Faculty of Engineering of the University of Porto

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Parallel and Distributed Computing - Project 1

Performance evaluation of a single core

Bachelor in Informatics and Computing Engineering

Class 05 - Group 11

Teacher: Pedro Alexandre Guimarães Lobo Ferreira Souto

Eduardo Martins Rodrigues da Cunha up202207126@up.pt

Mariana Pinto Pereira up202207545@up.pt

Rodrigo Gomes de Araújo up202205515@up.pt

Index

1. Problem Description

Modern computer architectures rely on memory hierarchies for optimized performance, making efficient data access crucial for large matrix operations like matrix multiplication, which is a key benchmark for processor performance.

In this project, we first implemented a baseline matrix multiplication algorithm in C/C++ for matrix sizes from 600×600 to 3000×3000, and tested an alternative line-by-line approach, with multi-core parallel versions.

We then implemented block-based matrix multiplication in C/C++, testing larger matrices (4096×4096 to 10240×10240) with various block sizes to optimize data locality.

Finally, we re-implemented the algorithms in Rust, comparing performance across different languages and implementations.

2. Algorithms for matrix multiplication

In this project, we implemented three different algorithms for matrix multiplication: simple matrix multiplication, line matrix multiplication and block matrix multiplication.

Our goal was to analyze how each algorithm differed in execution time for various matrix sizes. We aimed to measure single-core performance, focusing on how each algorithm handled memory allocation.

The first algorithm was provided to us. For both the first and second algorithms, we needed to implement them not only in C/C++ but also in another programming language of our choice. We chose Rust due to its syntactic similarity, built-in memory management system, and our interest in learning the language. The third algorithm was required to be implemented only in C/C++.

2.1. Simple (Row-by-Column Approach)

For the first algorithm, we were given C/C++ code that performed a basic matrix multiplication, multiplying each row of the first matrix by each column of the second. The corresponding pseudocode is as follows:

```
Unset
For i from 0 to matrix_size:
  For j from 0 to matrix_size:
    sum = 0
    For k from 0 to matrix_size:
      sum = sum + (pha[i][k] * phb[k][j])
    phc[i][j] = sum
```

2.2. Line

In this algorithm, we traverse each row of the second matrix instead of iterating through each row of the first matrix and each column of the second, as done in the first approach. This minimizes scattered memory accesses and improves cache locality since elements are more likely to already be in cache. Overall, even though it still uses three loops, performance is enhanced. The corresponding pseudocode is as follows:

```
Unset
For i from 0 to matrix_size:
  For j from 0 to matrix_size:
    sum = pha[i][j]
      For k from 0 to matrix_size:
        phc[i][k] = phc[i][k] + (sum * phb[j][k])
```

2.3. Block

For block multiplication, the matrices are divided into smaller submatrices, which are processed separately using the line-by-line algorithm. This approach reduces cache misses and enhances performance by improving cache locality, as values are repeatedly used and remain within the blocks. Among the three approaches, block multiplication is the fastest and most efficient, especially for large matrices. The corresponding pseudocode is as follows:

```
Unset
For ii from 0 to matrix_size with step block_size:
  For jj from 0 to matrix_size with step block_size:
    For kk from 0 to matrix_size with step block_size:
      For i from ii to min(ii + block_size, matrix_size):
        For k from kk to min(kk + block_size, matrix_size):
          pha_val = pha[i][k]
          For j from jj to min(jj + block_size, matrix_size):
              phc[i][j] = phc[i][j] + (pha_val * phb[k][j])
```

2.4. Line with Multi-core

Version 1

In this version, the directive `#pragma omp parallel for` is applied to the outermost loop, which means that its iterations will be parallelized, but the inner loops `k` and `j` will be executed in a regular sequential manner. This can be visualized here:

```
C/C++
# 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 ++)
      { }
```

Version 2

In this version, the parallelism is split between the loops. In the outermost is applied the directive `#pragma omp parallel`, that creates multiple threads. In the innermost loop, is applied `#pragma omp for`, leading to its parallelization and the iterations are divided among the threads created by the first directive. We can see that here:

```
C/C++
# 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 ++)
      { }
```

