Performance Evaluation of a Single Core

Report of the first project of CPD

Class 9 Group 5
Maria José Valente da Silva Carneiro (up201907726)
Rodrigo Tuna de Andrade (up201904967)

March 2022

Contents

List of Figures

1 Problem Description

As data increases, program efficiency must be consistent in order to guarantee it's high performance and scalability. From the data access pattern to the algorithm design, multiple factors influence a program's performance, so it's pertinent to observe and interpret their effects.

Therefore this project explores the effect on the processors' performance of the memory hierarchy when accessing large amounts of data. In this context, we studied and collected relevant performance indicators of the multiplication of two matrices with large dimensions, in order to analyse and better understand the processors' behaviour.

Various versions of the matrix multiplication algorithm were implemented in two different programming languages - `C++` and `JAVA`, so that both algorithmic and memory access factors could be analysed:

1. Multiplying one line of the first matrix by each column of the second matrix.
2. Multiplying an element from the first matrix by the correspondent line of the second matrix.
3. Multiplying using a block oriented algorithm that divides the matrices in blocks and uses the same sequence of computation as in 2 (only in `C++`).

All the metrics collected in `C++` were done with the aid of `PAPI`, a performance `API` that enables the analysis of the relation between software performance and processor events.

2 Algorithms

The different matrix multiplication algorithms allow us to observe and evaluate the execution of the processor and the way it behaves using different programming techniques and optimizations.

2.1 Naive Multiplication

The *Naive Multiplication Algorithm* does the classic product of matrices: one line of the first matrix is multiplied by each column of the second matrix. It has a time complexity of $O(n^3)$, assuming the product is between square matrices with the same dimensions.

```
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;
    }
}
```

2.2 Line Multiplication

The *Line Multiplication Algorithm* follows the same logic as the previous one but instead of multiplying one line of the first matrix by each column of the second, it multiplies an element of the first matrix by the correspondent line of the second matrix. It has a time complexity of $O(n^3)$, assuming the product is between square matrices with the same dimensions.

```
for(i=0; i<m_ar; i++){
    for( k=0; k<m_ar; k++){
      for( j=0; j<m_br; j++){
          phc[i*m_ar+j] += pha[i*m_ar+k] * phb[k*m_br+j];
    }
  }
}
```

2.3 Block Multiplication

The *Block Multiplication Algorithm* divides each matrix in square blocks of the same dimensions and multiplies each smaller matrix of the first matrix by the correspondent line of smaller matrices of the second matrix, that is, it applies the Line Multiplication Algorithm to smaller blocks of matrices and sums the results. It is useful for calculations with limited memory capacity since each block is stored in auxiliary memory and their product has manageable dimensions. It has a time complexity of $O(n^3)$, assuming the product is between square matrices with the same dimensions.

```
for(bi = 0; bi < m_ar; bi += bkSize){
    for(bk = 0; bk < m_ar; bk += bkSize){
        for(bj = 0; bj < m_br; bj += bkSize){
            for(i=0; i<bkSize; i++){
                for( k=0; k<bkSize; k++){
                    for( j=0; j<bkSize; j++){
                        phc[(bi + i)*m_ar+ (bj + j)] += pha[(bi + i)*m_ar+(bk + k)
    ] * phb[(bk + k)*m_br+(bj + j)];
                    }
                }
            }
        }
    }
}
```

3 Performance Metrics

The performance metrics chosen for the study are the execution time and data cache misses (both from L1 and L2 caches). To study the evolution of performance, the theoretical number of floating point operations (`FLOP`) were calculated and divided by the execution time to obtain the `FLOP/s`, using the following formula:

$$\texttt{FLOPS} = 2\frac{3^{size}}{time}$$

As there are 2 Floating Point Operations inside 3 nested loops, which will allow to compare the floating point efficiency of the different algorithms.

Data cache misses were measured using `PAPI` to study the correlation between memory access and the execution time.

4 Results and Analysis

All the measurements were tested on a Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz processor with 8 cores, and 3 values were taken for each test matrix size, in order to give more accurate results. The total CPU cache size is, for L1, 128 `KiB` and, for L2, 1 `MiB`.

The results collected and graphed in the following sections can be found in our group's repository in the `data.md` file as well as annexed in the final section of the report in annex A. The following graphs are also in our group's repository in the folder graphs.

4.1 Line Multiplication vs Naive Multiplication

In order to compare the performance of the algorithms *Naive Multiplication* and *Line Multiplication*, the algorithms were firstly implemented in `C++` and ran for different sized squared matrices with sizes `600x600`, `1000x1000`, `1400x1400`, `1800x1800`, `2200x2200`, `2600x2600` and `3000x3000`. For each algorithm and each matrix size, the values were measured three times and then averaged in order to account for variability in results.

