Faculdade de Engenharia da Universidade do Porto

Licenciatura em Engenharia Informática e Computação

Computação Paralela e Distribuída

Performance evaluation of a single core

Student	**Up**
Sofia Germer	up201907461
Sérgio Estêvão	up201905680
Miguel Rodrigues	up201906042

2021/2022

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Contents

List of Figures

1 Problem description

The aim of this project is to understand the effect on the processor performance of the memory hierarchy when accessing large amounts of data.
The product of two matrixes will be used for this study.
In order to collect relevant performance indicators of the program execution we used the Performance API (PAPI).

2 Algorithms explanation

All algorithms do 2*n^3 floating point operations = O(n^3) Flops

2.1 Column Multiplication

Column oriented matrix multiplication: multiplies one line of the first matrix by each column of the second matrix. This algorithm was already provided by the course teaching staff in C/C++ and we implemented it in RUST.

Algorithm 1 Column Multiplication

```
function ColumnMultiplication(A, B)
    m_ar ← matrix A size
    m_br ← matrix B size
    for i = 0 to m_ar do
        for j = 0 to m_ar do
            temp ← 0
            for k = 0 to m_ar do
                temp += A[i * m_ar + k] * B[k * m_br + j]
            end for
            C[i * m_ar + j] ← temp
        end for
    end for
     return C
end function
```

2.2 Line Multiplication

Row oriented matrix multiplication: multiplies an element from the first matrix by the correspondent line of the second matrix. We implemented this algorithm in C/C++ and in RUST.

Algorithm 2 Line Multiplication

```
function LineMultiplication(A, B)
    m_ar ← matrix A size
    m_br ← matrix B size
    for i = 0 to m_ar do
        for k = 0 to m_ar do
            temp ← 0
            for j = 0 to m_ar do
                temp += A[i * m_ar + k] * B[k * m_br + j]
            end for
            C[i * m_ar + j] ← temp
        end for
    end for
     return C
end function
```

2.3 Block Multiplication

Block oriented algorithm that divides the matrixes in cache-sized blocks and uses the same sequence of computation as in the previous algorithm. We implemented this algorithm in C/C++.

Algorithm 3 Block Multiplication

```
function BlockMultiplication(A, B)
    m_ar ← matrix A size
    m_br ← matrix B size
    bkSize ← block size
    for ii = 0 to m_ar incrementing bkSize do
        for jj = 0 to m_ar incrementing bkSize do
            for kk = 0 to m_ar incrementing bkSize do
                for i = ii to min(bkSize+ii, m_ar) do
                    for k = kk to min(bkSize+kk, m_ar) do
                        for j = jj to min(bkSize+jj, m_ar) do
                            C[i * m_ar + j] += A[i * m_ar + k] * B[k * m_br + j]
                        end for
                    end for
                end for
            end for
        end for
    end for
     return C
end function
```

3 Performance metrics

To evaluate the performance of each algorithm we used the time elapsed, capacity, which we calculate using **2*N³/time**, **N** being the size of the matrix, and the data cache misses of the L1 and the L2 cache memory, which we obtained via the performance API (PAPI), as mentioned before.

4 Results

These results were obtained in a Legion-Y540-15IRH-PG0 PC, running an Intel® Core™ i7-9750H CPU (Coffee Lake) with 6 cores and 12 threads with Hyper-Threading @ 2.60 GHz.

