Faculty of Engineering of the University of Porto

Evaluation of the performance of a single core

Parallel and Distributed Computing

T07G13
André Lima - up202008169@edu.fe.up.pt
Jorge Sousa - up202006140@edu.fe.up.pt
Mariana Lobão - up202004260@edu.fe.up.pt

March 10th, 2023

Problem Description

In this work, we are asked to evaluate the performance of a single core of a CPU when executing different algorithms and understand why differences in performance and execution time occur. We will be using three different algorithms for matrix multiplication to demonstrate these differences and a single CPU core to execute all algorithms. L1 and L2 Data Cache statistics from the CPU core will be extracted using PAPI (Performance API). It's important to note that this process will be done in both C++ and Java, to discriminate whether the language used affects the performance of said algorithms.

I. Algorithms Used

1. L×C Matrix Multiplication

The Line×Column (L×C) Matrix Multiplication algorithm multiplies the lines of matrix A with the columns of matrix B. This algorithm does only one write per position in the resulting matrix, as seen below:

```
for (int i = 0; i < m_size; i++) {
    for (int j = 0; j < m_size; j++) {
        double temp = 0;

        for (int k = 0; k < m_size; k++) {
            temp += m_a[i][k] * m_b[k][j];
        }

        m_c[i][j] = temp;
    }
}
```

2. L×L Matrix Multiplication

The Line×Line (L×L) Matrix Multiplication algorithm multiplies the lines of matrix A with the lines of matrix B. For that to work, however, each cell in the resulting matrix needs to be written to N times, where N is the width of the matrix.

```
for(int i=0; i<m_size; i++) {
    for(int j=0; j<m_size; j++) {
        for(int k=0; k<m_size; k++) {
            m_c[i][k] += m_a[i][j] * m_b[j][k];
        }
    }
}
```

3. Block Matrix Multiplication

In the Block Matrix Multiplication algorithm, all matrices are divided into blocks of a given size. The matrices and the corresponding blocks are then multiplied together using the Line×Line Matrix Multiplication algorithm.

```
for (int i = 0; i < m_size; i += bkSize) {
    for (int j = 0; j < m_size; j += bkSize) {
        for (int k = 0; k < m_size; k += bkSize) {
            for (int a = 0; a < bkSize; a++) {
                for (int b = 0; b < bkSize; b++) {
                    for (int c = 0; c < bkSize; c++) {
                        m_c[a + i][c + k] += m_a[a + i][b + j] * m_b[b + j][c + k];
                    }
                }
            }
        }
    }
}
```

II. Performance Metrics

To compare the performance of the different algorithms, we measured the time that each algorithm took to multiply two square matrices of various sizes, as well as the number of L1 and L2 Data Cache Misses, using PAPI. The size of the matrices was gradually increased.

1. The L×C and L×L algorithms were tested on matrices with sizes starting at 600×600 and

ending at 3000×3000, increasing by 400×400 in each iteration. This test was both performed in Java and in C++.

2. The C++ L×L implementation was also tested on matrices with sizes starting at 4096×4096 and ending at 10240×10240, increasing by 2048×2048 in each iteration. The Block Matrix Multiplication algorithm was also subjected to the same tests, but with block sizes varying from 64 to 1024 in each test.

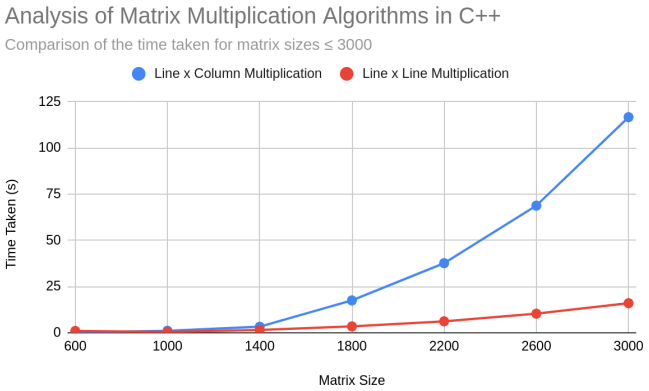
III. Notes

It is important to note that the values PAPI returns for the counters most likely aren't the absolute number of cache misses for the program's execution, since the returned values are always in the order of the billions. Instead, they're most likely related to the number of cache misses from all the programs executing during the tests' window of time. We tried our best to minimize the number of programs running when measuring our results, in order to get the most accurate number of cache misses possible.

