Performance of cache-aware matrix multiplication optimizations

Report

Mestrado Integrado em Engenharia Informática e Computação

Parallel Computing

Authors:
David Luís Dias da Silva - up201705373@fe.up.pt
Luís Pedro Pereira Lopes Mascarenhas Cunha - up201706736@fe.up.pt

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

March, 2021

1 Introduction

Matrix multiplication is an ubiquitous algorithm throughout computer science, found in many fields, from Computer Graphics ([Foley, 1994], [Szeliski, 2020]) to Machine Learning ([Mueller and Massaron, 2019]).

Due to the large volume of multiplications that need to be made and the large size of matrices used, these operations need to be done quickly. Adding this to the fact that the multiplication algorithm is "embarrassingly parallel", current techniques for speeding up the process use GPUs, which excel in massive parallel computations.

Regardless of this trend, there are still techniques that can be applied to sequential matrix multiplication algorithms in order to achieve better performance. These techniques use different operation ordering in order to account for the CPU's cache, lowering data cache miss rates. These techniques are especially relevant as this is an operation which is often applied to very large matrices, and designing an algorithm that performs well for large inputs — capable of sustained performance — is crucial.

In this report we present some of these techniques, comparing execution times and cache miss rates across multiple programming languages. We start by reviewing some of the concepts behind matrix multiplication and the CPU cache. Then we experiment with different algorithms, comparing and analysing them by measuring several metrics and their evolution across different problem sizes.

2 Background

2.1 Matrix multiplication

Let A be any $m \cdot p$ matrix, and let B be any $p \cdot n$ matrix, say:

$$A = (a_{ij})_{i,j=1}^{m,p} \quad \text{and} \quad B = (b_{ij})_{i,j=1}^{p,n} \tag{1}$$

Then the product $A \cdot B$ is defined to be the $m \cdot n$ matrix $C = (c_{ij})$ whose ij-entry is given by ([Apostol, 2003]):

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj} \tag{2}$$

As such, to perform a matrix multiplication between two square matrices of size n, $2n^3$ floating-point operations are required. The number of operations can be easily identified by looking at Source Code 1, which consists of a direct translation of Equation 2. In the innermost loop, two floating-point operations are performed (one addition and one multiplication). The body of the loop is executed n^3 times, since it is inside three nested loops, each with n iterations.

2.2 Cache

The CPU cache is a hardware component used to reduce memory access times. It works by keeping a small, high-speed memory (typically $SRAM$) near the CPU, allowing for faster memory accesses than main memory (using $DRAM$). In 2020, typical access times for $SRAM$ ranged from 0.5-2.5ns whereas $DRAM$ access times spanned 50-70ns ([Patterson and Hennessy, 2021]).

Although several times smaller than main memory, cache relies on the *principle of locality* to achieve good performance. *Temporal locality* states that if an item is referenced, it will tend to be referenced again soon. Similarly, *spatial locality* states that if an item is referenced, items whose addresses are close by will tend to be referenced soon [Patterson and Hennessy, 2021].

Modern CPU architectures typically have several levels of cache (normally three). *L2* cache, i.e. the second level of cache, is larger but slower than *L1* cache. A level is accessed when the data can't be found on the previous level (*cache miss*).

3 Algorithms

We analyse three matrix multiplication algorithms, which are described in this section.

3.1 Row x Column

The first algorithm is a naive approach which consists of a direct translation of Equation (2). We compute each element c_{ij} of the resulting matrix by computing the dot product between the row i of matrix A and the column j of matrix B.

This algorithm, presented in Source Code 1, has a $O(n^3)$ time complexity.

3.2 Row x Row

The second algorithm (Source Code 2) is a variation of the first, in which we multiply the elements of each row of the first matrix with the elements of each row of the second matrix sequentially, instead of multiplying rows of the first matrix with columns of the second matrix.

In this algorithm, the elements c_{ij} are computed incrementally rather than individually. The temporary result of each value is stored in the corresponding element of the result matrix.

We can see that the operations performed to compute the final result are exactly the same as in the previously described algorithm. The only difference is the order in which they are performed. Instead of computing each dot product at once, the term of the dot product corresponding to the rows being multiplied is computed for the full row of matrix B and added to the corresponding temporary value of the result matrix.

