U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

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

1st lab project

FEUP
Parallel and Distributed Computing

Group 15
Daniel dos Santos Ferreira - up202108771
Francisco Cardoso - up202108793
Mansur Mustafin - up202102355

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Index

Introduction **3**
Algorithms **3**
Normal Matrix Multiplication 3
Line per Line Matrix Multiplication 4
Block Matrix Multiplication 4
Line x Line Matrix Multiplication with multi-core 4
Version 1 4
Version 2 4
Performance metrics **5**
Results Analysis **6**
Naive matrix multiplication and line-by-line matrix multiplication with different languages. 6
Block Matrix Multiplication with different block sizes 6
Line-by-line with different threads 7
Conclusion **8**
Annexes **9**
A.1 Graph 9
A.2 Pseudocode 15
A.3 Final Results Mean 17

Introduction

This project aims to understand and explore how different programming languages, optimization techniques, and parallelization strategies affect an algorithm's performance. Due to its computational intensity and memory access patterns, matrix multiplication is the perfect example for this effect. We'll implement various algorithms to study their impact on efficiency. We'll examine both single-core and multi-core and different programming language implementations to understand their effects on performance.

Algorithms

For this project, we implemented the following matrix multiplication algorithms:

- Normal Matrix Multiplication
- Line per line Matrix Multiplication
- Block Matrix Multiplication

Each approach accesses the matrices' elements in main memory in a different order which makes more or less use of the cache, ultimately affecting performance. Additionally, we implemented the first two algorithms in both C++ and Java to verify how the programming language affects the performance. Furthermore, to verify how using multiple threads affects the performance we implemented the second algorithm both in single-core and multi-core, with the multi-core being implemented in two different ways and with a variable number of threads.

Normal Matrix Multiplication

In this first algorithm, the multiplication of the two matrices is done traditionally: for each row in the first matrix, we calculate the sum of the products of the elements of each column in the second matrix with the current row in the first matrix, sequentially. We start using the next column only when we have completed all the calculations in the current column. This algorithm is defined mathematically by the following formula, with *c* the result matrix, *a* and *b* the squared matrices to be multiplied, and *m* the number of columns of *a*:

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$$

Which can be translated in the following pseudocode.

As it will be verified in the results, this algorithm runs slower than the other ones because it doesn't take advantage of the cache's memory spatial and temporal location, as every column will be repeatedly fetched for each row of the first matrix. Because of that the values in cache will be rarely reused, resulting in worse performance.

Line per Line Matrix Multiplication

In this algorithm, instead of traversing each column in the second matrix for each row in the first matrix, we will traverse each row in the second matrix. This allows the algorithm to take advantage of the cache's memory spatial location as the probability of the next elements of a row being in the cache is high, resulting in better performance. That is the case since when a value in main memory is accessed, some of the values sequentially in front of it are stored in cache in addition to the value that was accessed. The pseudocode of this algorithm is presented here.

Block Matrix Multiplication

In this algorithm, we divide both matrices *a* and *b* into smaller blocks and apply the "Line x Line Matrix Multiplication" to those blocks. Because of that, in addition to taking advantage of the cache's memory spatial location with the algorithm above, we also take advantage of the cache's memory temporal location since the values inside the blocks will be repeatedly used, resulting in better performance. This process can be described in the following pseudocode.

Line x Line Matrix Multiplication with multi-core

This algorithm employs parallel computing to optimize matrix multiplication, utilizing OpenMP for distributing computations across multiple processor cores. There are two distinct versions, each with a different approach to parallelization:

Version 1

In this version, threads are created, and the parallelization is applied to the first *for* loop in the algorithm. This means that the iterations of the loop are divided among the available threads. The structure of this version is here.
Each thread independently executes a subset of the total iterations, which results in a distributed workload across the processor cores.

Version 2

This version initiates a parallel region using #pragma omp parallel, where each thread runs the enclosed code. Parallelization is specifically applied to the third *for* loop (#pragma omp for). Here, each thread independently executes the first and second loops and then synchronizes at the end of the third loop, for the parallel multiplication and accumulation of results for each column of matrix *b*.
The code structure is here.

Performance metrics

The evaluation of our algorithms' performance is centered on a variety of key metrics, each providing valuable insights into different aspects of computational efficiency.

1. **Execution Time:** This metric measures the duration required to complete the matrix multiplication algorithm. A shorter execution time is indicative of better performance;
2. **Cache Misses:** In the C/C++ implementation, we utilized the Performance API (PAPI) to monitor the number of cache misses at two levels: Level 1 (L1 Data Cache Misses, L1_DCM) and Level 2 (L2 Data Cache Misses, L2_DCM). We also measured the number of Level 3 cache misses in the execution of the Block Matrix Multiplication algorithm. Cache misses are a critical metric as they significantly influence processing time. A cache miss occurs when the CPU cannot locate the required data in the cache memory, leading to a time-consuming retrieval from the main memory or the cache below it. This metric is particularly insightful when comparing the effectiveness of different algorithmic approaches;
3. **Floating Point Operations Per Second (FLOPS):** We computed the FLOPS to gauge the raw processing power of our implementation. We anticipate variations in this metric across different matrix multiplication algorithms, with higher FLOPS indicating superior computational performance. GFLOPS ($10^9\ FLOPS$) was used for better understanding, as these values are generally high;
4. **Speedup and Efficiency**: For each execution of the parallelized algorithms we computed the speedup and efficiency in comparison with the single-core implementation.

Additional notes:

- All tests were conducted using an Intel Core i7-9700 processor with 8 cores and 1 thread per core, operating under Ubuntu 22.04;
- To guarantee the **accuracy** of our data, we repeated each test multiple times on the same system and calculated the average of these results.
- Java implementations did not include the Cache Misses metric in our analysis.
- The range of **matrix sizes** tested extended from 600x600 up to 10240x10240. 128, 256, and 512 block sizes were tested for the block-oriented multiplication algorithm.
- For the C++ version, the -O2 optimization flag was used for compilation.

