Faculdade de Engenharia da Universidade do Porto

CPD Project 1

Performance evaluation of a single core

Licenciatura em Engenharia Informática e Computação

Turma 5 - Grupo 11

Professor: Carlos Miguel Ferraz Baquero-Moreno

Estudantes:

Alexandre Guimaraes Gomes Correia - up202007042@edu.fe.up.pt

Fábio Araújo de Sá - up202007658@edu.fe.up.pt

João Paulo Moreira Araújo - up2020004293@edu.fe.up.pt

Index

1. Problem Description

In this report, we will study the effect on the processor performance on the memory hierarchy when accessing large amounts of data. For this effect, we used the product of two matrices, which is common in a wide range of applications across various fields, such as computer graphics and machine learning, and is still today a study field due to how computationally expensive it can be to solve them.

2. Algorithms

Even without the use of parallel computing, sequential programs can be improved if we take into account memory manipulation. For this project, we implemented three different algorithms that aim to measure the performance of a single core when exposed to a large amount of data, differing essentially in the way they take advantage of memory allocation. These algorithms are:

1. **Simple Matrix Multiplication** (already given)
2. **Line Matrix Multiplication**
3. **Block Matrix Multiplication**

For **Simple Matrix Multiplication** and **Line Matrix Multiplication**, we were given the task to develop code for C/C++ and a different language, we chose Java due to sharing the same level of abstraction with C/C++, having similar syntax, which helped to translate the code from one language to another, as well as it is considered both interpreted and compiled and is an entirely Oriented Programming Language. These factors may have a great influence on memory access and consequently execution time. For **Block Matrix Multiplication** we only developed code for C/C++.

2.1. Simple Matrix Multiplication

For the default program, we were given a basic C/C++ algorithm that multiplies two matrices, i.e. multiplies one line of the first matrix by each column of the second matrix. The complexity of the algorithm is **O(n^3)**. Here is the pseudo code of the algorithm:

```
Unset
for i=0 to matrix_size:
        for j=0 to matrix_size:
                temp = 0
                for k=0 to matrix_size:
                        temp += pha[i*matrix_size + k] * phb[k*matrix_size + j]
```

```
            phc[i*matrix_size + j] = temp
```

2.2. Line Matrix Multiplication

For this version, we implemented an improved algorithm that multiplies an element from the first matrix by the corresponding line of the second matrix. This algorithm, although having the same complexity **O(n³)**. Here is the pseudo code of the algorithm:

```
Unset
for i=0 to matrix_size:
 for k=0 to matrix_size:
   for j=0 to matrix_size:
     phc[i*matrix_size + j] += pha[i*matrix_size + k] * phb[k*matrix_size + j]
```

2.3. Block Matrix Multiplication

For the block matrix multiplication, we implemented a version of the previous algorithm that splits the matrices into smaller matrices (blocks) that are calculated separately and added up in the end. As well as the last one, it shares the same time complexity of complexity **O(n³)**.

This new approach should improve the access time to the data stored in memory, allowing more data (compared to the amount available on each block) to be stored more often in lower level and faster memory (such as L1 and L2 caches), reducing the time to fetch the data from higher level memories.

```
Unset
for ii=0 to matrix_size, ii += block_size:
 for jj=0 to matrix_size, jj += block_size:
  for kk=0 to matrix_size, kk += block_size:
   for i=0 to ii + block_size:
    for k=0 to kk + block_size:
     for j=0 to jj + block_size:
      phc[i*matrix_size +j] += pha[i*matrix_size + k] * phb[k*matrix_size + j]
```

3. Performance Metrics

To evaluate the performance of the algorithms for the C/C++ versions, we resorted to the Performance API (PAPI), which guarantees access to various measures on the CPU, as well as on the levels of CPU Cache memory used by the process. In addition to the execution time of the algorithm, the number of Floating Points Operations (FLOP) was accounted for, as well as the number of cache misses for both L1 and L2. A cache miss operation implies a considerable overhead in the processing, so the variation of this value is extremely relevant to evaluate the efficiency of the implementation. None of these attributes are derived, that is, they are exact and do not result from an approximation of the API.