In the following figures a comparison between the two algorithms in terms of execution time and `GFLOP's` can be seen.

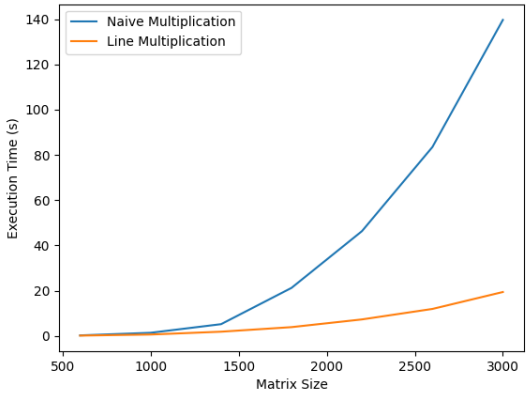

Figure 1: Execution time of Line and Naive Multiplication (`C++`)

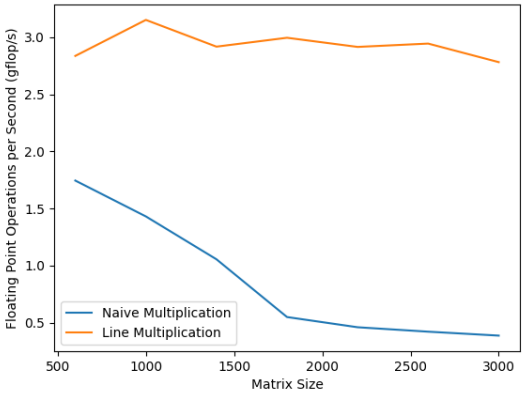

Figure 2: `GFLOPS` of Line and Naive Multiplication (`C++`)

From the results it can be concluded, that *Line Multiplication* outperforms *Naive Multiplication* given its execution is smaller and increases at a slower rate than its counterpart. The `GFLOP's` are greater in the *Line Multiplication* which means that its floating point efficiency is greater as the CPU disposes of a greater share of its time to floating point operations.

In order to understand why it exists such a difference of performance between the two algorithms, which are similar and have the same time complexity, the Data Cache Misses were measured for both L1 and L2 and the results are given by the following graphs:

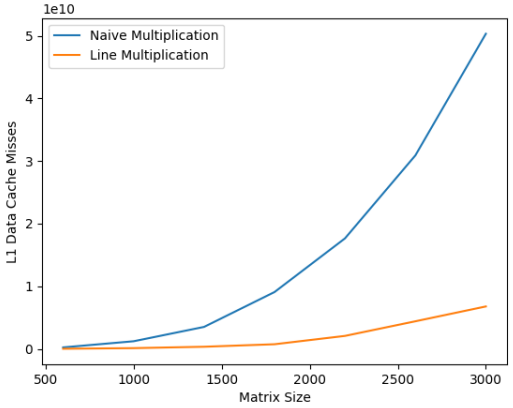

Figure 3: L1 Data Cache Misses of Line and Naive Multiplication (`C++`)

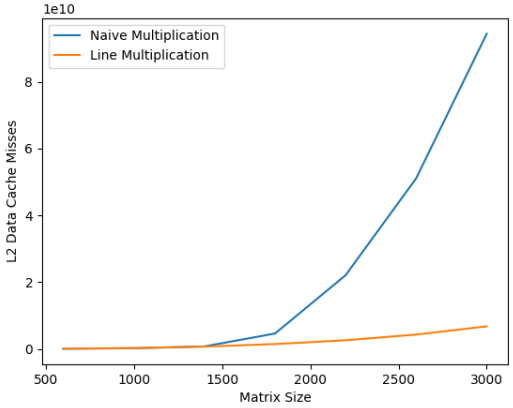

Figure 4: L2 Data Cache Misses of Line and Naive Multiplication (`C++`)

The results indicate that the amount of Data Cache Misses is greater and increases at a greater rate in the *Naive* implementation which will affect performance as accessing data that is not in cache is slower than accessing cached data. The reason why *Line Multiplication* has lower Data Cache Misses is because data is stored in cache in blocks and not individually, and as matrices are stored by aggregating lines, when a value is accessed the whole line is stored in cache memory. In the *Naive Multiplication* each value of the result matrix (C) is obtained by multiplying a row of the first matrix (A) with a column of the second matrix (B), and it is in accessing the values for the column of B that the cache misses occur as new blocks of data are being constantly stored in cache because the different values of the column of B are in different data blocks. On the contrary, the *Line Multiplication* does not calculate the value of C immediately but rather alters it successively. The algorithm takes a row of A and row of B and alters the row of C that is affected by the previous; by doing the operations row oriented, it takes advantage of the fact that the data is stored in blocks and so, when a value is stored, the values needed for the following operations are also stored which makes Data Cache Misses less common.