3. Performance Metrics

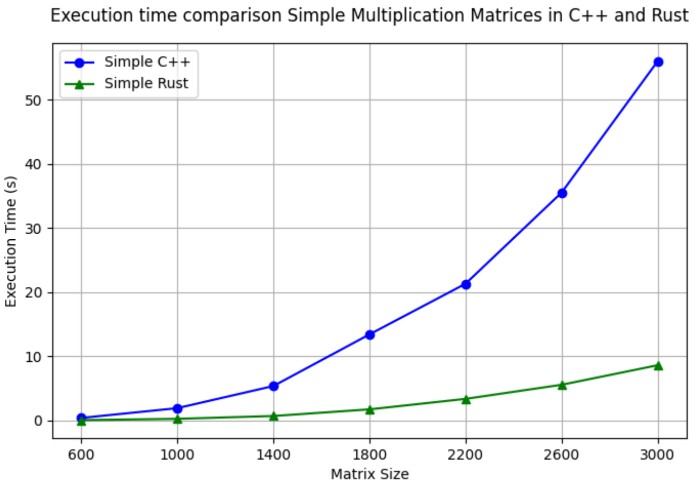
We used PAPI for precise hardware-level cache metrics (misses) and std::chrono for nanosecond-resolution timing to ensure accuracy. Compiling with g++ -O2 (instead of -O3, according to guidelines) provided a baseline for comparing optimizations fairly, avoiding unpredictable compiler aggressiveness while still enabling common speed-ups. Testing on FEUP's identical lab PCs eliminated hardware/OS variability, ensuring results reflect algorithm efficiency, not external factors.

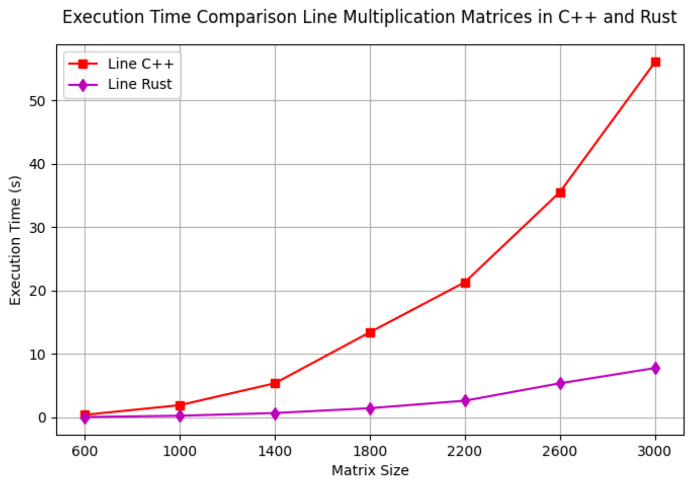
For the Rust implementation, we used the instant library for precise timing and leveraged Rust's idiomatic features (e.g., iterators, slices) to ensure a fair, language-realistic comparison. Code was compiled with cargo run --release, enabling Rust's equivalent of -O3 optimizations (auto-vectorization, loop unrolling). Due to permission constraints on FEUP's machines, C++ vs. Rust benchmarks were run on a personal MacBook Pro M2 (macOS Sonoma) to ensure both languages used identical hardware/OS environments. While this introduced a different testing platform, we maintained consistency by re-running C++ tests on the same machine, using identical matrix sizes and averaging results across multiple executions.

We automated runs with a .sh script guaranteed consistent testing conditions, while repeating tests 10x reduced noise from background processes (for larger matrix sizes we tested about 3 times for slower approaches due to the massive amount of execution time). We also saved results to CSV streamlined analysis, enabling direct comparisons and graph generation. This rigor ensures conclusions about cache efficiency and FLOPS are trustworthy and reproducible, for our project.

