Parallel and Distributed Computing

Report - Performance evaluation of a single core

Catarina Gonçalves (up201906638)
Mariana Monteiro (up202003480)
Vasco Gomes (up201906617)

March, 2022

Introduction

In this report, we analyse the impact that the memory hierarchy has in a processor's performance, in particular when the memory access involves reading a considerable amount of data. It is also worth noting the importance of optimizing the code, with the aim of having the least possible number of cache failures in the processor.

To implement the matrix multiplication algorithms, we use the C++ and Java languages. As for the in-memory storage of multi-dimensional arrays, in C++ these are stored in row-major order, with all matrix elements being stored in positions contiguous memory. On the other hand, in Java these are stored in memory blocks unrelated.

Problem Description

The problem in analysis on this report is the matrix multiplication, more specifically square matrices. Given two matrices A and B of size $n \times n$, the result of the multiplication of these two should be a matrix C of size $n \times n$, defined as:

$$C_{i,j} = \sum_{k=0}^{n-1} A_{i,k} \cdot B_{k,j}$$

Algorithms Explanation

In these section, the algorithms used for solving the proposed problem are analysed and explained. Furthermore, it's important to note that all algorithms have $O(n^3)$ complexity and, since each calculation inside the loops requires 2 FLOP, all of them execute $2n^3$ FLOP in total.

Column Multiplication

The implementation of Column Multiplication on this problem leads to an algorithm that multiplies each row of matrix A for each column of matrix B, in this way it's computing the value of the cells of matrix C one at a time.

```
1     for(i=0; i<m_ar; i++)
2     {
3       for(j=0; j<m_br; j++)
4       {
5         temp = 0;
6         for(k=0; k<m_ar; k++)
7         {
8             temp += pha[i*m_ar+k] * phb[k*m_br+j];
9         }
10        phc[i*m_ar+j]=temp;
11      }
12    }
```

Figure 1: Column multiplication code snippet.

Therefore, in just one iteration, the row i of the matrix A and every element of the matrix B are accessed and a row of matrix C is calculated and written to memory.

Line Multiplication

A more efficient implementation of the matrix multiplication algorithm is the line by line. This algorithm doesn't follow the steps performed in algebraic multiplication, so it takes advantage of the processor's architecture and way of working.

```
1     for(i=0; i<m_ar; i++)
2     {
3       for(k=0; k<m_ar; k++)
4       {
5         for(j=0; j<m_br; j++)
6         {
7           phc[i*m_ar+j] += pha[i*m_ar+k] * phb[k*m_br+j];
8         }
9       }
10    }
```

Figure 2: Line multiplication code snippet.

This algorithm assumes that the elements of matrix C are initialized to 0 and consists of successively multiplying elements of a row of matrix A by all rows of matrix B. Using this method, each value of matrix C is not computed at once, but one line is gradually filled, then the next and so on.

Block Multiplication

The algorithm of Block Multiplication divides the matrix into smaller blocks, performing the calculations contained within that sub-matrix. In this way, the multiplication can be scaled to large size matrices and the performance is tuned, finding an optimal block size for the processor in use.

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

Figure 3: Block multiplication code snippet.

As we can see, the Line Multiplication algorithm was used for each block-wise multiplication, however we introduced 3 outer loops in order to pass through the blocks and their sum parcels.

Performance Metrics

In order to do the necessary experiments, the algorithms were implemented in two different languages: C++ and Java and tested with different sized matrices. For the algorithm of Block Multiplication, for each size, multiple block sizes were used. For all experiments, the execution time was recorded, as well as cache performance in the C++ version with the PAPI library, namely Data Cache Misses (DCM) on L1 and L2 caches. The performance metrics chosen to compare different algorithms in different languages with different sized matrices were DCM per FLOP and FLOP per sec.

$$DCM/FLOP = \frac{DataCacheMisses(DCM)}{FLOP} = \frac{DCM}{2n^3}$$

$$FLOP/sec = \frac{FLOP}{ExecutionTime(sec)} = \frac{2n^3}{ExecutionTime(sec)}$$

All experiments were tested on a computer running the Ubuntu Linux distribution, with an Intel Core i7-9700 3.00GHz processor with 8 cores.

Results and Analysis

Figures 4 and 5 represent respectively the number of FLOP/s of the first and second algorithms in Java and C++ and the number of FLOP/s of line multiplication and block multiplication with different block sizes, 128, 256 and 512. We also vary the size of the matrix with values between 600 and 3000 for figure 4 and from 4096 to 10240 for figure 5.

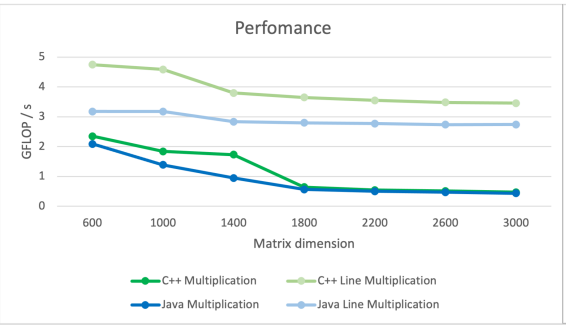

Figure 4: Performance comparison between C++ and Java.