Results Analysis

Naive matrix multiplication and line-by-line matrix multiplication with different languages.

Based on the graphs found in Annexes A.1.1.1 and A.1.1.2 which detail how the execution time and Gflops vary depending on the matrix size for both algorithms, we can easily conclude that the line-by-line algorithm is more efficient than the more naive algorithm. This superiority can be attributed to the second algorithm's utilization of cache memory's spatial locality, as elaborated earlier.

Furthermore, our evaluation indicates that the choice of programming language has minimal impact on the algorithm's runtime. Both our implementations in C++ and Java, across varying matrix sizes, demonstrated similar execution times and GFlops counts for both the naive and line-by-line matrix multiplication algorithms. Thus, it's apparent that the difference in performance between the algorithms is independent of the programming language used.

Block Matrix Multiplication with different block sizes

Based on the graphs found in Annexes A.1.2.1 and A.1.2.2 which detail how the execution time and Gflops vary depending on the matrix and block size for the Block Matrix Multiplication algorithm, we can conclude that, in this case, using a larger block size is advantageous since it generally caches a higher amount of information. However, with the matrix with 8192 lines and columns, the algorithm takes more time and the GFlops count lowers with the use of blocks with size 256 and 512. This sudden drop can be explained by a steep increase in the number of cache misses in the L3 cache which can be observed in the graphs found in Annex A.1.2.3. This might suggest that these block sizes don’t fit well in the L3 cache.

Based on the graphs found in Annexes A.1.3.1 and A.1.3.2 which compare the line-by-line and the block matrix multiplication algorithms we can conclude that the latter performs better than the former. That is to be expected since the block algorithm also utilizes the cache memory’s temporal locality in addition to the spatial locality. The only exceptions are the running times of the block algorithm with block sizes of 256 and 512 in matrices with 8192 lines and columns for the reason mentioned above.

Line-by-line single and multi-core with 6 threads

Based on the graphs found in Annexes A.1.4.1 and A.1.4.2 which detail how the execution time and Gflops vary depending on the matrix and block size for the various versions of the Line-by-Line algorithms we can easily conclude that the multi-core algorithms will always be faster than the single-core as they can take advantage of parallel processing, distributing the workload across multiple cores and executing computations concurrently. This advantage becomes particularly noticeable when handling larger matrices.

Additionally, we can also conclude that the first version of the parallel algorithm is more efficient than the second version, this is due to how the workload is divided by each version. In the first version, the parallelization is done in the outer loop, ensuring a better workload distribution among threads as each thread will be responsible for the multiplication of a line per column. In contrast, the second version parallelizes the inner loop, resulting in a worse workload distribution among threads as each thread will be responsible for only computing part of a single value in the resulting matrix, causing more idle time for synchronization.

Line-by-line with different threads

Based on the following graphs in Annex A: A.1.5.1, A.1.5.2, A.1.5.3 and A.1.5.4; we can analyze the effect of the number of threads in the speedup and efficiency of the parallelized versions of the line per line matrix multiplication algorithm.

The first aspect we can observe is that a higher speedup doesn't necessarily translate into a higher efficiency. That is because speedup measures how faster the parallelized version was compared to the algorithm with only one thread, which should generally give higher speedup values when more threads are used, whereas the efficiency measures the ratio between the speedup when N threads are used and the number of threads N, that is, in perfect conditions, the use of 8 threads should guarantee a speedup of value 8. Based on graphs A.1.5.1 and A.1.5.2 we can observe that, in general, a higher number of threads results in a higher speedup, and that efficiency decreases with the increase of the number of threads used. This last observation can be explained by an increase in the intensity of simultaneous requests of data from and to main memory, which becomes a bottleneck of the algorithm with the increase in the number of threads. This effect becomes even more noticeable with larger matrices, which results in even more requests to main memory.

From graphs A.1.5.3 and A.1.5.4, we can conclude that the speedup and efficiency are very low regardless of the number of threads that are used due to the synchronization that needs to happen at the end of every inner loop of the algorithm. Because of that, a higher number of threads results in a higher number of threads to be synchronized, which results in a higher overhead.

Conclusion

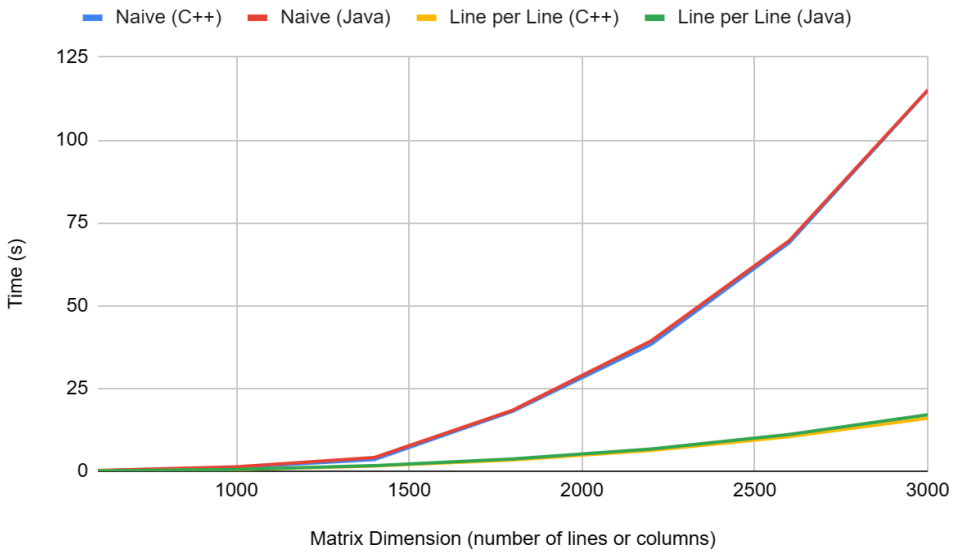
This project has provided valuable insights into memory management and its impact on program efficiency. We've observed how different algorithms that solve the same problem and which have the same time complexity can have vastly different performances due to memory access patterns.
Additionally, we were able to better understand how parallelization can drastically improve the performance of an algorithm if done correctly and have no significant improvement if done incorrectly, by putting the theory into practice using the OpenMP API.
Lastly, we were able to learn how the number of threads can change an algorithm's performance.
The complete set of results used to perform the analysis, which is the mean of every execution of each algorithm, can be found here.