4. Results and Analysis

4.1. Execution Time comparison between Simple and Line Matrix Multiplication in C++ and Rust

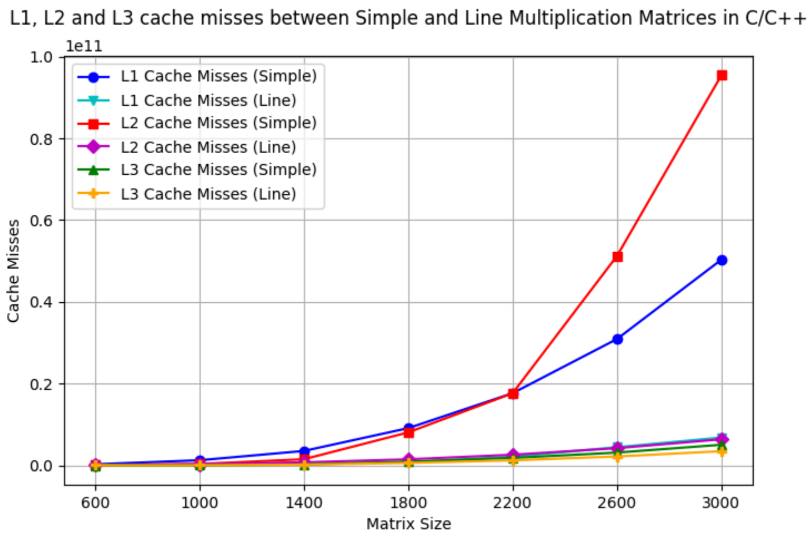




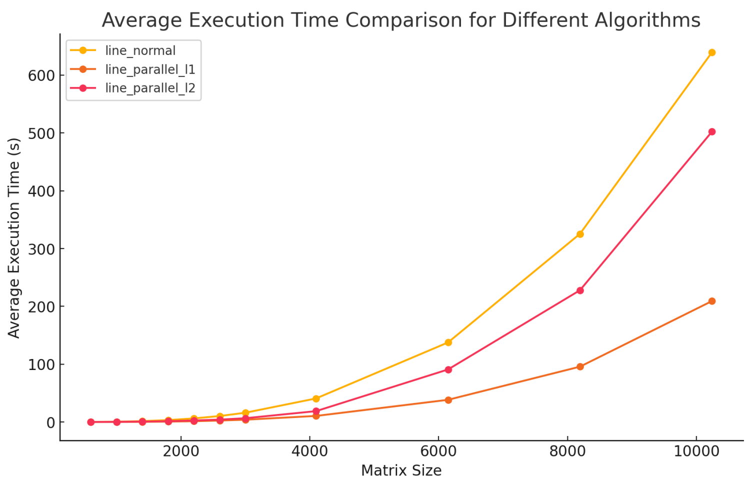
As the graphs indicate, Rust consistently outperformed C++ in both matrix multiplication implementations as matrix sizes increased, though execution times were comparable for smaller matrices. This growing performance gap stems from Rust's default aggressive optimizations under --release (equivalent to C++'s -O3), combined with its ownership model, which eliminates pointer aliasing and enables safer, more efficient memory access patterns. These compile-time guarantees allow Rust to minimize redundant memory operations and auto-vectorize loops without manual intervention. In contrast, C++ compiled with -O2 (used here for baseline fairness) lacks these optimizations by default, requiring -O3 or manual tuning (restrict keywords…) to match Rust's efficiency. While C++ could achieve parity with deeper tuning, this project's constraints prioritized -O2 as a common development standard, highlighting Rust's "optimized-by-default" advantage for memory-bound tasks.

4.2. L1, L2 and L3 data cache misses between Simple and Line Matrix Multiplication

The line-by-line matrix multiplication method demonstrates significantly fewer cache misses (L1, L2 and L3) compared to the simple method, especially for large matrices. This occurs because line multiplication implementation accesses memory sequentially, in a row-major order in its innermost loop, maximizing cache reuse and spatial locality. In contrast, the simple implementation uses column-major access for one matrix, leading to non-contiguous memory fetches and frequent cache evictions. The graph shows cache misses for OnMult rising sharply with matrix size, while OnMultLine's contiguous access pattern reduces memory bottlenecks, improving efficiency significantly for large datasets, making the cache misses barely go up with increased matrix size. Having said this, it is clear that optimizing loop order to prioritize row-major traversal is critical for improving cache performance.

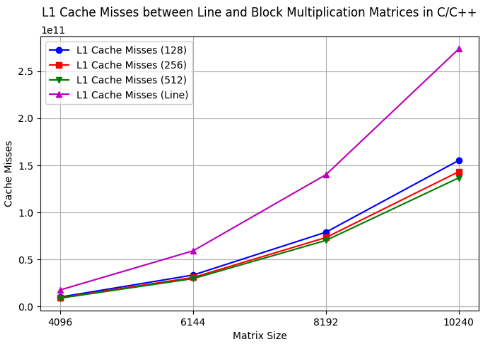


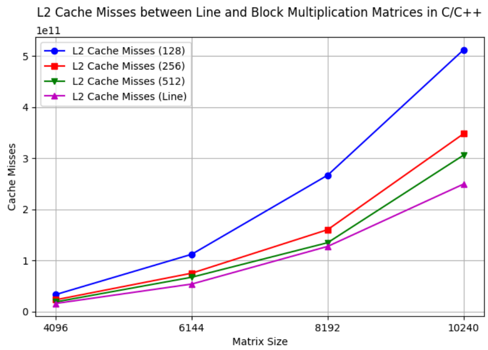
4.3. Execution Time comparison between Line Matrix Multiplication with its parallel implementation

