2023/24

Project 1

Grupo 17 – Turma 13

Licenciatura em Engenharia Informática e Computação

Eva Carvalho	up202006379@up.pt
Inês Soares	up202108852@up.pt
Joana Maia	up202108835@up.pt

Índice

1. Problem Description

This project focuses on examining the influence of memory hierarchy on processor performance during the manipulation of substantial data sets. Specifically, we will explore how accessing large amounts of data affects the overall efficiency of the processor. To achieve this, we will utilize the multiplication of two matrices as our computational task, a common operation in various scientific and computational fields.

2. Algorithms Explanation

Our matrix multiplication functions take two parameters, *m_ar* and *m_br*, which represent the dimensions of the square matrices to be multiplied. Firstly, we dynamically allocate memory for each matrix whose size is *m_ar*m_ar* and initialize matrices A and B, the ones to be multiplied.

This is the point at which they start to differ, i.e. when the multiplication algorithms are performed. Three different algorithms were implemented, in both C++ and Java: a. Simple Matrix Multiplication, b. Line Matrix Multiplication and c. Block Matrix Multiplication.

The time taken for the matrix multiplication is calculated as *(Time2 - Time1) / CLOCKS_PER_SEC*, *Time1* and *Time2* being the processor time before and after performing the multiplication algorithms. The last step of these functions is to call the free function to deallocate the memory that was previously allocated for the three matrices (important to prevent memory leaks).

a. Simple Matrix Multiplication

The simple matrix multiplication is performed in the nested *for loop* below, part of our *OnMult* function. A temporary variable temp is used to store the result of each multiplication and addition operation. Each element of the result matrix C is calculated as the sum of the products of the corresponding elements in the rows of A and the columns of B. The result is then stored in the corresponding element of matrix C.

```C/C++
void OnMult(int m_ar, int m_br) {
(...)
     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;
        }
      }
(...)}
```

b. Line Matrix Multiplication

The line matrix multiplication is performed by the following nested *for loop*. Each element of the result matrix C is calculated as the sum of the products of the corresponding elements in the rows of A and the columns of B. The result is then stored in the corresponding element of matrix C.

```
C/C++
void OnMultLine(int m_ar, int m_br){
(...)
     for (int i = 0; i < m_ar; i++) {
        for (int j = 0; j < m_ar; j++) {
           for (int k = 0; k < m_br; k++)
              phc[i * m_ar + k] += pha[i * m_ar + j] * phb[j * m_br + k];
              }
        }
(...)}
```

c. Block Matrix Multiplication

The block matrix multiplication works by having the outer three loops iterate over the blocks of the matrices, while the inner three loops perform the multiplication for each block. Each element of the result matrix C is calculated as the sum of the products of the corresponding elements in the rows of A and the columns of B. The result is then stored in the corresponding element of matrix C.

```
C/C++
void OnMultBlock(int m_ar, int m_br, int bkSize){
(...)
   for (int i = 0; i < m_ar; i += bkSize)
     for (int j = 0; j < m_br; j += bkSize)
       for (int k = 0; k < m_ar; k += bkSize)
         for (int ii = i; ii < min(i + bkSize, m_ar); ii++)
           for (int jj = j; jj < min(j + bkSize, m_br); jj++)
             for (int kk = k; kk < min(k + bkSize, m_ar); kk++)
               c[ii * m_ar + jj] += a[ii * m_ar + kk] * b[kk * m_br + jj];
(...)}
```

3. Performance Metrics

To evaluate the performance of the processor, we compared the results in two different programming languages, C/C++ and Java. For the C/C++ version, we use the Performance API (PAPI) which shows us the execution time and the number of cache misses for L1 and L2. The program was compiled using -O2 as an optimization flag, which increases the performance of the code.

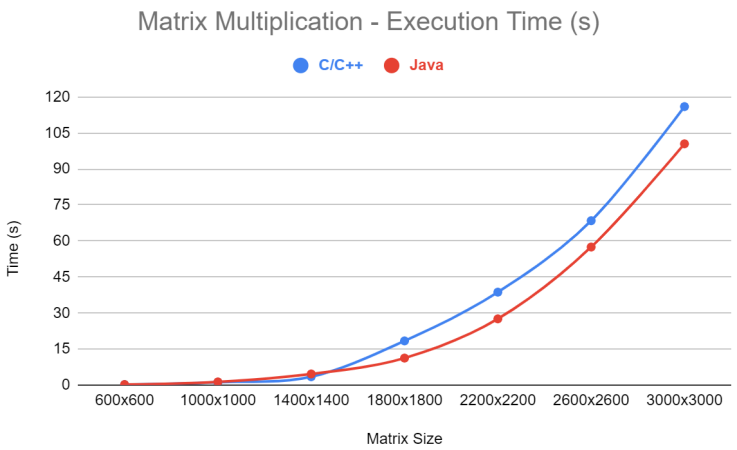
For the data collection, we used two different computers, so it is possible that the results slightly vary. Therefore, we test each measurement 3 times to balance out the differences between each one.