3.3 Block-based

The third algorithm (Source Code 3) consists of subdividing each matrix in blocks of the same size, and following the algorithm described in the previous section at two levels. In the outer level, each sub-matrix is seen as a matrix element, and the they are paired following the same order of the matrix multiplication algorithm described in the previous section. In the inner level, each of the paired sub-matrices are multiplied following the Row x Row algorithm. Again, the operations performed are the same as in the previous algorithms. The difference lies in the order in which the operations are performed, in an attempt to improve the reuse of cached data and reduce cache misses.

4 Experiments

4.1 Setup

We ran the three algorithms for square matrices with increasing side-length and different block sizes (when applicable). The Row x Column and Row x Row algorithms were ran for C++ and Java in order to test if the results are language-dependent.

We ran each experiment 3 times and calculated the mean of the time values and cache misses [1]. Measuring the cache miss values both for L1 and L2 levels, helps us explore if the different algorithms make a good use of the locality principles, and if this is the most determining (or only) factor of the algorithms' performance. We also computed the performance (in Gflops) following the Equation 3, being $2n^3$ the number of floating point operations required to compute the matrix multiplication between square matrices of size n. By measuring the performance we are able to better compare the algorithms since we eliminate the inherent increase in time measurements that arises from the increase in matrix sizes. It also allows us to compare the measured performance to the ideal performance, which would be constant independently of the problem size. Besides comparing the absolute number of cache misses, we also compare the cache misses per operation — by dividing the number of cache misses by the number of operations $2n^3$ — in order to normalize the number of cache misses and better compare which algorithms have better cache performance.

In the C++ implementation time is measured using the *time* function if the *time.h* header (user time + system time). Java uses the *nanoTime* function which measures elapsed time and is not related to any other notion of system or wall-clock time.

The experiments were ran on a quad-core, Intel Core i5-6500 Processor running at a 3.20 to 3.60 GHz, with 4x32KB L1 Data Cache and a 4x256KB L2 Unified Cache.

$$performance = \frac{2n^3}{time} \tag{3}$$

4.2 Results

4.2.1 Row x Column algorithm

In the first experiment, we measured the time of the Row x Row algorithm multiplying square matrices with side-lengths from 600 to 3000, with increments of 400, in C++ and Java. We can see that the behaviour of both metrics is similar for both languages, despite C++ being slightly faster. By analysing the plots (figures 1 and 2), we can see that this algorithm does not have sustained performance, *i. e.* is not able to keep the same performance as the size of the problem data increases.

[1] Note that we were not able to measure cache misses for the Java implementations due to the unavailability of the *PAPI* library

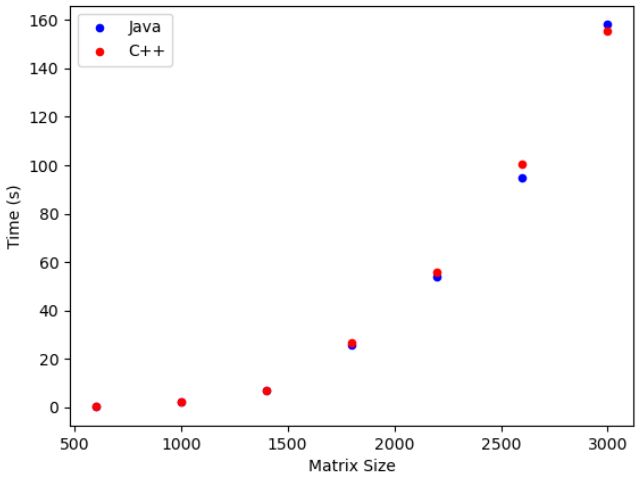

Figure 1: Time comparison (in seconds) of C++ and Java using the Row x Column algorithm (lower is better)

