CPAR – Computação Paralela

Autor:

Tiago Magalhães – up201607931

Algorithm 1

How This Algorithm Works

Algorithm 1 is the standard matrix multiplication algorithm where you multiply each line of the initial matrix by each column of the second matrix to obtain the values of the resulting matrix.

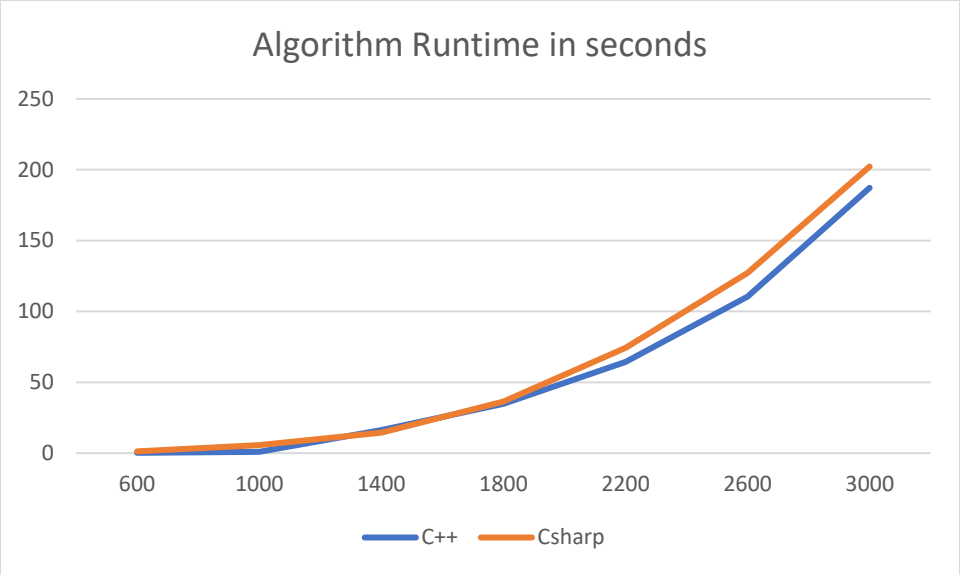
For this algorithm we are using a contiguous array to represent the matrix instead of the typical array of arrays representation for matrices. The following code was used to compute the product of the matrices using this algorithm:

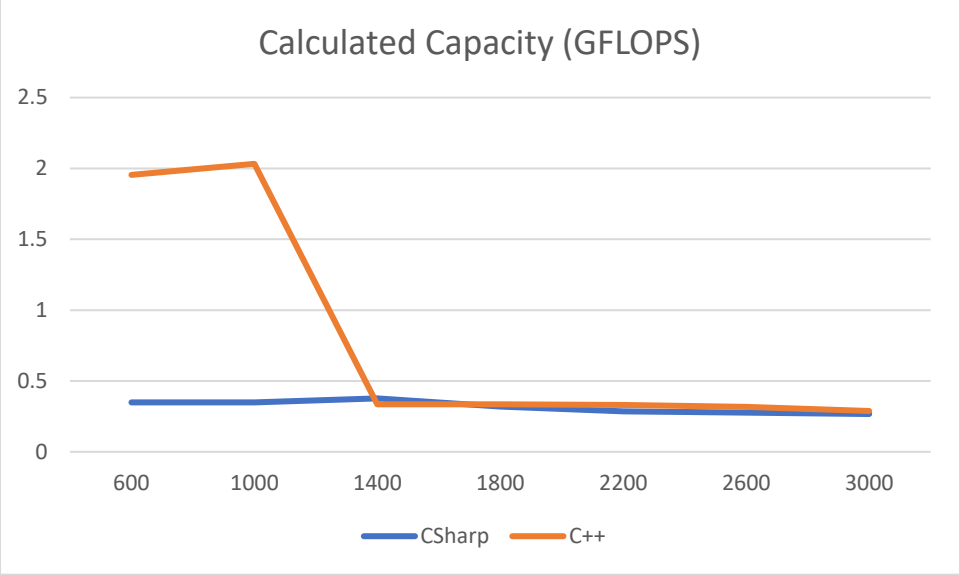
```
        size_t mA_height = this->m_height;
        size_t mB_height = matrix.get_height();
        size_t mB_width = matrix.get_width();

        if (this->algorithm == 1)
        {
            for (size_t matrix_A_line = 0; matrix_A_line < mA_height;
++matrix_A_line)
            {
                for (size_t matrix_B_col = 0; matrix_B_col < mB_width;
++matrix_B_col)
                {
                    for (size_t matrix_B_line = 0; matrix_B_line < mB_height;
++matrix_B_line)
                    {
                        res[matrix_A_line * this->get_height() + matrix_B_col] +=
this->get(matrix_A_line, matrix_B_line) * matrix.get(matrix_B_line,
matrix_B_col);
                    }
                }
            }
        }
```