Annexes

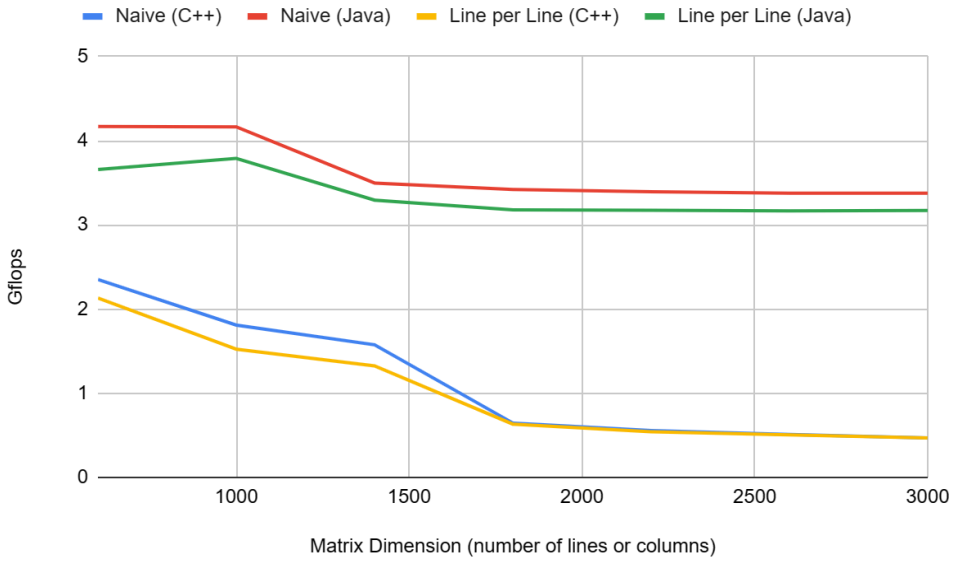
A.1 Graph

A.1.1 **Naive matrix multiplication and line-by-line matrix multiplication with different languages**

