Report.md 13.7 KB

CPD Project 1

Problem description and algorithms explanation

Problem 1

In this first problem, we were asked to use the traditional matrix multiplication algorithm - multiplying one line of the first matrix by each column of the second one. For that implementation we chose Python, in addition to the algorithm coded in C++ that was provided.

Problem 2

For this problem, the algorithm was different in the sense that elements from the first matrix were "blocked" in every iteration, and multiplied by the corresponding line of the second matrix. This implementation was more efficient when compared to the first one because, in theory, when accessing elements from the matrix, there is a bigger chance that a line is stored in cache and a column is not. That way, the values are retrieved faster (with fewer cache misses) in this second problem.

Problem 3

In this last algorithm, the original matrices should be parted in blocks of smaller matrices, facilitating the calculation and therefore improving efficiency.

Performance metrics

After analysing the metrics provided by PAPI, we decided to use the ones pre-coded (PAPI_L1_DCM and PAPI_L2_DCM) and add metrics that could allow us to calculate the CPI (cycles per instruction) to the tests. For this purpose, we included PAPI_TOT_INS and PAPI_TOT_CYC to the code.

The purpose of DCM metrics is to evaluate the memory efficiency of each algorithm by counting the total number of data cache misses. This value alone is not relevant, but when comparing results for different algorithms, it is possible to arrive to appropriate conclusions.

In the case of CPI, its relevancy lies in the fact that it gives us a perception of how much time the CPU stands idle. This is directly connected to the percentage of cache misses.

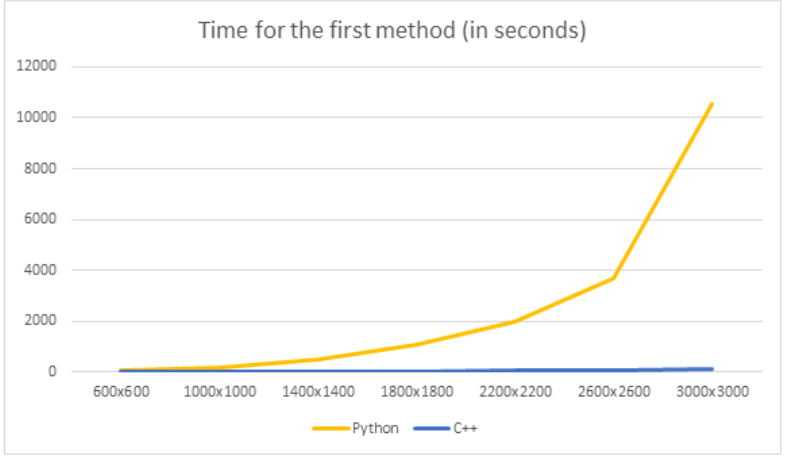
We planned on calculating the **percentage** of data cache misses. Since we already had information concerning the total data cache misses for Level 1 and 2 (PAPI_L1_DCM and PAPI_L2_DCM), we searched for PAPI commands that would allow us to receive information of total data cache accesses. After finding out that there was a useful command for Level 2 but not for Level 1, we tested the compatibility between PAPI_L2_TCA and PAPI_L2_DCM commands, and concluded that that percentage couldn't be caculated because both commands were not supported in the same test.

Results and analysis

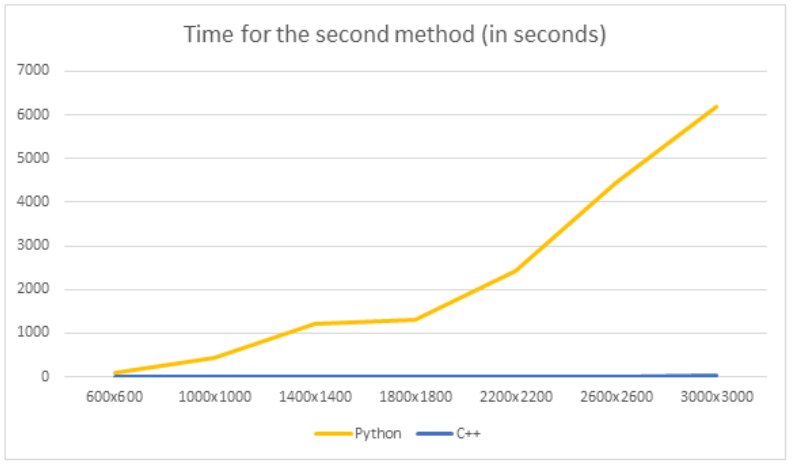
We chose to compare implementations to better demonstrate efficiency.