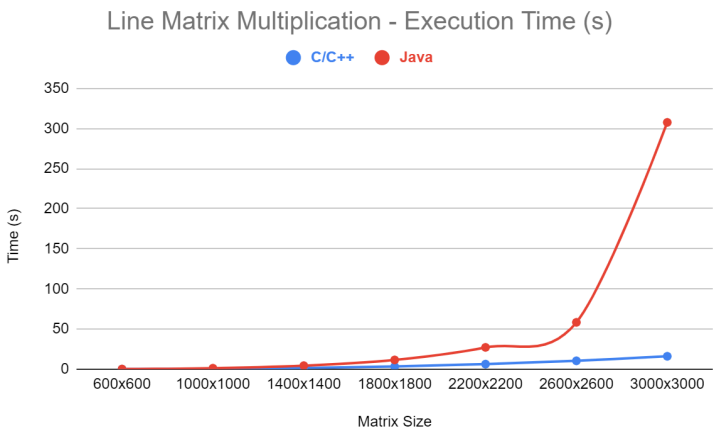
For part 2, we use the formulas below to calculate the Flops, Speed and Efficiency on the parallel versions of the Line Matrix Multiplication. Although we used different computers, for this part a single one was used whose processor had 12 cores.

$$Flops = 2 * \frac{Matrix\ Size^{3}}{CPU\ Time} \qquad Speed = \frac{Time\ Line}{Time\ Parallel} \qquad Efficiency = \frac{Speed}{Nº\ Cores}$$

4. Results and Analysis

4.1. Comparison between the Matrix Multiplication and Line Matrix Multiplication

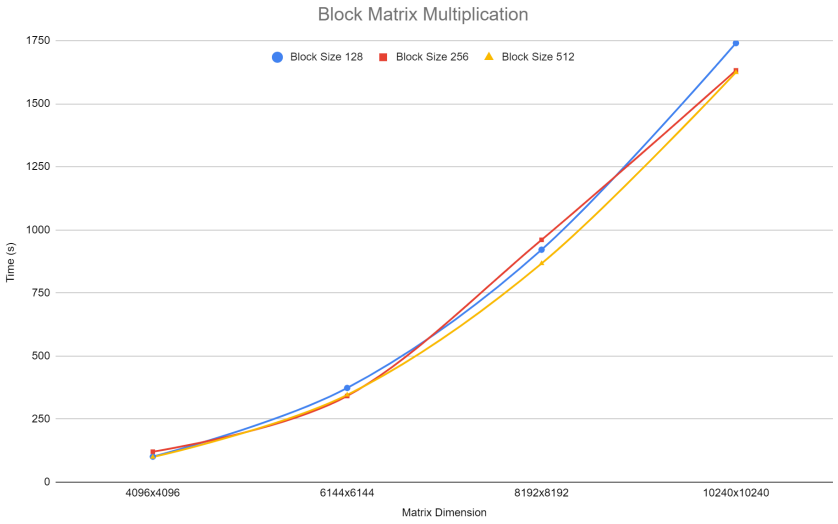




The Matrix Multiplication algorithm is less efficient than the Line Matrix Multiplication algorithm, which shows a huge improvement in the optimization of this last one. Therefore, the number of cache misses was reduced, improving its performance.

As mentioned before, we tested both algorithms written in two different programming languages. For Matrix Multiplication, the algorithm performed better using Java. However, the Line Matrix Multiplication algorithm had a significantly low average execution time in C/C++ (which is not the case for this algorithm using Java).