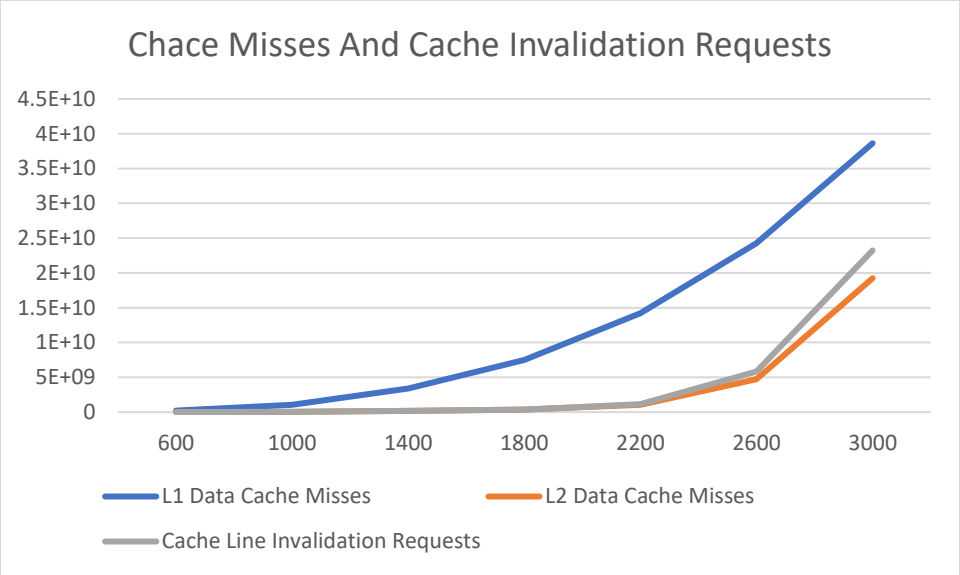
Here a class is being used to encapsulate the matrix, its related data and its functionality, the get method allows us to obtain a value of the matrix by passing the line and column that we want, then the formula line * height + column is used to obtain the value that is wanted.

Essentially as we can see here, we use 3 for blocks to go over every line of the matrix A, then for that we go over every column of matrix B and then we go over every element of that column to calculate the matrix values.

Result Analysis


Algorithm Runtime in seconds
250
200
150
100
50
0
600
1000
1400
1800
2200
2600
3000
C++
Csharp


Calculated Capacity (GFLOPS)
2.5
2
1.5
1
0.5
0
600
1000
1400
1800
2200
2600
3000
CSharp
C++



Here we can observe that the runtime and capacity difference between C++ and C# rapidly becomes almost nonexistent, while C++ starts out being significantly more powerful due to the inefficient usage of memory all the advantages of a compiled native language like C++ are thrown away.

Looking at the number of data cache misses we observer they grow exponentially and the data sets where the L2 cache misses start growing significantly are also the same data sets where the algorithm runtime starts to grow even more exponentially. We also observer that the growth in L1 data cache misses grow much more smoothly than the L2 data cache misses in this algorithm. We also observer that the L2 data cache misses and the number of cache line invalidations are nearly equal.

Performance analysis

After analyzing the results its quite clear we are running into a locality of reference cause performance bottleneck. In this case we are specifically dealing with a spatial locality of reference issue, this conclusion is due to the L2 cache misses causing most of the cache line invalidation and the analysis of the algorithm.

