Performance evaluation of a single core

Parallel and Distributed Computing

Degree in Informatics and Computing Engineering

Henrique Pinho
up201805000@up.pt

Luís Marques
up201104354@up.pt

Paulo Ferreira
up201804977@up.pt

March 2022

Problem Description

The problem that we are attempting to resolve is the analysis and study of the performance of a single core relative to memory hierarchy specifically when dealing with a big amount of data.

The proposed example to illustrate the problem is the multiplication of two matrices of different sizes with the same number of rows and columns using different algorithms with two different programming languages: C++ (where we use the PAPI library to get feedback of the performance) and Java.

$$c_{ij} = a_{i0}*b_{0j} + a_{i1}*b_{1j} + \ldots + a_{i(n-1)}*b_{j(n-1)} = \sum_{k=0}^{n-1} a_{ik}*b_{kj}, \text{ for n = number of rows}$$

The three different algorithms we used are **Column Multiplication** and **Line Multiplication** in both languages and **Block Multiplication** in C++.

Algorithms:

Column Multiplication:

With this approach the elements of the first row from Matrix A are multiplied by the elements of the first column of Matrix B and loops until the end of both matrices.

```
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;
            }
      }
```

Line Multiplication:

With this algorithm the first element of Matrix A is multiplied by all elements of the first row of Matrix B the results are accumulated to the values on the Matrix C by the corresponding row of the Matrix A and column of Matrix B.

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

Block Multiplication:

Both matrices are partitioned in smaller matrices (blocks) of the same size. For each block the line multiplication algorithm is applied.

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

All the algorithms have a complexity of O(n^3), since the first two algorithms have three for loops and the third algorithm even though it has two more for loops than the other two it does not impact the complexity since the outer loops are performed at most n / block size.

Performance Metrics

For the first two algorithms we timed the execution for different matrix sizes in both languages and for C++ the PAPI library was used to get Data Cache Misses for L1 and L2 caches. The third algorithm was only implemented in C++ and for each matrix size, blocks of different sizes were used to time the execution and PAPI was used for DCM in both caches L1 and L2. To compare the performance between the algorithms some derivatives metrics were calculated such as DCM/FLOP ($\frac{DCM}{2n^3}$) for both caches and GFLOP/s ($\frac{2n^3}{10^9 * Time(s)}$).

The C++ code was compiled using the -O2 optimization flag.

Results and Analysis

C++ and Java comparison:

For the column algorithm both languages present similar performance.

As the size of the matrix increases, more time is needed to perform all the calculations, for bigger matrices this is not the best approach (Fig. 1). The line multiplication algorithm reduces the execution time by almost tenth of the time in both languages although a performance increase is more noted with the C++ algorithm (Fig. 2). Even as good as it seems for larger matrices the time increases exponentially making this solution not a good option as we can see for matrix sizes superior to 4096 (Fig. 3).