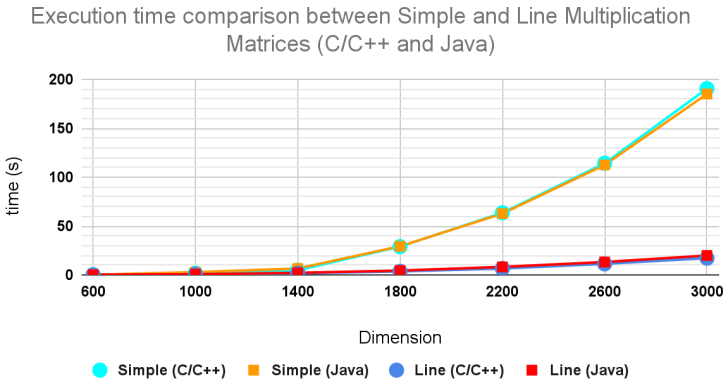
To ensure control and independence of the collected data, the same computer was used for all measurements and these were repeated five times. This computer was running Ubuntu 22.04 and has an i7 9700 single clocked at up to 4.7GHz, with, for each core, an L1 cache of 32KB for instructions and another chunk of 32KB for data, an L2 cache of 256KB, and a shared L3 cache of 12MB. The calculation of the average of the values made it possible to combat the slight differences obtained between measurements, resulting from the state of the computer. In each test performed, a new process was launched, instead of remaining in the loop of the original C/C++ code, in order to allow an independent memory allocation between processes. In the C/C++ version, the program was compiled using the **-O2** optimization flag, which increases the compilation time and performance of the generated code. We also experimented with the **-O3** optimization flag, but the performance gains were negligible.

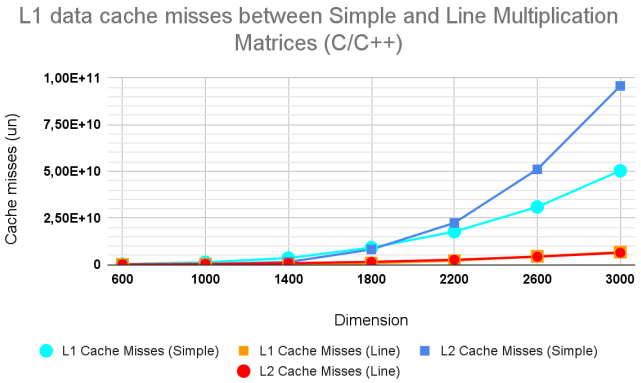
We also compare the execution time between C/C++ and Java versions of the algorithms.

4. Results and analysis

As stated previously, the results are the average of five consecutive tests, and they are presented on the vertical axis of each graph, with the according label beside it. On the horizontal axis, labeled as **Dimension**, is the matrix row/column dimension. The GFLOPS values obtained were the result of dividing the number of FLOP operations by the execution time. Note as well that the full results are available in the Annexes.

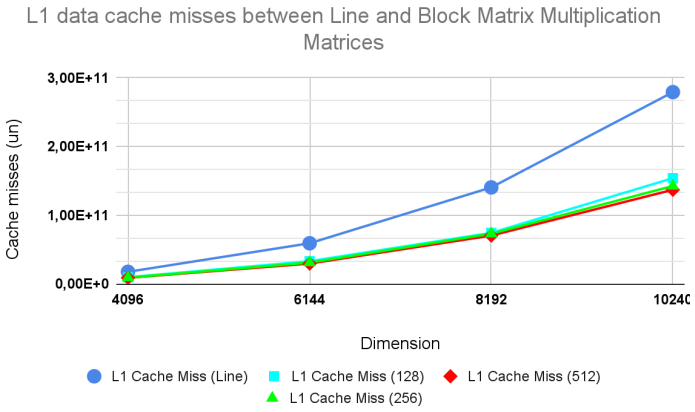
4.1. Execution time and data cache misses comparison between Simple and Line Matrix Multiplication Algorithms