In essence what this means is that there is a large amount of data that is being continuously fetched from main memory, this happens because this algorithm by reading in column format does not utilize the cache lines that have been already loaded and causes the matrix to be loaded many more time than needed, in fact this algorithm will load the entire matrix into cache once for every column of the second matrix that we read from. This conclusion is further supplemented if we pass this code through a performance profiler, revealing that we are spending 99% of our time in the line where we fetch the values from the matrices and multiply them together, this is an obvious indicator of cache misusage and excessive loading from main memory.

Considerations

We have now concluded that to improve our performance we must use the cache lines that have already been loaded better, we have a spatial locality of reference issue to solve.

We must keep in mind that in part this issue is caused by our data representation, in fact if we used the array of arrays matrix representation this problem would not exists as we would use the cache lines correctly. This means that when analyzing cache and memory performance bottlenecks we can't assume things just based on the algorithm and we must understand the memory representation of data and how they interact with the algorithm as the same algorithm might have completely different performance properties when used with different representations of the same data.

Algorithm 2

How This Algorithm Works

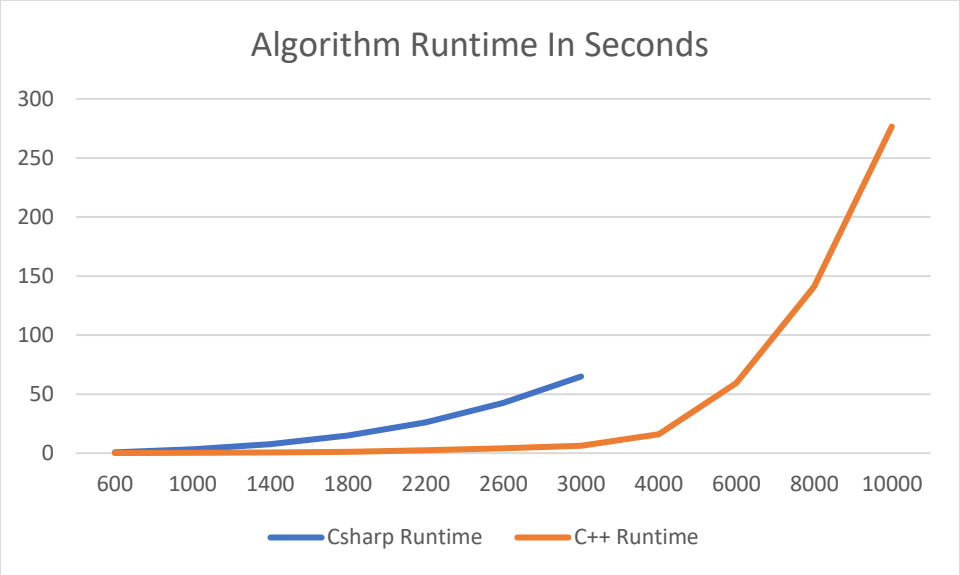
With this algorithm our focus is to solve the spatial locality of reference issue that was present in the first algorithm, to do this we will need to make better use of the cache lines that are loaded during calculation. This achieved through multiplying the lines in the first matrix by the lines of the second matrix and iteratively calculating the values of resulting matrix.

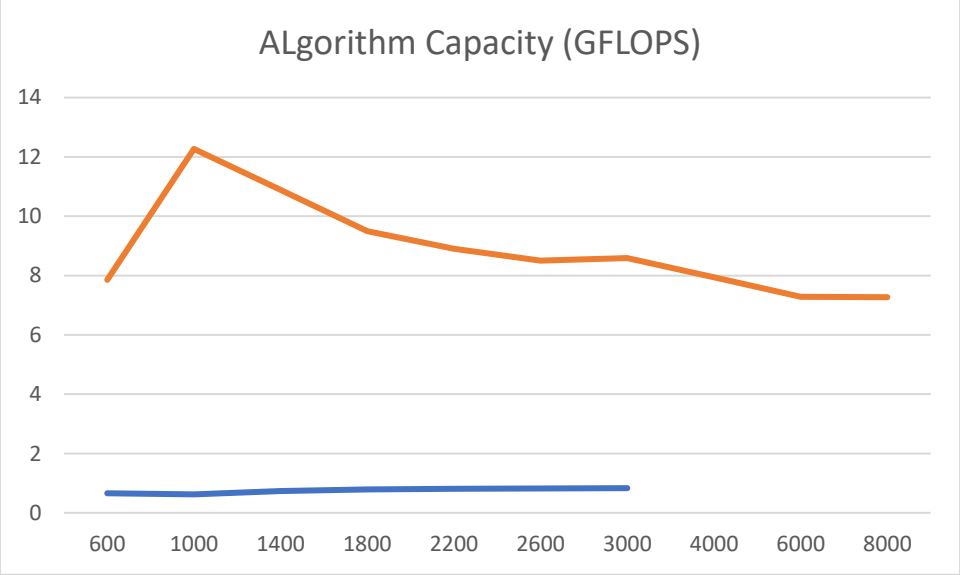
The following code executes this algorithm:

