Performance evaluation of a single core

Parallel Computing

Faculty of Engineering of the University of Porto

Integrated Master in Informatics and Computing Engineering

Daniel Marques
up201503822@fe.up.pt

João Damas
up201504088@fe.up.pt

March 22nd, 2019

Motivation

Problems can be solved in several ways. However, different algorithms for the same problem may result in substantially different performances. With this in mind, the goal of this project is to study the effect of the memory hierarchy on a processor's performance when it needs to access larges volumes of data. For this, 3 different algorithms for a common problem, matrix multiplication, were implemented in different programming languages. Their performances were subsequently compared in order to understand the impact of different implementations on processor performance and possible connections to the memory hierarchy.

Problem Description

The problem at hand is matrix multiplication, more specifically square matrices. Given matrices Ma and Mb of size nxn, the product matrix $Mc = Ma.Mb$, also of size nxn, can be defined as:

$$\forall i, j \in \{0, ..., n-1\}, Mc_{ij} = \sum_{k=0}^{n-1} Ma_{ik} Mb_{kj} \tag{1}$$

In other words, each entry Mc_{ij} is the result of the dot product between the i-th row of Ma and the j-th column of Mb.

Algorithms Explanation

In the subsections for each algorithm, n corresponds to the matrix size and, in block multiplication, b represents the size of each block when decomposing the matrix. Each algorithm assumes the result matrix Mc to be zero-initialized.

Column Multiplication

This algorithm implements the multiplication operation precisely as defined in eq. 1, i.e. it loops all i rows from Ma and, for each row, loops through all j columns of Mb, calculating the dot product for each row-column pair, looping through their k elements. In other words, this approach fully calculates each entry Mc_{ij} before moving to another one.

Algorithm 1 Column Multiplication

```
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        c = 0
        for k ← 0 to n − 1 do
            c += Ma_ik * Mb_kj
        end for
        Mc_ij = c
    end for
end for
```

Line Multiplication

In this algorithm, each entry Mc_{ij} is not calculated all at once. Instead, while looping Ma's i rows, it loops Mb's j rows as well. This way, when computing the dot product for each pair, a partial value for each entry in each row in Mc is calculated and added to the accumulated value already stored there.

Algorithm 2 Line Multiplication

for $i \leftarrow 0$ to $n-1$ **do**
    for $k \leftarrow 0$ to $n-1$ **do**
        for $j \leftarrow 0$ to $n-1$ **do**
            $Mc_{ij} \mathrel{+}= Ma_{ik} * Mb_{kj}$
        end for
    end for
end for

Block Multiplication

A block matrix is a matrix that is interpreted as having been partitioned into blocks (sub-matrices). If the partitions are conformable between the matrices Ma and Mb, (i.e., each block in Ma can be multiplied by its correspondent in Mb), then Mc's calculation can be done block-wise, yielding a matrix with as many row partitions as Ma and as many column partitions as Mb . Since the matrices are assumed to be square and of the same size, they will always be suitable for block multiplication. In this implementation, the line multiplication algorithm was used for each block-wise multiplication. The only major difference in this algorithm is the introduction of 3 outer loops (bi,bj,bk) to loop through the blocks and their sum parcels.

Algorithm 3 Block Multiplication

for $bi \leftarrow 0$ to $n-1$ step b **do**
    for $bj \leftarrow 0$ to $n-1$ step b **do**
        for $bk \leftarrow 0$ to $n-1$ step b **do**
            for $i \leftarrow bi$ to $min(bi+b, n)$ step 1 **do** ▷ Ensure no overflow if $n \mod b \neq 0$
                for $k \leftarrow bk$ to $min(bk+b, n)$ step 1 **do** ▷ Same as above
                    for $j \leftarrow bj$ to $min(bj+b, n)$ step 1 **do** ▷ Same as above
                        $Mc_{i,j} \mathrel{+}= Ma_{i,k} * Mb_{k,j}$
                    end for
                end for
            end for
        end for
    end for
end for

All algorithms have a complexity of $\mathcal{O}(n^3)$. For the first two this is obvious, as three n iteration loops are performed. In block multiplication, despite there being six loops, the first three only have approximately n/b iterations, while the three most inner loops iterate at most b times, cancelling out the block size factor for a common complexity. Since each calculation inside the loops requires 2 FLOP, all of them execute $2n^3$ FLOP in total.