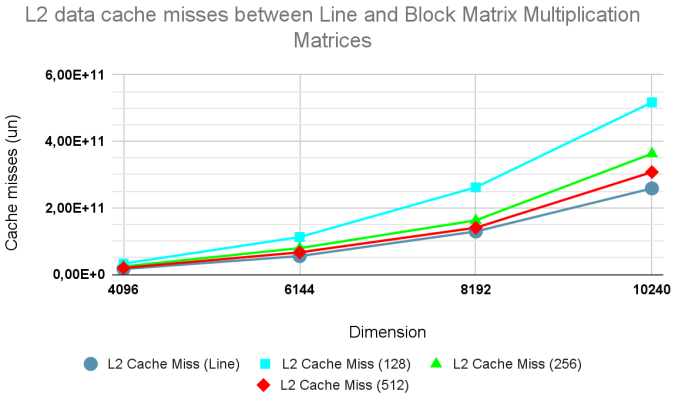




In these two algorithms, the two languages processed the data practically in the same execution time because they are both compiled and follow the same programming paradigm. The second algorithm is more efficient than the previous one, having fewer cache misses and consequent time execution. It uses the result matrix as an accumulator, using as well the intermediate cycle to access the matrix line by line, which guarantees a more frequent availability of the data in lower level memory, consequently lowering the cache misses.

4.2. L1 and L2 Data cache misses between block sizes on a Block Matrix Multiplication Algorithm





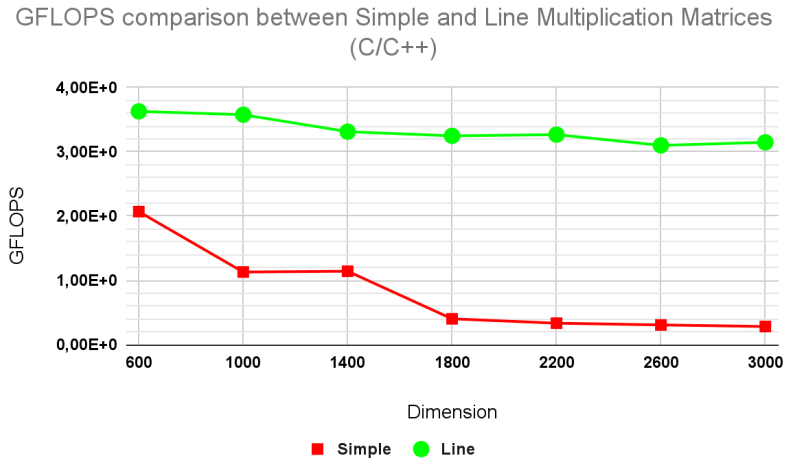
It was possible to observe that the Block Matrix Multiplication Algorithm allowed a lower cache miss rate in relation to the predecessor algorithm at the L1 level. Although the L2 cache misses may be marginally bigger for the Block Algorithm, the time lost fetching data from the L1 cache to higher memory levels ends up being more costly.

Block size did not significantly influence the performance of this cache level, as it is the primary and also the fastest, this may be due to the fact that, in a block of 128x128, housing a double type in each cell of 8 bytes, occupies 128KB in memory, which, for the i7 9700,

equals to the whole L1 cache, as well as most L2 cache. Although, for the bigger block sizes, 256 and 512, which occupy 512KB and 2MB in memory, respectively, they show to be more efficient in terms of L2 cache misses, which relates perfectly with the previous sentence, as there is still non occupied memory in the L2 cache for the 128 block.

All in all, the 512 block results in a lower cache miss rate and consequently, less processing time.