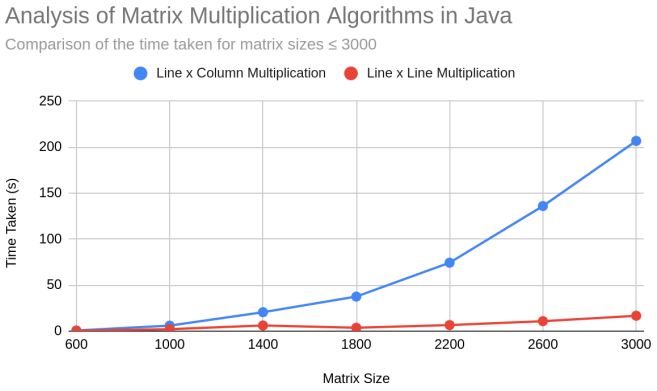
In our results regarding the Block Matrix Multiplication algorithm, we use "Block Multiplication (worst)" to identify the block size for which the Block Matrix Multiplication algorithm performed the worst. On the other hand, we use "Block Matrix Multiplication (best)" to identify the block size for which the Block Matrix Multiplication performed the best.

IV. Results and Analysis

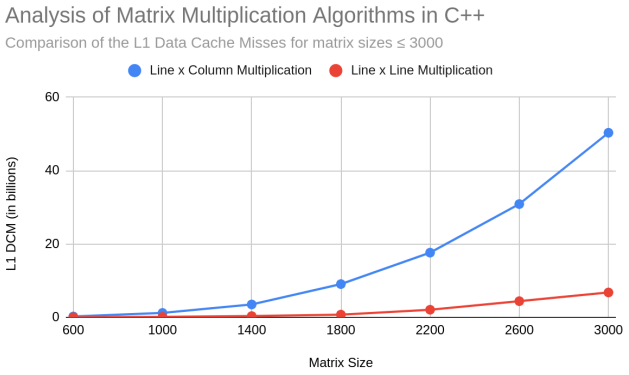
1. L×C vs. L×L

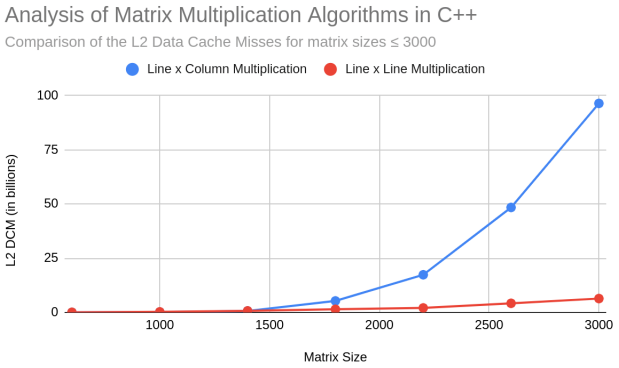


Our first test revealed that the Line×Line multiplication algorithm is faster than the Line×Column algorithm for almost every matrix size (except for 600×600). To verify that these results were not specific to C++, we executed the same tests in Java.



Even in languages other than C++, such as Java, the L×L multiplication algorithm is consistently faster than the L×C algorithm. With the help of PAPI, we were able to understand that the reason behind these results is that the L×L uses the L1 and L2 Data Caches in a more efficient way.





Our results showed that the number of cache misses in the L×L algorithm is significantly lower than that of the L×C algorithm.

