Performance evaluation of a single core

CPD project 1

Nuno Costa
up201906272@edu.fe.up.pt

João Baltazar
up201905616@edu.fe.up.pt

Pedro Gonçalo Correia
up201905348@edu.fe.up.pt

March 28, 2022

Contents

1 Introduction

This project seeks to investigate the effect of the memory hierarchy on the processor performance when accessing large amounts of data. The study focuses on three different matrix multiplication algorithms and their performance, cache utilization and scalability. The results of two programming languages, C++ and Rust, were compared.

2 Algorithms

2.1 Naive algorithm (ijk)

The most direct algorithm that can be conceived for matrix multiplication is to, for each line in the first matrix and column in the second matrix, sum the element wise product of the values in the line and column.

Written in C:

```
// Square matrices of size n
// mc = ma . mb
for(size_t i=0; i<n; i++) {
    for(size_t j=0; j<n; j++) {
        temp = 0;
        for(size_t k=0; k<n; k++) {
            temp += ma[i*n+k] * mb[k*n+j];
        }
        mc[i*n + j] = temp;
    }
}
```

2.2 Line oriented algorithm (ikj)

The performance of the previous algorithm can be improved by changing the order of operations so as to leverage spatial and temporal locality of reference. The elements in the matrices are stored in memory in row-major order, which means elements in the same row of the same matrix occupy consecutive spaces in memory, while elements in the same column of the same matrix or elements from different matrices are not. With this in consideration, a more cache efficient algorithm would be to multiply each element of the first matrix by the corresponding line in the second matrix. Since the inner loop iterates over a line of a single matrix, that is, over consecutive values, the number of data cache misses will be lower. This is effectively a version of the previous algorithm with the j loop and the k loop swapped.

In C, this algorithm would be:

```
// Square matrices of size n
// mc = ma . mb
for(size_t i=0; i<n; i++)
    for(size_t k=0; k<n; k++)
        for(size_t j=0; j<n; j++)
            mc[i*n + j] += ma[i*n + k] * mb[k*n + j];
```

2.3 Block oriented algorithm

In the previous algorithm, if the matrices are large enough, a single line might not fit in cache. In this case, iterating over a line of the second matrix for each element in the first matrix will result in multiple cache misses as different portions of the line are loaded in cache. In another element of the first matrix that iterates over the same line, the same cache misses would occur.

However, the matrices can be partitioned in equally sized blocks before executing the multiplication, and the multiplication can be done on the block level before being done on the element level with the line based algorithm. This way, a line of a single block may fit in the cache, thus avoiding the cache misses that would occur in between line segments because the operations that only require that segment are grouped.

For this method to be effective, the block size must be small enough that a single line may fit in cache, but large enough that there aren't many avoidable line changes that result in cache misses.

In C, the algorithm would be:

```
// Square matrices of size n and blocks of size bsize
// mc = ma . mb
size_t bnum = n / bsize;

for(size_t ii = 0; ii < bnum; ii++)
    for(size_t kk = 0; kk < bnum; kk++)
        for(size_t jj = 0; jj < bnum; jj++)
            for (size_t i = ii*bsize; i < (ii+1)*bsize; i++)
                for (size_t k = kk*bsize; k < (kk+1)*bsize; k++)
                    for (size_t j = jj*bsize; j < (jj+1)*bsize; j++)
                        mc[i*n + j] += ma[i*n + k] * mb[k*n + j];
```

3 Performance metrics

All algorithms were implemented in C in order to measure the execution time, number of instructions and the number of data cache misses for different matrix sizes. In order to be able to measure the hardware metrics, the Performance Application Programming Interface (PAPI) was used. The measurement results were used to assess the impact of the memory hierarchy in the processor performance. L1 and L2 cache misses were tracked, as they provide good insight into how well memory in cache is being reused - less misses mean better performance, as accesses to cache are much faster than DRAM and, as such, very desirable.

The algorithms were also implemented in Rust, but for this language only the execution time was measured. The results in this other language were compared with the results in C in order to determine the impact of the choice of the programming language in the experiments.

In both languages and for every algorithm, the capacity of the system in GFLOPS was estimated based on the formula:

$$FLOPS = \frac{\#floating\_point\_operations}{t} \tag{1}$$

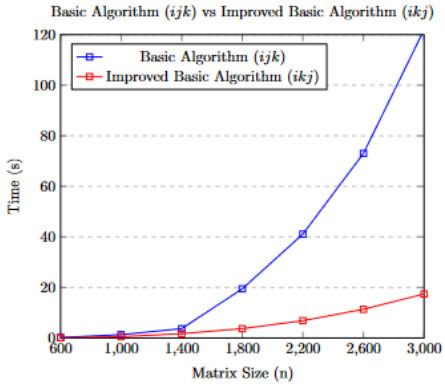
In all algorithms, $\#floating\_point\_operations = 2n^3$, since the floating point operations are the same, but in a different order.