Implementation in Python

In general, the efficiency of the implementations of the same algorithms were much slower in Python than in C++. One of the reasons for this is due to the fact that Python is an interpreted language which means it needs a higher level of abstraction before turning into executable machine code.



Graphic comparing the execution times of the multiplication of matrices of different sizes in Python and C++



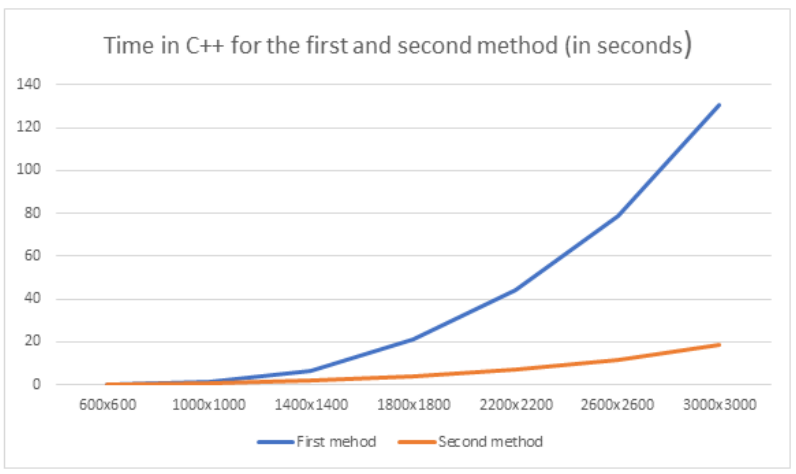
Graphic comparing the execution times of the multiplication of matrices through the lines method of different sizes in Python and C++

We were expecting the second method to be the most efficient one since it's accessing values in the same cache line, but we observed that this was not the case. In general, in the Python implementations, the first method is more efficient than the second one. This is because memory management in Python involves a private heap containing all Python objects and data structures. The management of this private heap is ensured internally by the Python memory manager. The Python memory manager has different components which deal with various dynamic storage management aspects, like sharing, segmentation, preallocation or caching.

grafico de comparação dos algoritmos em python

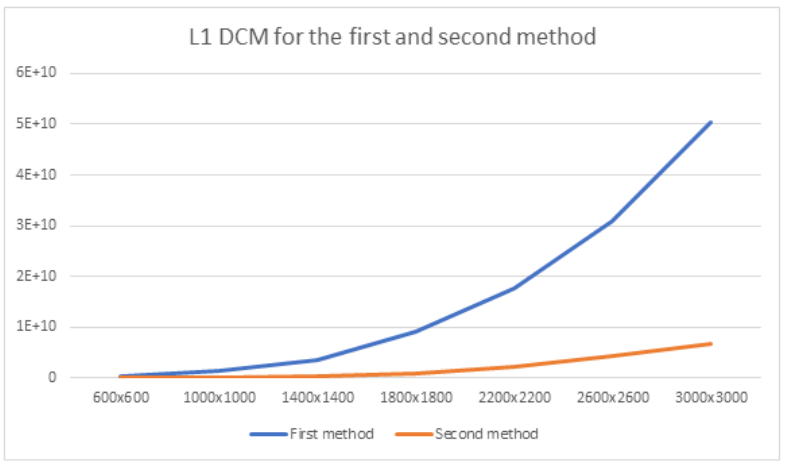
Comparison between first method and second method

When comparing implementations using C++ for the first and second algorithms, we noted that all metrics grow, but with different rates of change (derivatives). The first method had always bigger rates of growth than the second one.