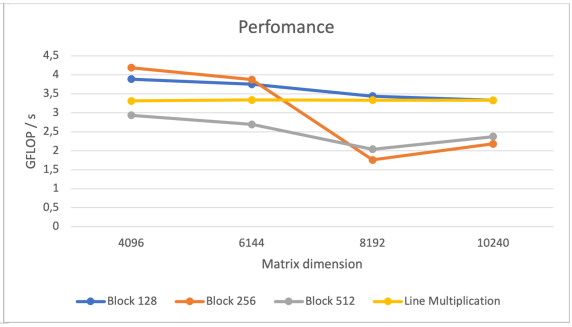

Figure 5: Performance comparison between Line and Block Multiplication.

As the results in Figure 4 show, the C++ algorithm has better performance than the Java version of the same algorithm. These results were expected because the just-in-time (JIT) compiler of the Java runtime system doesn't spend a lot of time optimizing the inner loops, and this hurts numerical code. Also, in Java there must be an index check on every array access, which requires executing extra instructions, further slowing down the computation.

In Figure 5, we can see that the performance of Block Multiplication algorithm where block size is 128 is better than others block sizes and the Line Multiplication. On the other hand, Block Multiplication algorithm with 256 and 512 block sizes generally has lower performance than Line Multiplication algorithm.

As we can see in the results, the second and third algorithms performed better than the first. The performance difference is more noticeable the larger the matrix size. The results between Line and Block Multiplication with larger dimensions were unexpected because we thought that the third algorithm would be overall better. Although the Block algorithm had the best performance values, it also had the worst values, while the Line algorithm had a more consistent performance across different sized matrices.

Figure 6 shows that the frequency of data cache misses on L1 level is lower for any block size in the Block Multiplication algorithm than in the Line Multiplication algorithm.

On the other hand, Figure 7 shows that the frequency of data cache misses for the L2 level is lower for Line Multiplication algorithm than in the Block Multiplication algorithm. Furthermore, the larger the size of the blocks in the Block Multiplication algorithm, the greater the frequency of data cache misses.

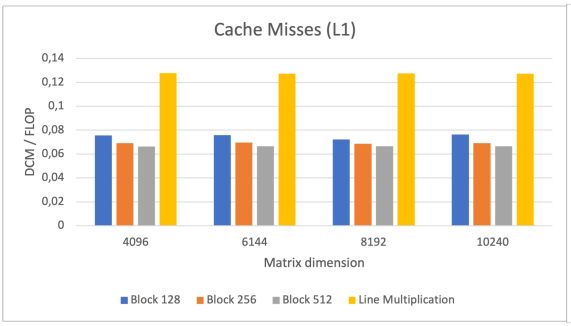

Figure 6: DCM/FLOP for L1.

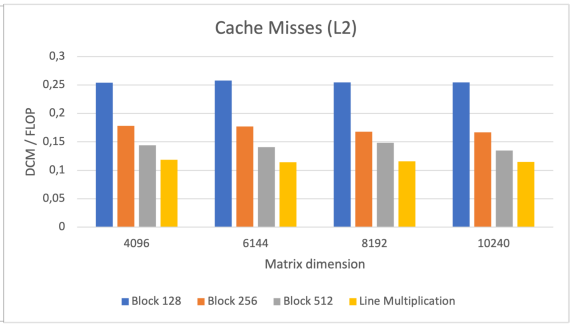

Figure 7: DCM/FLOP for L2.

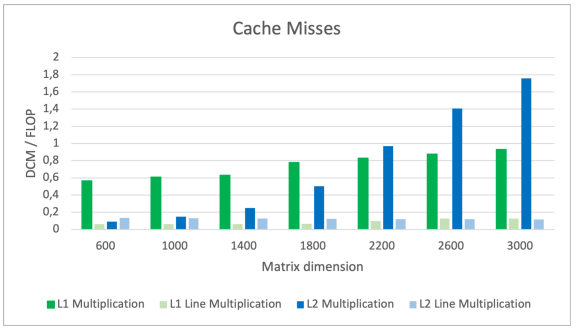

Figure 8: DCM/FLOP comparison between First and Second algorithm

In Figure 8 we can see that the frequency of data cache misses for the two levels is lower in the first algorithm (Multiplication) than in the second algorithm (Line Multiplication). This results are expected because the second algorithm has to access memory less often that the first.

A surprising fact is the value of L2 cache misses being higher than L1 misses in larger matrices, which was unexpected, since L1 is usually accessed first.

Conclusions

This project made it possible to understand the importance of taking into account the internal functioning of the processor, namely the temporal and spatial location of the cache, in the implementation of an algorithm. This prior knowledge allows us to make small changes to an algorithm, optimise it, and greatly improve its performance. This project also allowed us to study the impact of the memory hierarchy in two different programming languages, as well as their performance.