4.2 Results on a Different Language

A implementation in a different language was made for the sake of understanding that the results obtained in `C++` are not related to the language (eg. compiler optimization) and are true for other ones. For this the two algorithms were implemented in `JAVA` and the execution time and `GFLOP's` were measured for matrices of the same size as the ones used for the `C++` implementation. The results obtained are the following:

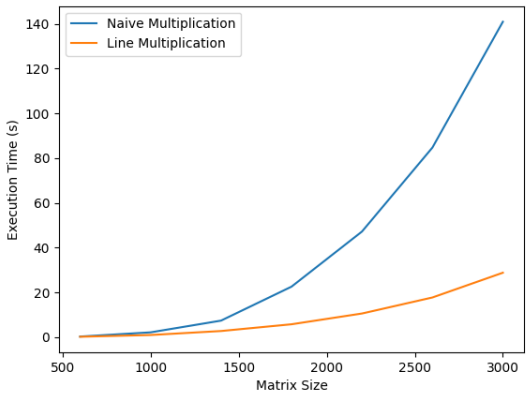

Figure 5: Execution time of Line and Naive Multiplication (`JAVA`)

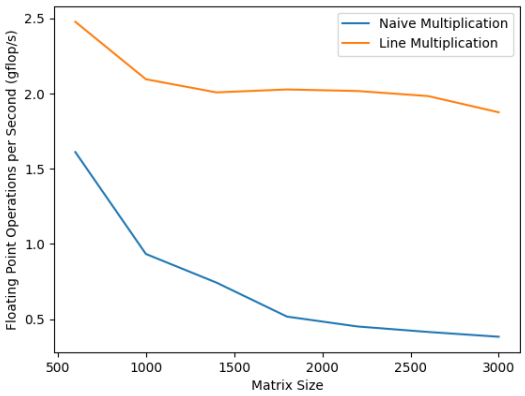

Figure 6: `GFLOPS` of Line and Naive Multiplication (`JAVA`)

The results obtained are similar to the ones previously obtained which validates the conclusions and indicates that there is no `C++` particularity that allowed that behaviour and that one algorithm is superior to the other in more than one language.

4.3 Line Multiplication vs Block Multiplication

In order to evaluate the effectiveness of the *Block Algorithm* and its cache performance, the algorithm was run for blocks of size `128`, `256`, `512`, `1024` and for matrices of `4096x4096`, `6144x6144`, `8192x8192` and `10240x10240`, and as a means of comparison the *Line Algorithm* was run for same sized matrices. The values measured were the same as the ones measured before and the following graphs display the results:

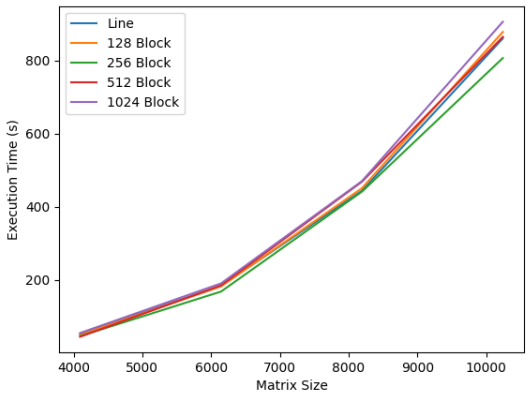

Figure 7: Execution time of Line and Block Multiplication (`C++`)

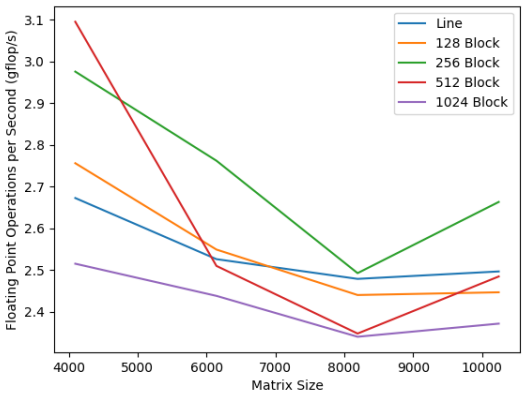

Figure 8: `GFLOPS` of Line and Block Multiplication (`C++`)

