Computação Paralela e Distribuída

Performance evaluation of a single core

Turma 2 - Grupo 9

Ana Rita Antunes Ramada - up201904565
Luísa Maria Pereira Araújo - up201904996
Maria Sofia Diogo Figueiredo - up201904675

Ano Letivo: 2021/2022

Conteúdo

1. Introduction 2
2. Algorithms Explanation 2
2.1 The Naive Matrix Multiplication Algorithm 2
2.2 Line Matrix Multiplication Algorithm 2
2.3 Block matrix multiplication Algorithm 3
3. Performance Metrics 3
3.1 Time 3
3.2 L1 DCM vs L2 DCM 3
4. Speed of different algorithms in C++ 4
5. Data Cache Misses with different algorithms 5
6. Speed of different algorithms in C++ vs Java 6
7. Conclusions 6

1. Introduction

In this project, we will use the product of two matrices to study the effect on the processor performance of the memory hierarchy, when accessing large amounts of data.

For this study, three different matrix multiplication algorithms were implemented, using the C++ language, and each performance was analysed. The first two were also implemented in Java, so that we can recognize that the performance tendency is not exclusive to C++ programs.

2. Algorithms Explanation

We'll discuss three matrix multiplication algorithms: the naive, line and block matrix multiplication.

2.1 The Naive Matrix Multiplication Algorithm

For this algorithm, it's directly applied the mathematical definition of matrix multiplication, ending up with three nested loops and a total time complexity of $O(N^3)$. Here is the resulting pseudocode:

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      C[i,j] = C[i,j] + A[i,k] * B [k, j]
    end for
  end for
end for
```

2.2 Line Matrix Multiplication Algorithm

For the second algorithm, it was implemented a version that multiplies an element from the first matrix by the correspondent line of the second matrix. This results in a pseudocode similar to the last one, but where the second and last loops have been swapped.

```
for (i = 0; i < n; i++)
  for (k = 0; k < n; k++)
    for (j = 0; j < n; j++)
      C[i,j] = C[i,j] + A[i,k] * B [k, j]
    end for
  end for
end for
```

2.3 Block matrix multiplication Algorithm

The third alternative is a divide-and-conquer approach of the problem, that partitions the initial matrix in smaller ones, of size equal to *blockSize.* The result will now consist of multiplications of pairs of submatrices, followed by an addition step. The pseudocode will look like this:

```
for (ii = 0; ii < n; ii += blockSize)
  for (jj = 0; jj < n; jj += blockSize)
    for (kk = 0; kk < n; kk += blockSize)
      for (i = 0; i < blockSize; i++)
        for (k = 0; k < blockSize; k++)
          for (j = 0; j < blockSize; j++)
            C[i,j] = C[i,j] + A[i,k] * B[k,j]
          end for
        end for
      end for
    end for
  end for
end for
```

3. Performance Metrics

3.1 Time

Time is going to be the main factor we are going to look into when it comes to classifying each algorithm's performance.

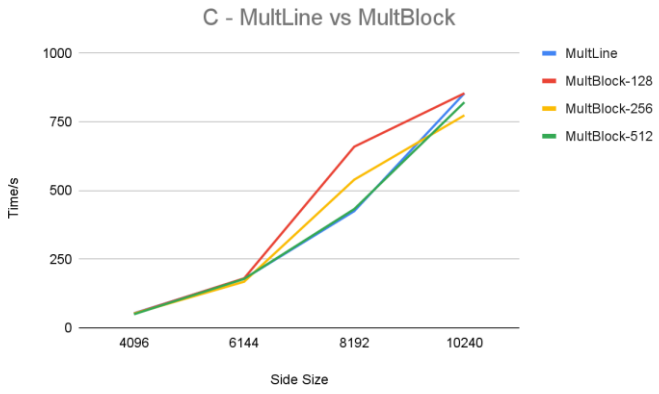
3.2 L1 DCM vs L2 DCM

Further, we are comparing how many cache misses occur with different caches: L1 DCM and L2 DCM. L1 caches are faster than L2 caches but L2 caches have a higher dimension than L1 caches.

4. Speed of different algorithms in C++

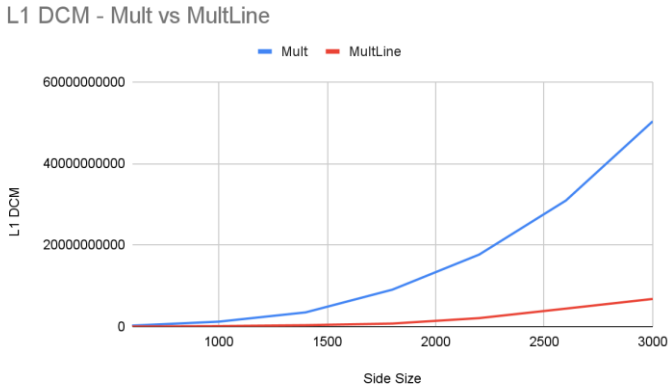
The graph below shows the amount of time that the different algorithms take to execute. Here, Naive Multiplication and Line Multiplication are compared. We can conclude that Line Multiplication is significantly faster. This can be explained by the fact that the Line Multiplication algorithm takes advantage of the values that are already on the cache. In contrast, Naive Multiplication does not do this and, every iteration, reloads an entire line of values from the matrix to the cache. This leads to fewer cache misses in the Line Multiplication, making it significantly faster.

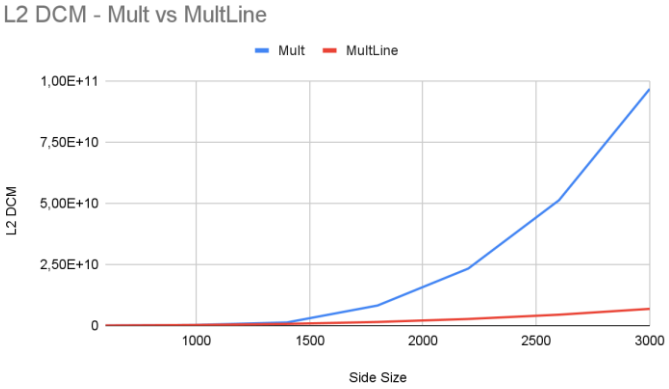
The following graph shows the difference between Line Multiplication and Block Multiplication. Block Multiplication also takes advantage of the values already loaded onto the cache and leads to execution times that are comparable to those of the Line Multiplication algorithm.



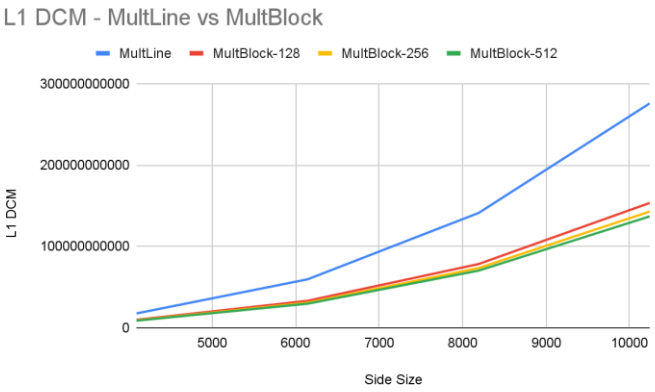
5. Data Cache Misses with different algorithms

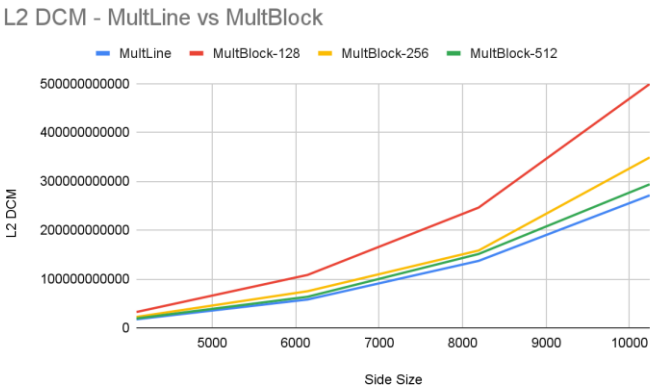
In the graphs below, Naive Multiplication and Line Multiplication are compared in terms of L1 and L2 DCM, Data Cache Misses. It's clear that Line Multiplication leads to less DCM. This happens because of the use of the data already present on the cache. As explained in 4. Speed of different algorithms in C++, with Naive Multiplication at every iteration, a new line is being loaded onto the cache. This leads to more data cache misses when compared to the Line Multiplication algorithm, which loads each line less times onto the cache.





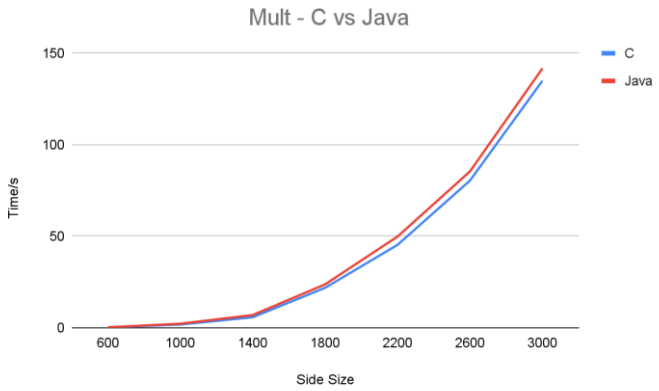
In the graphs below, Line Multiplication and Block Multiplication are compared in terms of L1 and L2 DCM, Data Cache Misses. It's clear that Block Multiplication leads to less L1 DCM.



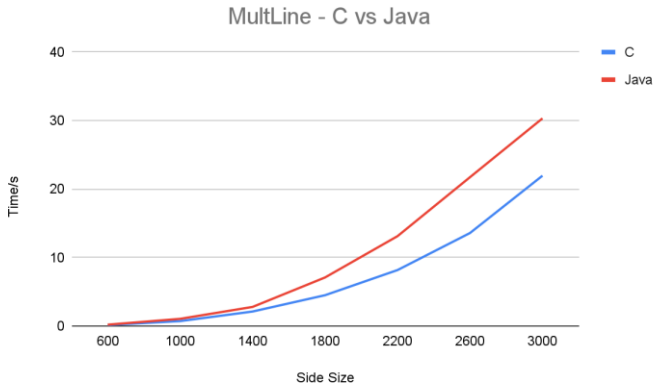


6. Speed of different algorithms in C++ vs Java

Here, Naive Multiplication in C++ and Java are compared. As we can see from the graph below, C++ is slightly faster, but comparable to Java.



The following graph compares Line Multiplication in C++ vs Java. The difference in execution time increases when compared to normal Multiplication, while C++ is still faster.



7. Conclusions

In this project, the main goal was to compare different matrix multiplication algorithms and their efficiency in different types of caches (L1 and L2), written in two different programming languages.

As expected, we obtained different results. When it comes to performance, Line Multiplication is similar to Block Multiplication, but it is much faster than Naive Multiplication.

Comparing the results obtained in C and Java, a significant difference can be noticed in the speed of both algorithms, Naive Multiplication and Line Multiplication, being faster when implemented in C.