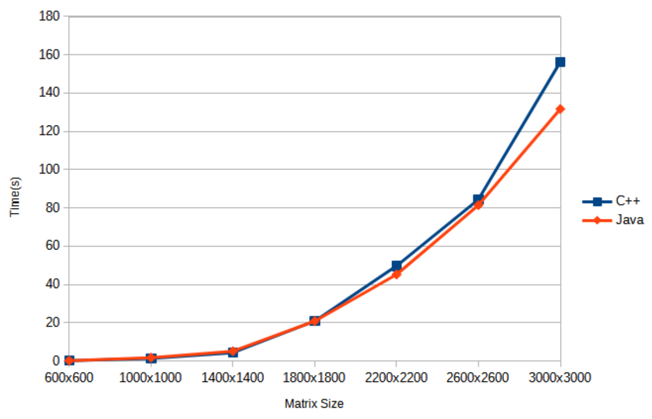


Figure 1: Column Multiplication

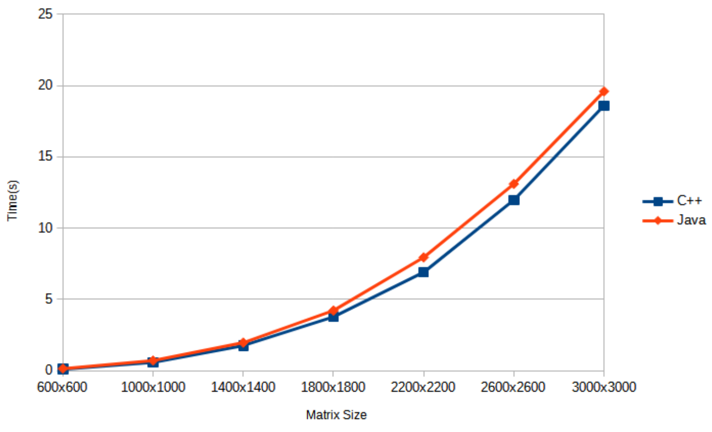


Figure 2: Line Multiplication

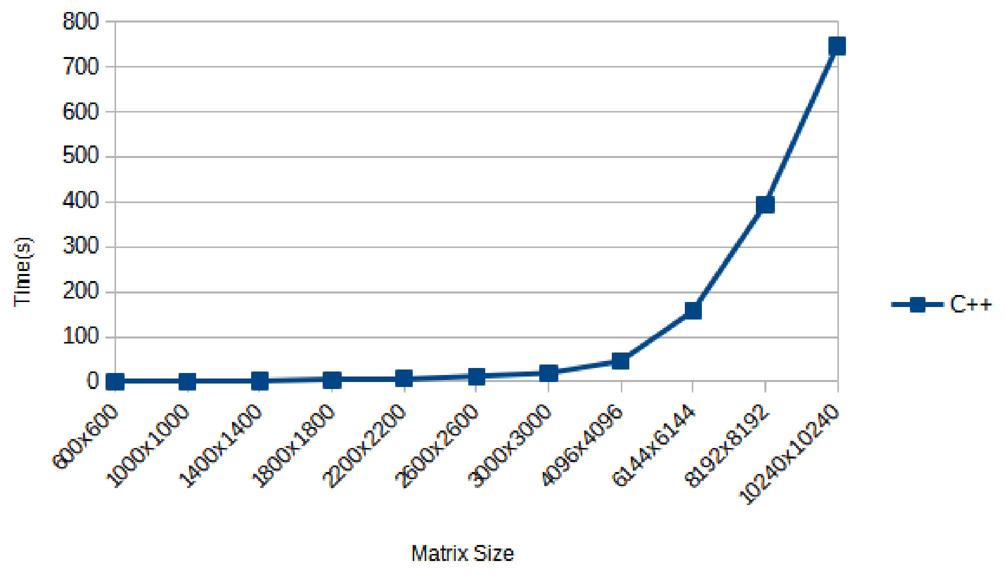


Figure 3: Line Multiplication

Comparing performance between the two languages it comes as clear that the line algorithm has a clear advantage. For matrices superior to 1400 the column algorithm has a performance drop of more than 50% for both languages. As for the line algorithm the performance keeps consistent (Fig. 4).

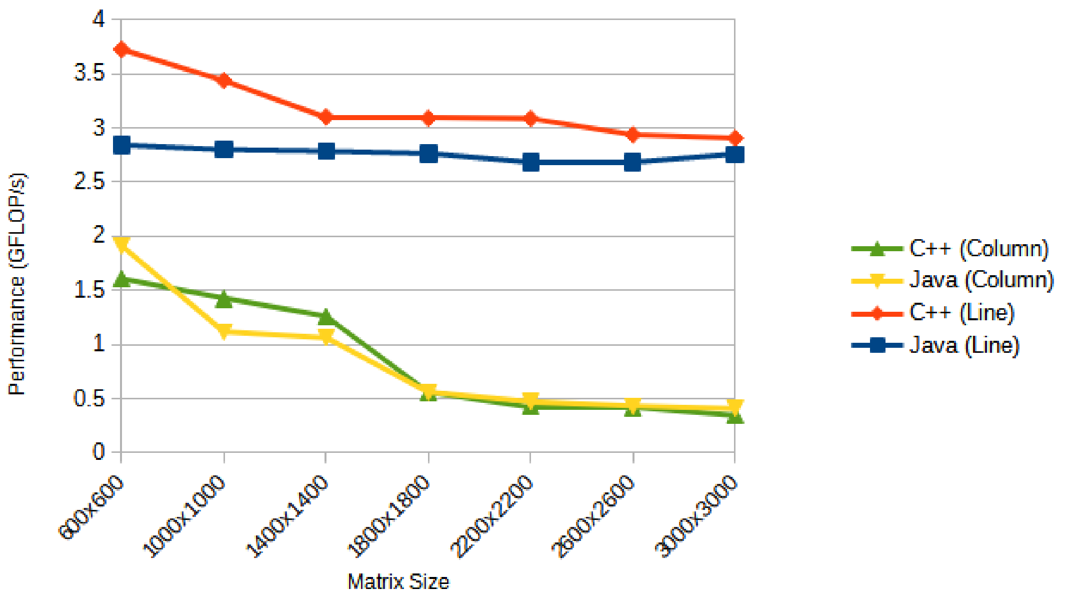


Figure 4: Performance C++ vs Java (higher is better)

DCM Impact on Performance:

Through the analysis made on both algorithms we can conclude that the line algorithm is a far better solution.
The cache memory was developed considering two key concepts: spatial location and temporal location. Spatial location states that when data elements are requested their neighbors will also be, since the rows of the matrices are stored next to each other the line algorithm will take advantage.
As the matrix size increases the DCM for the column algorithm intends to increase unlike the line algorithm which seems to maintain stability even for bigger matrices.