Evaluation Methodology

In order to perform the necessary experiments, the algorithms were implemented in two different languages: C++ and Java. Different sized matrices were tested, with common values between Column/Line and Line/Block multiplications for a better assessment of the performance difference. For block multiplication, for each size, multiple block sizes were tested.

For all experiments, the execution time was recorded, as well as cache performance metrics in the C++ version with the help of the PAPI library, namely Data Cache Misses (DCM) on both L1 and L2 caches. With these values, it will be possible to compare execution times of the same algorithm in different programming languages, as well as measure the difference in performance between algorithms through derived metrics, namely Data Cache Misses per FLOP, for both L1 and L2 caches (Eq. 2), and FLOP/s (Eq. 3).

$$\frac{CacheMisses(Lx)}{FLOP} = \frac{DCM(Lx)}{2n^3}, x \in \{1, 2\} \tag{2}$$

$$\frac{FLOP}{s} = \frac{FLOP}{ExecutionTime(s)} = \frac{2n^3}{ExecutionTime(s)} \tag{3}$$

All experiences were run on a laptop running the Solus Linux distribution, powered by an Intel i7-8750h 2.2GHz processor, which contains 6 cores, each with a 32KB L1 data cache and a 256KB L2 unified cache (as well as a common L3 9MB SmartCache).

Results and Analysis

Language performance comparison

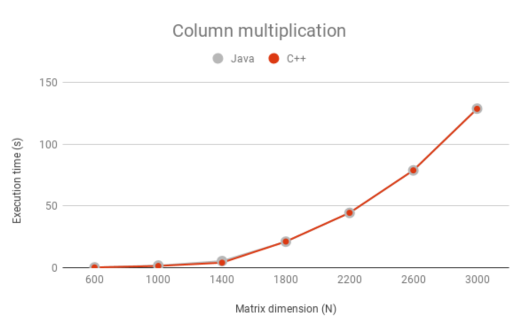

Figure 1: *Execution times for the column algorithm.*

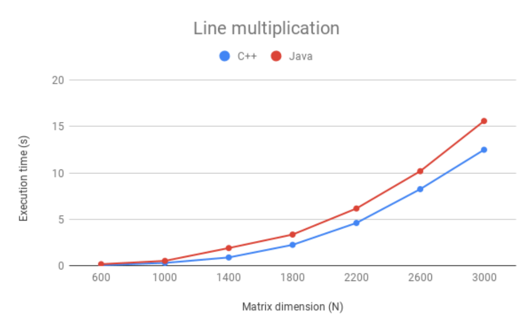

Figure 2: *Execution times for the line algorithm.*

The results show that C++, overall, proves to have a better execution time than the Java version of the same algorithm, with more emphasis on the line algorithm. This does not come as a surprise, since C++ is directly compiled into machine binary code, unlike Java, that runs the JVM and needs an extra layer of translation to be executed. One thing that might come as a surprise is the closeness of the times for the column algorithm: on average, the difference comes down to tenths of a second.

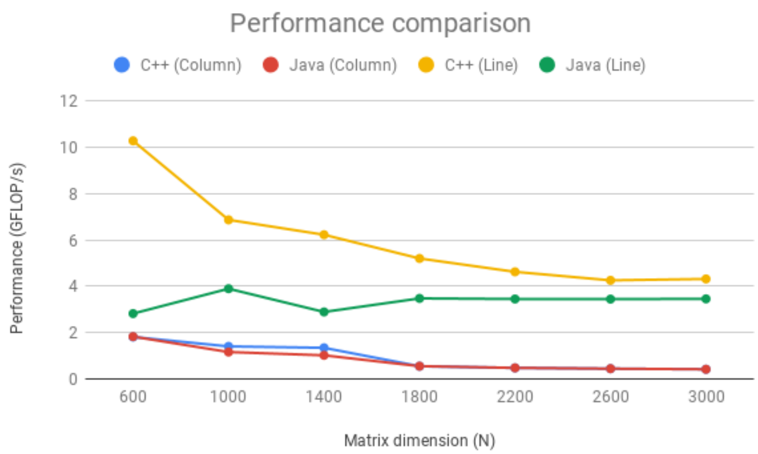

Figure 3: *Performance comparison between algorithms and languages.*

When translating execution times into performance, the difference is also noticeable (a little over 4GFLOP/s for C++ line, around 0.4 for column, regardless of language). Notice that after the 2200 mark the performance values seem to stabilize, cementing the difference between the two languages (e.g. on the line algorithm, the performance never goes below 4GFLOP/s on C++, but in Java it never reaches that point). This performance measurement also allows to assess the impact of cache usage, which is more thoroughly analyzed next.