4.1 Column Multiplication

4.1.1 C/C++

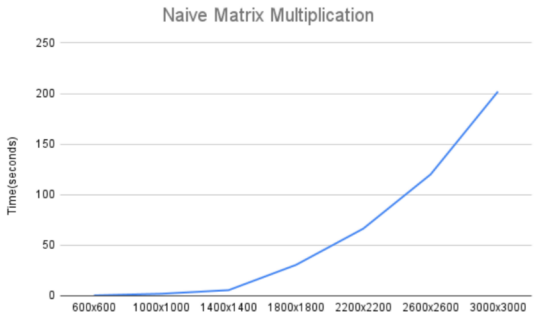

Figure 1: Time(seconds) of Column Matrix Multiplication in C++

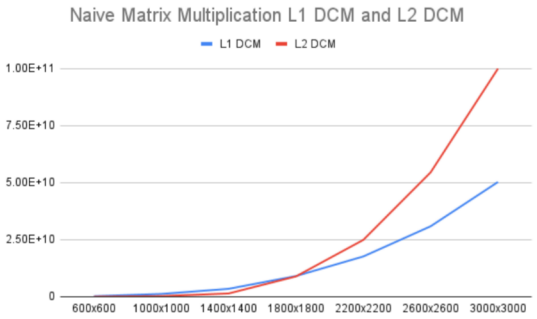

Figure 2: L1 DCM and L2 DCM of Column Matrix Multiplication in C++

4.1.2 RUST

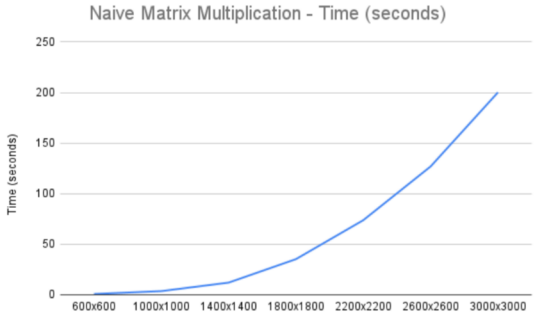

Figure 3: Time(seconds) of Column Matrix Multiplication in RUST

4.2 Line Multiplication

4.2.1 C/C++

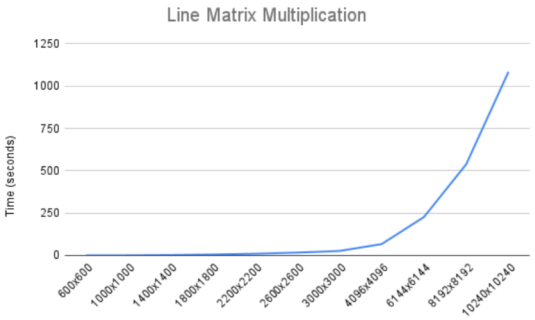

Figure 4: Time(seconds) of Line Matrix Multiplication in C++

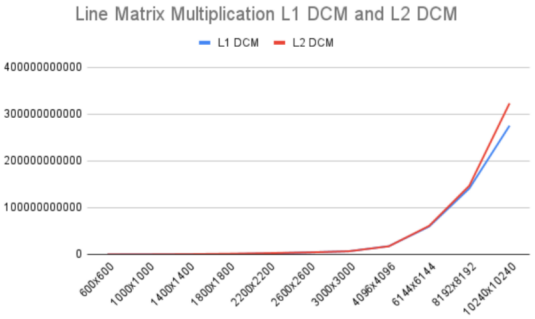

Figure 5: L1 DCM and L2 DCM of Line Matrix Multiplication in C++

4.2.2 RUST

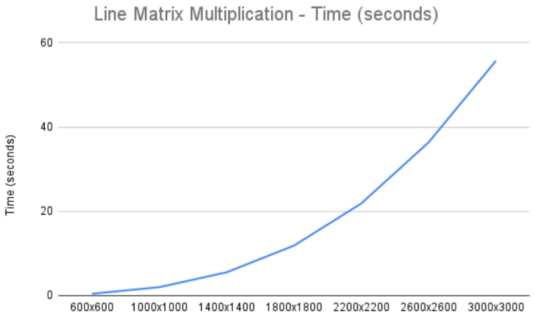

Figure 6: Time(seconds) in Line Matrix Multiplication in RUST

4.3 Block Multiplication

4.3.1 C/C++

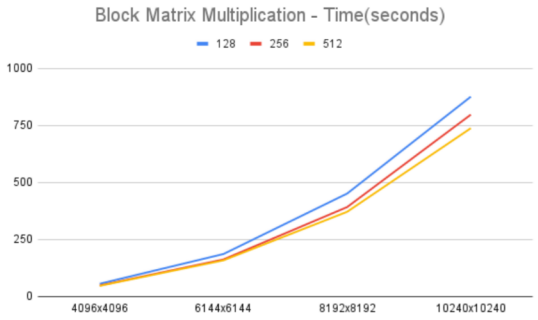

Figure 7: Time(seconds) of Block Matrix Multiplication in C++

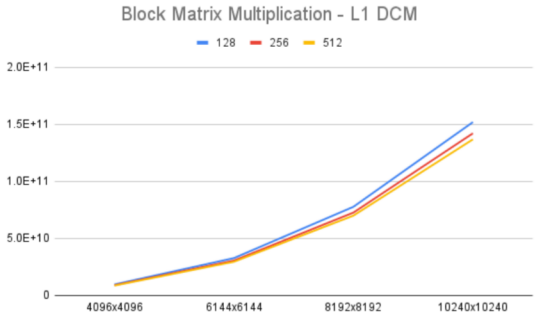

Figure 8: L1 DCM of Block Matrix Multiplication in C++

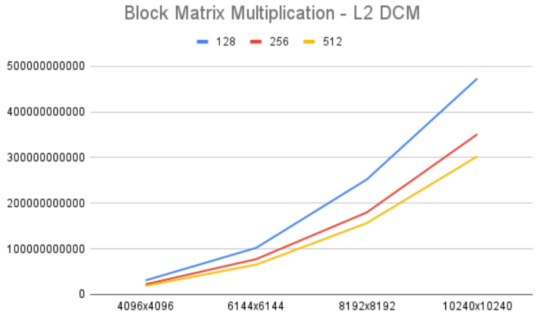

Figure 9: L2 DCM of Block Matrix Multiplication in C++

5 Analysis

After analysing the results, we verified that the column algorithm gave worst results compared to its line algorithm counterpart.
These results confirm the known fact that reading memory in contiguous locations is faster than jumping around among locations. This happens because when reading a value from the matrix, the CPU tries to fetch it from the cache. If the operation is unsuccessful (Data Cache Miss) it will fetch the value from the main memory and copy a chunk of memory next to that value to the cache, making new readings next to the previous values faster since the data will more often be in cache and therefore closer to the CPU. Since the line algorithm is more cache friendly and reads data in a row-major order, that means there are few data cache misses and consequently the algorithm runs in a shorter time.

The block matrix algorithm gave the best results overall, specially for larger matrixes, having the least amount data cache misses and executing in the least amount of time among all block sizes. The larger the block size the more efficient is the algorithm was, however the block must fit in the cache (architecture dependent), so we cannot make these blocks arbitrarily large.
This results occurred because the block algorithm takes advantages of the memory cache block system. By using smaller blocks that can fit in the L1 and L2 the access of data, compared to accessing further level cache, achieving smaller execution times.

6 Conclusions

Through this project we were able to study and fully comprehend the impact of the data cache access on the performance of algorithm and how a deficient cache access can hinder the execution of a program. We were able to register these implications in real time with different approaches to the multiplication of matrixes.