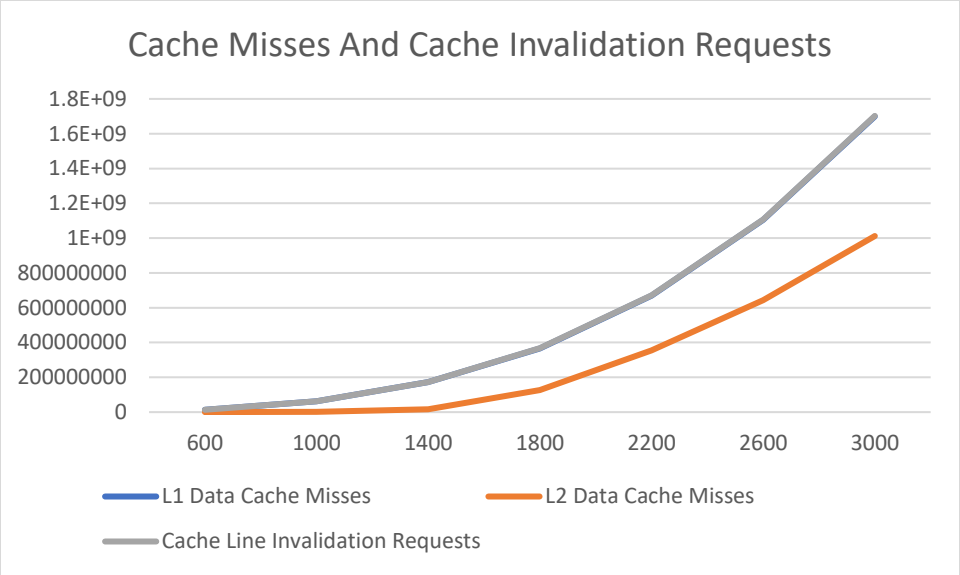
```
        else if (this->algorithm == 2)
        {
            for (size_t matrix_A_line = 0; matrix_A_line < mA_height;
++matrix_A_line)
            {
                for (size_t matrix_B_line = 0; matrix_B_line < mB_height;
++matrix_B_line)
                {
                    for (size_t matrix_B_col = 0; matrix_B_col < mB_width;
++matrix_B_col)
                    {
                        res[matrix_A_line * this->get_height() + matrix_B_col] +=
this->get(matrix_A_line, matrix_B_line) * matrix.get(matrix_B_line,
matrix_B_col);
                    }
                }
            }
        }
```

As can be seen we go over every line of matrix A and then we go over every line of matrix B and then over every column of that line, the rest of the calculation is equal to the previous algorithm, in fact the only difference is that the two last loops have been switched around.

Result Analysis


Algorithm Runtime In Seconds
300
250
200
150
100
50
0
600
1000
1400
1800
2200
2600
3000
4000
6000
8000
10000
Csharp Runtime
C++ Runtime


ALgorithm Capacity (GFLOPS)
14
12
10
8
6
4
2
0
600
1000
1400
1800
2200
2600
3000
4000
6000
8000



We can now observe that we are no longer throwing away the performance advantage of a natively compiled language with C++ handily beating C# in both runtime and computational capacity by a large margin.