4.3. GFLOPS comparison between Simple and Line Matrix Multiplication Algorithms



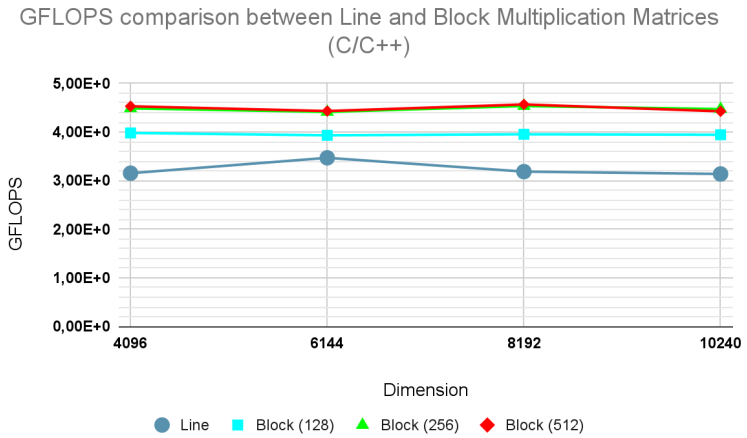
As expected, the number of floating point operations (FLOP) was maintained between algorithms (A1.4. and A2.4.) since the number of operations remained the same.

Although, the higher number of floating points operations per second (FLOPS) on the second algorithm correlates with the lower cache misses both on L1 and L2 caches showed previously.

We can also see a tendency of the Simple algorithm to be slower as the dimension grows. It aligns with our initial thought that the further it increases, the more regularly it may need to access higher level memories to fetch data.

On the other hand, the Line algorithm manages to keep an almost perfect straight line, which correlates to the number of data available in the cache with the sequential FLOPs it does.

4.4. GFLOPS comparison between Line and Block Matrix Multiplication Algorithms



Expectedly, the FLOP was maintained between algorithms (A2.4. and A3.3.) since the number of operations remained the same. We may verify a higher FLOPS on the second algorithm, since, as stated previously, it correlates with the lower cache misses on the lower levels.

As observed, we may point out a difference inside the Block algorithm: the 128x128 block does not fully use the L2 cache for its FLOP, unlike the 256x256 and 512x512 blocks. For this reason, it misses out on maxing the number of operations it could have done.

We may also add that increasing the block size would be worthless and most likely costly (e.g. 1024x1024) for this CPU since it would be more dependent on the L3 cache, which is also shared between the system and may require access to the system RAM. Also, decreasing the block size (e.g. 64x64) would also be slower since it is not using most of the lower level memory, needing to fetch up data to higher memory more often than the other cases.

5. Conclusions

In conclusion, the realization of this work allowed us to have a deeper knowledge of the importance of memory management to improve program efficiency. Even without the use of parallel computing, sequential programs can be further improved if techniques are taken into consideration to manipulate memory to its favor, having in mind the hardware that is going to run it.

The improvements in execution time between algorithms demonstrate how critical it is to plan the code implementation to reuse as much lower level memories (e.g. L1 and L2 caches) to not waste essential time in higher level and latency ones.

References

Intel. **Intel® Core™ i7-9700 Processor**. 2019. Accessed 28 February 2023. <https://www.intel.com/content/www/us/en/products/sku/191792/intel-core-i79700-processor-12m-cache-up-to-4-70-ghz/specifications.html>

Terpstra, D., Jagode, H., You, H., Dongarra, J. "Collecting Performance Data with PAPI-C", Tools for High Performance Computing 2009, Springer Berlin / Heidelberg, 3rd Parallel Tools Workshop, Dresden, Germany, pp. 157-173, 2010.

University of Tennessee. **Papi Wiki**. 2022. Accessed 03 February 2023: <https://bitbucket.org/icl/papi/wiki/Home>

Annexes

A1. Default Matrix Multiplication

A1.1. C/C++ version - execution time (s)

Matrix Dimension	Run 1	Run 2	Run 3	Run 4	Run 5	Average
600x600	0.219	0.203	0.203	0.219	0.203	**0.209**
1000x1000	1.750	1.750	1.828	1.688	1.828	**1.769**
1400x1400	4.781	4.797	4.781	4.766	4.875	**4.800**
1800x1800	29.234	28.969	28.625	28.781	29.188	**28.959**
2200x2200	63.000	63.969	63.500	63.734	63.562	**63.553**
2600x2600	115.156	113.516	114.344	113.984	113.953	**114.191**
3000x3000	189.670	189.343	191.250	191.795	190.625	**190.537**

