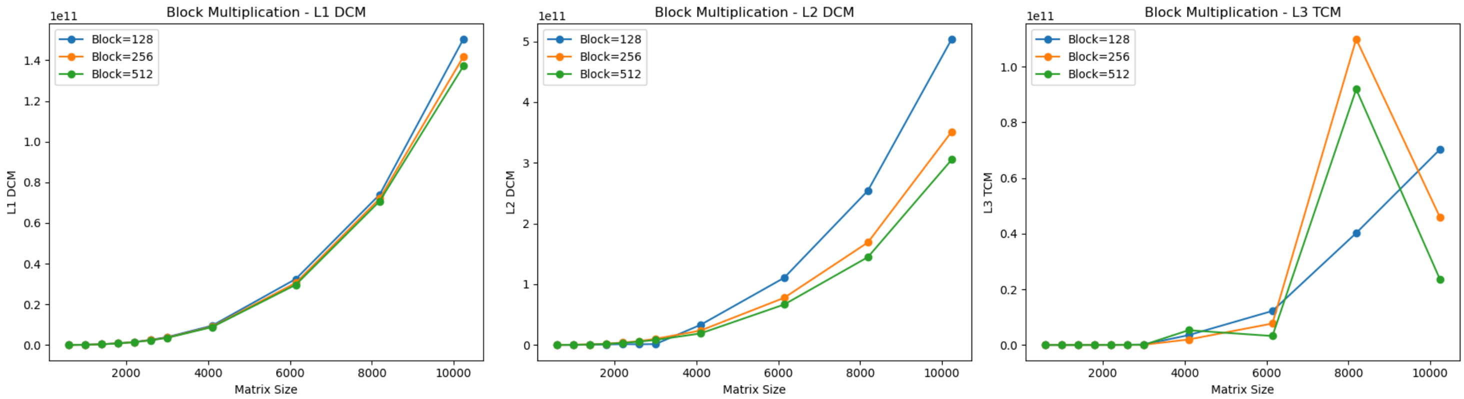
(Image 10 - block multiplication times)

In these tests, block multiplication clearly improves performance by leveraging cache locality, but there's no single block size that wins across every matrix dimension. At 4096×4096, block 256 is the fastest; for 6144×6144 and 10240×10240, block 512 does better; and at 8192×8192, block 128 comes out on top. This variation really drives home the point that you can't just pick one block size and be done—you either need to experiment with multiple options or auto-tune for each problem size, because the ideal block configuration changes as the matrix gets bigger or smaller (and depends on the machine's cache structure).

4. Cache Miss Analysis:

Using PAPI, we measure the number of cache misses at L1, L2, and L3 levels. Since L1 and L2 caches have lower latency, misses at these levels are less impactful compared to L3 cache misses, which significantly increase memory access time. We compare cache misses across different matrix sizes, evaluating how each implementation utilizes the cache hierarchy and the impact of cache efficiency on performance.

Comparing Different Block Sizes Cache misses

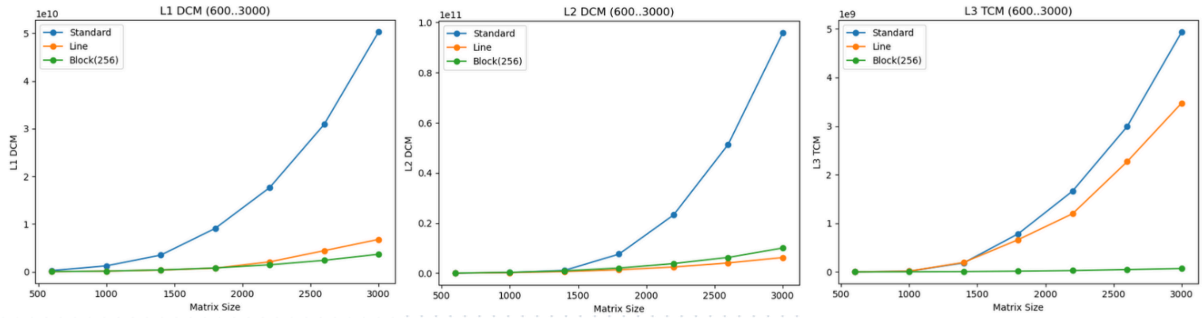


(Image 11 - block multiplication cache misses)

When we vary block size (e.g., 128, 256, 512), the number of cache misses changes with both the matrix dimension and how well that block size fits into the CPU's cache hierarchy. Generally:

- Smaller matrices (600 to 3000) often benefit most from Block(512), which reduces L1 and L2 misses by keeping sub-blocks in faster caches for longer.
- Medium to large matrices (4096, 6144, 8192, 10240) may find Block (256) or Block (128) more suitable, depending on the specific size and hardware constraints. A single "best" block size doesn't exist for all dimensions.

