Computação Paralela e Distribuida 2021 - 2022

Matrix Multiplication

João Silva 201906478

Nuno Castro 202003324

Pedro Vale 201806083

Index

1. Problem description and Algorithm Analysis

In this project we evaluated the performance of a single core using as an object of study the product of two matrices.

To collect relevant processor performance of the memory hierarchy when accessing large amounts of data, in this case the rows and columns of the matrices, we used the Performance API (PAPI). PAPI show us the relation between software performance and processor events.

The first iteration of the project was given to us as an example to run and to collect performance data related to a basic algorithm that multiplies two matrices.

This first algorithm is the basic approach one can do to multiply two matrices. Basically, we multiply one line of the first matrix by each column of the second matrix.

In terms of implementation, we can do this by doing 3 nested for loops, the first one to get the lines of the first matrix, second one to get the columns of the second matrix and third one to get the resulting element on the third matrix. Basically, we multiply each element of each row of the first matrix with corresponding element of each column of the second matrix.

```
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;
    }
}
```

Fig.1 - Implementation

In the second version of the solution to multiply two matrices we followed the algorithm of multiplying two matrices by doing line x line, i.e. multiplies an element from the first matrix by the correspondent line of the second matrix.

This can be achieved in the same 3 nested for loops, but in this algorithm don't need to use a temporary variable to keep the intermediate results of the operation, we can just add this intermediate result to the correspondent position on the result matrix.

```
for (i = 0; i < m_ar; i++)
{
    for (j = 0; j < m_br; j++)
    {

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

Fig.2 - Implementation of Line x Line Multiplication of Matrices

In the final iteration of the solution, we collect performance data related to the block multiplication of matrices algorithm.

In this algorithm, we divide the matrices in blocks of varied sizes and then we multiply these blocks using the same computation we used on the Line x Line multiplication algorithm.

To do this we need to increase the number of our nested loops to 6 instead of 3, since we now need to access the correspondent blocks of each matrix before we do the calculations.