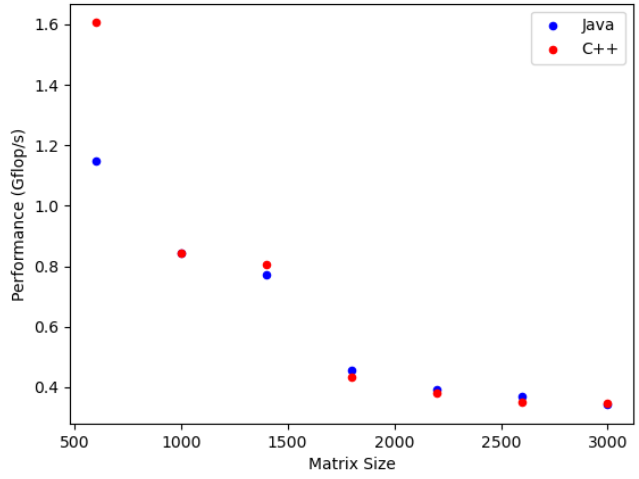

Figure 2: Performance comparison (in Gflops) of C++ and Java using the Row x Column algorithm (higher is better)

The real performance of an algorithm is a function of several factors, only one of which is the number of operations required and the theoretical complexity of the algorithm. Other factors, dependent on the computer architecture and hardware, such as I/O speed, memory hierarchy, and the algorithm's data access pattern are also relevant. In addition, according with the results, we can safely say that the problem at hand is not language dependent. In this example, the order in which the operations are executed makes a poor use of the CPU cache, which significantly degrades the algorithm's performance for larger matrices. We plotted the number of cache misses for both L1 and L2 cache (figures 3 and

4), as well as the number of cache misses relative to the number of operations of the problem by dividing the number of cache misses by the number of operations ($2n^3$ for square matrices of size n).

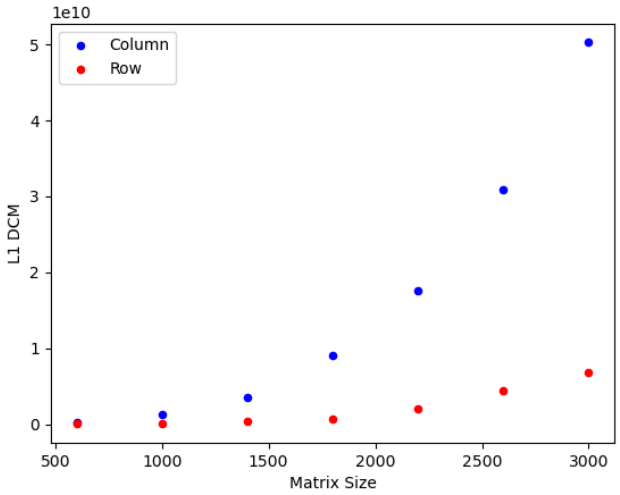

Figure 3: Number of L1 data cache misses for the C++ algorithm (lower is better). Row x Row algorithm in red and Row x Column in blue.

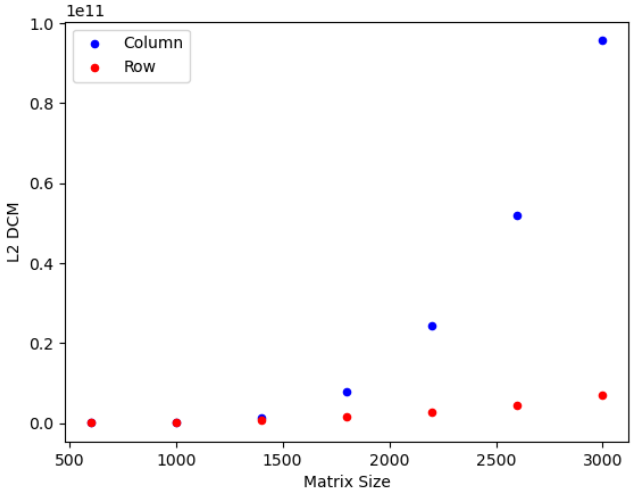

Figure 4: Number of L2 data cache misses for the C++ algorithm (lower is better). Row x Row algorithm in red and Row x Column in blue.

We can see that the number of cache misses (both L1 and L2) per operation (figures 5 and 6) increases as the problem size increases, which shows that the pattern of data access and poor cache usage is what's degrading the performance. As we can see by the representation of this access pattern in figure 7, with each iteration of the innermost loop, although the result value and first operand are

contiguous to the previous ones, the second operand is one matrix-length away [2] from the previous and, therefore, is unlikely to be in cache. This means that the Row x Column algorithm does not take advantage of spatial proximity. Also, the algorithm does not take advantage of temporal proximity, since the elements of a matrix row stored in cache are only reused after several iterations, thus the elements could have been displaced from the cache if the cache is not big enough.