Cache usage performance impact

As seen in Fig. 3, regardless of the programming language, the line algorithm always prevails in terms of performance. This is to be expected, since this algorithm access lines sequentially on both matrices (whereas the column algorithm loops through one of the matrices' columns) and both C++ and Java follow a row-major order policy when storing multidimensional arrays (matrices).

The line algorithm can then take advantage of this policy because of the *Principle of Locality*, more notoriously *spatial locality*, which states that items whose addresses are near one another tend to be referenced close together in time and, therefore, are loaded together in an effort to improve performance. This obviously applies to the line algorithm, since *Mb*'s rows are stored next to each other and their elements are accessed in close succession. Figures 4, 5 and 6 allow one to understand better these differences.

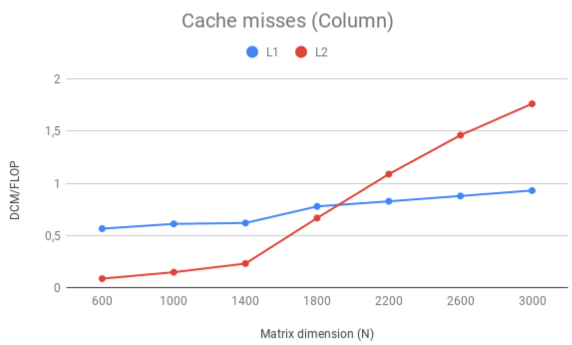

Figure 4: *DCM/FLOP for the column algorithm.*

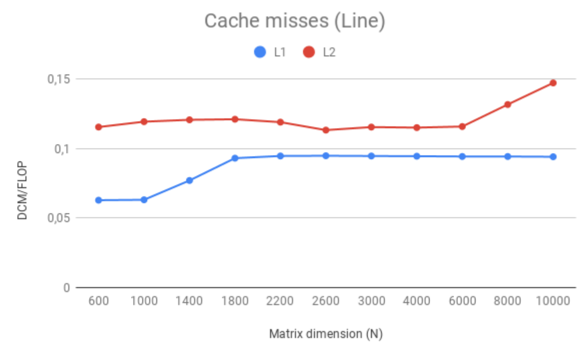

Figure 5: *DCM/FLOP for the line algorithm.*

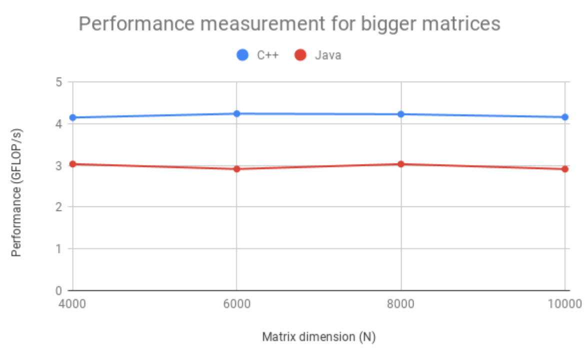

Figure 6: *Further performance measurements for the line algorithm.*

Figures 4 and 5 show that the frequency of data cache misses, for both levels (L1 and L2) is significantly lower in the line algorithm (by approximately a factor of 10). In the case of 5, these metrics were measured for even bigger matrices (from sizes between 4000 and 10000), showing that the L1 cache misses values are probably already stable (~0.09), despite the L2 showing some growth signs for bigger matrices, and shouldn't be subject to big changes as the input size grows (unlike the column algorithm in which both seem to grow much more significantly). Figure 6 also corroborates this conclusion, as the performance seems to be stable (~4.2GFLOP/s and ~3GFLOP/s in C++ and Java, respectively).

A surprising fact is the value of L2 cache misses being higher than L1 misses, which was unexpected, since L1 is usually accessed first. With this in mind, we established contact with the PAPI tool maintainers (see App.D), which quickly provided a compelling answer: prefetching is also being counted in L2_DCM, which increased its value considerably. This was corroborated by some quick further tests that also measured a new event, PAPI_PRF_DM (data prefetch misses), available in appendix E.