A.1.1.1 Time x Dimension

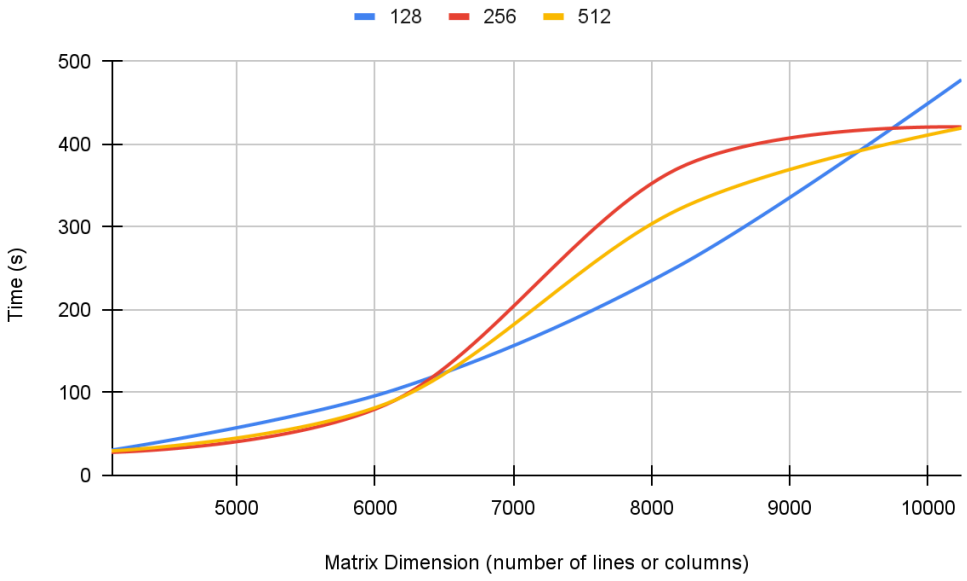


A.1.1.2 Gflops x Dimension

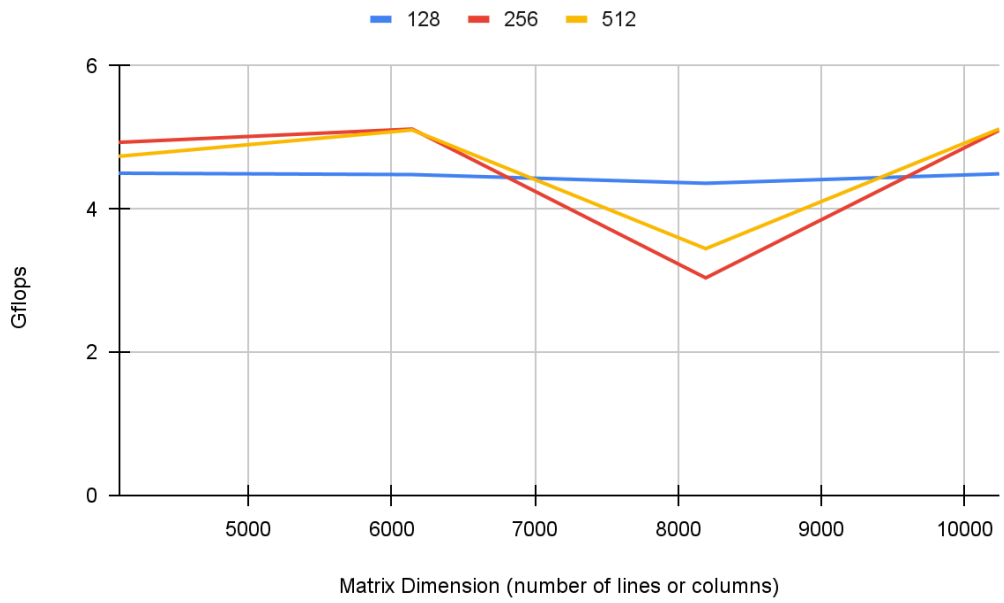


A.1.2 **Block Matrix Multiplication with different block sizes**

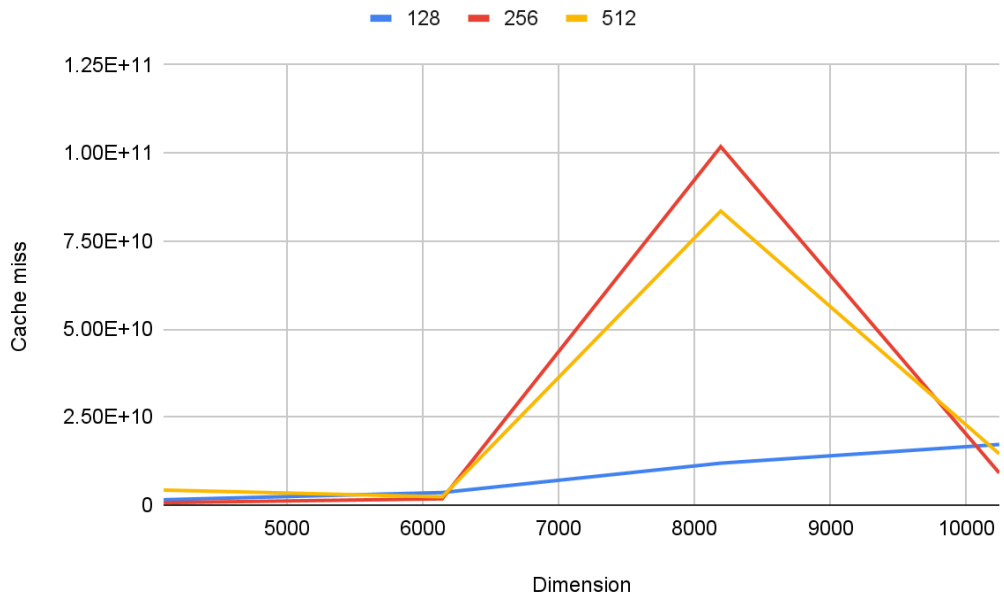
A.1.2.1 Time x Dimension



A.1.2.2 Gflops x Dimension

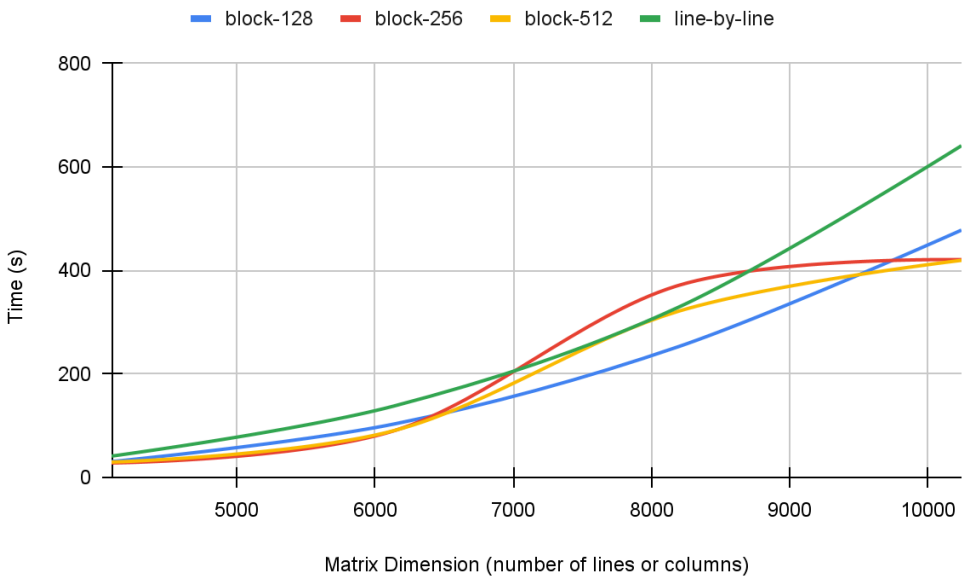


A.1.2.3 L3 Cache miss x Dimension

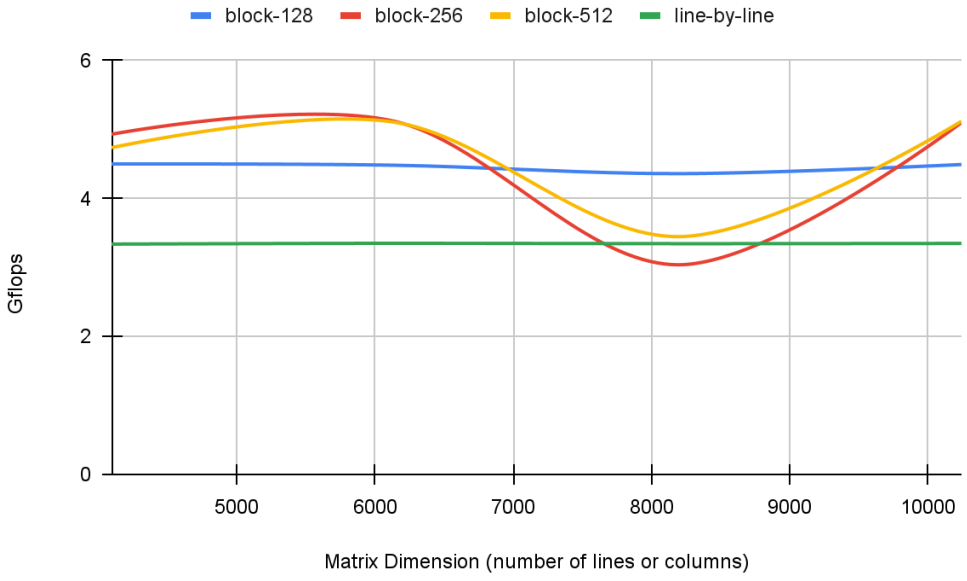


A.1.3 **Block Matrix Multiplication and Line-by-Line Multiplication**

A.1.3.1 Time x Dimension

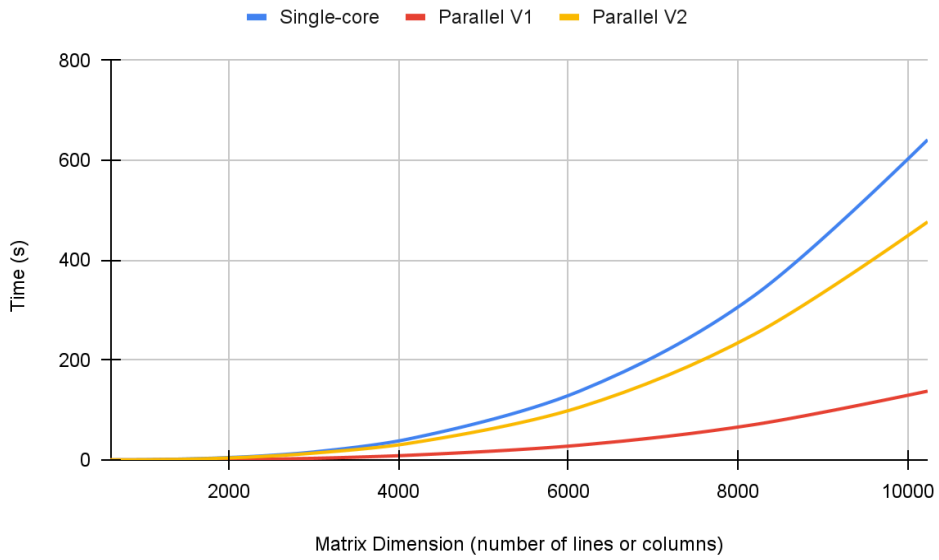


A.1.3.2 Gflops x Dimension

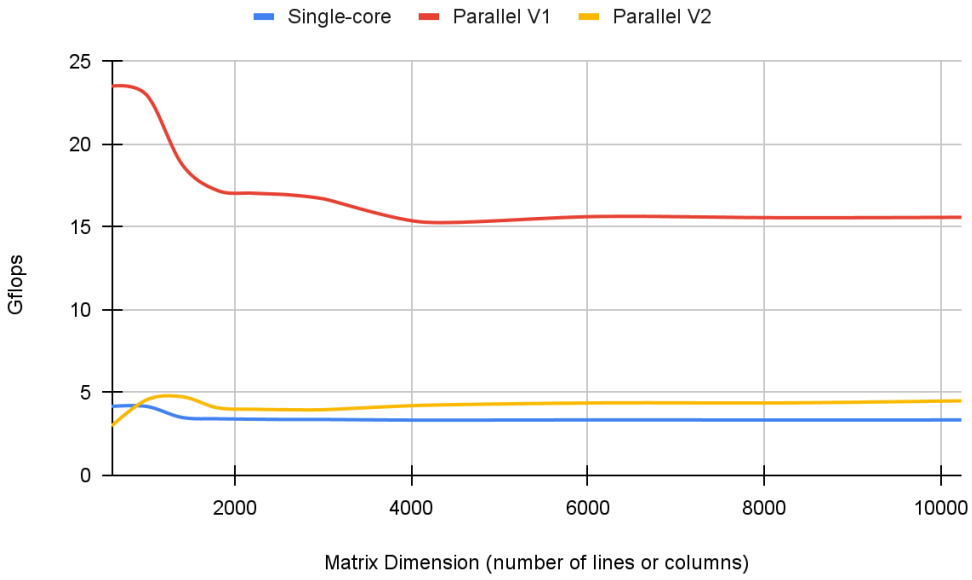


A.1.4 **Line-by-line single and multi-core with 6 threads**

A.1.4.1 Time x Dimension