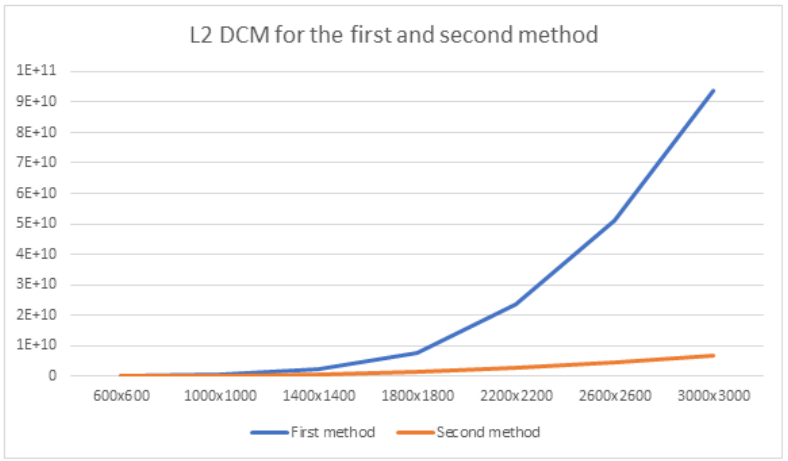


Graphic comparing the execution times of the multiplication of matrices through the first and second method of different sizes in C++

For both cache misses charts (for L1 and L2), the first algorithm has higher values and growth because of the way it is implemented. When storing data in the cache, the memory manager prioritizes values stored close together in-code. That is, a line of a matrix would have a higher chance of being stored in cache than the nth element of every line of that matrix. That is why the second algorithm takes less time to calculate the result of the multiplication and has a smaller chance of missing the data needed for such calculation.

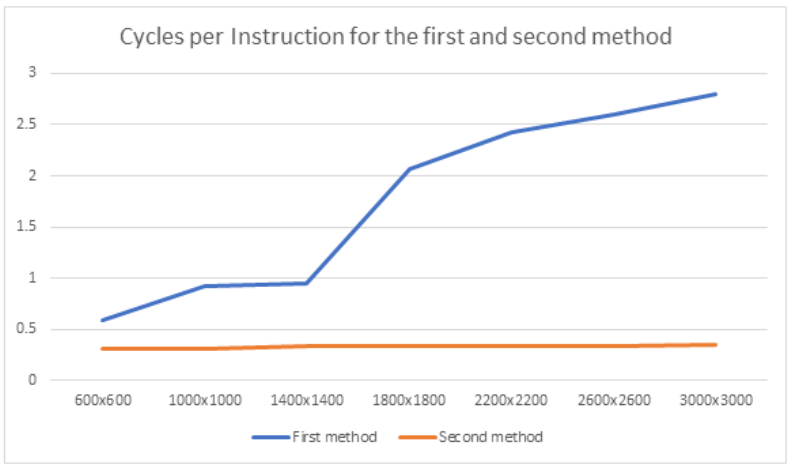


Graphic comparing the data cache misses of L1 when multiplicating matrices through the first and second method of different sizes in C++



Graphic comparing the data cache misses of L2 when multiplicating matrices through the first and second method of different sizes in C++

"Cycles per Instruction" is a metric that allows to observe the amount of time the CPU stands idle when executing the program's instructions. When comparing cycles between the two implementations, it is possible to deduce the inefficiency of the first method. The fact that the first method has higher values for this metric is justified by the amout of times the processor needs to get data from the main memory after cache miss.