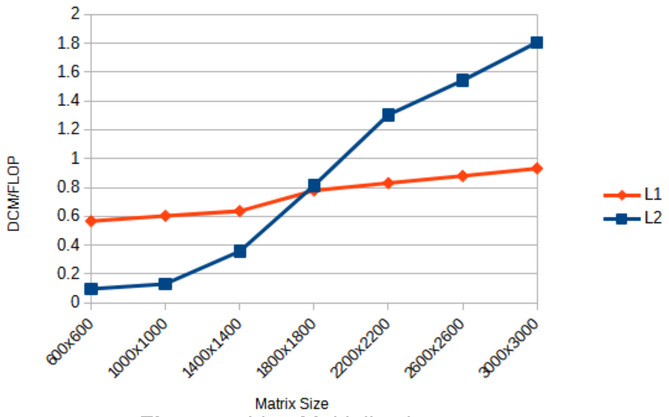


Figure 5: Line Multiplication (lower is better)

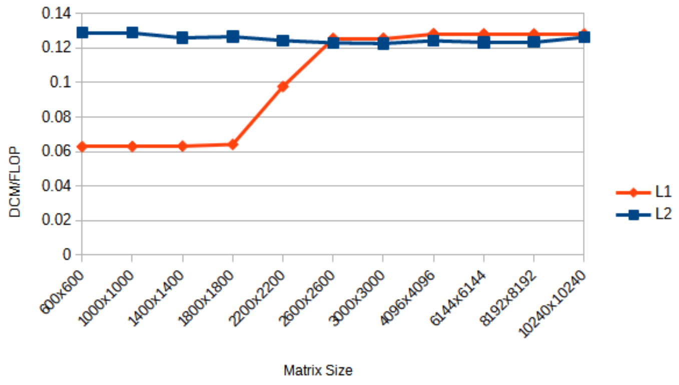


Figure 6: Column Multiplication (lower is better)

Block Algorithm Analysis:

Compared with the Line Algorithm and Column the Block Algorithm has better execution time and performance (Fig. 8). This difference in performance can be explained through temporal locality, which is the tendency for a processor to access memory locations that have been used recently. The algorithm makes use of this because it loads a block, performs the necessary multiplications with it, and discards the data.
Also, comparing the overall DCM/FLOP of both algorithms (Fig. 9), we conclude that the block algorithm has a loss less cache misses, even though the L2 cache in the block algorithm has overall more DCM/FLOP.
Analyzing the stats that we got (available in the appendix) we can conclude the performance gets better as the size of the blocks get bigger to a certain extent. When we start dealing with matrices of a bigger size the division in bigger sized blocks starts slowing down the performance of the algorithm.

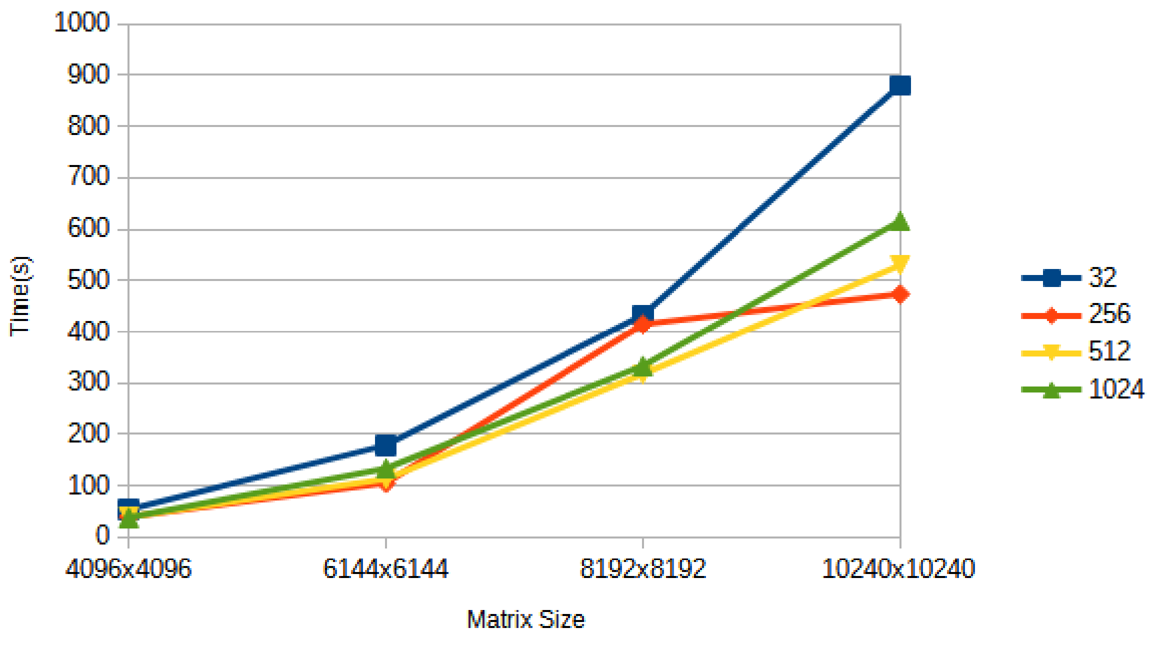
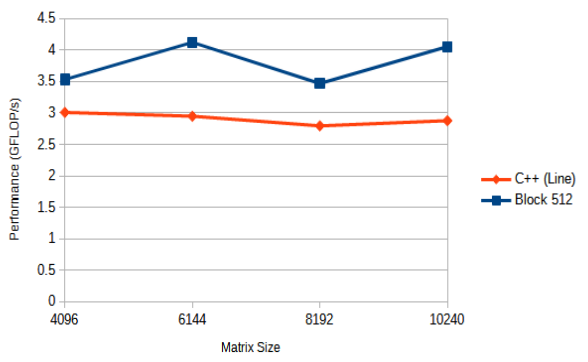