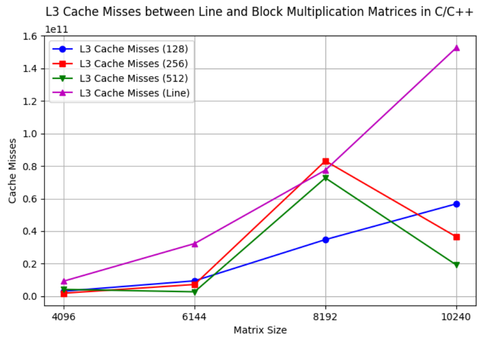


The graph shows that the outer-loop parallel method (line_parallel_1) outperforms both the sequential (line_normal) and nested-loop parallel (line_parallel_2) implementations, especially for large matrices. By parallelizing the outer loop (i), line_parallel_1 maintains contiguous memory access and minimizes thread overhead, leveraging multi-core CPUs effectively. In contrast, line_parallel_2 suffers from excessive thread contention and scheduling costs due to nested parallelism and small chunk sizes degrading performance. While line_parallel_1 scales well with matrix size, line_parallel_2 performs worse being closer to the sequential method due to fragmented memory access and synchronization overhead. The results highlight the importance of coarse-grained parallelism and cache-aware scheduling for efficient matrix multiplication.

4.4. Data cache misses between Line and Block Matrix Multiplication







For large matrix sizes, block-based matrix multiplication outperforms the line-based method in both cache efficiency (Cache L1 and L3, although for L2 it proved to be the most optimized) and runtime. While the line method accesses memory sequentially (row-major), large matrices exceed cache capacity, forcing frequent reloads of data and higher cache misses (L1/L2/L3). Block methods break matrices into smaller chunks that fit into faster cache layers, reusing cached data within blocks to minimize memory latency. For example, 128x128 blocks optimize L1/L2 cache use, while 512x512 blocks leverage L3 cache effectively, avoiding slower RAM access. This efficient data reuse, combined with predictable access patterns reduces stalls and speeds up computation. As a result, block methods achieve fewer cache misses and faster runtimes, with performance gains growing significantly at larger sizes, highlighting the importance of cache-aware algorithms for high-performance tasks.

5. Conclusions

This project demonstrates that cache-aware algorithms and language design paradigms are pivotal for high-performance matrix multiplication. Block-based methods prove superior for large matrices by strategically exploiting data locality, minimizing cache misses through localized computation. Rust's modern toolchain, with its ownership model and zero-cost abstractions, streamlines achieving peak performance by default, whereas C++ requires manual tuning to match similar efficiency. Parallel implementations further emphasize the necessity of coarse-grained parallelism to mitigate thread overhead and maintain cache coherence. Ultimately, the results highlight that optimizing for memory hierarchy — leveraging faster cache tiers and minimizing RAM access — is as critical as raw computational power for performance-critical tasks. These insights validate the centrality of algorithm design and language choice in unlocking hardware potential.

Annexes

A.1. Average Results for C/C++ and Rust on Mac (Execution Time)

A.1.1. Simple Matrix Multiplication

Dimension	Rust Execution Time (s)	C/C++ Execution Time (s)
600	0.0546769414	0.394128
1000	0.2622749416	1.91784
1400	0.6970145999999999	5.36383
1800	1.6647115	13.3975
2200	3.3875901375000006	21.3339
2600	5.7014830334	35.5447
3000	8.693898795700001	56.0194

A.1.2. Line Matrix Multiplication

Dimension	Rust Execution Time (s)	C/C++ Execution Time (s)
600	0.0587481291	0.052931
1000	0.2557755749	0.248332
1400	0.6976220498	0.679813
1800	1.6469846416	1.51107
2200	3.3640210209	2.88322
2600	5.5722279709	5.43044
3000	8.6017306082	7.8342

A.2. Average Results for C/C++

A.2.1. Simple Matrix Multiplication

Dimension	Execution Time (s)	Cache L1 misses	Cache L2 misses	Cache L3 misses	FLOPS
600	0.1864893	244746909	39650826	163385	432000000
1000	1.135391	1224855165	316932084	7423584	2000000000
1400	3.338227	3504346526	1471699812	181429536	5488000000
1800	18.10855	9088237337	8050019144	872507420	11664000000
2200	38.33927	17633163340	17633163340	1781935594	21296000000
2600	68.22986	30895738335	51226290069	3098624125	35152000000
3000	113.7859	50302223677	95427883577	5010847179	54000000000

A.2.2. Line Matrix Multiplication

Dimension	Execution Time (s)	Cache L1 misses	Cache L2 misses	Cache L3 misses	FLOPS
600	0,100195	27109199	57927022	196242	432000000
1000	0,4702675	125765602	264527982	6473679	2000000000
1400	1,534644	346263842	710633295	157113420	5488000000
1800	3,354327	746043099	1452571859	566433941	11664000000
2200	6,214872	2075596726	2574411736	1198107104	21296000000
2600	10,35419	4413574600	4176765810	2140415980	35152000000
3000	16,08105	6781583602	6346146671	3441710789	54000000000
4096	40,78428	17537567235	16053976157	9272725880	137438953472
6144	137,6005	59148757447	53982060421	32352362751	463856467968
8192	325,5035	14009042209	1276458526	77493076097	1099511627776