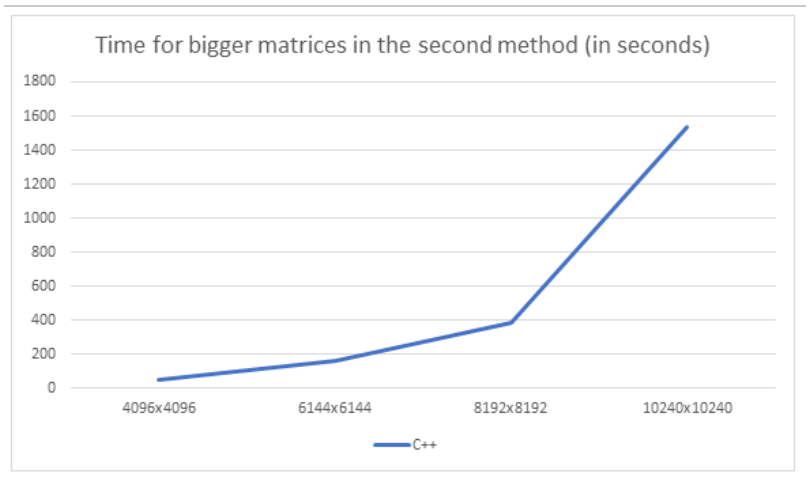
Graphic comparing the cycles per instruction when multiplicating matrices through the first and second method of different sizes in C++

Comparison between the second and third methods

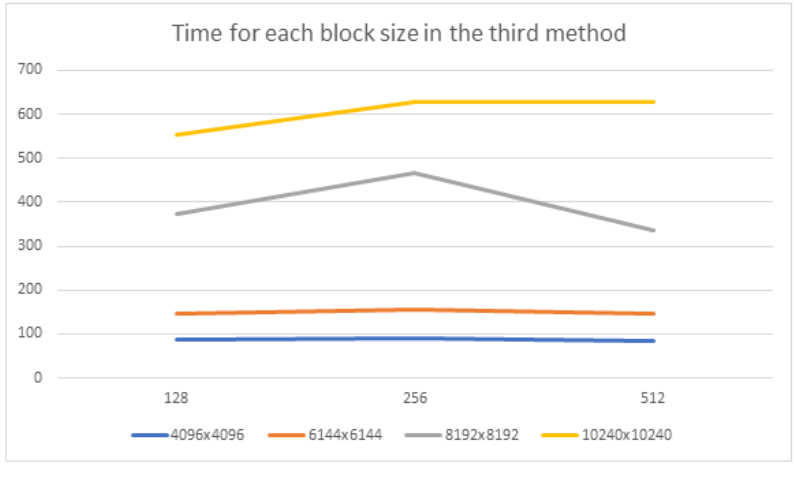
As expected in general the block method is more efficient, this is because by dividing a big matrix into smaller blocks, all the values necessary for each block's multiplication can be stored into cache, lowering the chances of cache misses, and therefore lowering the execution time.

Comparing the execution time of both methods,it's possible to see that the third method is more efficient, culminating in a difference of 985.818 seconds when comparing the multiplication of the 10240x12040 in both methods.

We were not able to identifiy the effect of block sizes in same size matrices in the third method, our tests came to inconclusive results.

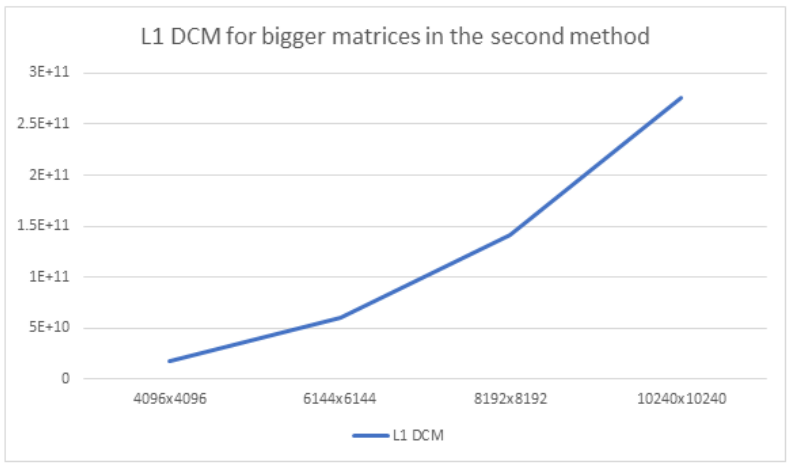


Graphic displaying the execution times when multiplicating matrices through the second method of different sizes in C++