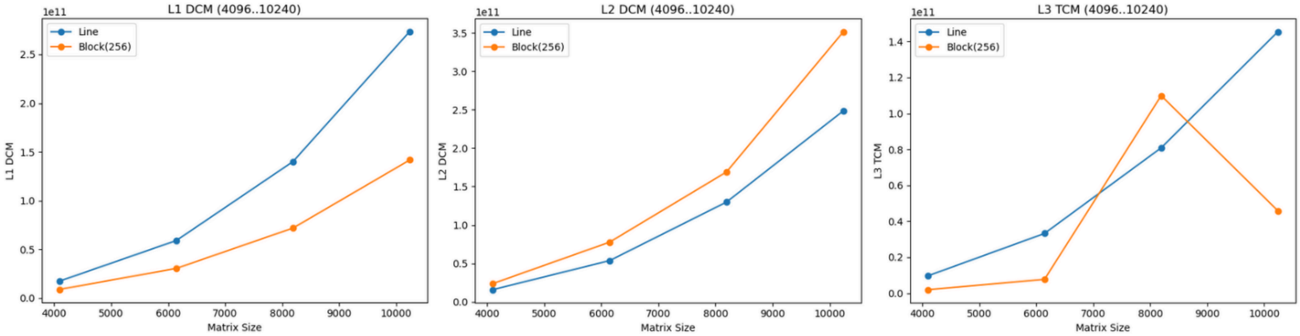
Observations for Small/Medium Matrices (600 to 3000)



(Image 12 - multiplications cache misses (600-300))

- Standard: Tends to show the highest L1, L2 and L3 misses. It reuses data less efficiently, leading to more frequent cache evictions.
- Line: Reduces misses compared to Standard, due to more sequential analysis (line-by-line).
- Block: Typically achieves the lowest misses overall. Partitioning the matrix into 256×256 sub-blocks keeps active data in caches longer, reducing memory fetches significantly.

Observations for Larger Matrices (4096 to 10240)



(Image 13 - multiplications cache misses (4096-10240))

- **Line vs. Block**: Line can still do fairly well around 4096, but as sizes move to 8192 or 10240, L2 and L3 misses increase sharply. Block remains comparatively lower in L2 and L3 misses, indicating better data locality at these scales. Both methods see large absolute miss counts, though, because total data far exceeds cache capacity.
- **L3 Misses**: The gap in L3 misses becomes especially large at 8192 and 10240. Block(256) often shows significantly fewer L3 misses than Line, correlating with better overall performance.

Impact on Performance:

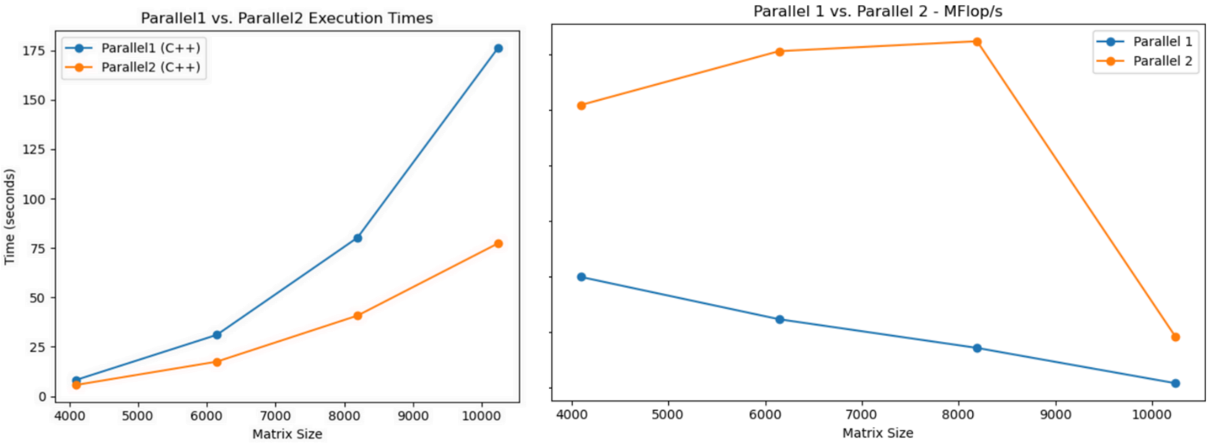
An L1 or L2 miss just means the data wasn't in those smaller, faster caches, but it can still be found in the next level (often quickly). An L3 miss is more severe because it forces a fetch from main memory, which is much slower. So while higher L1 or L2 misses can hurt performance, increases in L3 misses usually have a far bigger impact on overall runtime.