Having the knowledge about the cache system and its impact on performance of algorithms will without a doubt prove very useful in future software development.

7 References

https://sites.cs.ucsb.edu/~tyang/class/240a17/slides/Cache3.pdf

8 Attatchment - Data Tables

Data obtain and used to build the graphics in the **Results** chapter.

8.1 Column Multiplication

8.1.1 C/C++

	Time (seconds)	Capacity	L1 DCM	L2 DCM
600x600	0.352	1227272727	244762464	39841821
1000x1000	2.006	997008973	1218157879	276262266
1400x1400	5.569	985455198	3466755326	1432246366
1800x1800	30.532	382025416	9070153335	8909795348
2200x2200	66.406	320693913	17648107946	24908687242
2600x2600	120.043	292828403	30879423474	54515371324
3000x3000	201.983	267349232	50290601209	99971337871

8.1.2 RUST

	Time (seconds)	Capacity
600x600	0.771	560311284
1000x1000	3.620	552486188
1400x1400	12.032	456117021
1800x1800	35.261	330790392
2200x2200	73.776	288657558
2600x2600	126.897	277012064
3000x3000	200.171	269769347

8.2 Line Multiplication

8.2.1 C/C++

	Time (seconds)	Capacity	L1 DCM	L2 DCM
600x600	0.186	2322580645	27113257	55718004
1000x1000	0.899	2224694105	125996252	256468072
1400x1400	2.637	2081152825	348148541	702658658
1800x1800	5.671	2056780109	749770733	1491520816
2200x2200	10.309	2065767776	2081077549	2684837731
2600x2600	17.088	2057116105	4413026981	4401794464
3000x3000	26.769	2017258769	6780085362	6767939477
4096x4096	67.513	2035740575	17671959034	17521428978
6144x6144	227.372	2040077353	59599254338	61060666820
8192x8192	538.848	2040485680	141144884618	146913291991
10240x10240	1084.880	1979466529	275297970049	323195794878

8.2.2 RUST

	Time (seconds)	Capacity
600x600	0.433	997690531
1000x1000	2.000	1000000000
1400x1400	5.529	992584554
1800x1800	11.862	983308042
2200x2200	21.866	973932132
2600x2600	36.378	966298312
3000x3000	55.732	968922702

8.3 Block Multiplication

8.3.1 C/C++

Time (seconds)

	4096x4096	6144x6144	8192x8192	10240x10240
128	56.167	186.508	451.495	876.049
256	48.680	162.753	392.226	797.342
512	47.272	158.880	371.593	737.901

Capacity

	4096x4096	6144x6144	8192x8192	10240x10240
128	2446969812	2487059365	2435268669	24513282340
256	2823314574	2850064010	2803260436	2693303059
512	2907407207	2919539702	2958913724	2910259842

L1 DCM

	4096x4096	6144x6144	8192x8192	10240x10240
128	9737005032	32836986502	77805627812	152010232860
256	9093389692	30670193418	72731401993	142193945198
512	8768586165	29638701788	70087463189	136912695044

L2 DCM

	4096x4096	6144x6144	8192x8192	10240x10240
128	30974558911	102257295908	252115437880	473371071155
256	22385672350	77514066163	179799536500	351220976308
512	19151009744	65404387284	156503742971	302765944518