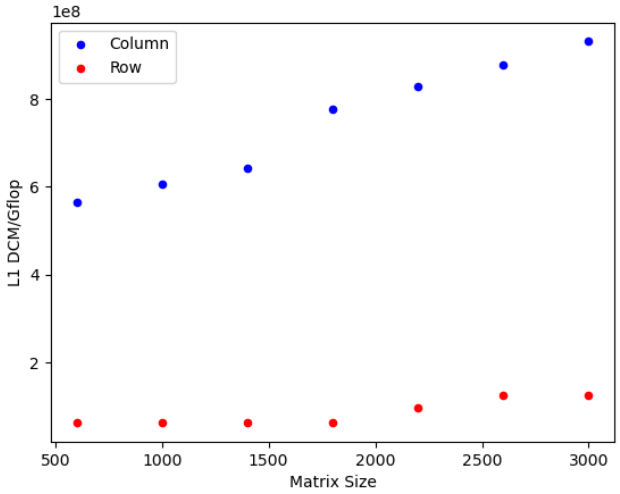

Figure 5: Number of L1 data cache misses for the C++ algorithm (lower is better) per GFLOP. Row x Row algorithm in red and Row x Column in blue.

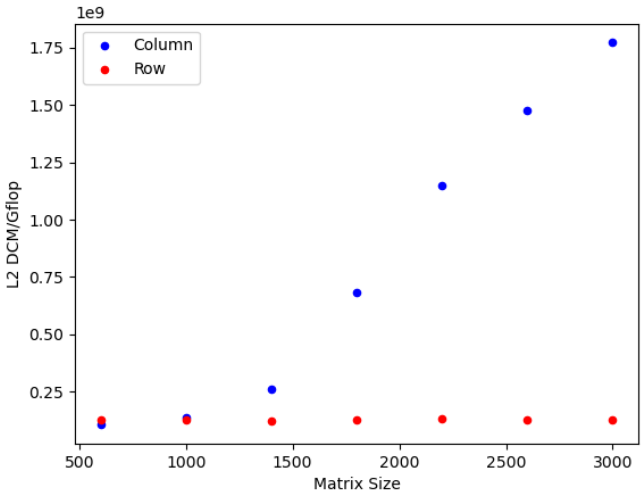

Figure 6: Number of L2 data cache misses for the C++ algorithm (lower is better) per GFLOP. Row x Row algorithm in red and Row x Column in blue.

[2]Note that although the value seems close in the matrix, it is represented in memory as a single sequence, which leads it to be one row-length apart

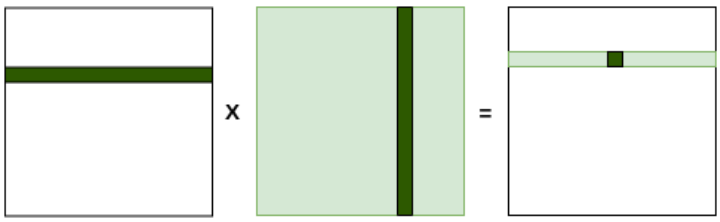

Figure 7: Matrix access pattern for Row x Column algorithm. The light-green area represents the data accessed during the j-loop. The dark-green area represents the data accessed by the inner-most loop (k-loop).

4.2.2 Row x Row algorithm

In this experiment the algorithm was run for both languages with the same ranges, and the C++ algorithm was ran further for matrices of sizes 4096 to 10240 with increments of 2048. Both languages also produced similar execution times and performance (figures 8 and 9), however, in comparison with Row x Column, this algorithm reduced the execution times significantly as can be seen in (figure 10). Furthermore, it was also able to reasonably sustain an higher performance (figure 11).