We also observe that the number of cache misses has gone significantly down, this indicates that most of the time when there was an L1 cache miss the value was able to be fetched from L2 cache, therefore reducing the usage of the main memory and improving the performance of our software.

Performance analysis

As we observed the number of cache misses reduced significantly, this happens because the cache lines are used much more efficiently, the second matrix is now loaded entire once per every line of the fist matrix, instead of being loaded once for every element of the initial matrix like in algorithm 1.

This is caused by the usage of cache lines, when the CPU fetches data from the main memory it actually takes an entire blob of sequential data and loads it into the cache, so by reading in line format we are making full use of these preloaded blobs of data instead of forcing the CPU to continuously load in the same data from memory because we didn't use what was already loaded.

There is still a bottleneck, we can observe that the L1 cache misses are directly aligned with the number of cache invalidations, this means that we are not using fully the data that we already have, this is most likely an issue with temporal locality of reference. This means that we need to access the same values many times, but the way we are using the memory is not making full use of when these elements are in cache and we are still loading them into cache more than needed.

Considerations

Once again, its fundamental to understand that the performance improvement obtained here is only valid due to the data representation in use, and that for another representation the figures might be much worse.

Algorithm 3

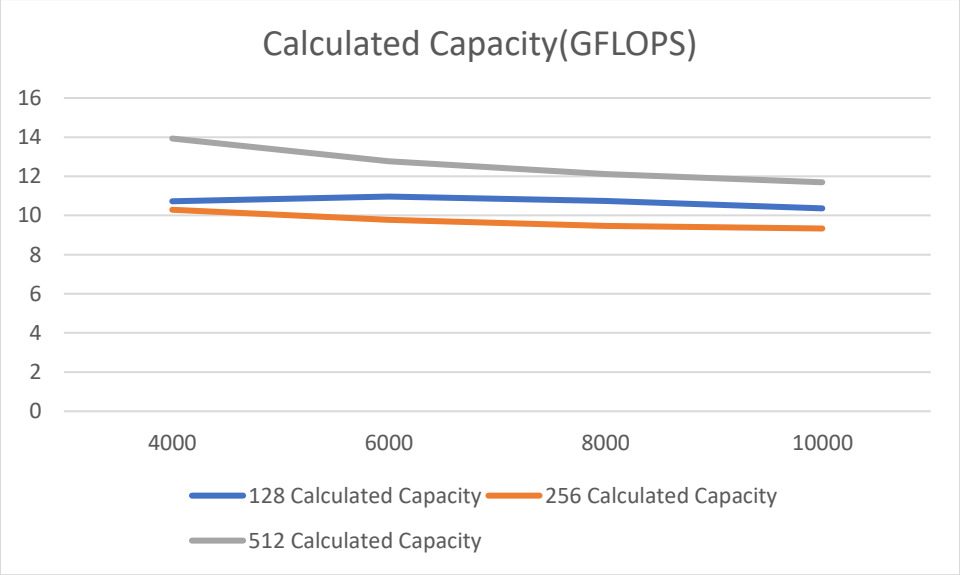
How This Algorithm Works

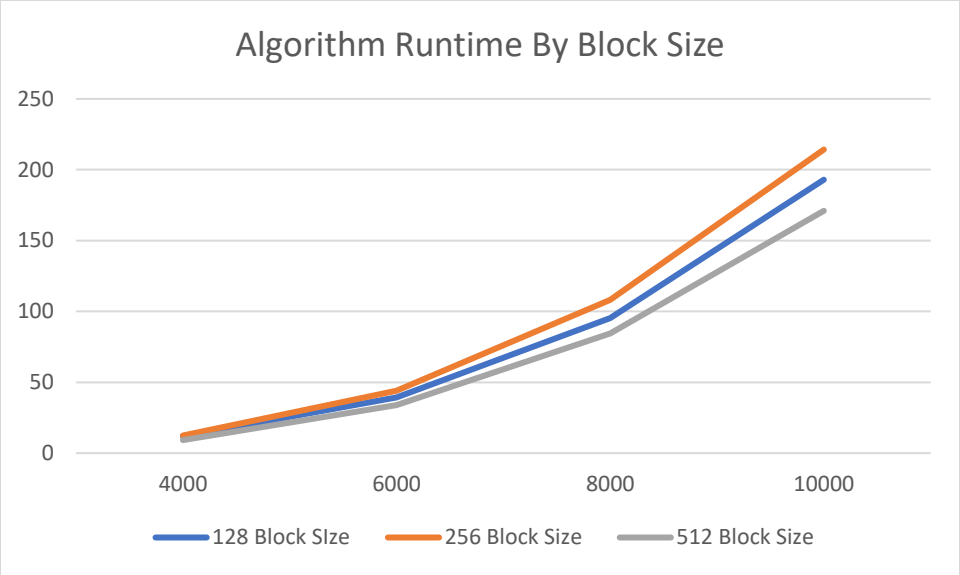
With this final algorithm our goal is to improve the temporal locality of reference, to do this we are going to use a technique called blocking that helps to improve the amount of cycles that data stays in the cache. In our case we will divide the matrix into blocks and execute the multiplication on these blocks and their submatrices.