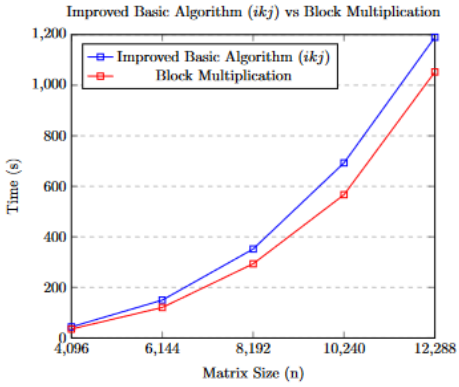
4 Results

The measurements taken were plotted in order to compare the execution time, number of cache misses, and FLOPS of the three implemented algorithms, as well as the impact on execution time of the block size in the third algorithm and the impact on execution time of the choice of the programming language in the first two algorithms.

4.1 Algorithm Comparison

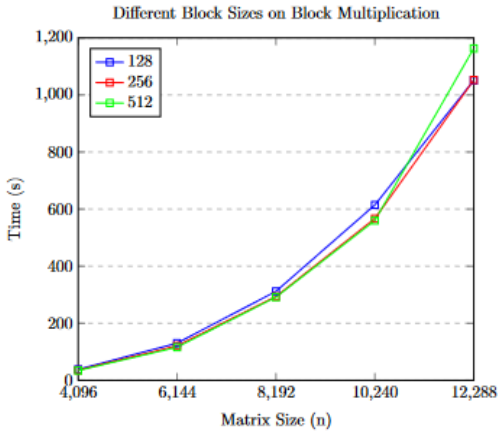
To properly compare these algorithms, the execution time of each of them for the C implementation was plotted over an increasing matrix size. For the third algorithm, the average execution time for all block sizes measured was used. As expected, the basic algorithm was slower than the other two, and the block multiplication algorithm was slightly faster than the improved basic algorithm.





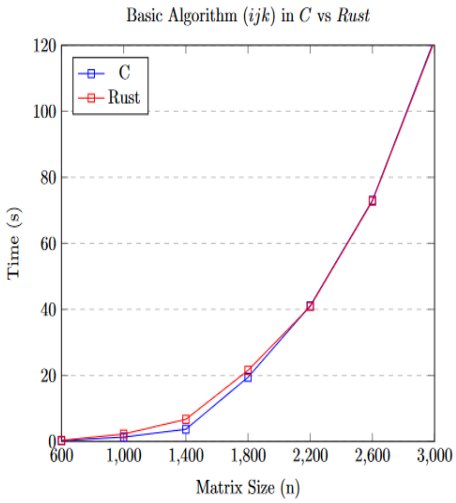
4.2 Different Block Sizes in Matrix Block Multiplication

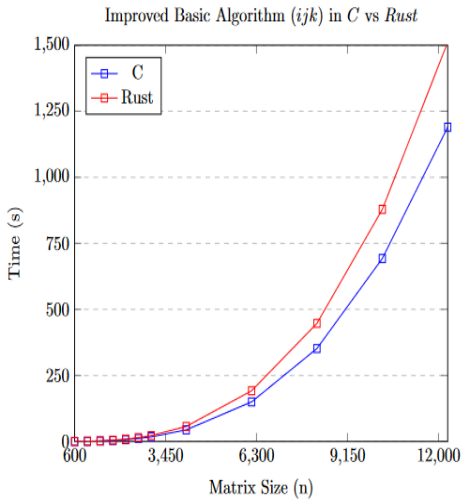
To verify how the block size affects the execution time of the Block Multiplication Algorithm, block-sizes of 128, 256 and 512 were plotted for the C implementation over a matrix size of 4096 up to 12288, with increments of 2048. The time differences between them were slight and the data points were insufficient to understand the evolution. Thus, the analysis is inconclusive.

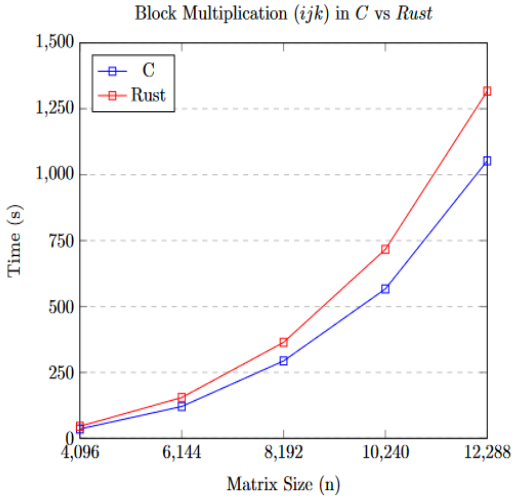


4.3 Comparison between C and Rust implementations

In order to see how different languages executing the same algorithm can have different results, particularly in terms of execution time, the direct comparison for each of the algorithms was plotted over an increasing matrix size. The results show that C performed better than Rust in all algorithms and, most importantly, that it scales better for larger matrix sizes on the faster algorithms.