4.2. Comparison between block sizes on The Block Matrix Multiplication

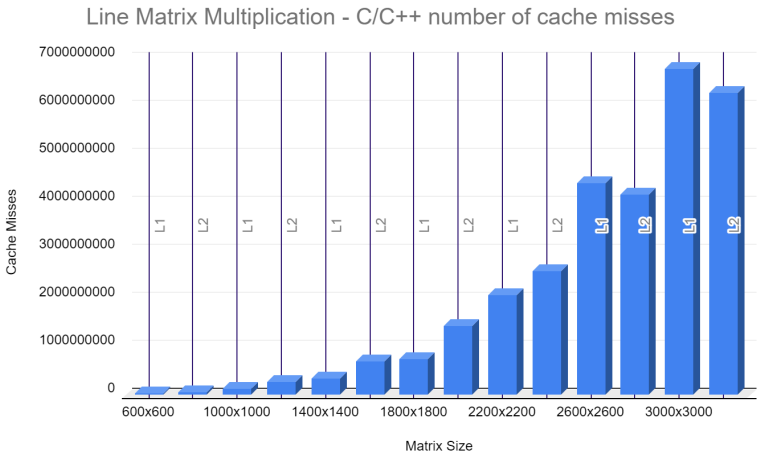


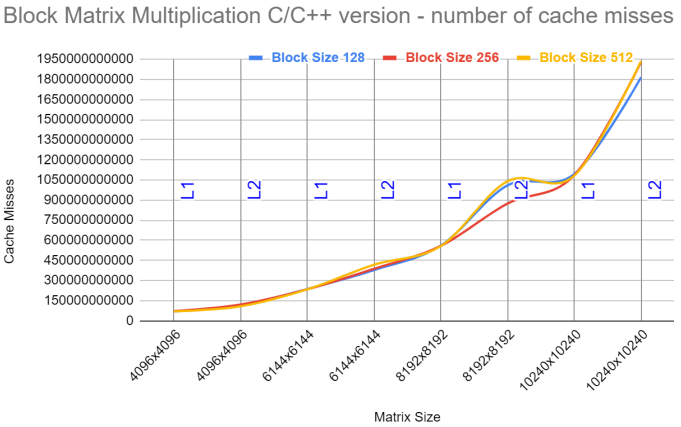
It was expected that for different block sizes, the performance of the same algorithm would differ between them. However, the difference was not considerably noticeable.

The key finding from our comparison analysis indicates that when dealing with significantly large matrix sizes, such as 10240, the performance notably improves with larger block sizes. However, for matrix sizes that are still considerable, like 8192, we observed a different trend. Surprisingly, smaller block sizes, such as 128, outperformed larger block sizes like 256. Notably, when utilizing an exceptionally large block size, such as 512, the performance consistently improved across all matrix sizes.

In summary, our analysis underscores the intricate relationship between matrix size, block size, and performance efficiency. It highlights the necessity of carefully optimizing block sizes based on the specific characteristics of the computation and data sets involved, particularly for scenarios with varying matrix sizes.

4.3. Comparison between cache misses from Line Matrix Multiplication and Block Matrix Multiplication





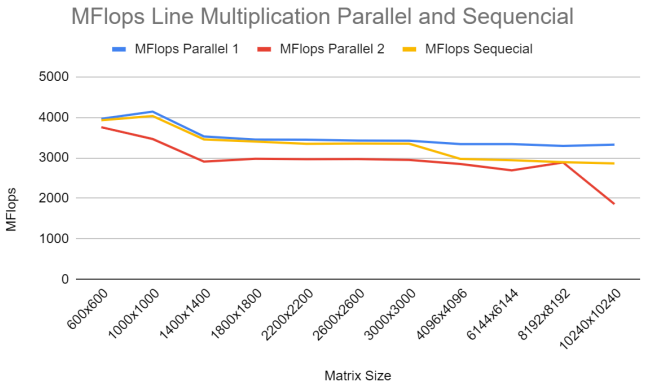
In the Line Matrix Multiplication approach, our observation reveals that for smaller matrix dimensions, the cache misses in L1 and L2 are relatively similar. However, as the matrix size expands, particularly to substantial dimensions, we notice a shift where the L2 cache misses tend to surpass those of L1. Interestingly, upon reaching significantly large matrix sizes, such as 2600, we observe a reversal, where the L1 cache misses become more prominent than L2.

This observation underscores the dynamic behavior of cache utilization in relation to matrix size, highlighting the complex interplay between cache hierarchy and computational workload.

In the Block Matrix Multiplication scenario, our results indicate that for smaller matrix sizes, variation in cache misses across different block sizes is minimal. However, we observe distinct patterns as we scale up to significant matrix sizes, such as 8192. Specifically, for both small and exceptionally large block sizes (128 and 512), the number of cache misses increases noticeably, while it remains relatively stable for a moderate block size like 256. Upon further scaling to a large matrix size like 10240, the cache misses exhibit a linear increase once again.