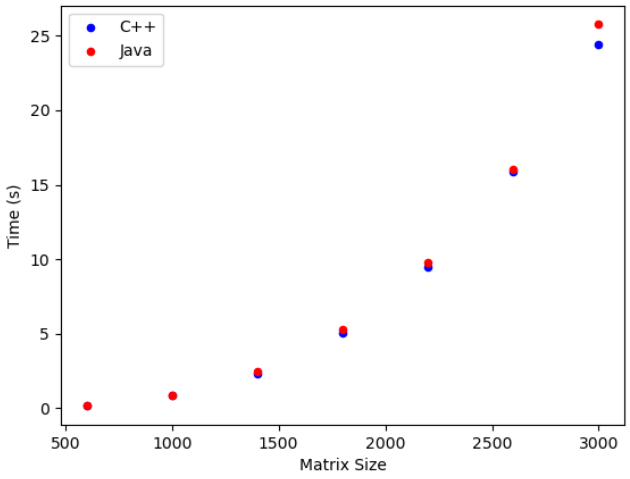

Figure 8: Time comparison (in seconds) of C++ and Java using the Row x Row algorithm (lower is better)

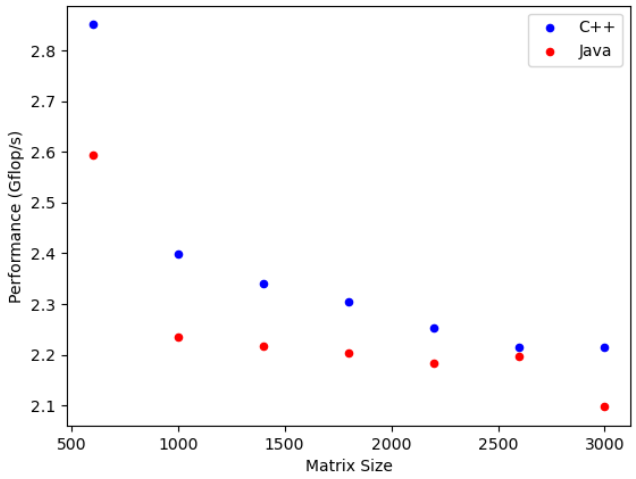

Figure 9: Performance comparison (in Gflops per second) of C++ and Java using the Row x Row algorithm (higher is better)

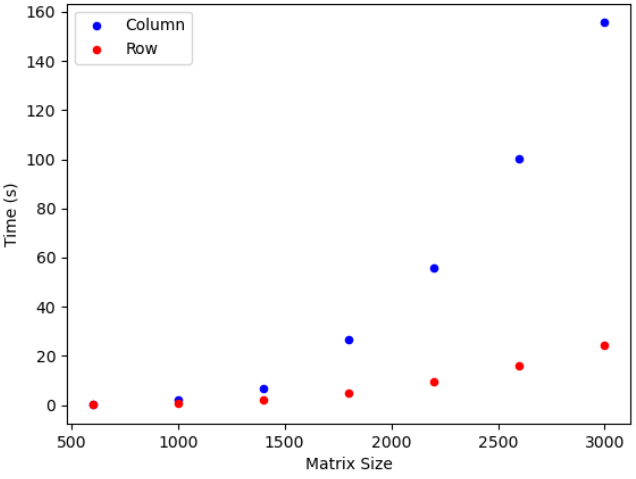

Figure 10: Time comparison (in seconds) the Row x Column and Row x Row algorithms for C++ (lower is better)

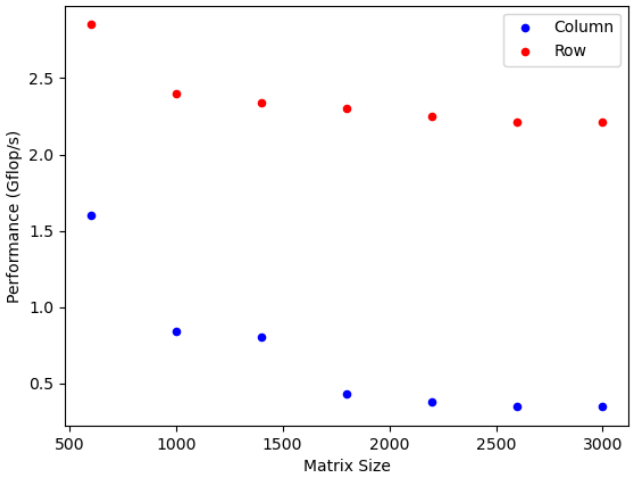

Figure 11: Performance comparison (in Gflops per second) the Row x Column and Row x Row algorithms for C++ (higher is better)

In comparison to the previous algorithm, the accesses made to the matrices are to elements that are near each other in memory (spatial locality) and that were used recently (temporal locality). Taking for example the innermost for loop and the three values that are accessed (result value, first operand and second operand), we can see that the result value and the second operand are contiguous to the ones used on the previous iteration (therefore are likely in cache) and the first operand is the same one that is used on the previous iteration (therefore is likely still in cache). This access pattern can be better seen in figure 12 is supported by the plots of the L1 (figure 5) and L2 (figure 6) data cache misses for both algorithms.