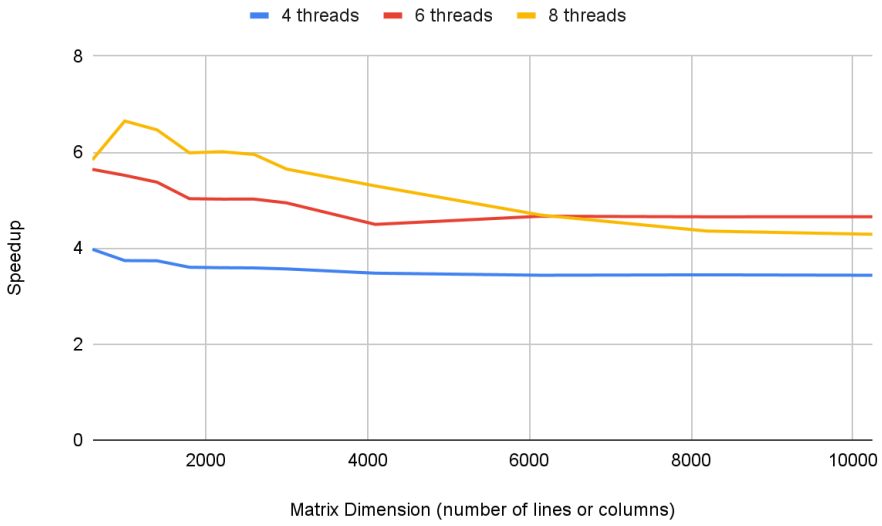


A.1.4.2 Gflops x Dimension

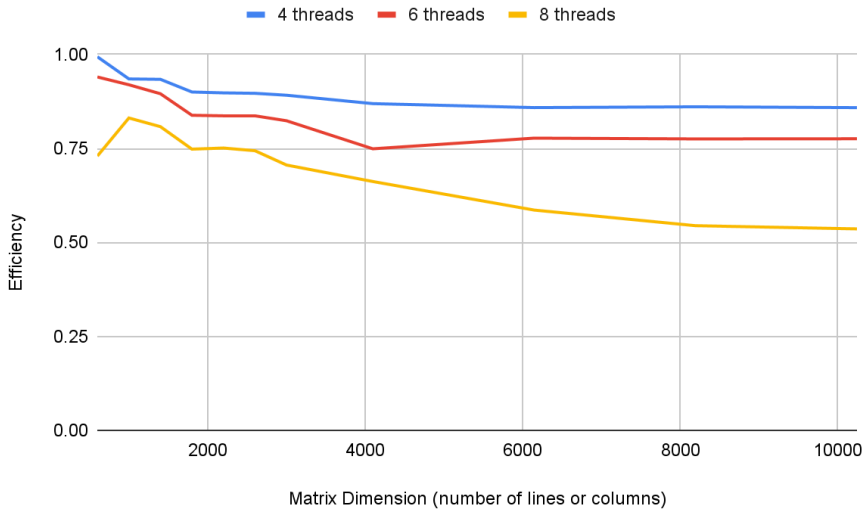


A.1.5 **Parallelized algorithms speedup and efficiency with different numbers of threads**

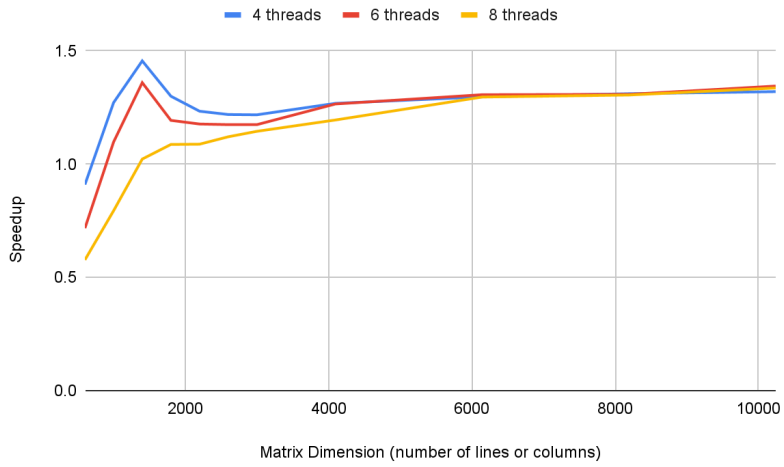
A.1.5.1 Parallelized outer loop line per line algorithm's speedup



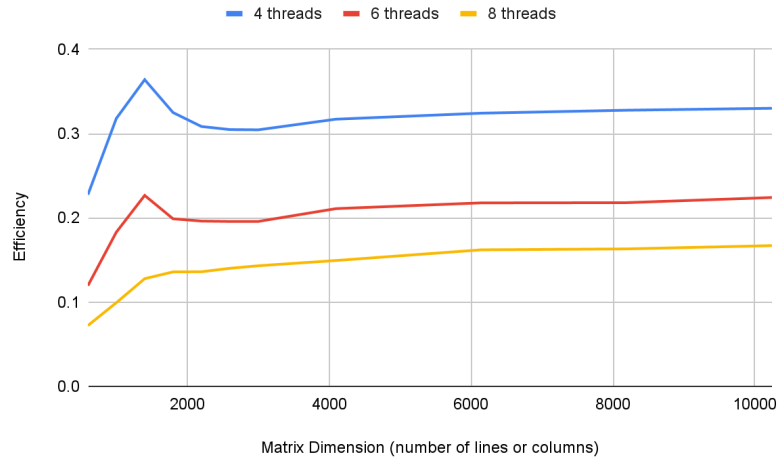
A.1.5.2 Parallelized outer loop line per line algorithm's efficiency



A.1.5.3 Parallelized inner loop line per line algorithm's speedup



A.1.5.4 Parallelized inner loop line per line algorithm's efficiency



A.2 Pseudocode

A.2.1 Normal matrix multiplication

```
Set m to number of columns of matrix a

for each line in matrix a:
    for each col in matrix b:
        for k from 0 to m:
```

```
            c[line, col] += a[line, k] * b[k, col]
```

A.2.2 Line per Line Matrix Multiplication

```
Set m to number of columns of matrix a

for each line in matrix a:
      for k from 0 to m:
            for each col in matrix b:
                c[line, col] += a[line, k] * b[k, col]
```

A.2.3 Block Matrix Multiplication

```
// Matrix.block(n,m) is a squared submatrix starting on line n and column m
with size BlockSize

set number_blocks as (m / BlockSize)
for i from 0 to number_blocks:
      for k from 0 to number_blocks:
            for j from 0 to number_blocks:
                  Result[i,j] += LineMultiplicationMatrix(A.block(i,k) ,
B.block(k,j))
```