```
//-----------Insert code-----------------
for (int ii = 0; ii < m_ar / bkSize; ii++)
{
    for (int jj = 0; jj < m_br / bkSize; jj++)
    {
        for (int kk = 0; kk < m_ar / bkSize; kk++)
        {
            for (int i = 0; i < bkSize; i++)
            {
                for (int j = 0; j < bkSize; j++)
                {
                    for (int k = 0; k < bkSize; k++)
                    {
                        int x = ii * bkSize + i;
                        int y = jj * bkSize + j;
                        int z = kk * bkSize + k;
                        phc[x * m_ar + z] += pha[x * m_ar + y] * phb[y * m_ar + z];
                    }
                }
            }
        }
    }
}
```

Fig.3 - Implementation of Block Multiplication of Matrices.

2. Performance metrics

To evaluate the performance of our program we opted to extract some execution details through PAPI. The chosen parameters were:

- Execution time;
- L1 DCM – Data cache misses of the L1 cache counter;
- L2 DCM – Data cache misses of the L2 cache counter;
- GFlops – Floating-Point operations per second;

With these variables we will conduct a performance analysis between the three different solutions for the same problem.

The variables above were only obtained for the program coded in C++ language. As PAPI isn't compatible with Java, for our Java coded program we calculated the execution time and retrieved the GFlops in every run of the algorithms. To establish comparison between the performance of the two languages solutions we'll use the execution time.

3. Results and analysis

- NormalMul vs LineMul

Line Multiplication algorithm is much faster than the Normal Multiplication algorithm, in C++ and in Java. In Java, the difference between them is bigger than in C++.

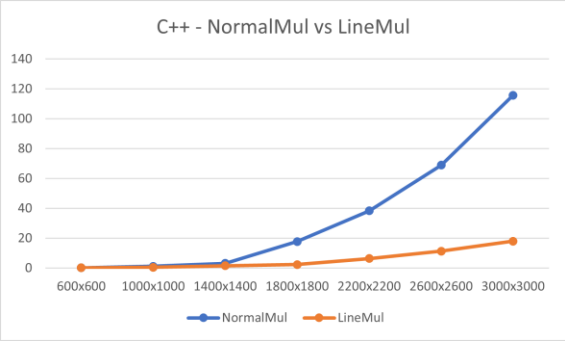

Fig.4 - C++ NormalMul vs LineMul

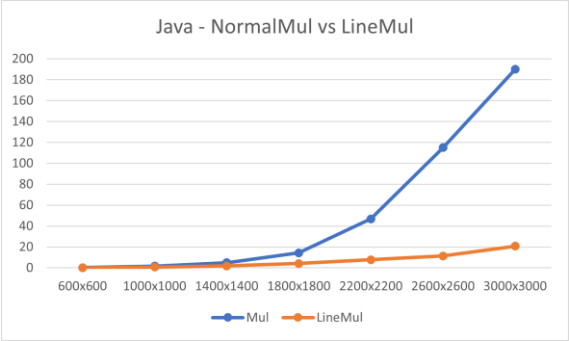

Fig.5 - Java NormalMul vs LineMul

In Java we obtained the following results from the NormalMul and LineMul in terms of GFlops:

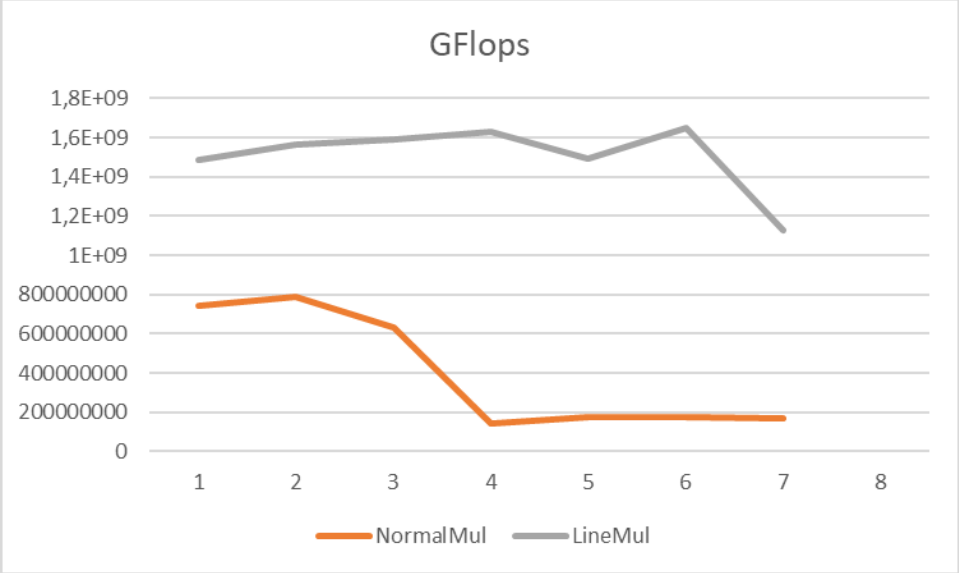

Fig.6 - Java NormalMul vs LineMul (GFlops)

- C++ vs Java

Both in NormalMul and LineMul algorithms, C++ is faster than Java. The difference between them is more significant in NormalMul and as matrix size increases.

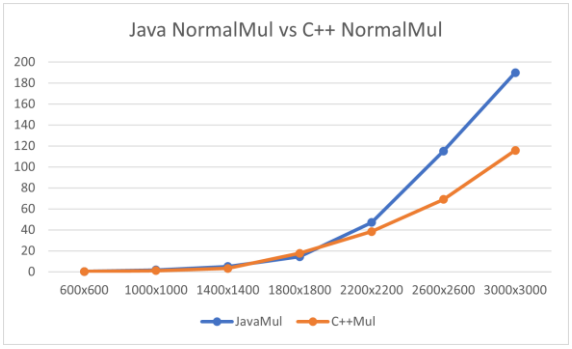

Fig.7 - NormalMul Java vs C++

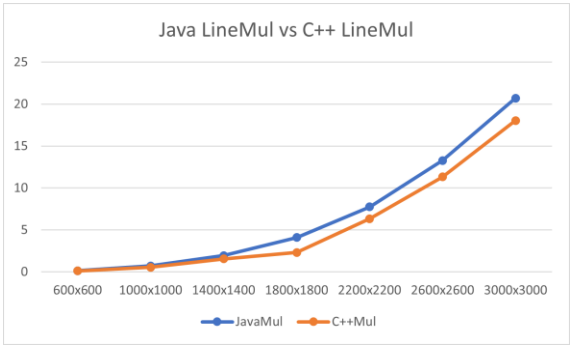

Fig.8 - LineMul Java vs C++

- LineMul vs BlockMul

BlockMul has better time performance than LineMul in all sizes we analyzed. The difference between these two algorithms also increases as the matrix size increases. Inside the BlockMul, we tested three block sizes (128, 256 and 512). The one with the best performance was the one with block sizes equal to 256.

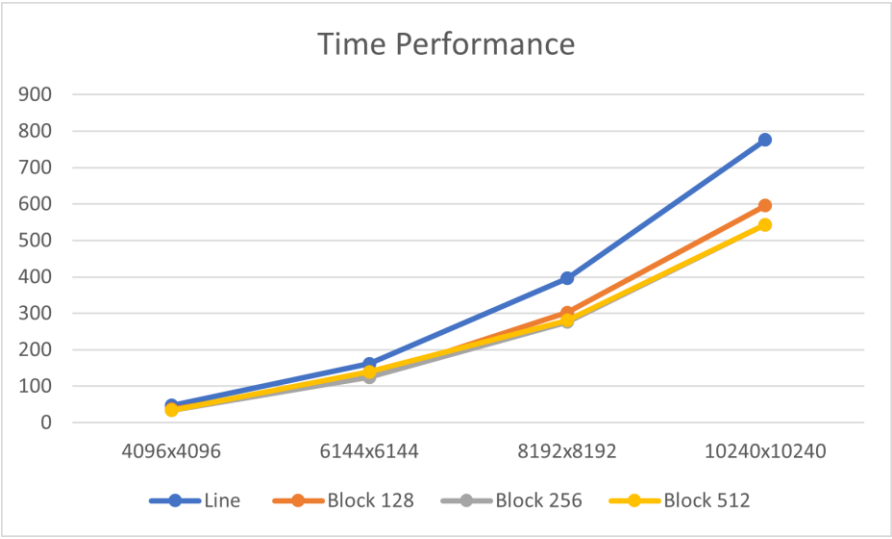

Fig.9 - LineMul vs BlockMul

- PAPI Measurements

When comparing the number of data cache misses of the L1 cache (L1 DCM), we can see that LineMul has the highest in all different sizes. In the BlockMul algorithm, there is not much difference between the different block sizes, however the one with 512 has the smallest values across all the different matrix sizes.