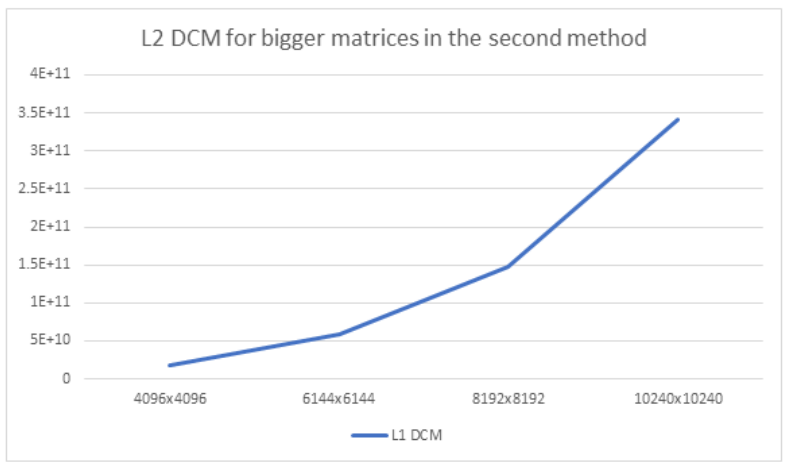


Graphic comparing the execution times when multiplicating matrices through the third method of different sizes and different block sizes in C++

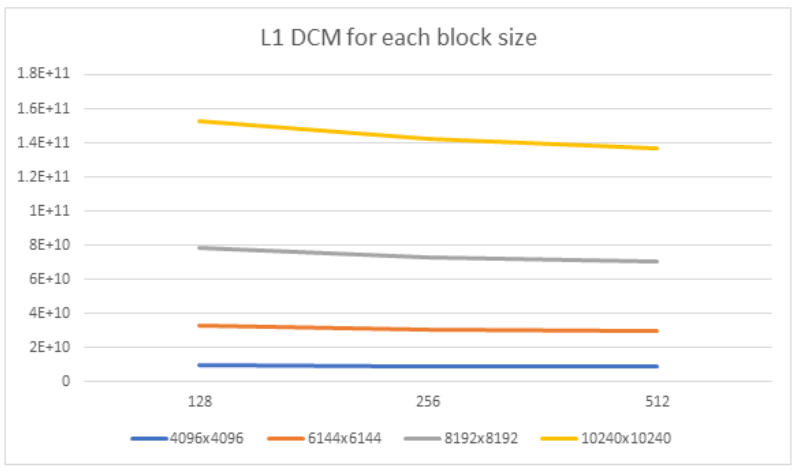
As expected the cache misses are lower in the third method for both the first and second caches,the difference grows, and at its peek, when comparing the misses first cache in the multiplication of 10240x10240 matrices there is a difference of 138441438900.
The differece is expected because for bigger matrices, whole lines don't fit in cache lines, by dividing the matrix into smaller blocks we assure that each one can be stored into cache, making the number of misses a lot smaller.



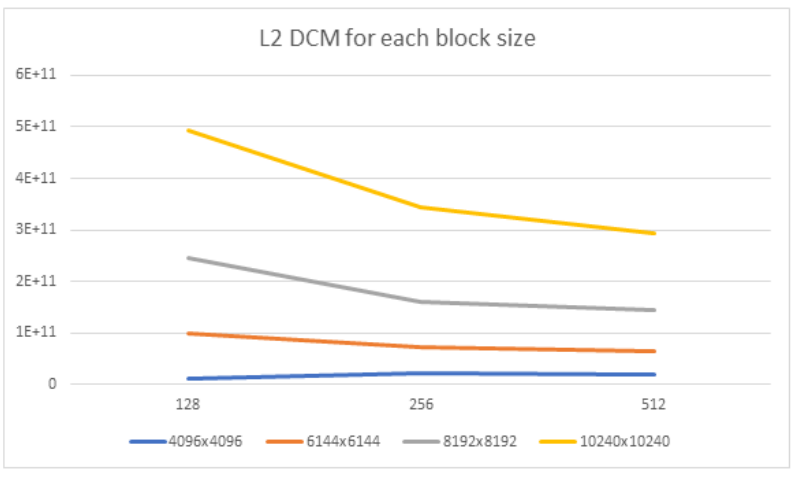
Graphic displaying the number of cache 1 misses when multiplicating matrices through the second method of different sizes in C++