This finding underscores the importance of selecting an optimal block size relative to the matrix size to effectively manage cache utilization and minimize cache misses, thereby enhancing overall computational efficiency.

4.4 Comparison of Flops between The Line Matrix Multiplication and its Parallel Versions



The Line Multiplication Parallel 2 algorithm exhibits the poorest performance among the compared methods. Conversely, the performances of both the Line Multiplication and the Line Multiplication Parallel 1 approach are nearly identical.

In terms of speed, the Parallel 1 version outperformed the Sequential version, as indicated by speed values exceeding 1. Conversely, the Parallel 2 version exhibited the opposite effect, with speed values lower than 1 when compared to the Sequential version.

5. Conclusions

The development of this project allowed us to study and understand the major effect programming languages and memory hierarchy have in the optimization of algorithms' performance.

While C/C++ allows us, the developers, to manually manage the memory's access and usage, Java abstracts the memory management from the programmer through its automatic systems of memory management. Consequently, C/C++ enables us to optimize memory usage to obtain better performance. On the other hand, this is not the case for Java, since its memory management systems introduce overhead that negatively affects performance. For that reason, C/C++ performed better.

The metrics of our study can be found in the Annexes section.

6. Annexes

Part 1.1 - Matrix Multiplication

A. C/C++ version - execution time (s)

Matrix Dimension	Run 1	Run 2	Run 3	Average
600x600	0.193	0.196	0.197	0.195
1000x1000	1.246	1.111	1.194	1.184
1400x1400	3.449	3.441	3.534	3.475
1800x1800	18.134	18.786	18.264	18.395
2200x2200	38.675	38.620	38.806	38.700
2600x2600	68.517	68.820	68.032	68.456
3000x3000	114.099	116.698	117.135	115.977

B. Java version - execution time (s)

Matrix Dimension	Run 1	Run 2	Run 3	Average
600x600	0.212	0.224	0.242	0.226
1000x1000	1.330	1.228	1.395	1.318
1400x1400	4.814	4.283	4.777	4.625
1800x1800	12.165	9.653	11.913	11.244
2200x2200	27.401	26.909	28.446	27.585
2600x2600	58.858	56.929	56.682	57.490
3000x3000	101.759	100.252	99.410	100.474

C. MFlops

Matrix Dimension	Average C/C++	MFlops C/C++	Average Java	MFlops Java
600x600	0.195	2215,38	0.226	1911,50
1000x1000	1.184	1689,19	1.318	1517,45
1400x1400	3.475	1579,18	4.625	1186,59
1800x1800	18.395	634,09	11.244	1037,35
2200x2200	38.700	550,28	27.585	772,01
2600x2600	68.456	513,50	57.490	611,45
3000x3000	115.977	465,10	100.474	537,45

Part 1.2.1 - Line Matrix Multiplication

A. C/C++ version - execution time (s)

Matrix Dimension	Run 1	Run 2	Run 3	Average
600x600	0.109	0.112	0.110	0.110
1000x1000	0.499	0.491	0.497	0.496
1400x1400	1.570	1.581	1.614	1.588
1800x1800	3.413	3.423	3.449	3.428
2200x2200	6.383	6.323	6.377	6.361
2600x2600	10.498	10.489	10.448	10.478
3000x3000	16.176	16.110	16.088	16.125

B. C/C++ version - number of caches miss

Matrix Dimension	Cache	Run 1	Run 2	Run 3	Average
600x600	L1	27110201	27113236	27110724	27111387
	L2	57906739	56709973	56872349	57163020
1000x1000	L1	125736418	125737919	125824016	125766118
	L2	260155171	260734348	260605500	260498340
1400x1400	L1	346198634	346198867	346346492	346247998
	L2	701397245	700102338	705447620	702315734
1800x1800	L1	745578598	745561381	745285782	745475254
	L2	1433312355	1434414391	1454700610	1440809119
2200x2200	L1	2073604443	2073808316	2073696497	2073703085
	L2	2591819667	2549319815	2592589485	2577909656
2600x2600	L1	4412860736	4412880235	4412865032	4412868668
	L2	4206475449	4145202088	4146254400	4165977312
3000x3000	L1	6780797335	6780634319	6780835415	6780755690
	L2	6273999062	6321787244	6310933844	6302240050