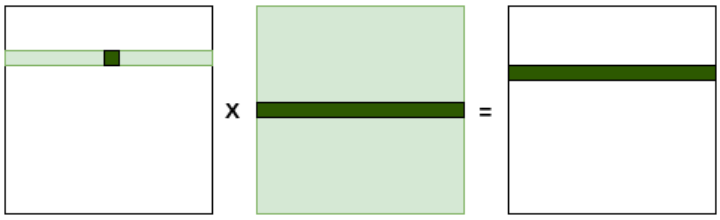

Figure 12: Matrix access pattern for Row x Row algorithm. The light-green area represents the data accessed during the k-loop. The dark-green area represents the data accessed by the inner-most loop (k-loop).

4.2.3 Block-based algorithm

The goal of this algorithm is also to improve cache reuse and reduce misses by dividing the data in blocks. Since all the operations regarding a sub matrix multiplication (multiplication between pairs of blocks) are done sequentially, if the block size is chosen in a way that allows for the needed data to be fully stored in cache, it can be more effectively reused reducing memory access. Figure 13 represents the data access pattern of the this approach which, once again, is sim-

ilar to the Row x Row algorithm however, with the sub-division of the original matrix.

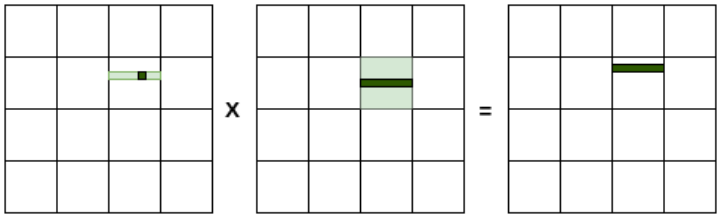

Figure 13: Matrix access pattern for the Block-based algorithm algorithm. This algorithm applies the Row x Row algorithm to sub-matrices of the original matrix.

For our experiments, we ran the block-based for matrices of sizes 4096 to 10240 with increments of 2048 and with block-sizes ranging from 64 to 1024 in powers of two. As can be seen in the figure 14, blocking produces increasingly lower times than the Row x Row algorithms. This is also reflected in the performance measurements (figure 15), however, the fact that this performance is diminishing with the increase of the matrix size shows that there are other factors that influence the algorithms' performance, rather than the algorithm itself.

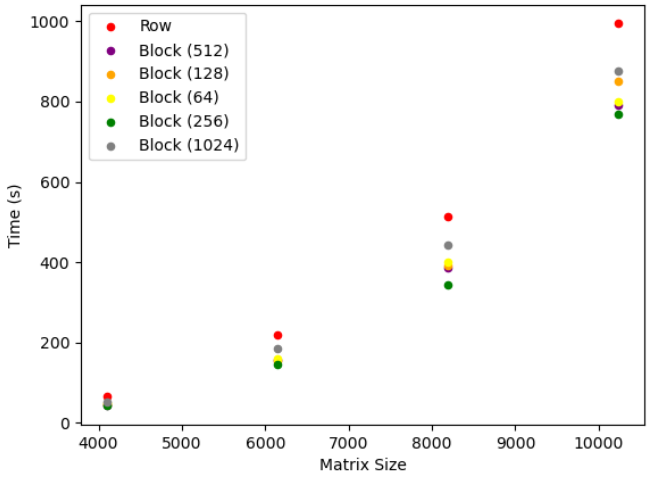

Figure 14: Time comparison (in seconds) of the Block-based algorithms for C++ (lower is better). The Row x Row algorithm is presented for comparison.