Block decomposition performance impact

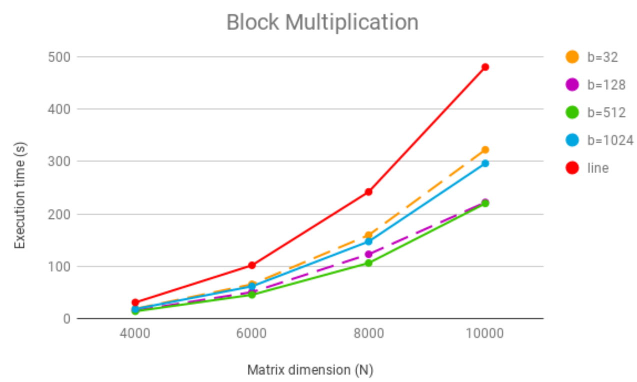

Figure 7: *Execution times for the block algorithm.*

Block decomposition algorithm proves to, overall, have an even better execution time than the line algorithm. The difference in performance can be explained through the exploitation of *temporal locality*, which states that recently accessed items are likely to be accessed in the near future, through block wise multiplication: the algorithm loads a block into the cache, perform all the necessary operations with it, discarding it after in favor of the next one. Nonetheless, the *block_size* parameter must be finely tuned for optimal results. Figure 7 compares different values for the block size and the line algorithm for reference. Even though all tested values provide better execution times than the line algorithm, block sizes can significantly affect them. Other block size values, namely 64 and 256, were tested, and their results are visible in the Appendix section. However, by not presenting significant changes to close neighbors, were removed from the chart, improving its readability.

Results show that the optimal *block_size* appears to be 512, with an average performance of 9.36 GFLOP/s. A memory efficiency comparison is found in figure 9. The algorithm's performance tends to get better with bigger blocks, however, if too big, it seems to actually slow the execution because the blocks needed may be too big to fit inside the cache, thus introducing misses. On the other hand, too small blocks may also slow the execution because the algorithm is probably not being fully exploited, introducing more block wise multiplication iterations and unnecessary overhead.

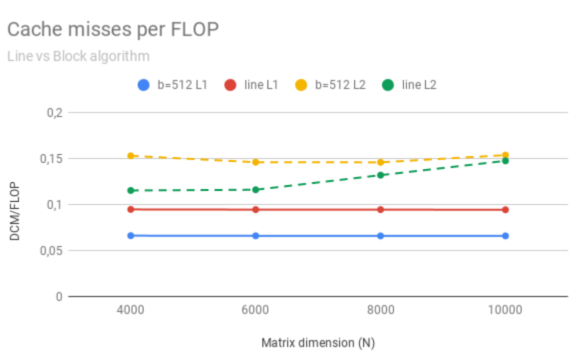

Figure 8: *Cache misses comparison between the block and line algorithms.*

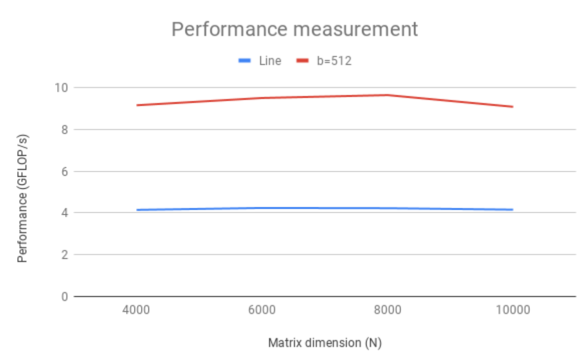

Figure 9: *Performance comparison between the block and line algorithms.*

Figure 8 shows that the block algorithm has a better cache usage, with its stable L1 values always being lower than the line's. Although the L2 cache's average misses is higher, it also appears to have a stable value, unlike the line algorithm which shows growth signs and would probably surpass it for larger matrices. Figure 9 shows that block's performance is much better (~2.3 times higher) and also has a stable value, hinting its scalability.

Conclusions

With this project it was possible to study the impact of the memory hierarchy in different programming languages, through simple, yet very different approaches for a common problem such as matrix multiplication. The results showed that the line algorithm is more efficient when implementing solutions in languages that follow row-major order storage, achieving 10 times the performance of the traditional column algorithm. Moreover, block decomposition (with line multiplication block wise), given the right block size choice, allowed for an extra increase in performance, with a peak of around 9.6GFLOP/s.

