FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Computação Paralela e Distribuída

1º trabalho prático

Licenciatura em Engenharia Informática e Computação

Diogo Fernandes (202108752)
Jaime Fonseca (202108789)
João Pereira (202108848)
José Oliveira (202108764)

Março 2023

Contents

1 Project Overview

1.1 Project Description

This project aims to study the effect on the processor performance of the memory hierarchy when accessing large amounts of data.

For the first part of the project, we were asked to implement three different versions of the hlmatrix multiplication algorithm in two different programming languages in order to compare the performance of both languages. The first language should be C++ and the second one could be any language of our choice. We chose to implement the second version of the algorithms in Julia, a high-level, general-purpose dynamic programming language that although being commonly used for numerical analysis and computational science, it is also a good choice for high-performance computing. This high-performance is easily perceived when comparing Julia Benchmarks with other popular languages such as Rust, Go, Lua, Python, etc.

For the second part of the project, we were asked to implement a parallel version of the second algorithm in C++ following two different approaches and using the OpenMP library. This allowed us to compare the performance of both parallelization strategies and the sequential version of the algorithm.

In order to collect relevant performance indicators of the program execution, we used the Performance API (PAPI) to measure the performance of the different versions of the matrix multiplication algorithm. The PAPI library provides a set of interfaces for accessing hardware performance counters available on the Performance Monitoring Unit (PMU) of the processor.

1.2 Algorithms Explanation

The matrix multiplication algorithm is a simple algorithm that multiplies two matrices and stores the result in a third matrix. The algorithm is defined as follows:

$$C = A \times B \tag{1}$$

Where both A and B are square matrixes.

1.2.1 First Algorithm - OnMult

The OnMult algorithm is a straightforward implementation of the matrix multiplication algorithm with a time complexity of $O(n^3)$. This algorithm iterates over each element of C and calculates its value by dot multiplying the corresponding row of A with the corresponding column of B. This process involves three nested loops: the outer loop iterates over each row of matrix A, the inner loop iterates over each column of matrix B, and the innermost loop computes the dot product by iterating over each element of the row of A and the column of B. The result is stored in the corresponding element of matrix C. This algorithm is suitable for small to medium-sized matrices and provides a simple and efficient way to perform matrix multiplication. However, for larger matrices, more optimized approaches, such as parallelization or algorithmic optimizations, may be necessary to improve performance.

1.2.1.1 C++ Implementation

```
for (i=0; i<m_ar; i++) {
    for (j=0; j<m_br; j++) {
        temp = 0;
        for (k=0; k<m_ar; k++) {
            temp += pha[i*m_ar+k] * phb[k*m_br+j];
        }
        phc[i*m_ar+j]=temp;
    }
}
```

1.2.1.2 Julia Implementation

```
for i in 1:m_ar
    for j in 1:m_br
        temp = 0.0
        for k in 1:m_ar
            temp += pha[i, k] * phb[k, j]
        end
        phc[i, j] = temp
    end
end
```

1.2.2 Second Algorithm - OnMultLine

The OnMultLine algorithm presents an alternative approach to matrix multiplication, differing from OnMult in loop ordering and potential for parallelization. With a time complexity of $O(n^3)$, OnMultLine iterates over each row of matrices A and B separately within nested loops. Unlike OnMult, where the dot product computation occurs within the innermost loop, OnMultLine's rearranged loop structure may influence memory access patterns and cache efficiency. This algorithm introduces potential for parallelization, particularly with parallelizing the outer loop iterations, offering opportunities for concurrent computation of independent elements of matrix C. Despite these differences, OnMultLine shares similar computational complexity and remains suitable for small to medium-sized matrices. However, for larger matrices, considerations for optimization, such as parallelization and algorithmic improvements, may be required to enhance performance on multi-core systems.

1.2.2.1 C++ Implementation

```
for (i=0; i<m_ar; i++) {
    for (k=0; k<m_ar; k++) {
        for (j=0; j<m_br; j++) {
                phc[i*m_ar+j] += pha[i*m_ar+k] * phb[k*m_br+j];
        }
    }
}
```

1.2.2.2 Julia Implementation

```
for k in 1:m_ar
    for j in 1:m_br
        for i in 1:m_ar
            phc[i, j] += pha[i, k] * phb[k, j]
        end
    end
end
```

1.2.3 Third Algorithm - OnMultBlock

The algorithm divides the matrices into smaller blocks. By dealing with smaller blocks, the chance of being able to utilize the processor cache is higher, which can lead to a significant performance improvement.

The OnMultBlock algorithm introduces a more sophisticated approach to matrix multiplication, particularly advantageous for large matrices. By partitioning the matrices into smaller blocks, this algorithm enhances cache utilization, potentially leading to significant performance gains. The nested loops iterate over blocks of the matrices, enabling more efficient access to data stored in the processor cache. Within each block, the algorithm computes the dot product of corresponding elements of matrix A and B, accumulating the results into the corresponding elements of matrix C. This block-based approach effectively reduces memory access latency and improves data locality, making it well-suited for large-scale matrix multiplication tasks. Despite its seemingly complex structure, OnMultBlock offers promising performance benefits and remains a valuable optimization technique for matrix multiplication algorithms on modern computing systems.

1.2.3.1 C++ Implementation

```
for(ii=0; ii<m_ar; ii+=bkSize) {
    for( kk=0; kk<m_ar; kk+=bkSize){
        for( jj=0; jj<m_br; jj+=bkSize) {
            for (i = ii ; i < ii + bkSize ; i++) {
                for (k = kk ; k < kk + bkSize ; k++) {
                    for (j = jj ; j < jj + bkSize ; j++) {
                        phc[i*m_ar+j] += pha[i*m_ar+k] * phb[k*m_br+j];
                    }
                }
            }
        }
    }
}
```

1.2.3.2 Julia Implementation

```
for bi in 0:bkSize:m_ar
    for bj in 0:bkSize:m_br
        for bk in 0:bkSize:m_ar
            for i in bi:min(bi+bkSize, m_ar)
                for j in bj:min(bj+bkSize, m_br)
                    for k in bk:min(bk+bkSize, m_ar)
                        phc[i, j] += pha[i, k] * phb[k, j]
                    end
                end
            end
        end
    end
end
```

2 Performance Metrics

In our study, we employed the Performance API (PAPI) to examine the effects of programming languages and optimization strategies on the performance of matrix multiplication algorithms. We concentrated on metrics such as execution time, floating-point operations (FLOPs), and cache miss rates at both the L1 and L2 levels, and CPU cycles. Given their substantial influence on processing efficiency, cache miss rates were deemed a critical component of our evaluation.

The entirety of our testing was executed on a computer equipped with an Intel i5 7300U CPU, which features a 2.6 GHz clock speed and four cores. This uniform testing environment was key to ensuring the reliability of our data collection efforts. To address potential variability, we calculated the average results from three executions for each test, initiating a new process on each occasion to guarantee independent memory allocation for every run. For the compilation of C/C++ code, we applied the -O2 optimization flag, thereby enhancing performance without significantly increasing compilation time.

Our comparative analysis spanned implementations in C/C++ and the Julia programming language, tracking execution times for matrices ranging from 600x600 to 3000x3000 elements, increasing in increments of 400. For the line-by-line (onMultLine) and block-oriented (onMultBlock) multiplication techniques, we not only examined standard sizes but also significantly larger matrices. We methodically incremented block sizes in the block-oriented variant, testing blocks of 128, 256, and 512 to determine the most efficient block size in terms of processing time and other vital performance indicators.