A1.2. Java version - execution time (s)

Matrix Dimension	Run 1	Run 2	Run 3	Run 4	Run 5	Average
600x600	0.238	0.260	0.230	0.265	0.284	**0.255**
1000x1000	2.526	2.916	2.521	3.171	3.184	**2.864**
1400x1400	5.782	6.055	6.131	7.667	7.098	**6.547**
1800x1800	28.361	28.925	29.087	29.447	30.681	**29.300**
2200x2200	63.604	62.385	62.468	62.125	63.018	**62.720**
2600x2600	112.086	114.235	112.350	112.173	111.864	**112.542**
3000x3000	189.260	188.260	186.388	180.568	180.892	**185.074**

A1.3. C/C++ version - number of caches miss

Matrix Dimension	Cache	Run 1	Run 2	Run 3	Run 4	Run 5	Average
	L1	2447756	2447675	2447179	2447704	2447185	**244750000**

600x600		19	77	01	00	04	
	L2	40224353	38782507	38817201	39811666	39561091	**39439364**
1000x1000	L1	1223975445	1220871645	1225389956	1227778553	1216262163	**1222855552**
	L2	315200971	298677212	304857772	290098338	268415479	**295449954**
1400x1400	L1	3417902801	3424984506	3511447807	3512796100	3521154517	**3477657146**
	L2	1226597863	1289720345	1310916055	1618551064	1456064823	**1380370030**
1800x1800	L1	9063255949	9086169518	9083764663	9077599632	9096003728	**9081358698**
	L2	8731386580	8160187817	8464943292	7444470944	7707833167	**8101764360**
2200x2200	L1	17653547712	17635508809	17634414614	17630533642	17663052177	**17643411391**
	L2	24064464720	17286435118	23362421479	23251721227	23629954475	**22318999404**
2600x2600	L1	30877692256	30877657516	30882244801	30877636624	30883581624	**30879762564**
	L2	50586430012	51113592886	50958783118	51167105763	51436811410	**5105254638**
3000x3000	L1	50305012093	50304896777	50296356782	50308670760	50297956583	**50302578599**
	L2	95664255369	95681929702	95553848706	96463790233	95355456769	**95743856156**

A1.4. Average data collected throughout the algorithm 1

Matrix Dimension	C/C++ Time (s)	Java Time (s)	L1 miss	L2 miss	FLOP
600x600	0.209	0.255	244750000	39439364	432000000
1000x1000	1.769	2.864	1222855552	295449954	2000000000
1400x1400	4.800	6.547	3477657146	1380370030	5488000000

1800x1800	28.959	29.300	9081358698	8101764360	11664000000
2200x2200	63.553	62.720	17643411391	22318999404	21296000000
2600x2600	114.191	112.542	30879762564	51052544638	35152000000
3000x3000	190.537	185.074	50302578599	95743856156	54000000000

A2. Optimized Matrix Multiplication

A2.1. C/C++ version - execution time (s)

Matrix Dimension	Run 1	Run 2	Run 3	Run 4	Run 5	Average
600x600	0.109	0.125	0.125	0.109	0.125	**0.119**
1000x1000	0.562	0.562	0.562	0.562	0.547	**0.559**
1400x1400	1.703	1.641	1.625	1.641	1.672	**1.656**
1800x1800	3.469	3.656	3.734	3.484	3.609	**3.590**
2200x2200	6.438	6.391	6.609	6.406	6.734	**6.516**
2600x2600	10.906	10.672	10.812	11.547	12.750	**11.337**
3000x3000	17.422	16.844	18.141	16.453	16.891	**17.150**
4096x4096	41.688	43.750	44.500	44.438	43.562	**43.588**
6144x6144	143.125	138.125	141.750	141.312	104.125	**133.687**
8192x8192	349.885	355.650	339.500	342.110	338.375	**345.104**
10240x10240	687.969	689.062	683.438	680.156	681.562	**684.437**