Appendix

A Column algorithm results

Size	Time (s)	L1 DCM	L2 DCM
600	0.238	244410437	37145031
1000	1.421	1222992943	295259414
1400	4.078	3402835816	1266090534
1800	21.194	9090274121	7787178923
2200	44.520	17632426833	23185403553
2600	78.950	30913445343	51412764629
3000	128.775	50309040447	95264584181

Table 1: *C++ column algorithm measurements*

Size	Time (s)
600	0.236
1000	1.719
1400	5.371
1800	21.089
2200	44.448
2600	78.894
3000	128.928

Table 2: *Java column algorithm measurements*

B Line algorithm results

Size	Time	L1 DCM	L2 DCM
600	0.042	27179046	49925742
1000	0.291	126413703	238868231
1400	0.881	423172266	662809043
1800	2.241	1086665273	1413540298
2200	4.602	2017757334	2536253246
2600	8.252	3335966999	3985295129
3000	12.502	5115391380	6239518139
4000	30.828	12105845708	14740545891
6000	101.782	40772360492	50087545084
8000	241.964	96633023386	134948903997
10000	480.425	188365974378	294660596488

Table 3: *C++ line algorithm measurements*

Size	Time
600	0.153
1000	0.513
1400	1.896
1800	3.352
2200	6.166
2600	10.191
3000	15.608
4000	42.164
6000	148.106
8000	337.373
10000	685.983

Table 4: *Java line algorithm measurements*

C Block algorithm results

Size	Block Size	Time (s)	L1 DCM	L2 DCM
4000	32	18.725	1005289860	3199334185
4000	64	14.376	8995820515	1749611879
4000	128	15.256	9320229143	10253616040
4000	256	15.250	8765444769	22628110503
4000	512	13.966	8457619701	19555546304
4000	1024	18.324	8351959629	18131524596
6000	32	65.499	3435890497	11425375406
6000	64	48.800	33776599009	5552792972
6000	128	50.573	31621458441	32729697377
6000	256	48.063	29639956795	73311333698
6000	512	45.407	28475967096	63005794666
6000	1024	61.618	28242839733	59830993116
8000	32	159.413	8590937821	27507482261
8000	64	118.382	58507921451	14277630103
8000	128	122.794	74568096178	86534113535
8000	256	113.041	69961918128	172415369137
8000	512	106.099	67406924924	149212478706
8000	1024	147.178	66730851284	142021820683
10000	32	322.422	15873288499	55479989588
10000	64	232.897	156727104139	23375828237
10000	128	221.905	146023702969	29388892817
10000	256	237.505	136742374266	359480463994
10000	512	219.961	131689147587	307092092117
10000	1024	296.414	130265632050	284834573913

Table 5: *(C++) block algorithm measurements*

D Email exchange with PAPI maintainers

Anthony Danalis <adanalis@icl.utk.edu>

Hi Daniel,

Prefetching could be the culprit. Try to see if your machine offers events that measure prefetching, and try them with your code. Also, you can look at "demand reads", to see if the discrepancy remains. Here are two (native events) that you probably have and should be illuminating:

L2_RQSTS:DEMAND_DATA_RD_MISS

L2_RQSTS:PF_MISS

Also, since L2_DCM is derived on your platform can you post the results of papi_avail -e PAPI_L2_DCM

Daniel Marques <up201503822@fe.up.pt> (Original email)

Greetings,

My name is Daniel Marques and I'm a student at FEUP.

I've been assigned to use PAPI to measure cache usage and performance of a single core on different matrix multiplication algorithms.

While measuring, I've noticed that the L2 Data Cache Misses (DCM) is much bigger than the L1 counterpart. I can't explain why, so I hope you could help me find out the reason behind it.

Here are some facts that may be useful: - All the testing was performed on an Intel Core i7 8750h running Solus Linux. - Running 'papi_avail -a' shows L1_DCM as a native event and L2_DCM as derived event.

Thank you for your attention.

Cheers, Daniel Marques

E (Some) Prefetching tests

Size	L1 DCM	L2 DCM	PRF DM
3500	8108336349	9914824082	8033540137
5000	23599558132	29079035234	23581252119
6500	51791862795	68190032615	53901830118
8000	96610930486	130189891340	100898772243
9500	161475315328	240014159848	189243397496

Table 6: *Some DCM misses measurements and comparison with data prefetch misses. Notice the common pattern $L1\_DCM > L2\_DCM - PRF\_DM$, which corroborates the hypothesis that prefetch misses are indeed included in L2's counter.*