3 Results and Analysis

In order to run the tests, we created a python script that can be found in the assign1/src directory. The script will run the tests for the different algorithms and for the different matrix and block sizes and will output the results to the assign1/src/results directory in the form of .csv files. Through the analysis of these results and by plotting the data, we were able to draw some conclusions about the performance of the different algorithms and the impact of the different programming languages and optimization strategies on the performance of the matrix multiplication algorithms. Each output table contains the following columns: Matrix Size, Elapsed Time, L1 Data Cache Misses, L2 Data Cache Misses, Total Instructions Completed, and Total Cycles. The following sections will present the results and analysis for the different algorithms and programming languages.

3.1 C++ vs Julia

The first comparison we made was between the C++ and Julia implementations of the matrix multiplication algorithms. As expected, the C++ implementation outperformed the Julia implementation in all tests. This is due to the fact that C++ is a lower-level language and allows for more control over the hardware, which can lead to better performance. However, it is important to note that the Julia implementation is also quite performant, especially when compared to other high-level languages such as Python.

3.2 OnMult vs OnMultLine

Even though, at first glance, the OnMult and OnMultLine algorithms seem to be very similar, the results show that the OnMultLine algorithm is consistently faster and produces fewer cache misses than the OnMult algorithm, it also runs in much fewer cycles. The analysis of total instructions completed was similar for both algorithms, with the OnMult algorithm completing slightly fewer instructions than the OnMultLine algorithm. The results show that the OnMultLine algorithm is more efficient than the OnMult algorithm, which is consistent with our expectations, given the different loop ordering and potential for parallelization of the OnMultLine algorithm. This analysis was verified in both the C++ and Julia implementations.

3.3 OnMultLine vs OnMultBlock

The OnMultBlock algorithm is known for its efficiency when dealing with large matrices, therefore, we only tested this algorithm with matrices of size 4096, 6144, 8192, and 10240. We tested the algorithm OnMultLine with the same matrices in order to compare the performance of both algorithms. The comparison between both algorithms varied with the language used. For this reason, we will present the results and analysis for both the C++ and Julia implementations separately.

3.3.1 C++ Implementation

Our results revealed a proximity in the elapsed time for both algorithms, with the OnMultBlock algorithm exhibiting slightly faster execution times than the OnMultLine algorithm. On the other hand, the OnMultBlock algorithm produced a significantly lower number of cache misses at the L1 level. It would be to expect that the OnMultBlock algorithm would produce more cache misses at the L2 level as it is designed speciffically to take advantage of the cache at lower levels. The OnMultLine algorithm has fewer cache misses at the L2 level than the OnMultBlock as most of its accesses to cache fail at the L1 level. In terms of total instructions completed and total cycles, there wasn't a significant difference between the two algorithms, with the OnMultBlock algorithm completing slightly fewer instructions and running in fewer cycles on average than the OnMultLine algorithm.

3.4 Julia Implementation

The Julia implementation of the OnMultLine revealed itself slightly faster than the OnMultBlock algorithm. Contrary to the C++ implementation and to the expected results, the OnMultBlock produced a higher number of cache misses at both the L1 and L2 levels. This is likely due to the fact that the Julia implementation of the OnMultBlock algorithm is not as optimized as the C++ implementation, which could lead to a higher number of cache misses. The behavior of the two algorithms in terms of total instructions completed and total cycles wasn't significantly different except for a surprising and spontaneous spike in the number of total instructions completed by the OnMultLine algorithm for the 10200 matrix size.

3.5 Block Size Analysis

The OnMultBlock algorithm was tested with block sizes of 128, 256, and 512. The results show that the bigger the block size, the least cache misses the algorithm produces. However, the execution time of the algorithm increases with the block size. This is likely due to the fact that the bigger the block size, the more data the algorithm has to process at each iteration. The number of total instructions completed and the number of total cycles was almost the same for all block sizes, almost as if the block size didn't have any impact on these metrics.

3.6 Parallelization Strategies

The parallelization of the OnMultLine algorithm was implemented using the OpenMP library. We tested two different parallelization strategies: parallelizing the outer loop and parallelizing the inner loop:

```
// #pragma omp parallel for
for (i=0; i<m_ar; i++) {
    for (k=0; k<m_ar; k++) {
        for (j=0; j<m_br; j++) {
                phc[i*m_ar+j] += pha[i*
                    m_ar+k] * phb[k*m_br+
                    j];
            }
    }
}
```

```
// #pragma omp parallel private(i,j,k)
for (i=0; i<m_ar; i++) {
    for (k=0; k<m_ar; k++) {
        // #pragma omp for
        for (j=0; j<m_br; j++) {
                phc[i*m_ar+j] += pha[i*
                    m_ar+k] * phb[k*m_br+
                    j];
            }
    }
}
```

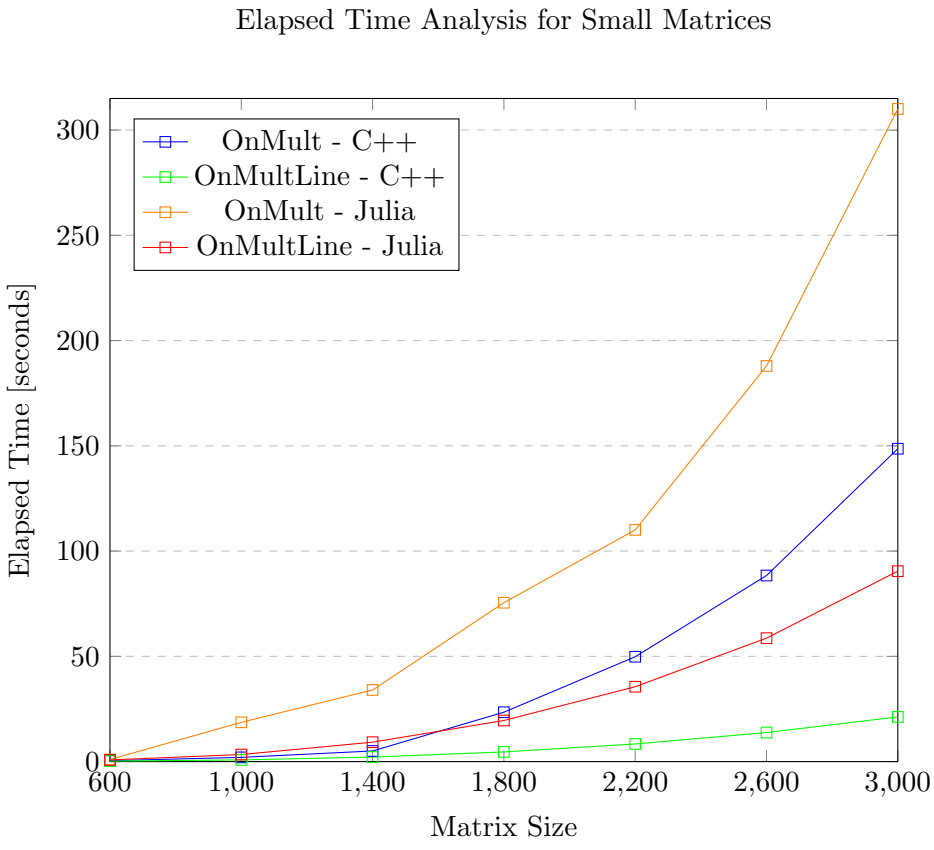
Both strategies were also compared to the default OnMultLine algorithm. The comparison between the default OnMultLine algorithm and the parallelized versions of the algorithm showed that the parallelized versions of the algorithm were consistently more efficient than the alternative. Between the two parallelization strategies, the one that standed out the most was the first approach, which only parallelized the outer loop. This approach was not only consistently faster than the default algorithm but also produced fewer cache misses at L1 level (the reason for the higher number of L2 cache misses already was explained in a previous section). The outter loop parallelization approach also completed fewer instructions but ran in more cycles than the inner loop parallelization aproach.