5. Parallel Implementations in C++:

The performance of two parallelized matrix multiplication implementations, Parallel 1 and Parallel 2, is evaluated. Both implementations utilize OpenMP for multi-core execution, but they differ in their parallelization strategy. We examine the impact of these differences on speedup and efficiency.

Speedup, Efficiency, and Cache Misses

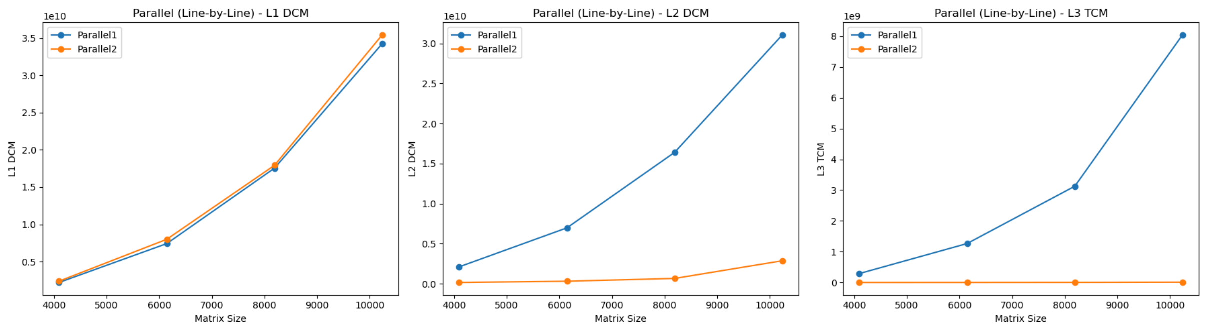
Although both implementations carry out the same total number of floating-point operations (MFlop) for a given matrix size, Parallel 2 finishes in less time—hence it achieves higher MFlop/s (more millions of operations per second). At smaller sizes, the difference may be moderate, but as the matrix grows, Parallel 2’s more efficient work distribution and fewer cache misses become increasingly evident, allowing it to maintain significantly higher MFlop/s compared to Parallel 1.



(Image 14 - line-by-line parallel 1 vs parallel 2)

Cache Misses Parallel (L1, L2, L3)

- The **three-panel** cache miss plot (L1 DCM, L2 DCM, L3 TCM) shows how each implementation uses the cache hierarchy:
    1. **Parallel 1** generally exhibits **higher** L1 and L2 DCM for larger sizes than Parallel 2, possibly reflecting less optimal data access patterns or scheduling.
    2. **Parallel 2** has noticeably **lower** L2 and L3 misses, suggesting it reuses data in caches more effectively or reduces overhead that invalidates caches. This aligns with its lower run times.
    3. Because **L3** misses have the highest latency, Parallel 2’s lower L3 TCM helps it sustain faster execution times.



(Image 15 - cache misses parallel 1)

Both parallel approaches are faster than running line-by-line on a single core, but Parallel 2 clearly does better when the matrix is large. It finishes in less time and has fewer L2 and L3 cache misses, meaning it’s making smarter use of the CPU’s resources. Parallel 1 does speed things up compared to a single thread, but it runs into more overhead and worse load balancing at bigger sizes, which shows up in higher miss counts. We only showed

MFlops for Parallel 1, but the fact that Parallel 2 runs faster and has fewer misses suggests it likely matches or beats Parallel 1's throughput in practice.

5. Conclusions

Overall, our results confirm that C++ outperforms Python by a substantial margin in matrix multiplication, largely due to compiler optimizations and more direct memory access. Within Python itself, the line-by-line method is generally more efficient than the standard row-by-column approach, but whether 1D or 2D indexing wins depends heavily on how the code is structured.

Additionally, when looking at Standard, Line, and Block (in C++) at the ranges we tested, Line consistently outperforms Standard for smaller matrices (up to 3000), while Block shows the greatest advantage on larger matrices (4096 and beyond). Although we didn't run all three methods across the exact same sizes, it's clear that Line helps more at moderate dimensions where standard row-by-column misses caches more often, and Block becomes essential at higher dimensions to keep working sets in faster caches.

For parallel C++ implementations, Parallel 2 demonstrates superior scheduling and resource utilization compared to Parallel 1, achieving faster times and fewer cache misses at large matrix sizes. Blocking in C++, especially with a well-chosen block size, reduces cache misses (notably at L2 and L3 levels), leading to significant performance gains on larger matrices. However, there is no single optimal block size for every dimension.

In short, cache efficiency is central to high-performance matrix multiplication. Approaches that minimize costly L3 misses and distribute work evenly across CPU cores (such as line-by-line parallel or block multiplication) show the greatest speedups and best overall efficiency.