The following code does this:

```
size_t n = mA_height;
            size_t en = this->BLOCK_SIZE * (n/this->BLOCK_SIZE); /* Amount that
fits evenly into blocks */

            for (size_t kk = 0; kk < en; kk += this->BLOCK_SIZE) {
                for (size_t jj = 0; jj < en; jj += this->BLOCK_SIZE) {
                    for (size_t i = 0; i < n; i++) {
                        for (size_t k = kk; k < kk + this->BLOCK_SIZE; k++) {
                            for (size_t j = jj; j < jj + this->BLOCK_SIZE; j++) {
                                res[i* this->get_height()+ j] += this-
>get(i,k)*matrix.get(k,j);
                            }
                        }
                    }
                }
            }
```

Result Analysis





Here we can observe that the computational capacity is much more consistent, and the runtime has greatly improved, we also observer that the block size has an impact on the algorithm performance.

Performance Analysis

What happens here is that due to the way the algorithm works once we are done using a block, we will rarely have to load it again, therefore we are improving our usage of that block when its in cache, meaning that we will overall have less cache misses, meaning we improved the temporal locality of reference.

While blocking is a general technique there is not a one size fits all way to implement it into an algorithm, its depends on the problem beings solved, however a key idea here is that the size of the blocks should be powers of two so that they fit neatly into the cache lines, and they should be smaller than the actual matrix otherwise we are just executing the previous algorithm.

Considerations

One consideration to be had is that algorithm can have different properties in different machines depending on cache sizes and how much data can be fit into the cache such that we are not preserving so much data in cache that we have nothing left for the rest of the data we need and instead of lowering cache misses we end up increasing the misses so as to be able to execute the computation.

While blocking has performance improvements that are still significant it should be noticed that the difference in solving this temporal locality of reference problem is much lower than when the spatial locality of reference problem was solved, this is an important consideration when we take into account that we might need a long time to tweak this algorithm for different hardware and to get it to where it always works well, that being said in general this technique will always lead to a performance improvement, so using it should always be a positive but spending too much time trying to identify the best block size might be a fruitless endeavor.

Conclusions

With this project we can conclude that serious consideration must be given to how we use our data. In many ways Cache usages are the new floating-point divides, being the major source of performance bottlenecks in algorithms.

Cache problems are much more insidious than floating point division, the way that data is modeled, and access greatly changes that algorithms properties and there is no way to fix these issues that always works.

Attaining good locality of reference requires a deep understand of the data model, the algorithms in use, how cache works and its accessed and many more factors such as the prefetchers, compiler instruction reordering and processor instruction reordering, as well as the impacts of language features such as polymorphism.

In fact, while its was explored here experiences were conducted with the usage of polymorphism during the project, that lead to the conclusion that when using polymorphism, we not only pay for the cost of the virtual table lookups, but we also pay for the impact these lookups have on cached data. In fact, when using polymorphism, we witnessed a 5x performance decrease for algorithm 2.

In conclusion it is obvious that throwing more hardware at a problem as many propose is not and will never be the solution for performance issues, this project clearly demonstrates that the memory access issues handily outpaced the power of modern hardware and the ridiculous idea of using better hardware instead of building better algorithms and understand our hardware.