A.2.3. Line Matrix Multiplication (Parallel L1)

Dimension	Execution Time (s)	Cache L1 misses	Cache L2 misses	Cache L3 misses	FLOPS
600	0,02888049	3385646	7329247	32037	54000000
1000	0,1195131	15695535	33829336	501081	250000000
1400	0,3468626	43142042	90045244	6454036	686000000
1800	0,7473303	92927109	190501964	16811348	1458000000
2200	1,399922	264706748	343610672	32563730	2662000000
2600	2,578749	552431617	565772305	65384999	4394000000
3000	4,009077	848768669	866319965	99222325	6750000000
4096	10,52549	2173735487	2202384545	366193428	17179869184
6144	38,41368	7334370035	7367046706	1495367834	57982058496
8192	95,74746	17384410834	17760665641	4760834701	137438953472
10240	209,1768	33903932828	36655789621	9325924298	268435456000

A.2.4. Line Matrix Multiplication (Parallel L2)

Dimension	Execution Time (s)	Cache L1 misses	Cache L2 misses	Cache L3 misses	FLOPS
600	0,0840651	3469045	7582438	83543	54045000
1000	0,3081834	16120713	33939027	2059737	250125000
1400	0,7659245	48765188	90643931	15326032	686245000
1800	1,47047	148357697	192058652	35842849	1458405000

2200	2,693251	357548979	335868360	74304365	2662605000
2600	4,188479	607364239	546168083	121483601	4394845000
3000	6,592056	921774122	821201779	224195871	6751125000
4096	18,81433	2305298810	1952990466	749808882	17181966336
6144	90,8655	7611280493	5809452849	3886156949	57986777088
8192	227,9243	10245686068	137447342080	10245686068	137447342080
10240	502,133	28736894552	28736894552	21267378580	268448563200

A.2.5. Block Matrix Multiplication

Dimension	Block Size	Execution Time (s)	Cache L1 misses	Cache L2 misses	Cache L3 misses	FLOPS
600	128	0.108531	32289897	30533928	668742	432000000
	256	0.10554	30368000	70084365	313870	432000000
	512	0.1015566	5296545639	19087905940	2980525424	432000000
1000	128	0.499881	147247546	149165109	3996306	2000000000
	256	0.47023	136752954	368950176	1972378	2000000000
	512	0.4630	131304775	312653667	1654111	2000000000
1400	128	1.39482	402417101	419893596	14826758	5488000000
	256	1.297158	377789271	1001377895	6831130	5488000000
	512	1.296552	360163008	846255567	6385366	5488000000
1800	128	2.79663	848414391	905565731	18356432	11664000000
	256	2.73007	798892979	2123368024	14052589	11664000000

	512	2.81525	773425964	1865261978	14680553	11664000000
2200	128	5.10634	1553569525	1616153150	37826329	21296000000
	256	4.997058	1463457855	3979138487	26448494	21296000000
	512	4.872562	1599802504	3350369854	22970826	21296000000
2600	128	8.46799	2612781115	2394207057	68552498	35152000000
	256	8.178374	2414693106	6296773878	46734517	35152000000
	512	9.727518	2573123442	5430872490	34389495	35152000000
3000	128	12.41328	4056472454	3456955328	106392326	54000000000
	256	12.65342	3685761143	10036202125	66846147	54000000000
	512	12.87421	3546411194	8332055479	41172510	54000000000
4096	128	36.7899	9925207087	33641338758	4060425689	137438953472
	256	33.928276	9179971029	23193241225	2219898322	137438953472
	512	39.014798	5650187474	8315257156	2067172610	137438953472
6144	128	124.106	33490066056	113132000000	11318786767	463856467968
	256	117.72776	30756956389	75251128816	8644911665	463856467968
	512	107.35069	11831451342	66895471	1297685159	463856467968
8192	128	278.282	79427354406	262580000000	39111457048	1099511627776
	256	389.722094	73271620972	160392600000	98065066433	1099511627776
	512	338.5348	45804799663	67333973880	36608217592	1099511627776

10240	128	606.108	154263000000	506930000000	68688328892	2147483648000
	256	551.20899	143249600000	348131800000	43878822590	2147483648000
	512	513.63430	136724400000	306253800000	23120296320	2147483648000