Regarding the number of data cache misses of the L2 cache (L2 DCM), the algorithm BlockMul with 128 block sizes has the highest values. The one with the lowest values is still the BlockMul with 512 block sizes.

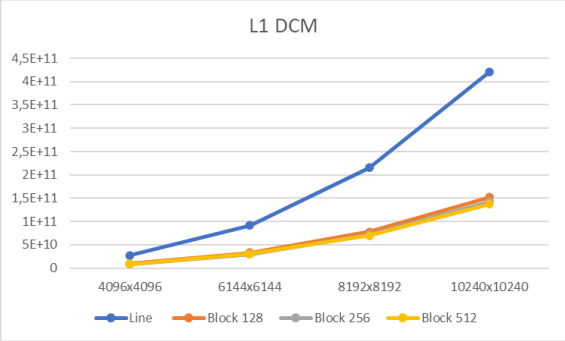

Fig.10 - L1 DCM Comparison

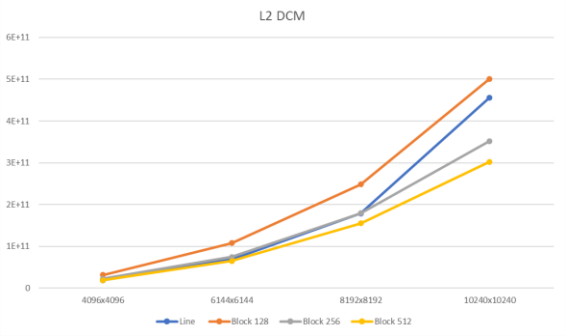

Fig.11 - L2 DCM Comparison

4. Conclusions

On the normal multiplication mode, the way the values are accessed is not as efficient as in the line multiplication one because through line multiplication the values are stored in memory by the order we want to access them later, while in the normal multiplication mode an even more extensive search is required.

Java is slower because the code must first be interpreted during run-time. C++ is compiled to binaries, so it runs immediately and therefore faster than Java programs. From the Fig.6 we can conclude that in Java Language the LineMul algorithm is more efficient since the computer executes substantially more floating-point operations per second in comparison to the NormalMul algorithm, the reason behind this is that the access to the elements of the matrices in memory is done way faster in the LineMul algorithm.

We can also conclude that block multiplication is more efficient than line multiplication because the matrix is divided in blocks with an arbitrary size and if that size is equal to the cache memory size, the access to the elements is done way faster because there's no need to search for the elements in the main memory. This isn't possible if the size of the blocks is way bigger than the cache memory size.

5. Attachement

- Normal C++

Normal	600x600	1000x1000	1400x1400	1800x1800	2200x2200	2600x2600	3000x3000
TIME	0,3076	1,7693	5,054	14,319	49,951	115,079	189,886

- Line C++

Normal	600x600	1000x1000	1400x1400	1800x1800	2200x2200	2600x2600	3000x3000
TIME	0,121	0,549	1,555	2,319	6,339	11,32	18,038

- Normal Java

Normal	600x600	1000x1000	1400x1400	1800x1800	2200x2200	2600x2600	3000x3000
TIME	0,307	1,759	5,054	14,319	46,912	111,079	189,866

- Line Java

Normal	600x600	1000x1000	1400x1400	1800x1800	2200x2200	2600x2600	3000x3000
TIME	0,14	0,716	1,937	4,102	7,758	13,26	20,715

- Line C ++

Line Multiplication	4096x4096	6144x6144	8192x8192	10240x10240
TIME	46,972	161,68	395,767	775,763
L1 DCM	27231636841	91493008152	2,15607E+11	4,20166E+11
L2 DCM	18877008833	68619688086	1,78873E+11	4,56009E+11

- Block Mul C++

BLOCK Multiplication	4096x4096			6144x6144			8192x8192			10240x10240		
	128	256	512	128	256	512	128	256	512	128	256	512
TIME	37,156	34,266	34,312	125,547	125,094	139,906	301,891	276,219	421,482	595,641	543,438	793,054
L1 DCM	9743486132	9100403793	8778944157	32886538944	30719039929	29710923755	78169767638	71870837025	7027034432	5,00576E+11	1,42254E+11	1,37426E+11
L2 DCM	31601286776	22769331175	18796033972	1,07972E+11	74674805497	64829443622	2,48487E+11	1,79606E+11	1,55158E+11	1,52204E+11	3,5217E+11	3,02384E+11