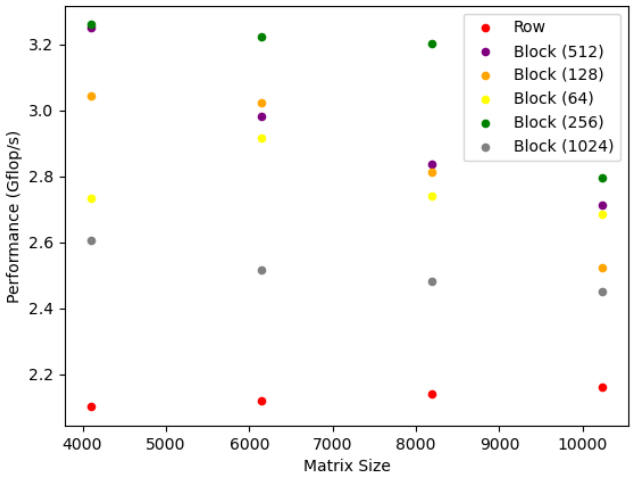

Figure 15: Performance comparison (in Gflop/s) of the Block-based algorithms for C++ (lower is better). The Row x Row algorithm is presented for comparison.

Regarding the block-based algorithm, if the choice of the ideal block-size was dependant exclusively on the cache size and on the goal of maintaining, in cache, the data needed to perform the sub-matrix multiplication, than we would expect that the performance between increasing block-sizes would show an increase in performance until the point where this data is no longer fits in cache, after which the performance would decrease [3]. Instead we can see that the performance comparison between different block-sizes changes with different matrix sizes.

In addition, the L1 and L2 data cache misses (figures 16 and 17) are much less when compared to the Row x Row algorithm, which shows that a blocking approach uses the cache more efficiently, taking advantage of the locality principles.

[3]Note that, increasing the block-size will eventually lead to the Row x Row algorithm (the blocks' size is equal to the original matrix's size).

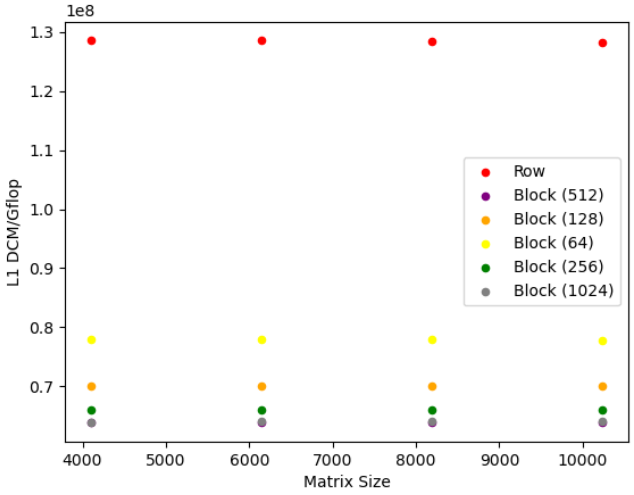

Figure 16: Number of L1 data cache misses for the C++ algorithm for the block-based algorithm (lower is better). Row x Row algorithm is shown for comparison.

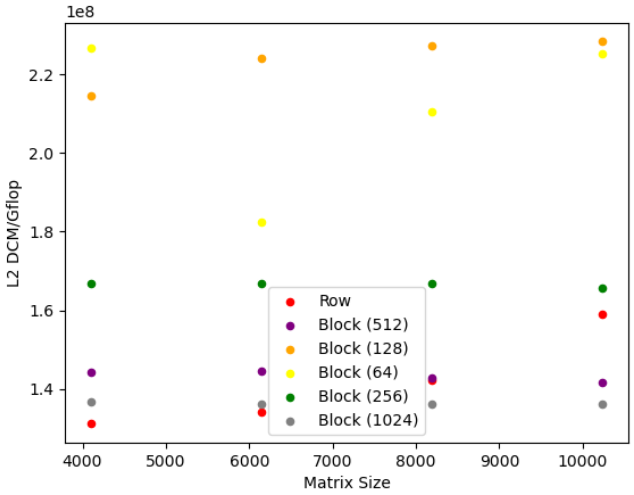

Figure 17: Number of L2 data cache misses for the C++ algorithm for the block-based algorithm (lower is better). Row x Row algorithm is shown for comparison.

We also point that the algorithm with the least cache misses is not always the one with better performance. For instance, even though using a block size of 1024 leads to less L1 and L2 cache misses when comparing against other block sizes, it is the one with least performance across all matrix sizes. This suggests that the number of cache misses is not the only factor impacting the performance. We hypothesize that other aspects outside of this work's scope, such as cache prefetching, compiler optimizations, or even limitations of the

setup in which the execution times were measured (such as the core where the process is running being shared with other processes), may be influencing the performance.