Figure 7: Block Multiplication



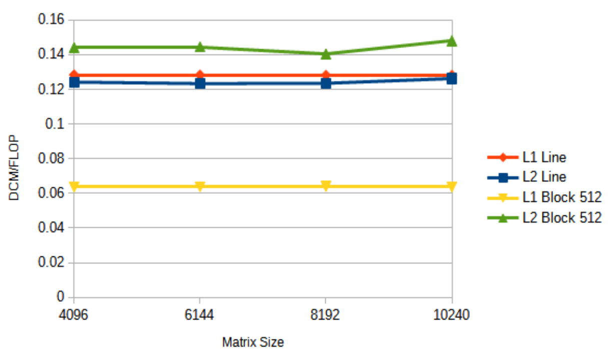


Figure 8: Performance C++(line) vs Block 512 (higher is better)

Figure 9: DCM/FLOP Comparison (lower is better)

Conclusions

Through the results that we got from analysis we concluded that overall the Block Multiplication is better than the rest of the algorithms and between the Line and Column Algorithm, the Line Algorithm is far more efficient, being slightly better in its C++ implementation.

Appendix

Column multiplication algorithm results

Size	TIME(s)	L1 DCM	L2 DCM
600	0.269	244742456	41106208
1000	1.402	1206206410	261425887
1400	4.360	3489550575	1961083847
1800	20.888	9082498139	9490801783
2200	49.719	17662426414	27743180726
2600	84.452	30883262701	54198519828
3000	156.303	50301750035	97491669606

1. **C++ Column algorithm measurement.**

Size	Time(s)
600	0.226
1000	1.790
1400	5.164
1800	20.869
2200	45.239
2600	81.518
3000	131.700

2. Java Column algorithm measurement.

Line multiplication algorithm results

Size	TIME(s)	L1 DCM	L2 DCM
600	0.116	27131790	55698828
1000	0.582	125784392	257869042
1400	1.770	346263751	690653537
1800	3.777	746922459	1475643464
2200	6.902	2079391052	2646424290
2600	11.973	4413898464	4321611401
3000	18.596	6781841655	6620173709
4096	45.694	17636620477	17064651108
6144	157.365	59491723628	57182451673
8192	393.589	140904879294	135752770381
10240	746.685	275116699028	271002648925

3. C++ Line algorithm measurement.

Size	TIme (s)
600	0.152
1000	0.715
1400	1.971
1800	4.222
2200	7.946
2600	13.098
3000	19.585

4. Java Line algorithm measurement.

Block multiplication algorithm results

Size	bkSize	TIME(s)	L1 DCM	L2 DCM
4096	32	54.015	13905913418	37539879743
4096	256	39.494	9112571677	22215221946
4096	512	38.932	8757663970	18863918994
4096	1024	38.136	8797594883	18577345752
6144	32	178.574	47010463996	110006846440
6144	256	104.779	30724421743	77545637349
6144	512	112.558	29593951994	66454897411
6144	1024	133.521	29740588402	62210985204
8192	32	432.616	114160142485	324028442596
8192	256	414.27	73096139619	159011209020
8192	512	317.074	70176732891	148931944640
8192	1024	333.296	70487942803	141116022356
10240	32	880.378	218003990384	616683545193
10240	256	473.398	142238039364	346626612698
10240	512	530.027	136844838742	306603996447
10240	1024	616.026	137718157546	291840557719

5. C++ Column algorithm measurement.