A2.2. Java version - execution time (s)

Matrix Dimension	Run 1	Run 2	Run 3	Run 4	Run 5	Average
600x600	0.143	0.148	0.164	0.147	0.150	**0.150**
1000x1000	0.699	0.688	0.723	0.744	0.760	**0.723**
1400x1400	2.022	2.112	2.233	2.216	2.310	**2.179**
1800x1800	4.183	4.334	4.348	4.845	4.626	**4.467**
2200x2200	7.620	8.273	8.610	7.761	8.878	**8.228**

2600x2600	12.717	13.765	13.582	12.796	13.180	**13.208**
3000x3000	19.080	19.443	19.542	20.981	20.276	**19.864**

A2.3. C/C++ version - number of caches miss

Matrix Dimension	Cache	Run 1	Run 2	Run 3	Run 4	Run 5	Average
600x600	L1	27109603	27108713	27110919	27109231	27109363	**27109566**
	L2	58155162	56864780	56884992	56879147	57409090	**57238634**
1000x1000	L1	125732759	125737111	125739574	125734328	125733330	**125735420**
	L2	264436236	268037490	267649347	268027454	267613943	**267152894**
1400x1400	L1	346086761	346092881	346103766	346095138	346089668	**346093643**
	L2	708030675	716373998	709857163	700765068	707007452	**708406871**
1800x1800	L1	745205873	745404054	745230776	745322872	745354753	**745303666**
	L2	1457417580	1433274106	1457172060	1458426766	1442764500	**1449811002**
2200x2200	L1	2073669941	2073748162	2074091525	2073822260	2073351084	**2073736594**
	L2	2560079625	2560294563	2582654606	2564470728	2574617895	**2568423483**
2600x2600	L1	4412939596	4412935671	4412942046	4412856604	4412887634	**4412912310**
	L2	4200885417	4199670595	4197422076	4224339341	4225412250	**4209545936**
3000x3000	L1	6780487856	6780469409	6780839915	6780494340	625576118	**6675610540**
	L2	6396653325	6418916564	6289394350	6403284593	625576118	**6352802002**

A2.4. Average data collected throughout the algorithm 2

Matrix Dimension	C/C++ Time (s)	Java Time (s)	L1 miss	L2 miss	FLOP
600x600	0.119	0.150	27109566	57238634	432000000
1000x1000	0.559	0.723	125735420	267152894	2000000000
1400x1400	1.656	2.179	346093643	708406871	5488000000
1800x1800	3.590	4.467	745303666	1449811002	11664000000
2200x2200	6.516	8.228	2073736594	2568423483	21296000000
2600x2600	11.337	13.208	4412912310	4209545936	35152000000
3000x3000	17.150	19.864	6675610540	6352802002	54000000000
4096x4096	43.588	---	17547738310	16250786429	137438953472
6144x6144	133.687	---	59148276307	55059490119	463856467968
8192x8192	345.104	---	140512705024	129005422329	1099511627777
10240x10240	684.437	---	279326040920	258549787781	2147483648000

A3. Block Matrix Multiplication

A3.1. C/C++ version - execution time

Matrix Dimension	Block size	Run 1	Run 2	Run 3	Run 4	Run 5	Average
4096x4096	128	34.562	34.781	34.266	34.594	34.594	**34.519**
	256	30.922	31.078	30.547	30.539	30.219	**30.625**
	512	30.312	30.125	30.359	29.672	31.219	**30.337**
6144x6144	128	114.844	118.875	119.250	118.219	118.500	**117.938**
	256	102.000	105.750	105.375	106.781	105.469	**105.075**
	512	99.094	103.875	102.938	109.781	107.531	**104.644**
	128	278.500	280.125	277.375	276.750	277.500	**278.050**