4 Conclusions

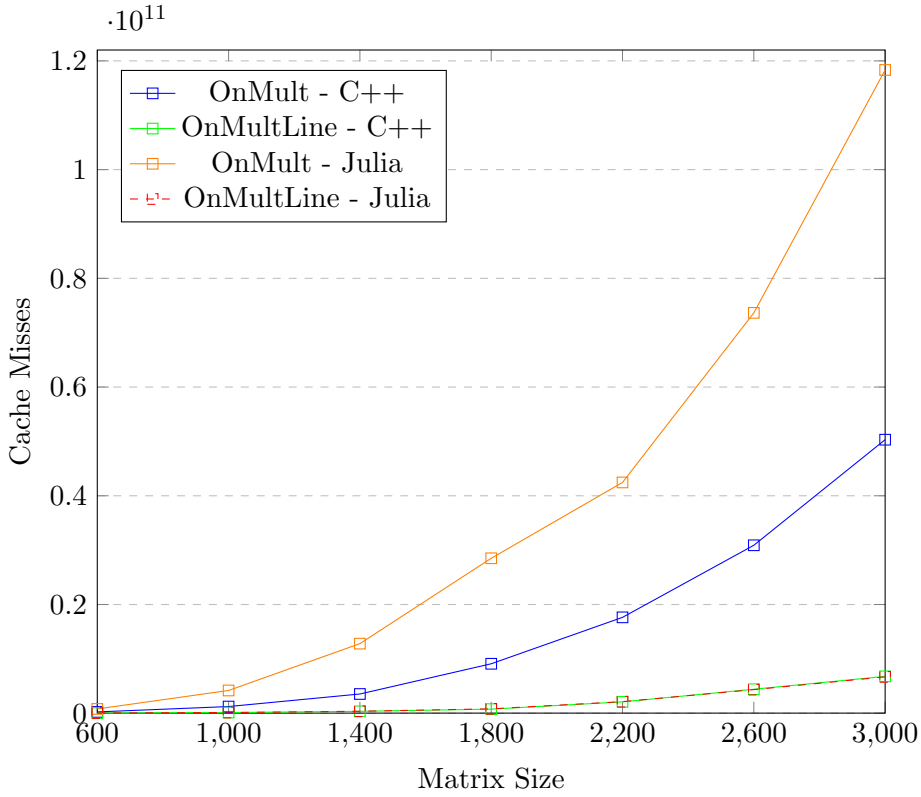
Our analysis has highlighted the critical role of efficient memory management in enhancing the performance of matrix multiplication algorithms. The C++ implementation outperformed the higher-level language Julia, with its efficiency attributable to lower cache miss rates, emphasizing the importance of cache optimization in low-level programming.

In conclusion, our work underscores the significance of cache-aware strategies for processor-intensive tasks like large matrix multiplications. By effectively minimizing cache misses and exploiting data locality, substantial improvements in execution times were achieved, demonstrating the value of optimizing for the cache hierarchy even in non-parallelized, sequential program execution.

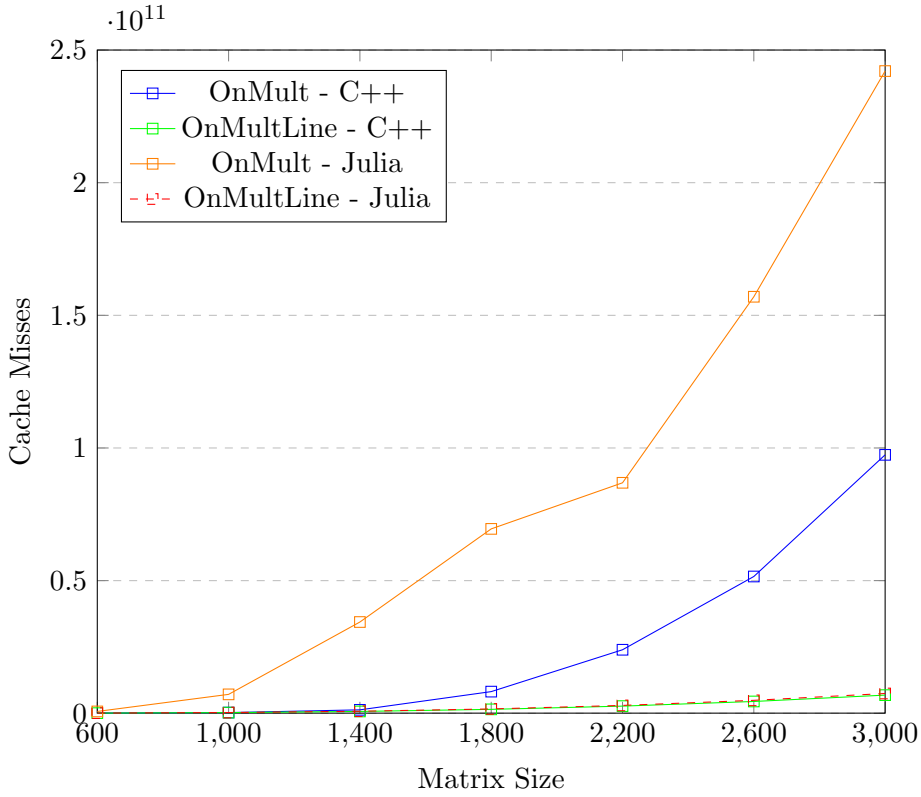
5 Annex



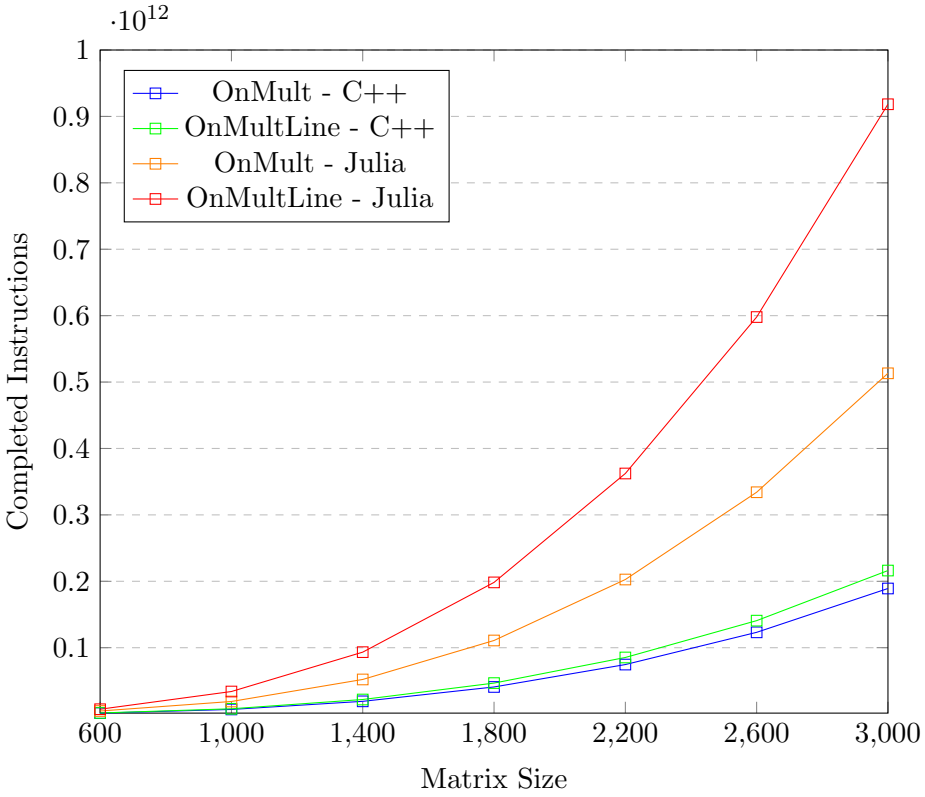
Level 1 Data Cache Misses Analysis for Small Matrices