C. Java version - execution time (s)

Matrix Dimension	Run 1	Run 2	Run 3	Average
600x600	0.219	0.228	0.232	0.226
1000x1000	1.203	1.306	1.278	1.262
1400x1400	4.210	4.545	4.609	4.454
1800x1800	11.488	11.631	11.621	11.580
2200x2200	27.567	27.104	27.005	27.225
2600x2600	58.005	58.004	59.233	58.414
3000x3000	102.838	101.919	102.777	307.534

D. MFlops

Matrix Dimension	Average C/C++	MFlops C/C++	Average Java	MFlops Java
600x600	0.110	3927,27	0.226	1911,50
1000x1000	0.496	4032,26	1.262	1584,79
1400x1400	1.588	3455,92	4.454	1232,15
1800x1800	3.428	3402,57	11.580	1007,25
2200x2200	6.361	3347,90	27.225	782,22
2600x2600	10.478	3354,84	58.414	601,77
3000x3000	16.125	3348,84	307.534	175,59

Part 1.2.2 - Line Matrix Multiplication

A. C/C++ version - execution time (s)

Matrix Dimension	Run 1	Run 2	Run 3	Average
4096x4096	46.315	46.252	46.097	46.221
6144x6144	157.547	156.116	158.982	157.548
8192x8192	378.225	371.646	389.475	379.782
10240x10240	758.284	747.746	744.554	750.195

B. C/C++ version - number of caches miss

Matrix Dimension	Cache	Run 1	Run 2	Run 3	Average
4096x4096	L1	17536187443	17544411672	17541223287	17540607467
	L2	15629537486	15676489589	15647468996	15651165357
6144x6144	L1	59031937694	59091627946	59085053836	59069539825
	L2	53923018149	52331926312	52488980468	52914641643
8192x8192	L1	139828779560	139820063955	139984394429	139877745981
	L2	127859550394	131123265545	138182984479	132388600139
10240x10240	L1	273260714731	273033197186	273149117047	273147676321
	L2	265270788542	260007284173	251582494563	258953522426

C. MFlops

Matrix Dimension	Average	MFlops
4096x4096	46.221	2973,52
6144x6144	157.548	2944,22
8192x8192	379.782	2895,11
10240x10240	750.195	2862,57

Part 1.3 - Block Matrix Multiplication

A. C/C++ version - execution time (s)

Block Size	Matrix Dimension	Run 1	Run 2	Run 3	Average
128	4096x4096	91.845	100.504	111.307	101.219
	6144x6144	305.780	327.604	487.091	373.492
	8192x8192	726.218	934.027	1104.086	921.444
	10240x10240	1358.346	1755.824	2107.096	1740.422

256	4096x4096	89.469	134.133	139.098	120.900
	6144x6144	303.458	358.195	363.207	341.62
	8192x8192	948.181	893.134	1040.595	960.637
	10240x10240	1712.947	1605.235	1578.520	1632.234
512	4096x4096	99.559	98.698	100.426	99.561
	6144x6144	376.694	333.918	327.461	346.024
	8192x8192	919.211	839.484	841.486	866.727
	10240x10240	1777.022	1528.932	1567.005	1624.320

B. C/C++ version - number of caches miss

Block Size	Matrix Dimension	Cache	Run 1	Run 2	Run 3	Average
128	4096x4096	L1	70005903349	70064901153	70032142348	7003431561 7
		L2	103815721531	123689234894	119935966113	1158136408 46
	6144x6144	L1	236280489537	2363901708 80	236190606565	2362870889 94
		L2	344587811397	4125698883208	377304822126	3781541722 44
	8192x8192	L1	560087842137	559997003325	560367068436	5601506379 66
		L2	109769047852 8	953477957432	978778954401	1,009982463×10¹²
	10240x10240	L1	109403529085 1	109328591001 1	109349163337 2	1,093604278×10¹²
		L2	200575528672 1	176857528472 9	167025011076 7	1,814860227×10¹²

256	4096x4096	L1	69453824670	73153957947	73706238524	72104673714
		L2	113269732731	124663644745	124677664520	120870347332
	6144x6144	L1	234400205265	234465603874	234387628801	234417812647
		L2	337836784421	414124730517	409895879045	387285797994
	8192x8192	L1	557387916518	556211937751	560546364894	558048739721
		L2	587583629294	101615536823 8	101852819336	874089272289
	10240x10240	L1	1084947530380	1085104347833	1084905262910	1,084985714×10¹²
		L2	1940893334837	1904256952558	194135819008	1,928836369×10¹²
512	4096x4096	L1	70219828410	69191432256	69223050576	69544770414
		L2	73609220453	126176946313	122513662076	107433276281
	6144x6144	L1	235579914289	233530704608	233528013792	234212877563
		L2	408677490212	409184901395	435355802625	417739398077
	8192x8192	L1	555982140669	556268571798	556843521382	556364744616
		L2	102848193216 1	104694348902	104696522037 2	1,040796881×10¹²
	10240x10240	L1	108149349169 0	108096880487 9	108309478949 2	1,081852362×10¹²
		L2	194214584694 8	193924502725 4	192652088791 5	1,935970587×10¹²