Graphic displaying the number of cache 2 times when multiplicating matrices through the second method of different sizes in C++

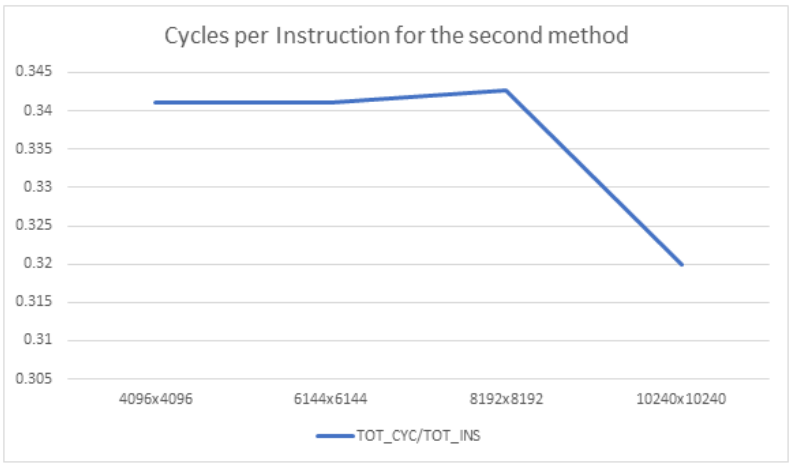


Graphic comparing the number of cache 1 misses when multiplicating matrices through the third method of different sizes and different block sizes in C++

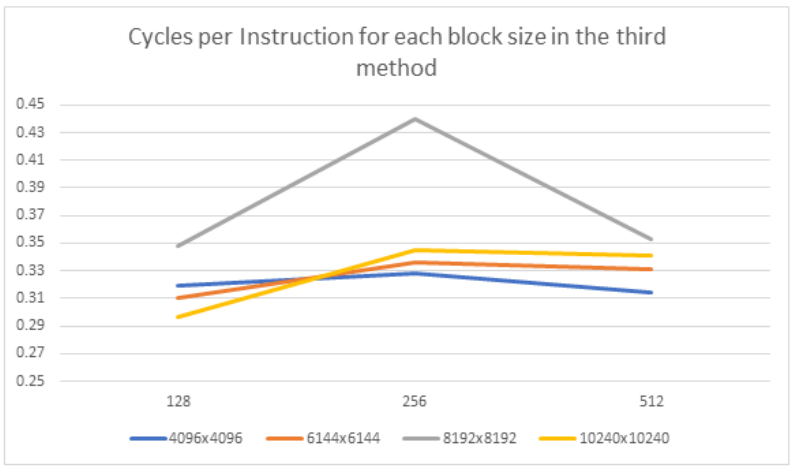


Graphic comparing the number of cache 1 misses when multiplicating matrices through the third method of different sizes and different block sizes in C++

The number of cycles per intruction is similar in both methods and it doesn't vary a lot through matrix size. The same can be said when comparing the third method with different block sizes.



Graphic comparing the number of cache 1 misses when multiplicating matrices through the third method of different sizes and different block sizes in C++



Graphic comparing the number of cache 1 misses when multiplicating matrices through the third method of different sizes and different block sizes in C++

Conclusions

While examining the time spent on each implementation, we concluded that the first algorithm is the least efficient in C++ and the most efficient in Python. In addition, C++ was substantially faster that Python in every single test made.

We could also conclude that one of the main reasons for a higher time of execution (in C++) was the number of cache misses. A significantly lower number of cache misses leads to an also significantly lower time of execution, as seen in the charts, so, this means that all programmers should bear in mind this fact to garantee a fast implementation, specially when dealing with low level languages such as C++.