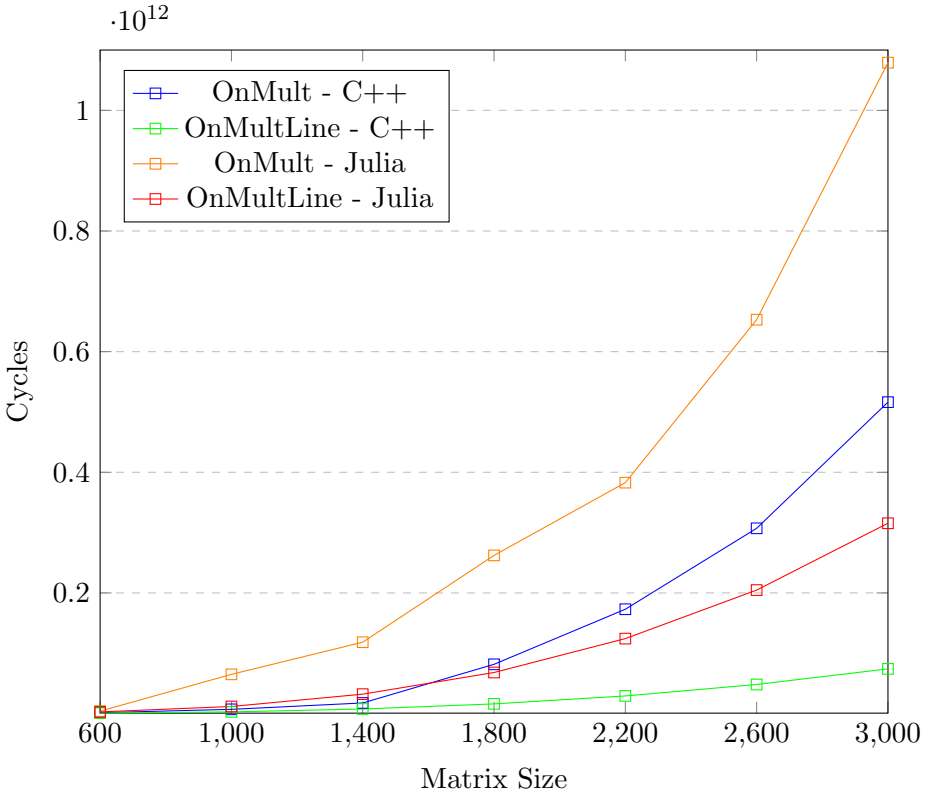
Level 2 Data Cache Misses Analysis for Small Matrices



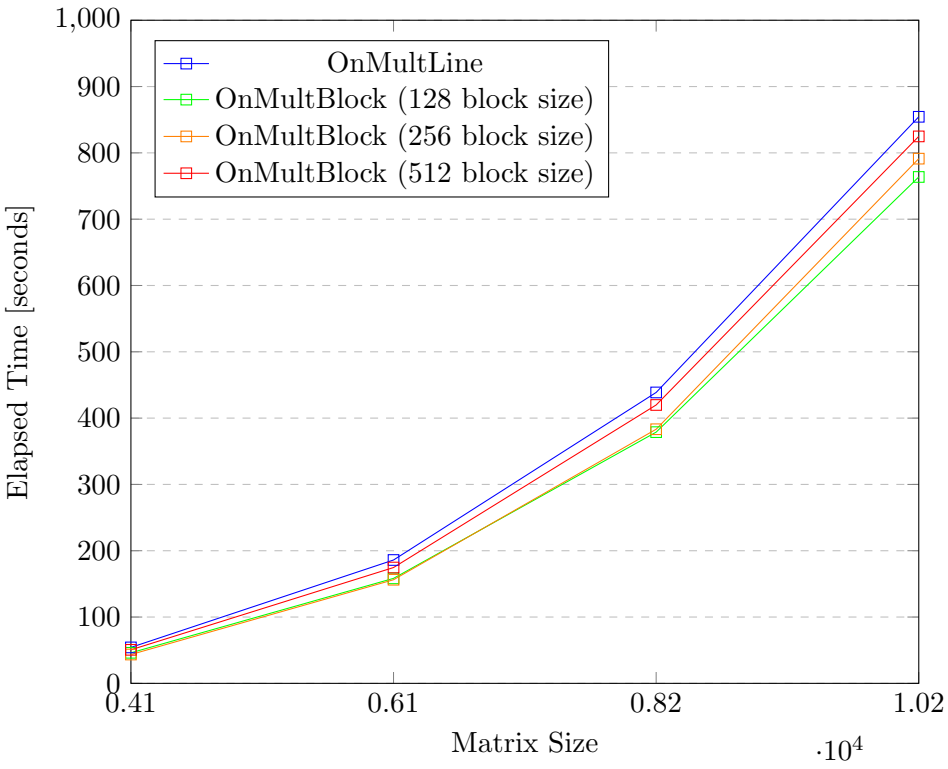
Total Instructions Completed Analysis for Small Matrices



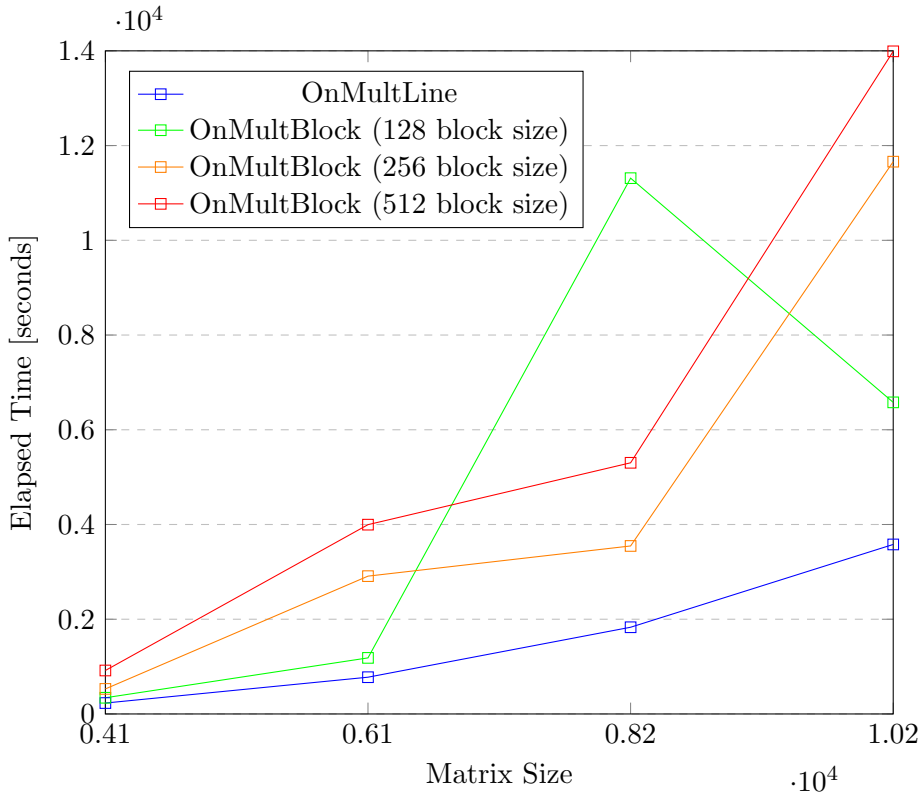
Total Cycles Analysis for Small Matrices