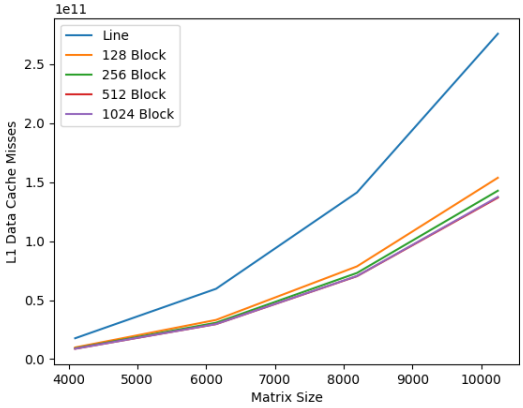

Figure 9: L1 Data Cache Misses of Line and Block Multiplication (`C++`)

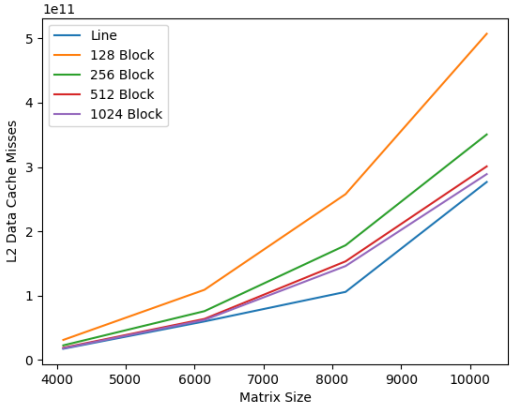

Figure 10: L2 Data Cache Misses of Line and Block Multiplication (`C++`)

It can be noted that the size of block with least Data Cache Misses is `1024`, and the misses seem to increase as the block size decreases, which can be explained by the fact that if a bigger sized chunk of data is loaded into cache memory then it will be more likely that the value the program needs is in cache. Exception made for the *Line Multiplication* that has the least Data Cache Misses for L2. However the impact on performance does not translate directly on performance as the block of size 256 has the best overall performance, which can be caused by the fact that the size fits best inside the cache, and so the performance is better than other bigger sized blocks, that may be too big to make use of the cache system.

5 Conclusions

This project explored the effects of memory access on the processor when accessing large amounts of data by calculating the product of square matrices of various sizes, using different algorithms and programming languages. We successfully analysed the results given by the algorithms implemented using our chosen performance metrics (time and data cache misses) and compared the results against each other to better interpret the processors' behaviour. Thus we were able to accomplish all the proposed project objectives.

During the development of the project we were able to learn more about many concepts related to memory access, such as memory hierarchy and cache memory. It was fascinating exploring how a subtle change in a simple algorithm, without changing its time complexity, can have such an impact on performance, due to reasons that can be often overlooked. Overall, this project had a positive impact on us as it allowed us to deepen our knowledge to hopefully become better future engineers.

Appendix A Result Data Tables

A.1 1. Naive Multiplication in CPP

Size		Time	L1 DCM	L2 DCM
	1	0.248	244567864	39314259
600x600	2	0.249	244444047	39020937
	3	0.246	244478967	39933474
	1	1.335	1216993178	188705416
1000x1000	2	1.374	1226433575	196642105
	3	1.486	1226466093	192798322
	1	5.128	3496708772	720096635
1400x1400	2	5.191	3520701252	767861178
	3	5.286	3521342910	732707506
	1	21.140	9068198210	4504534434
1800x1800	2	21.266	9078764278	5190185403
	3	21.309	9086526668	4120265003
	1	44.456	17642644537	22239312462
2200x2200	2	49.951	17642622097	21540711253
	3	44.539	17643525589	22419759033
	1	82.563	30901240812	50911386405
2600x2600	2	84.404	30900291855	50853210377
	3	83.500	30900917727	51532188583
	1	139.442	50316207768	94749120913
3000x3000	2	138.376	50316461057	95167903575
	3	141.202	50312090830	93383496102

A.2 1. Naive Multiplication in Java

Size		Time
	1	0.242095
600x600	2	0.283412
	3	0.278737
	1	2.263389
1000x1000	2	2.125062
	3	2.037778
	1	7.339578
1400x1400	2	7.396729
	3	7.416124
	1	22.549400
1800x1800	2	22.553031
	3	22.566179
	1	47.337149
2200x2200	2	46.635489
	3	47.564372

Size		Time
	1	84.899614
2600x2600	2	84.806130
	3	84.410386
	1	140.887719
3000x3000	2	140.450531
	3	141.489318