A.2.4 Line x Line Matrix Multiplication with multi-core V1

```
#pragma omp parallel for num_threads(n_threads) private(i, k)
for each line in matrix a:
      for k from 0 to m:
            for each col in matrix b:
                c[line, col] += a[line, k] * b[k, col]
```

A.2.4 Line x Line Matrix Multiplication with multi-core V2

```
#pragma omp parallel num_threads(n_threads) private(i, k)
for each line in matrix a:
      for k from 0 to m:
            #pragma omp for
            for each col in matrix b:
```

```
c[line, col] += a[line, k] * b[k, col]
```

A.3 Final Results Mean

A.3.1 Naive Matrix Multiplication in C++

Dimension	BlockSize	Time	L1_DCM	L2_DCM	Gflops
600	0	0.18377	244779770	38682131	2.351
1000	0	1.114836	1225949849	236773461	1.8088
1400	0	3.495259	3471191580	978095214	1.576
1800	0	18.071686	9090172841	5730662022	0.6459
2200	0	38.294471	17630657821	20302880969	0.5564
2600	0	69.0226	30895931144	50719685849	0.5095
3000	0	114.973857	50305095600	96485036230	0.4697

A.3.2 Naive Matrix Multiplication in Java

Dimension	Time	Gflops
600	0.20273	2.1312
1000	1.319123	1.5232
1400	4.143914	1.3251
1800	18.415449	0.6334
2200	39.342968	0.5414
2600	69.682669	0.5045
3000	115.062472	0.4694

A.3.3 Line-by-line Matrix Multiplication in C++

Dimension	Time	L1_DCM	L2_DCM	Gflops
600	0.10373	27108266	57162689	4.1682
1000	0.48045	125673776	263534205	4.165
1400	1.570402	345982100	705462398	3.4976
1800	3.411581	745296306	1436094201	3.4198
2200	6.273155	2074023721	2557854864	3.3951
2600	10.413975	4412536352	4167976751	3.3756
3000	15.99335	6779756162	6367598825	3.3765
4096	41.170812	17536031042	16132329082	3.3387
6144	138.437	59145286722	54131404398	3.351
8192	328.7155	140028713686	127012178410	3.3453
10240	641.245	273424271178	252406172548	3.3491

A.3.4 Line-by-line Matrix Multiplication in Java

Dimension	Time	Gflops
600	0.118207	3.659
1000	0.527974	3.7896
1400	1.666489	3.2941
1800	3.677329	3.1777
2200	6.714822	3.1729
2600	11.102308	3.1668
3000	17.02255	3.1725

A.3.5 Block Matrix Multiplication in C++

Dimension	BlockSize	Time	L1_DCM	L2_DCM	L3_TCM	Gflops
4096	128	31.1459	9714617150	32876612253	1511819258	4.4133
6144	128	103.62825	32794675815	110509886140	3553499340	4.4763

8192	128	245.18875	77736190952	260777071819	11897357585	4.4846
10240	128	482.5865	151858418653	515000465708	17210079673	4.4501
4096	256	27.26935	9078442948	23191762157	679815131	5.0415
6144	256	90.746875	30637275508	78222562296	1775922383	5.1117
8192	256	395.0055	73018820054	155996722314	101783009326	2.7836
10240	256	419.35075	141832697756	365956126522	9106665976	5.121
4096	512	34.294825	8764910727	19222733832	4262298991	4.0409
6144	512	89.2555	29609086840	66468005243	2339974787	5.197
8192	512	327.06725	70358024589	129655190654	83493579253	3.3618
10240	512	421.87475	136917569032	304458422957	14538189034	5.0905

A.3.6 Parallel Line-by-line Matrix Multiplication V1 in C++

Dimension	Time	L1_DCM	L2_DCM	Gflops	N_Threads	Speedup	Efficiency
600	0.02609	6779416	14371891	16.5601	4	3.975853	0.993963
600	0.018388	4523712	9563773	23.5075	6	5.641179	0.940197
600	0.016991	3892388	8235956	25.4388	7	6.104997	0.872142
600	0.01774	3394520	7180503	26.0532	8	5.847238	0.730905
1000	0.128421	31424573	65960031	15.5802	4	3.741211	0.935303
1000	0.087084	20995060	43931333	22.9683	6	5.517087	0.919514
1000	0.077738	17978786	37739367	25.7315	7	6.180375	0.882911
1000	0.072236	15719515	32943602	27.7337	8	6.651116	0.831389
1400	0.420342	86904279	173941711	13.0571	4	3.73601	0.934003
1400	0.292254	58102892	116577739	18.7794	6	5.373415	0.895569
1400	0.260103	49663608	100108790	21.0996	7	6.037616	0.862517
1400	0.242828	43459207	87563121	22.7307	8	6.467137	0.808392
1800	0.947277	187092896	362591143	12.321	4	3.601461	0.900365
1800	0.678144	124744777	245272075	17.2174	6	5.030762	0.83846