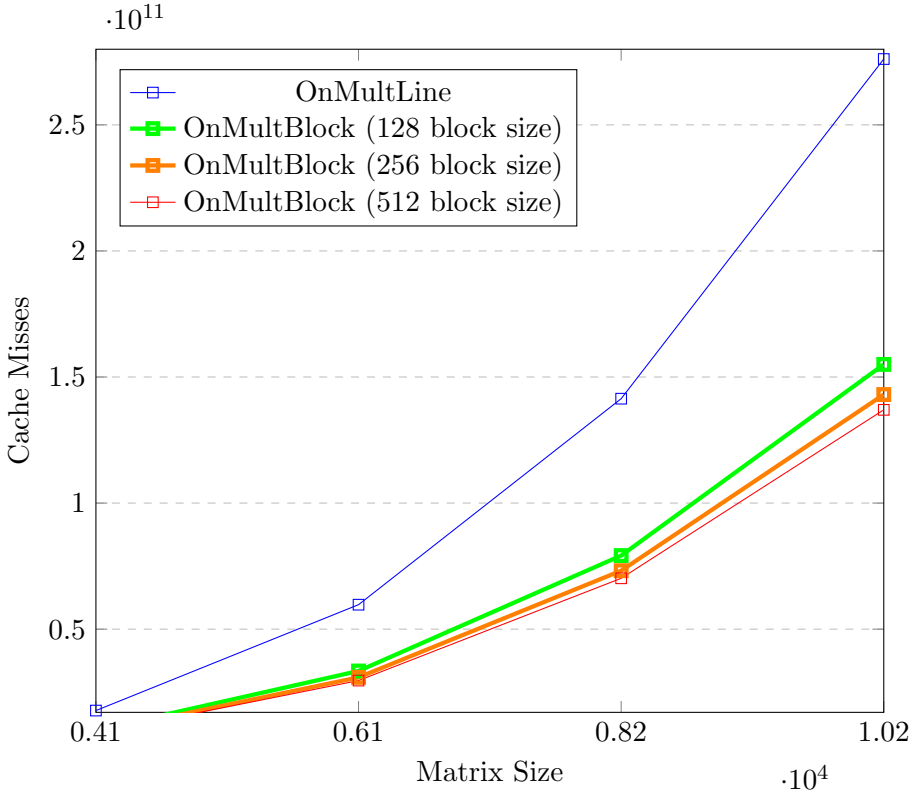
Elapsed Time Analysis for Big Matrices (C++)



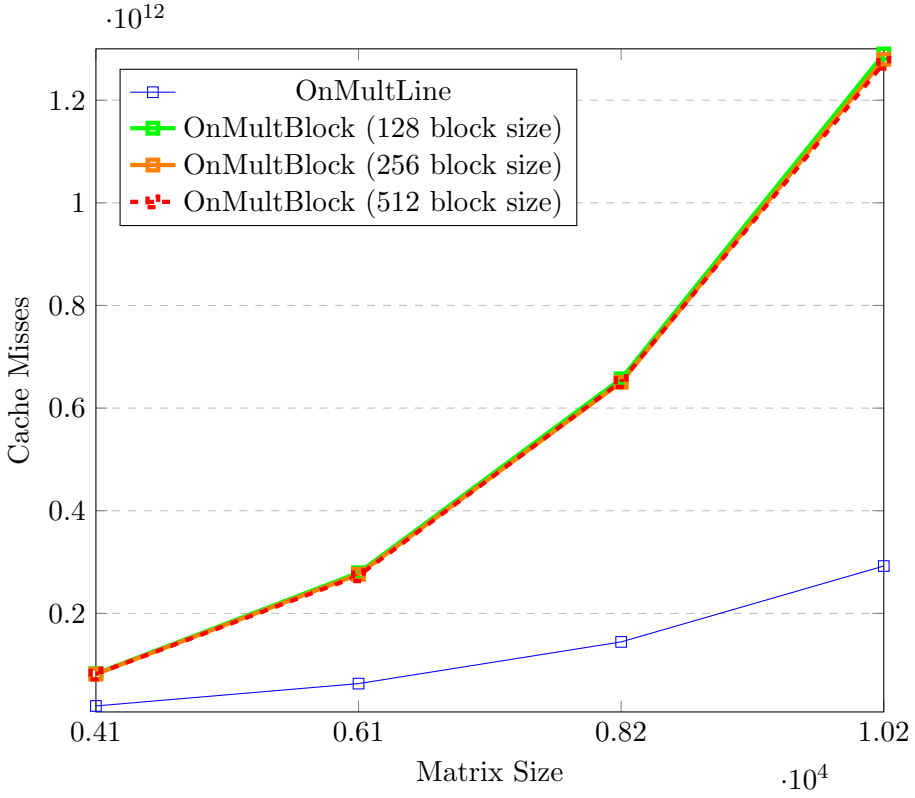
Elapsed Time Analysis for Big Matrices (Julia)



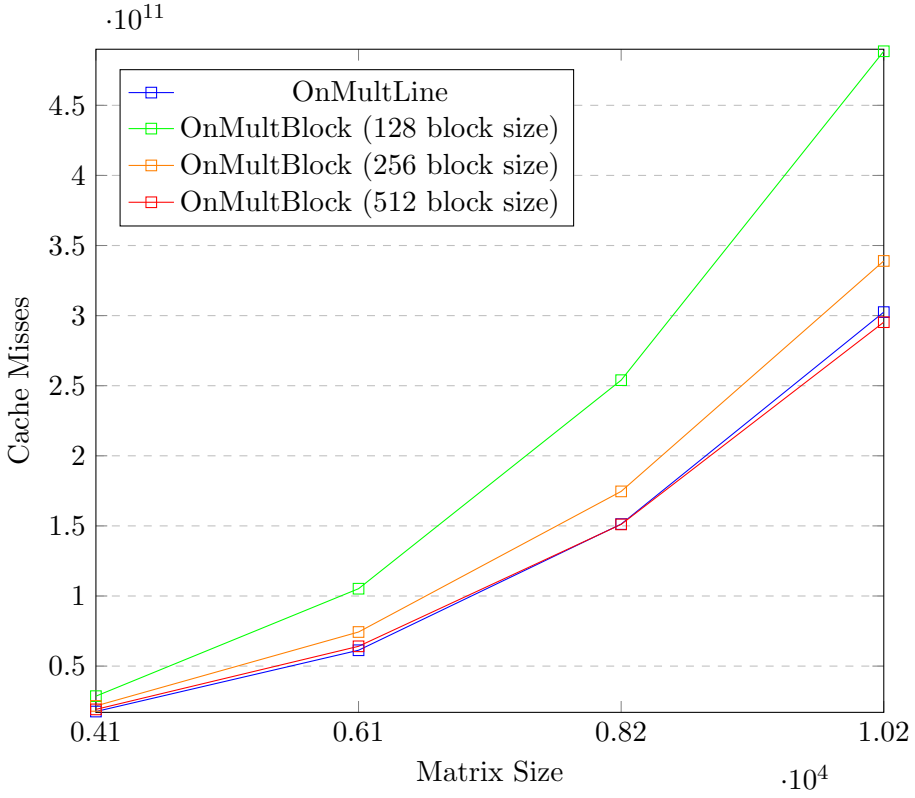
Level 1 Data Cache Misses Analysis for Big Matrices (C++)