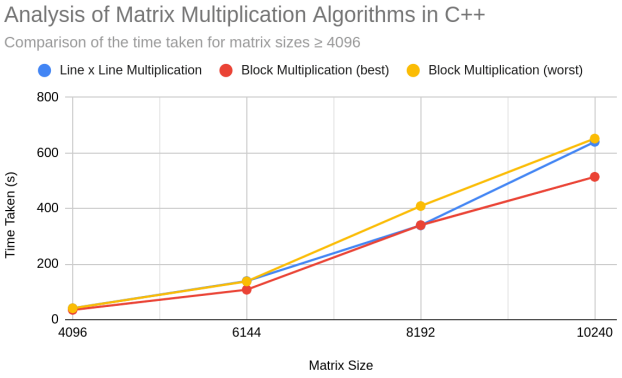
An explanation for this is as follows: the matrices in our program were stored in memory using row-major order - that is, numbers in the same row were stored contiguously in memory. Furthermore, when the value stored in a memory address is requested and the value is not present in the CPU cache (cache miss), instead of only loading the requested position into the cache, a contiguous block of memory containing that position is loaded, which generally helps performance due to spatial locality. This means that, most times, only elements from the same row (or from the neighboring rows) will be cached.

Because the L×C algorithm iterates the matrix B in a column-major order, it is very likely that accessed elements will not be cached, increasing the number of L1 and L2 Data Cache Misses and the time taken by the algorithm, since RAM accesses take a lot more time to perform than L1 or L2 Data Cache accesses. The L×L algorithm overcomes this problem by only iterating through the matrices in row-major order, which means that, most times, the requested elements will be cached in either the L1 or L2 Data Caches, leading to an increase in performance.

2. L×L vs. Block Mult.

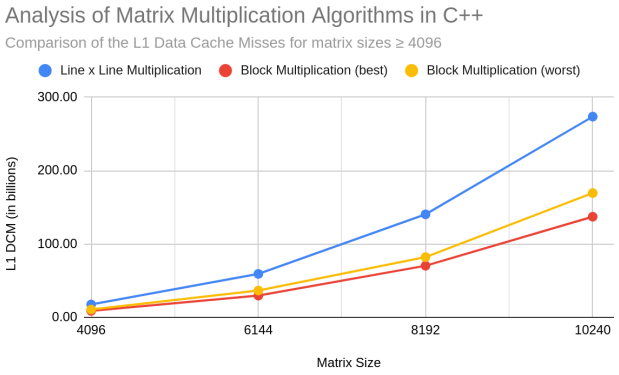
Our second test was aimed at comparing the performance of the Line×Line algorithm with the Block Matrix Multiplication algorithm.

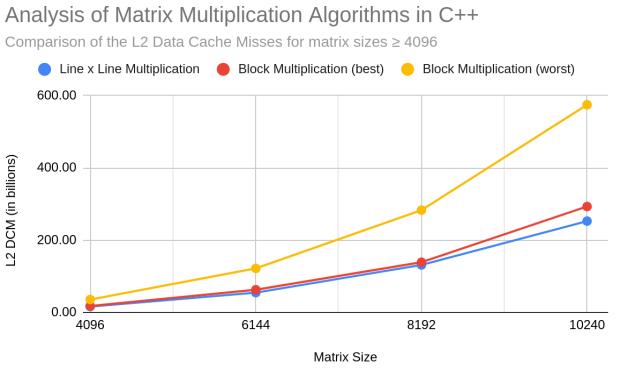
The Block Matrix Multiplication algorithm was developed to overcome the cache size limitations of the L×L algorithm for large matrices by dividing the matrices into blocks. This means that smaller matrices (blocks) will be processed for longer periods of time, which should help with the temporal and spatial locality of the program and, thus, improve its performance.



We observed that, for matrix sizes within the 4096 to 8192 range, the Block Matrix Multiplication (BMM) algorithm and the L×L algorithm performed similarly. For matrix sizes greater than 8192, we observed that the performance

of the BMM algorithm is significantly better than that of the L×L algorithm, if the block size is chosen adequately.

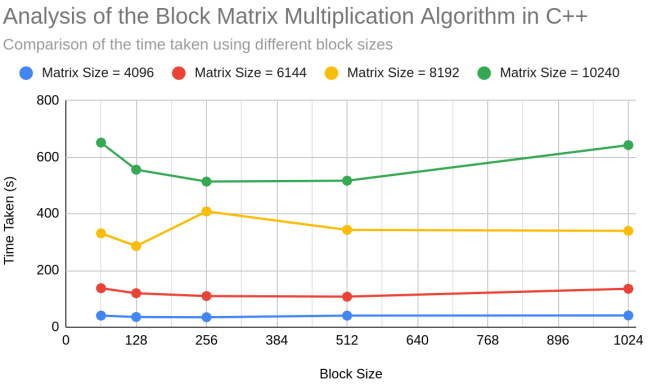




Furthermore, for every matrix size and block size tested, the BMM algorithm has fewer L1 Data Cache Misses than the L×L algorithm. On the other hand, the L×L algorithm has slightly fewer L2 Data Cache Misses than the BMM algorithm. Since the L2 Data Cache is slower to access than the L1 Data Cache, those two effects balance out for matrices with sizes up to 8192, but favor the BMM algorithm for larger matrices.