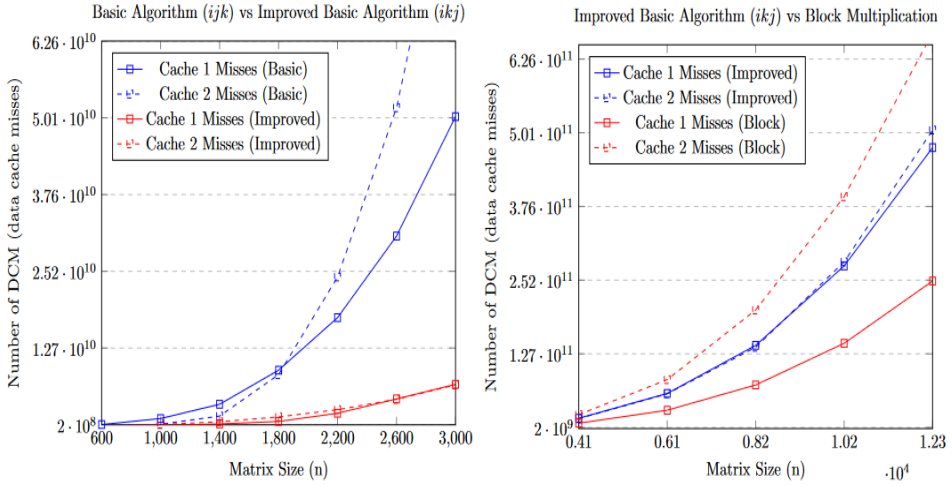






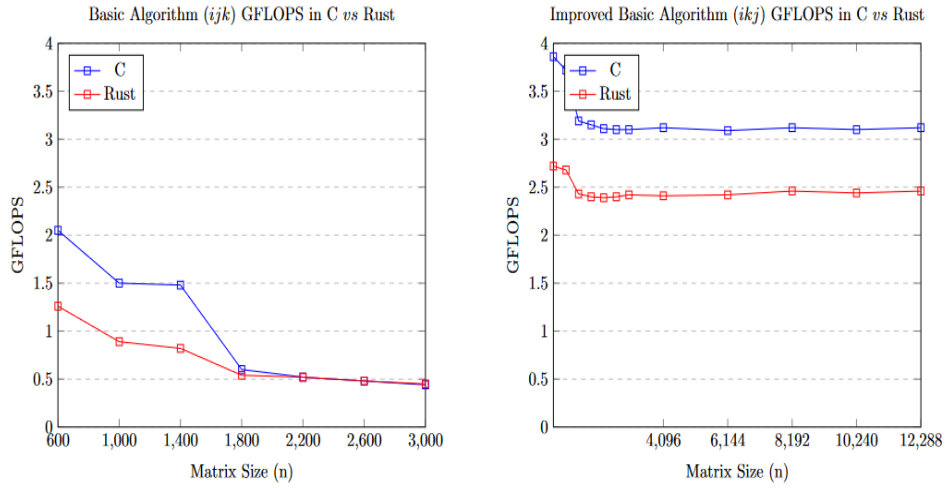
4.4 Cache Performance Comparison between different algorithms

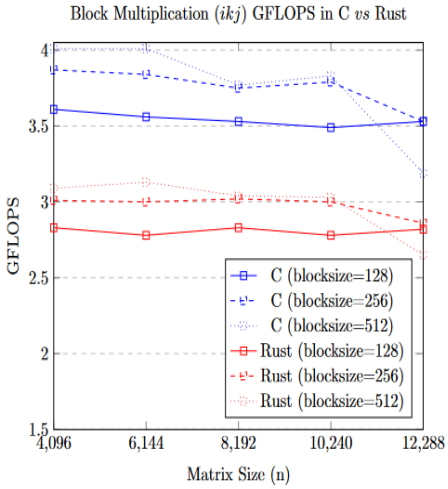
To verify how the algorithm optimizations impact cache usage, all three algorithms were plotted over increasing values of matrix sizes. The results make clear that the improvement over the basic algorithm drastically reduces the number of cache misses (on both levels), while the block multiplication, when compared with the improved basic algorithm, only reduces the number of cache 1 misses, at the cost of increasing the number of cache 2 misses.



4.5 Algorithm FLOPS comparison

In order to see how efficient the algorithms are, a good measurement is to calculate the GFLOPS for each previously calculated matrix size. As such, it is clear that C exhibits higher GFLOPS on all three algorithms, as expected, and there's also a clear improvement as we move towards more refined algorithms.





5 Conclusion

For data-heavy tasks such as large matrix multiplications, cache-aware algorithms can have a very noticeable impact on the processor's performance. Minimizing L1 and L2 cache misses by taking advantage of temporal and spatial locality when handling data allows for maximum usage of the much faster access times these components provide. This, in turn, speeds up the overall program run time by a significant amount, worthy of optimization efforts in critical tasks.