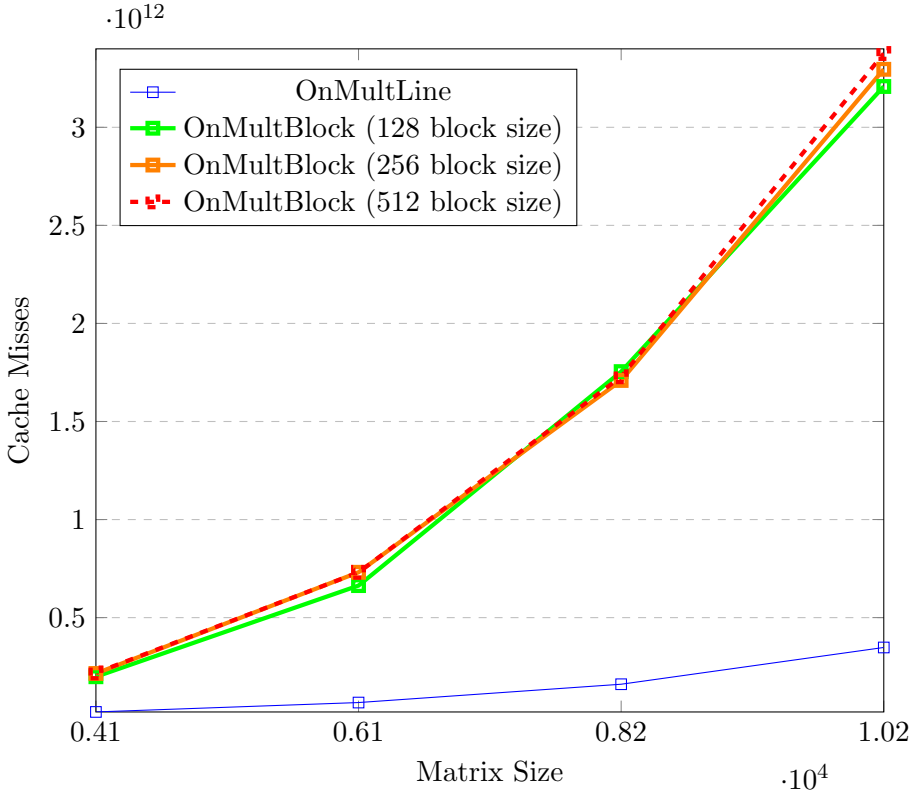
Level 1 Data Cache Misses Analysis for Big Matrices (Julia)

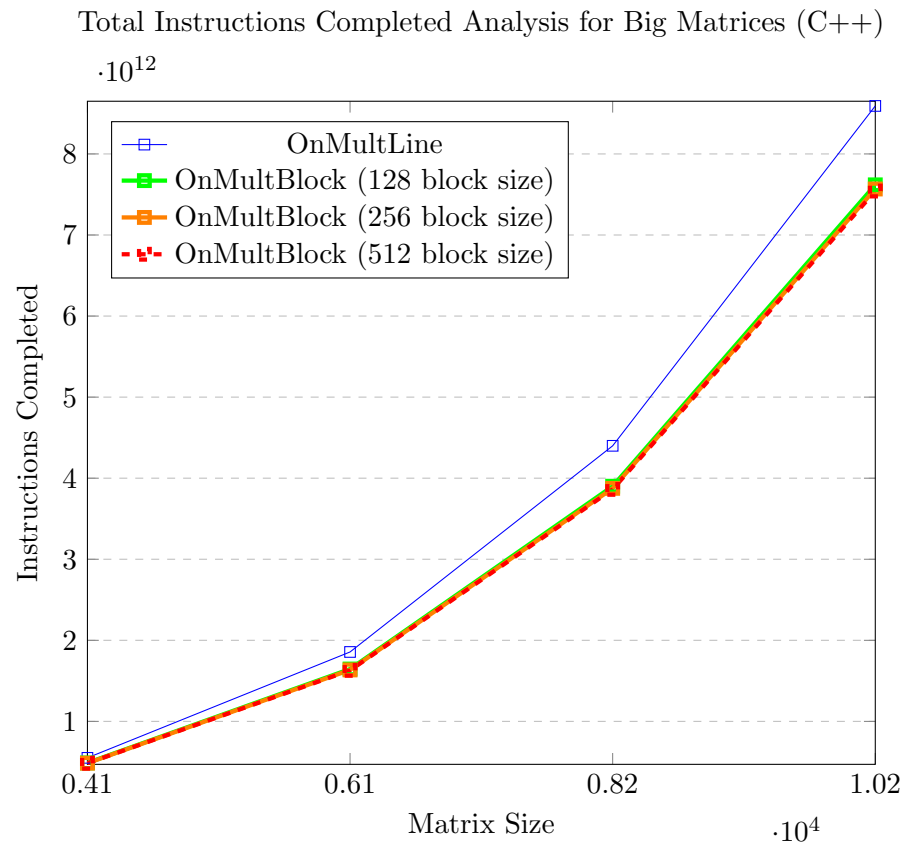


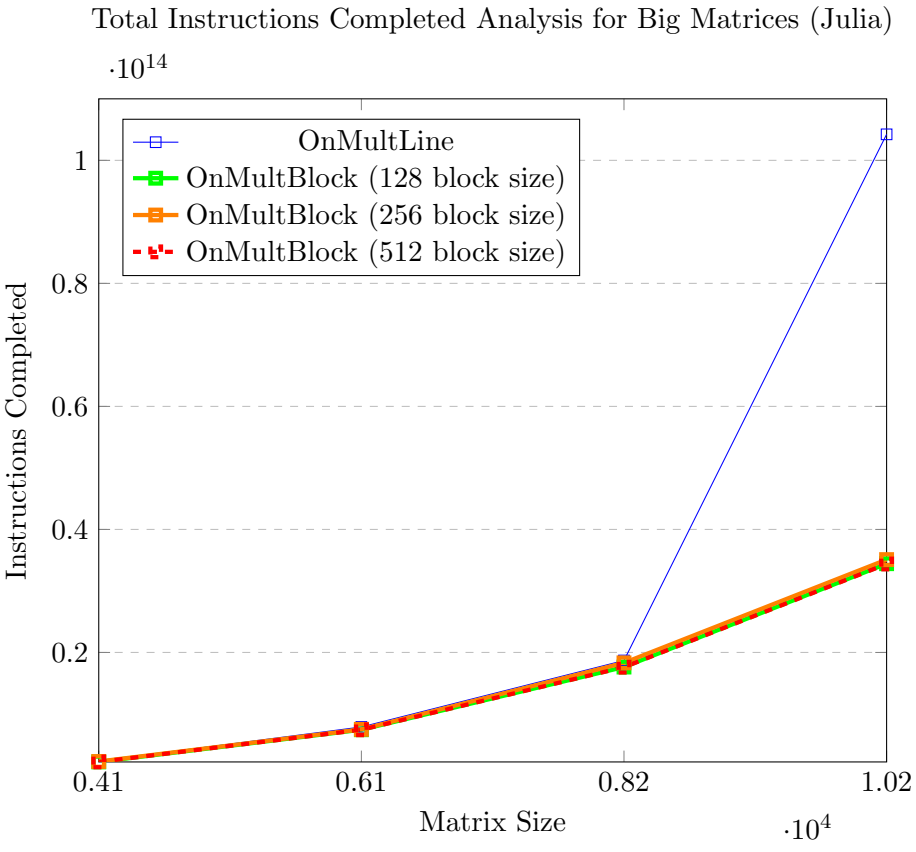
Level 2 Data Cache Misses Analysis for Big Matrices (C++)