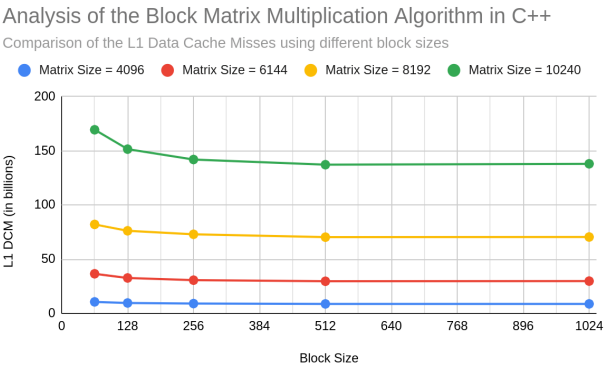
3. Different Block Sizes

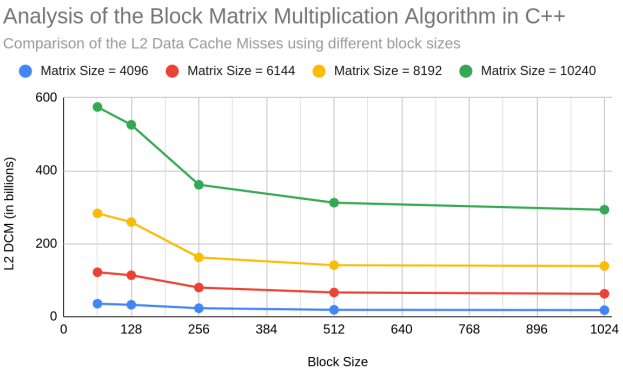
Our third test was aimed at analyzing the impact of using different block sizes in the BMM.



Analyzing this graph, we understand that, depending on the matrix size, there is an optimal block size to use. Our testing revealed that the optimal block size for the BMM algorithm, in our setup, was around 512 doubles, which is equivalent to 512 × 512 × 8B = 2MiB.

Regarding the L1 and L2 Data Cache Misses, we got similar results.





The number of L1 Data Cache Misses is the lowest for a block size of 512. On the other hand, the number of L2 Data Cache Misses is the lowest for a block of

1024 doubles. This is to be expected, since the L2 Data Cache has more capacity than the L1 Data Cache. However, as we saw in the L×L vs. BMM comparison, the L1 Data Cache has a bigger impact on the performance of the CPU, which explains why the optimal block size in our testing came out to be 512 doubles.

V. Conclusion

As we saw in the previous sections, even though an algorithm can be sped up by using a faster programming language (C++ vs. Java), it can be sped up a lot more if the algorithms use the CPU cache in a better way (L×C vs. L×L). Algorithms that take into account cache limitations, such as capacity, also perform better (L×L vs Block Mult.).

Out of three algorithms tested, we concluded that the Block Matrix Multiplication algorithm is the fastest one, if the block size is optimal. In our testing, we couldn't determine with certainty why the optimal block size is around 512 doubles, but we hypothesize it is related to the fact that, in Linux, with most CPUs, the default page size is 4KiB, which is equal to the space that a single matrix block row occupies (512 × 8B = 4KiB). This means that, when iterating over a row, most of its elements will be cached. If this hypothesis were true, it would mean that the optimal block size would be highly dependent on the CPU and OS the algorithm is being executed on.

Finally, more testing would need to be done (with different page sizes, for instance), in order to determine the factors that impact the optimal block size.

Overall, the project was a great learning experience and allowed us to learn more about the cache behavior and the impact that it has on software performance. We also learned that software and hardware need to be thought of as a whole when designing performant systems.

Index