8192x8192	256	240.125	240.500	240.500	240.500	249.375	**242.400**
	512	236.875	237.250	240.125	239.625	249.500	**240.675**
10240x10240	128	534.375	545.312	547.656	557.500	537.344	**544.438**
	256	492.344	474.688	471.250	474.688	487.500	**480.094**
	512	480.781	475.000	472.031	488.281	509.375	**485.094**

A3.2. C/C++ version - number of caches miss

Matrix Dimension	Block size	Cache	Run 1	Run 2	Run 3	Run 4	Run 5	Average
4096x4096	128	L1	9602477766	9685232577	9625451365	9655222557	9614568846	**9636590622**
		L2	32472457550	33026910001	32056155512	3292691551 3	3156516852 5	**32409521420**
	256	L1	9076616398	9092896372	9102449631	8972516302	9122856332	**9073467007**
		L2	23844195600	23524816214	23449482114	2412481620 4	2405401551 4	**23799465129**
	512	L1	8789278412	8789208589	8824568480	8686465449	8883242460	**8794552678**
		L2	19332148561	19102201137	20092356133	19202201137	19368206116	**19419422617**
6144x6144	128	L1	32991961446	33000478471	32913257021	3281240040 0	3350342844 0	**33044305156**
		L2	113279566634	111495204740	113503444351	11023452452 2	11258522475 6	**112219593001**
	256	L1	30761534109	30859154074	31052452344	31192496020	3095937273 9	**30965001857**
		L2	79995675530	78688695121	78723423423	7859372930 3	7888302413 4	**78976909502**

	512	L1	2964381 9394	296396 90854	297218 34702	29028 37823 9	29722 39709 1	**29551224 056**
		L2	6680212 1759	662960 87917	662193 78921	66028 33283 8	65938 72387 2	**66256929 061**
8192x8192	128	L1	7509402 3906	743132 61073	739828 37283	74002 98378 2	74290 29384 4	**74336679 978**
		L2	2639530 17747	261547 064845	260382 732834	26292 39827 31	26023 89239 23	**26180914 4416**
	256	L1	7325556 2967	722606 74496	731398 48384	73083 87483 2	73340 92130 3	**73016176 396**
		L2	1607042 48554	170497 712988	161937 493274	15802 39329 34	16138 39237 83	**16250946 2307**
	512	L1	7043089 7031	702733 82346	703938 28344	70493 79202 8	70192 78328 0	**70356936 606**
		L2	1452057 21767	136440 297501	143392 394548	14139 27847 28	13493 73929 34	**14027371 8296**
10240x10240	128	L1	1554275 08473	153788 789890	153891 258318	15325 94438 52	15374 24316 87	**15402188 6444**
		L2	5195688 15875	524775 893450	514529 006350	51088 72862 86	51634 02713 95	**51722025 4671**
	256	L1	1424175 23622	142727 308670	142386 411346	14308 89321 88	14240 78952 35	**14260561 4212**
		L2	3606164 26123	359881 828658	364161 953304	36524 55628 90	36500 94556 32	**36298304 5321**
	512	L1	1370238 15865	137017 084554	137043 440380	13715 78221 32	13717 93614 52	**13708430 4877**
		L2	3088436	307515	306048	30767	30789	**30759722**

			85088	787299	715266	8324560	9612963	**5035**

A3.3. Average data collected throughout the algorithm 3

Matrix Dimension	Block	Time (s)	L1 miss	L2 miss	FLOP
4096x4096	128	34.519	9636590622	32409521420	137438953472
	256	30.625	9073467007	23799465129	
	512	30.337	8794552678	19419422617	
6144x6144	128	117.938	33044305156	112219593001	463856467968
	256	105.075	30965001857	78976909502	
	512	104.644	29551224056	66256929061	
8192x8192	128	278.050	74336679978	261809144416	1099511627776
	256	242.400	73016176396	162509462307	
	512	240.675	70356936606	140273718296	
10240x10240	128	544.438	154021886444	517220254671	2147483648000
	256	480.094	142605614212	362983045321	
	512	485.094	137084304877	307597225035	