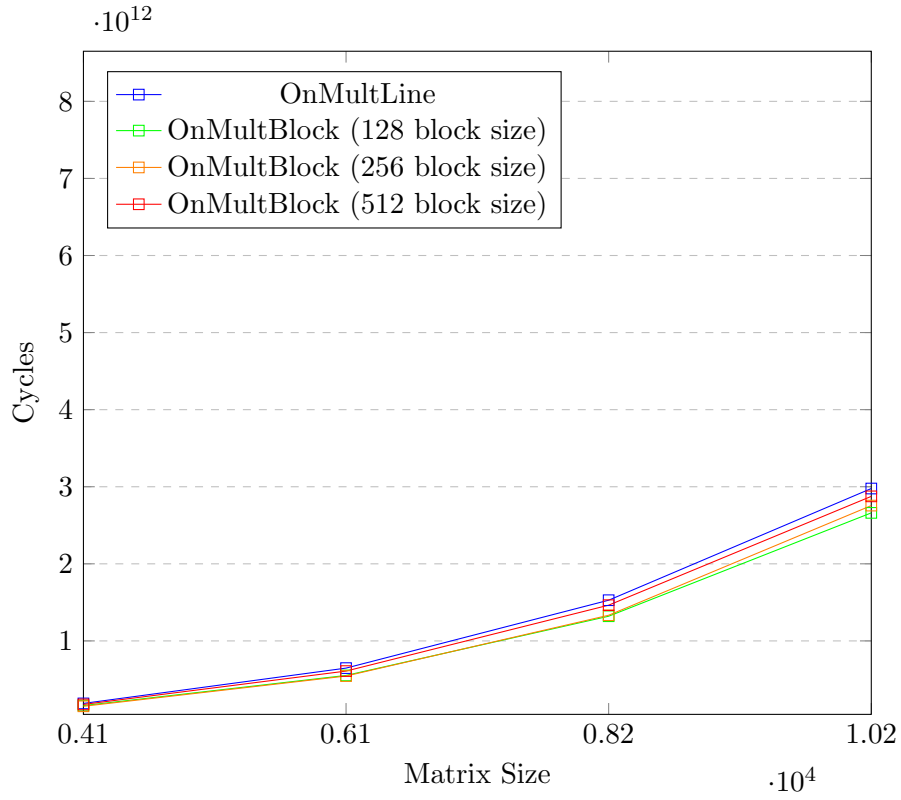
Level 2 Data Cache Misses Analysis for Big Matrices (Julia)



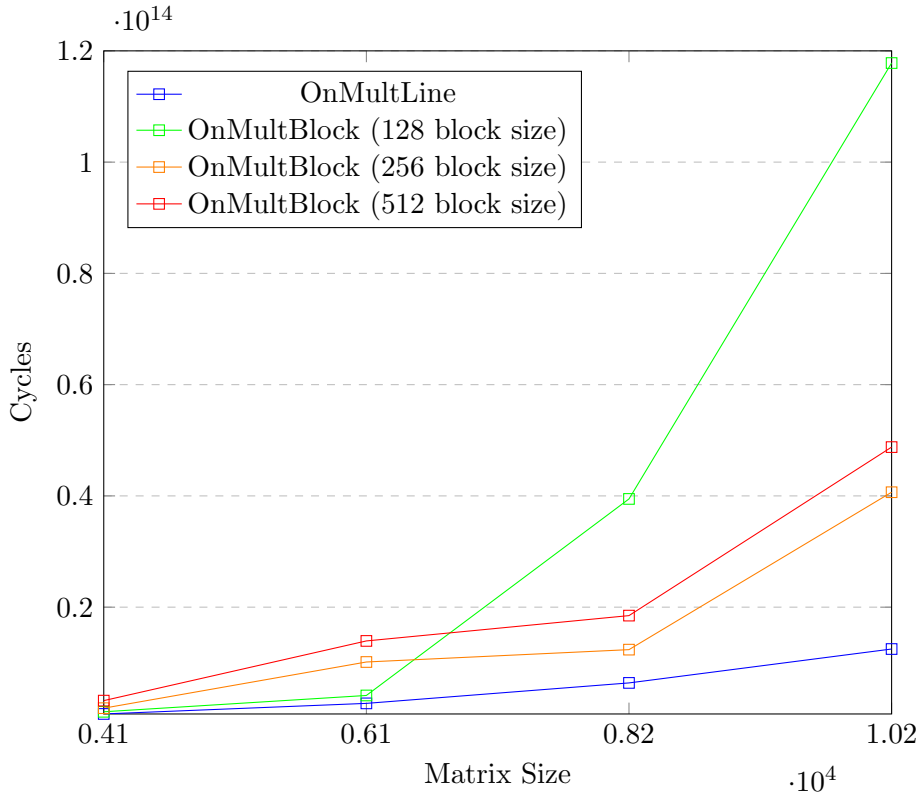

Total Instructions Completed Analysis for Big Matrices (C++)
$\cdot 10^{12}$
OnMultLine
OnMultBlock (128 block size)
OnMultBlock (256 block size)
OnMultBlock (512 block size)
Instructions Completed
1
2
3
4
5
6
7
8
0.41
0.61
0.82
1.02
Matrix Size
$\cdot 10^4$


Total Instructions Completed Analysis for Big Matrices (Julia)
$\cdot 10^{14}$
OnMultLine
OnMultBlock (128 block size)
OnMultBlock (256 block size)
OnMultBlock (512 block size)
Instructions Completed
0.2
0.4
0.6
0.8
1
0.41
0.61
0.82
1.02
Matrix Size
$\cdot 10^4$

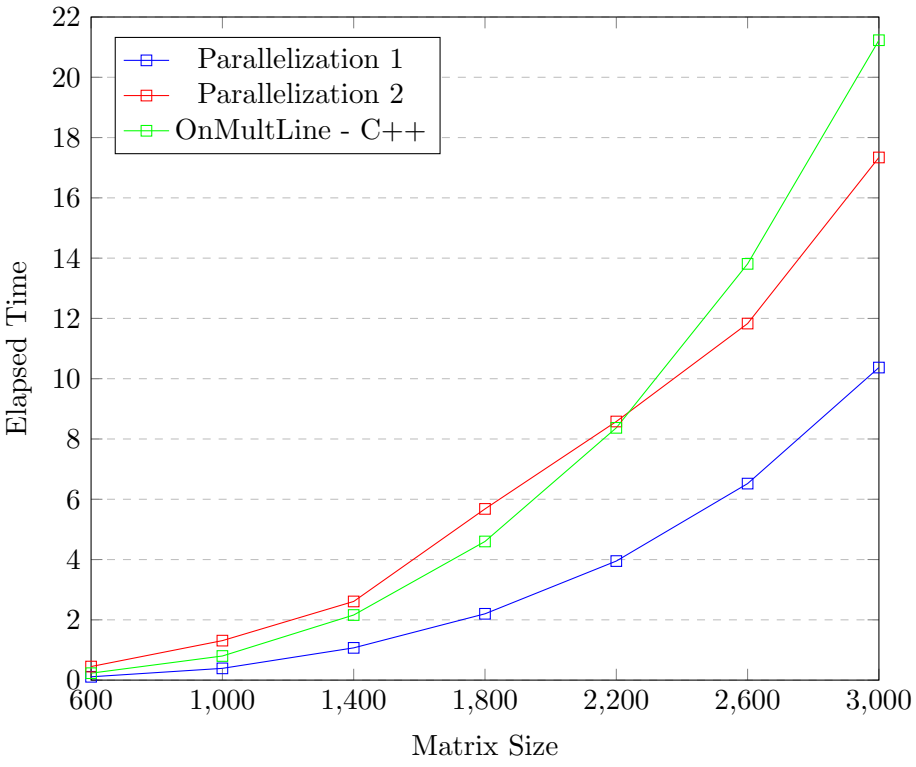
Total Cycles Analysis for Big Matrices (C++)