C. MFlops

Block Size	Matrix Dimension	Average	MFlops
128	4096x4096	101.219	1357.84
	6144x6144	373.492	1241.94
	8192x8192	921.444	1193.25
	10240x10240	1740.422	1233.89
256	4096x4096	120.900	1136.80
	6144x6144	341.62	1357.81
	8192x8192	960.637	1144.57
	10240x10240	1632.234	1315.67
512	4096x4096	99.561	1380.45
	6144x6144	346.024	1340.53
	8192x8192	866.727	1268.58
	10240x10240	1624.320	1322.08

Part 2.1.1- Line Matrix Multiplication - Parallel 1

A. C/C++ version - execution time (s)

Matrix Dimension	Run 1	Run 2	Run 3	Average
600x600	0.111	0.107	0.109	0.109
1000x1000	0.490	0.474	0.484	0.483
1400x1400	1.542	1.547	1.578	1.556
1800x1800	3.349	3.357	3.428	3.378
2200x2200	6.193	6.147	6.187	6.176
2600x2600	10.456	10.156	10.160	10.257
3000x3000	15.675	15.857	15.789	15.774

4096x4096	40.898	41.522	41.037	41.152
6144x6144	141.782	137.541	137.207	138.843
8192x8192	333.495	333.231	334.382	333.703
10240x10240	646.477	647.810	643.151	645.813

B. MFlops, Speed

Matrix Dimension	Average Parallel	MFlops Parallel	Average Line	Speed	Efficiency
600x600	0.109	3963.30	0.110	1.009	0.084
1000x1000	0.483	4140.79	0.496	1.027	0.086
1400x1400	1.556	3526.99	1.588	1.021	0.085
1800x1800	3.378	3452.93	3.428	1.015	0.085
2200x2200	6.176	3448.19	6.361	1.030	0.086
2600x2600	10.257	3427.12	10.478	1.022	0.085
3000x3000	15.774	3423.35	16.125	1.022	0.085
4096x4096	41.152	3339.79	46.221	1.123	0.094
6144x6144	138.843	3340.87	157.548	1.135	0.095
8192x8192	333.703	3294.88	379.782	1.138	0.095
10240x10240	645.813	3325.24	750.195	1.162	0.097

Part 2.1.2- Line Matrix Multiplication - Parallel 2

A. C/C++ version - execution time (s)

Matrix Dimension	Run 1	Run 2	Run 3	Average
600x600	0.113	0.119	0.113	0.115
1000x1000	0.576	0.564	0.592	0.577
1400x1400	1.800	1.805	2.057	1.887
1800x1800	3.862	4.019	3.872	3.918
2200x2200	7.175	7.269	7.077	7.174

2600x2600	11.778	11.923	11.800	11.834
3000x3000	18.154	18.684	18.083	18.307
4096x4096	50.511	47.792	46.534	48.279
6144x6144	169.788	174.502	172.819	172.370
8192x8192	381.971	380.736	379.979	380.895
10240x10240	749.833	763.446	738.562	750.614

B. MFlops, Speed

Matrix Dimension	Average Parallel	MFlops Parallel	Average Line	Speed	Efficiency
600x600	0.115	3756,52	0.110	0,957	0,078
1000x1000	0.577	3466,20	0.496	0,860	0,072
1400x1400	1.887	2908,32	1.588	0,842	0,070
1800x1800	3.918	2977,03	3.428	0,875	0,073
2200x2200	7.174	2968,50	6.361	0,887	0,074
2600x2600	11.834	2970,42	10.478	0,885	0,074
3000x3000	18.307	2949,69	16.125	0,881	0,073
4096x4096	48.279	2849,76	46.221	0,957	0,080
6144x6144	172.370	2691,05	157.548	0,914	0,076
8192x8192	380.895	2886,65	379.782	0,997	0,083
10240x10240	750.614	1860,97	750.195	0,999	0,083