In contrast to the last conclusion, we could also notice that what is faster in a low level language isn't necessarly faster in a higher level language. As referenced before, Python has several levels of abstraction to deal with memory management, this makes it easier for the programmer to implement the algorithm and creates less bugs but leads to a more unpredictable use of memory and, consequently, makes it harder for the programmer to ensure an efficient memory management.

The elaboration of this assignment helped us to understand the way the cache stores data and the impact of memory management in the efficiency of the code.

Group Members - T08G01

- Carolina Figueira (up201906845)
- Frederico Lopes (up201904580)
- Marta Mariz (up201907020)

Appendix

Results of both implementations of the first method from sizes 600 to 3000

Size	Time Python	Time C++	L1 DCM	L2 DCM	TOT CYC / TOT INS
600x600	32.825	0.247	244581211	40604611	0.585162
1000x1000	176.106	1.626	1224082329	253435997	0.922570
1400x1400	501.886	6.146	3502175317	2157620748	0.949127
1800x1800	1075.895	21.153	9067037714	7766235164	2.069857

Size	Time Python	Time C++	L1 DCM	L2 DCM	TOT CYC / TOT INS
2200x2200	1972.397	44.515	17645600468	23667105892	2.425441
2600x2600	3676.853	78.971	30875674782	51287734723	2.595588
3000x3000	10540.293	130.494	50306730596	93873031998	2.804447

Results of both implementations of the second method from sizes 600 to 3000

Size	Time Python	Time C++	L1 DCM	L2 DCM	TOT CYC / TOT INS
600x600	96.244	0.160	27186228	57230196	0.304
1000x1000	443.657	0.617	125904690	250461478	0.313
1400x1400	1212.661	1.798	346548710	685572254	0.332
1800x1800	1312.105	3.839	747161424	1467384400	0.335
2200x2200	2429.491	7.029	2078429926	2654616524	0.336
2600x2600	4439.133	11.664	4412826640	4337525336	0.338
3000x3000	6186.579	18.499	6779624994	6654314525	0.348

Results of C++ implementation of the second method from sizes 4096 to 10240

Size	Time C++	L1 DCM	L2 DCM	TOT CYC / TOT INS
4096x4096	45.974	17643051018	17241584470	0.341
6144x6144	157.904	59552433660	59184606000	0.341
8192x8192	381.700	141071022550	147439678353	0.3427
10240x10240	1539.626	275407687044	341739839613	0.320

Results of C++ implementation of the third method from sizes 4096 to 10240 with block sizes 128, 256, 512

Matrix Size	Block size	Time	L1 DCM	L2 DCM	TOT CYC / TOT INS
4096x4096	128	86.865	9822455456	9822455456	0.319
4096x4096	256	88.915	9122690340	21743476462	0.3277
4096x4096	512	85.142	8799062551	18905275917	0.3139
6144x6144	128	147.691	33153739599	99407563746	0.3101
6144x6144	256	154.592	30754243470	73101556122	0.3357
6144x6144	512	146.919	29678762167	63011906717	0.331
8192x8192	128	374.136	78557858543	244749071511	0.348
8192x8192	256	467.282	73071510874	160784451040	0.440
8192x8192	512	336.235	70271688872	145035076639	0.353
10240x10240	128	553.808	152969965769	494147986928	0.297
10240x10240	256	629.589	142248950630	343528485118	0.345
10240x10240	512	628.317	136966248144	293690465442	0.341

Information of the computer used for testing

1. L1 Data TLB:

- Page Size: 4 KB
- Number of Entries: 64
- Associativity: 4

Cache Information.

2. L1 Data Cache:

- Total size: 32 KB
- Line size: 64 B
- Number of Lines: 512
- Associativity: 8

3. L1 Instruction Cache:

- Total size: 32 KB
- Line size: 64 B
- Number of Lines: 512
- Associativity: 8

4. L2 Unified Cache:

- Total size: 256 KB
- Line size: 64 B
- Number of Lines: 4096
- Associativity: 4

5. L3 Unified Cache:

- Total size: 8192 KB
- Line size: 64 B
- Number of Lines: 131072
- Associativity: 16