A.3 2. Line Multiplication in CPP

Size		Time	L1 DCM	L2 DCM
	1	0.151	27153282	54529367
600x600	2	0.152	27141381	54559552
	3	0.154	27284049	57544134
	1	0.637	125953821	248623757
1000x1000	2	0.629	126041514	258320343
	3	0.638	126031628	257570301
	1	1.897	347166511	671668621
1400x1400	2	1.917	347904673	689758554
	3	1.830	346710720	681399256
	1	3.892	746820416	1446189392
1800x1800	2	3.900	745937990	1422612495
	3	3.892	745898301	1466887340
	1	7.566	2079608217	2561609484
2200x2200	2	7.175	2077201559	2613662751
	3	7.178	2077393985	2570689873
	1	12.028	4413044878	4245737000
2600x2600	2	11.895	4412951269	4315073550
	3	11.899	4413186706	4304612351
	1	19.114	6780645975	6706541756
3000x3000	2	19.332	6780836067	6766952494
	3	19.797	6780760320	6794866473
	1	50.802	17627031321	17386767508
4096x4096	2	52.221	17663468196	17540023925
	3	51.234	17655830594	17408211501
	1	183.496	59600016129	59955964085
6144x6144	2	183.270	59578900971	59993437753
	3	184.048	59607573320	59908786138
	1	441.876	141259169229	140611193241
8192x8192	2	447.068	141260024754	137620859854
	3	441.589	141236755018	39413103996
	1	853.970	275802709027	278205615291
10240x10240	2	855.411	275868009574	275234620222
	3	870.774	275826307054	276909511501

A.4 2. Line Multiplication in JAVA

Size		Time
600x600	1	0.176985
	2	0.164318
	3	0.181724
1000x1000	1	0.942510
	2	0.935275
	3	0.985335
1400x1400	1	2.702358
	2	2.805933
	3	2.690271
1800x1800	1	5.740973
	2	5.762401
	3	5.754250
2200x2200	1	10.512228
	2	10.623340
	3	10.534425
2600x2600	1	17.382626
	2	18.148559
	3	17.632482
3000x3000	1	28.994091
	2	28.324997
	3	29.042478

A.5 3. Block Multiplication in CPP with 128 Block Size

Size		Time	L1 DCM	L2 DCM
4096x4096	1	49.519	9854266520	30771319946
	2	50.006	9853971819	31200439372
	3	50.062	9851673218	31576031436
6144x6144	1	177.960	33282003050	108868647304
	2	184.641	33159536737	109849265910
	3	183.238	33140161934	108726499012
8192x8192	1	449.120	78664786443	256953815477
	2	451.506	78854372638	258019132080
	3	450.996	78221462751	258676756813
10240x10240	1	882.110	153624732961	508267833814
	2	877.005	153765918968	507479032292
	3	873.727	153806062400	506135277676

A.6 3. Block Multiplication in CPP with 256 Block Size

Size		Time	L1 DCM	L2 DCM
	1	45.570	9131400162	22486963091
4096x4096	2	46.845	9130969409	22953124290
	3	46.131	9131869471	22674048019
	1	168.270	30795580754	75282664426
6144x6144	2	166.984	30798607919	77255827159
	3	168.501	30837897893	75160605133
	1	438.957	73135750850	179719151476
8192x8192	2	443.564	72956611213	177722649739
	3	440.670	72948180989	177683628239
	1	803.002	142716874485	348436296418
10240x10240	2	814.442	142757916610	355153829961
	3	801.297	142752040331	348493193191

A.7 3. Block Multiplication in CPP with 512 Block Size

Size		Time	L1 DCM	L2 DCM
	1	44.233	8754377404	18725675590
4096x4096	2	44.614	8755740444	19347541673
	3	44.349	8755114462	18710795579
	1	184.065	29609932259	63045210466
6144x6144	2	186.388	29640661332	65495650841
	3	183.917	29618093798	63038756240
	1	468.177	70255011021	154841344303
8192x8192	2	465.885	70225748041	153781920064
	3	470.756	70252668750	151484889087
	1	873.059	136883271398	299774035916
10240x10240	2	857.396	136864887773	301046194963
	3	862.220	136875730920	302320450526

A.8 3. Block Multiplication in CPP with 1024 Block Size

Size		Time	L1 DCM	L2 DCM
	1	54.174	8783612716	18031890278
4096x4096	2	54.895	8788436324	18032840409
	3	54.843	8782312091	18406769685
	1	189.592	29700369001	61806773550
6144x6144	2	191.053	29708051303	62951534873
	3	190.056	29708634563	61837539610
	1	469.027	70384815130	144524164126
8192x8192	2	470.852	70428404452	147388655290
	3	469.536	70484408796	146292039179
	1	903.137	137538226664	293450920671
10240x10240	2	906.935	137534007011	286534408820

Size	Time	L1 DCM	L2 DCM
3	906.168	137504205620	287059612481