5 Conclusion

In this work we implemented and analysed different versions of the matrix multiplication algorithm taking into account the data access pattern. We verified that the different versions achieve different levels of sustained performance. While for small problems their performance is similar, as the size of the problem increases, the difference between performances grows larger and larger. We also learned that programmers must be aware of the computer architecture and memory layout while developing and implementing an algorithm, especially those that are used in applications where performance is key and those that deal with huge amounts of data. Even though most of the time it is thought that the programmer does not need to be aware of details of the lower abstraction levels, there are certain situations where it is important, because optimizations such as the ones performed in this work which regard order of operations and are not performed by the compiler. It is notorious how something as simple as switching the order of two loops could have such a massive impact on performance. In future work, we would like to better understand the relationship between performance and the block size used, as well as the optimal block size depending on the memory layout. We could also compare the algorithms using different types of caches, such as direct-mapped, fully associative or set associative caches, analysing the data that is displaced from cache due to interferences. Another aspect we could explore is to represent the matrices as two dimensional arrays and explore the influence of using languages that use row-major order or column-major order.

Bibliography

[Apostol, 2003] Apostol, T. M. (2003). *Calculus*. John Wiley & Sons.

[Foley, 1994] Foley (1994). *Introduction to computer graphics*. Addison-Wesley.

[Mueller and Massaron, 2019] Mueller, J. and Massaron, L. (2019). *Deep learning*. John Wiley & Sons, Inc.

[Patterson and Hennessy, 2021] Patterson, D. A. and Hennessy, J. L. (2021). *Computer organization and design: the hardware/software interface*. Morgan Kaufmann.

[Szeliski, 2020] Szeliski, R. (2020). *Computer Vision: algorithms and applications*. Springer Nature.

A Source code

A.1 Row x Column

```
void simpleCycle(double* op1Matrix, double* op2Matrix, double* resMatrix, int
↪  matrixSize) {
  int i, j, k, rowOffsetI;

  for (i = 0; i < matrixSize; i++) {
    rowOffsetI = i * matrixSize;
    for (j = 0; j < matrixSize; j++) {
      for (k = 0; k < matrixSize; k++) {
        resMatrix[rowOffsetI + j] += op1Matrix[rowOffsetI + k] * op2Matrix[k *
        ↪  matrixSize + j];
      }
    }
  }
}
```

Source code 1: Matrix multiplication in C (Row x Column)

A.2 Row x Row

```
void optimCycle(double* op1Matrix, double* op2Matrix, double* resMatrix, int
↪  matrixSize) {
  int i, j, k, rowOffsetI, rowOffsetK;

  for (i = 0; i < matrixSize; i++) {
    rowOffsetI = i * matrixSize;
    for (k = 0; k < matrixSize; k++) {
      rowOffsetK = k * matrixSize;
      for (j = 0; j < matrixSize; j++) {
        resMatrix[rowOffsetI + j] += op1Matrix[rowOffsetI + k] *
        ↪  op2Matrix[rowOffsetK + j];
      }
    }
  }
}
```

Source code 2: Matrix multiplication in C (Row x Row)

A.3 Block-based

```
double blockOptimCycle(double* op1Matrix, double* op2Matrix, double* resMatrix,
↪  int matrixSize, int blockSize) {

  int jj, kk, i, j, k, rowOffsetI, rowOffsetK;

  for (jj = 0; jj < matrixSize; jj = jj + blockSize)
    for (kk = 0; kk < matrixSize; kk = kk + blockSize)
```

```
7         for (i = 0; i < matrixSize; i = i + 1){
8           rowOffsetI = i * matrixSize;
9           for (k = kk; k < kk + blockSize; k = k + 1) {
10            rowOffsetK = k * matrixSize;
11            for (j = jj; j < jj + blockSize; j = j + 1)
12              resMatrix[rowOffsetI + j] += op1Matrix[rowOffsetI + k] *
                ↪  op2Matrix[rowOffsetK + j];
13          };
14        }
15  }
```

Source code 3: Matrix multiplication in C (Block-based)