1800	0.610436	107288543	211639904	19.1082	7	5.588761	0.798394
1800	0.569609	93581856	184365190	20.6737	8	5.989338	0.748667
2200	1.746516	517165173	662145917	12.1978	4	3.591811	0.897953
2200	1.249384	345148641	445274363	17.0488	6	5.020998	0.836833
2200	1.134683	296256768	383562517	18.7684	7	5.528553	0.789793
2200	1.04339	258646489	335303262	20.504	8	6.012282	0.751535
2600	2.903005	1099258636	1095593284	12.1107	4	3.587309	0.896827
2600	2.073793	733986855	736432958	16.9572	6	5.021704	0.836951
2600	1.9244	629129292	636268160	18.2789	7	5.411544	0.773078
2600	1.748542	549639713	552421636	20.1765	8	5.955805	0.744476
3000	4.483564	1690015837	1689838990	12.0486	4	3.567106	0.891777
3000	3.234804	1126716862	1138573594	16.7164	6	4.944148	0.824025
3000	3.174967	966719623	983098031	17.0178	7	5.037328	0.719618
3000	2.828935	845058388	850217588	19.2032	8	5.653488	0.706686
4096	11.836123	4407909173	4316033737	11.6171	4	3.478404	0.869601
4096	9.153131	2941377685	2892802714	15.3299	6	4.498003	0.749667
4096	8.242457	2524705201	2504473066	16.68	7	4.994968	0.713567
4096	7.771289	2204401438	2162463956	17.7229	8	5.29781	0.662226
6144	40.295716	14885685910	14695499040	11.5162	4	3.435526	0.858882
6144	29.667017	9929442450	9844024749	15.647	6	4.666361	0.777727
6144	28.2015	8512299426	8467960181	16.4495	7	4.908852	0.701265
6144	29.515132	7433794109	7168623350	15.7671	8	4.690374	0.586297
8192	95.438815	35275294718	34880897302	11.5245	4	3.444254	0.861063

8192	70.641451	23538544588	23336068548	15.5699	6	4.653295	0.775549
8192	67.893367	20175062555	20068707152	16.195	7	4.841644	0.691663
8192	75.42843	17604598574	16761041528	14.6254	8	4.357979	0.544747
10240	186.692967	68882849009	68809383761	11.5073	4	3.434757	0.858689
10240	137.727615	45959857286	46074405426	15.5971	6	4.655893	0.775982
10240	138.811	39365912816	38966486526	15.4773	7	4.619555	0.659936
10240	149.454144	34379532019	33016779615	14.3976	8	4.29058	0.536323

A.3.7 Parallel Line-by-line Matrix Multiplication V2 in C++

Dimension	Time	L1_DCM	L2_DCM	Gflops	N_Threads	Speedup	Efficiency
600	0.113962	10799676	37865461	3.7928	4	0.910216	0.227554
600	0.144588	9825370	38023821	2.9884	6	0.717418	0.11957
600	0.159152	8972229	34763659	2.7144	7	0.651767	0.09311
600	0.179644	9075200	36659137	2.4064	8	0.57742	0.072177
1000	0.377731	44878399	138306310	5.2971	4	1.271937	0.317984
1000	0.437446	34945002	124836502	4.5761	6	1.098307	0.183051
1000	0.525699	32705194	121749854	3.8045	7	0.913926	0.130561
1000	0.603653	31358390	119766863	3.4194	8	0.795904	0.099488
1400	1.077534	106862009	280687083	5.1206	4	1.457404	0.364351
1400	1.15417	85559911	260475621	4.7571	6	1.360633	0.226772
1400	1.29333	77578756	249028287	4.2433	7	1.214231	0.173462
1400	1.536636	72556965	242570149	3.6242	8	1.021974	0.127747
1800	2.627022	223421910	533618004	4.4489	4	1.29865	0.324662
1800	2.860502	168727729	447536781	4.0808	6	1.192651	0.198775
1800	2.98124	153268287	423587083	3.9125	7	1.14435	0.163479
1800	3.137156	140767116	392428817	3.7189	8	1.087476	0.135934

2200	5.084544	378902281	821352480	4.1891	4	1.233769	0.308442
2200	5.328933	290668990	710337365	3.997	6	1.177188	0.196198
2200	5.44191	261354517	664687694	3.9133	7	1.152749	0.164678
2200	5.762697	240682821	616103745	3.6962	8	1.08858	0.136072
2600	8.541682	632014629	1322827411	4.1175	4	1.219195	0.304799
2600	8.866128	461425575	1058804426	3.966	6	1.17458	0.195763
2600	8.94681	414250619	997288893	3.929	7	1.163987	0.166284
2600	9.289744	378071772	915369162	3.7844	8	1.121019	0.140127
3000	13.132577	931816053	1832250343	4.1128	4	1.217838	0.304459
3000	13.616445	685151821	1504226936	3.9672	6	1.174561	0.19576
3000	13.4372	612475285	1399850189	4.0187	7	1.190229	0.170033
3000	13.966899	559369652	1282685570	3.8665	8	1.14509	0.143136
4096	32.454551	2328754266	4360249024	4.2366	4	1.268568	0.317142
4096	32.530889	1676968930	3549417175	4.2265	6	1.265591	0.210932
4096	31.5714	1474226063	3100505092	4.3533	7	1.304054	0.186293
4096	34.443327	1278957702	2631177780	3.9911	8	1.19532	0.149415
6144	106.736239	7838493184	13948381403	4.3461	4	1.297001	0.32425
6144	105.924744	5283688919	9776612417	4.3794	6	1.306937	0.217823
6144	107.017	4538796927	8844620333	4.3344	7	1.293598	0.1848
6144	106.770409	4161767807	8227084359	4.3466	8	1.296586	0.162073
8192	250.677545	20547025129	32486673063	4.3863	4	1.311308	0.327827
8192	251.235354	126104587	231178540	4.3781	6	1.308397	0.218066

		58	29				
8192	248.166	10818973788	17934813750	4.4306	7	1.324579	0.189226
8192	251.780089	9483041671	16384085209	4.3685	8	1.305566	0.163196
10240	485.638133	68713170443	60516777679	4.4221	4	1.320417	0.330104
10240	476.781064	25651633811	42481677602	4.5043	6	1.344946	0.224158
10240	481.183	21273197411	35916095315	4.4629	7	1.332643	0.190378
10240	479.751935	18563483883	30693220656	4.4769	8	1.336618	0.167077