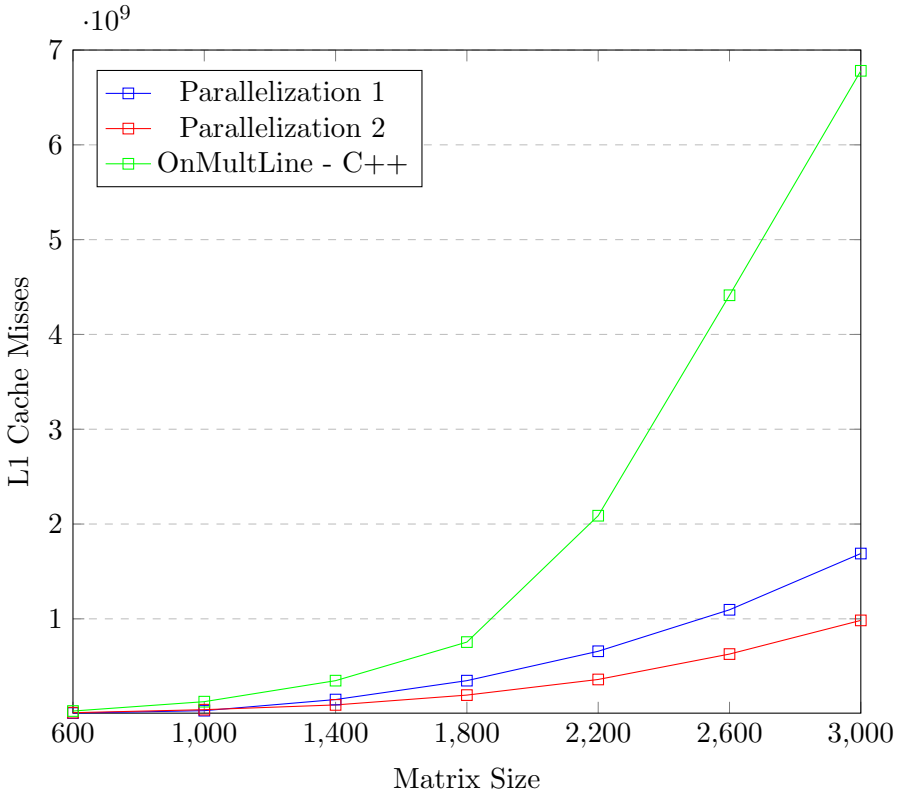
Total Cycles Analysis for Big Matrices (Julia)



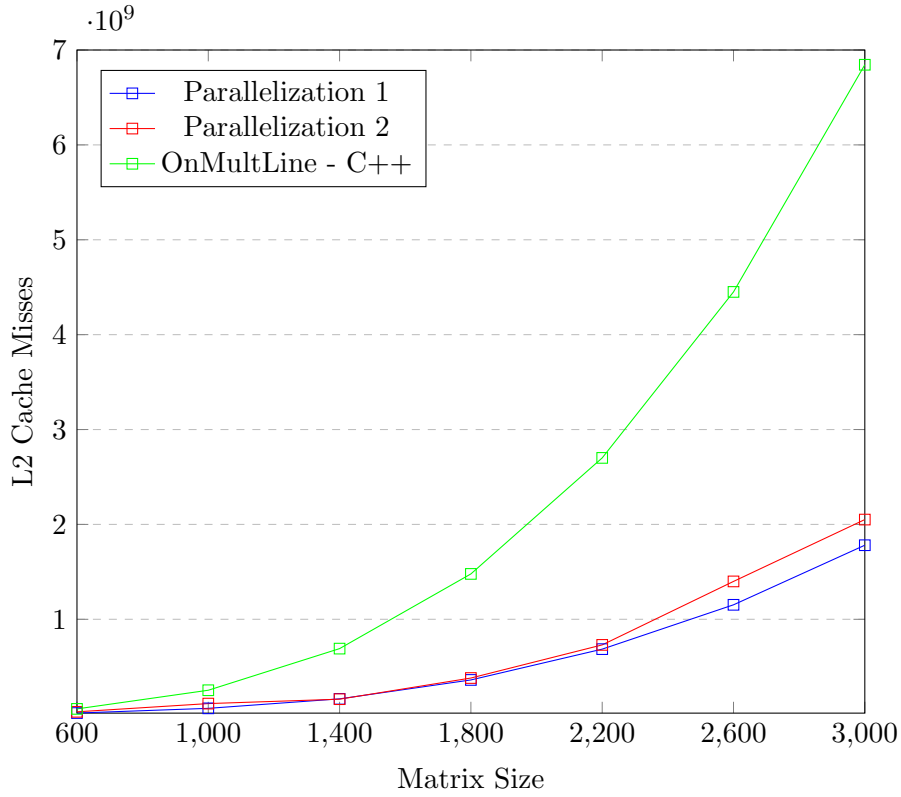
Elapsed Time Analysis for Parallelization



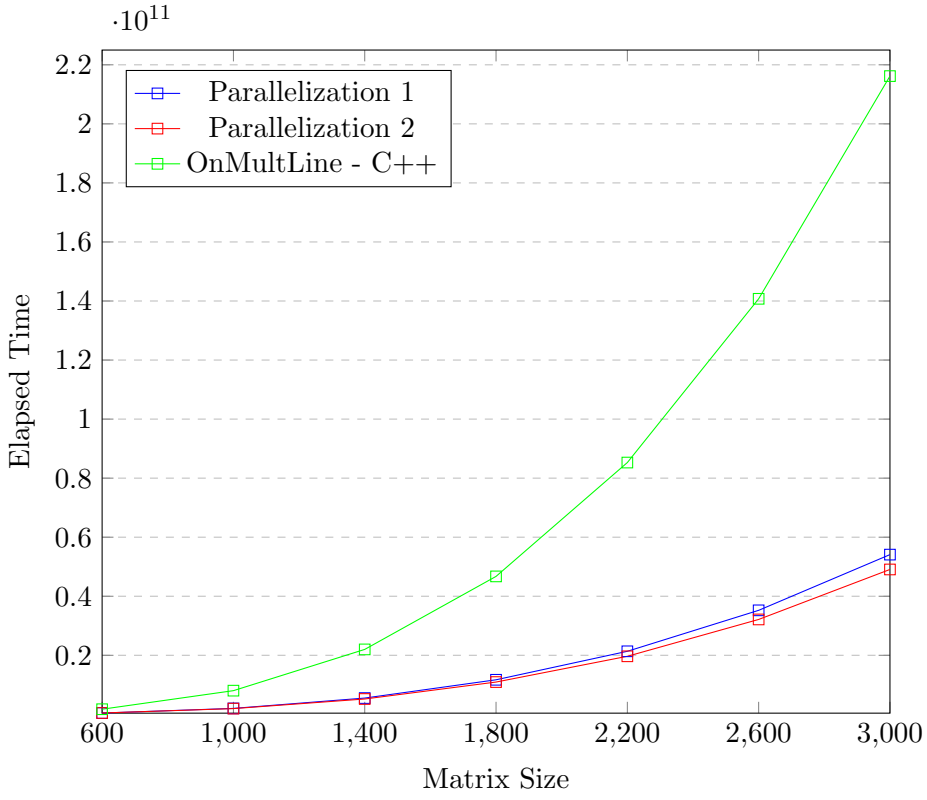
Level 1 Data Cache Misses Analysis for Parallelization

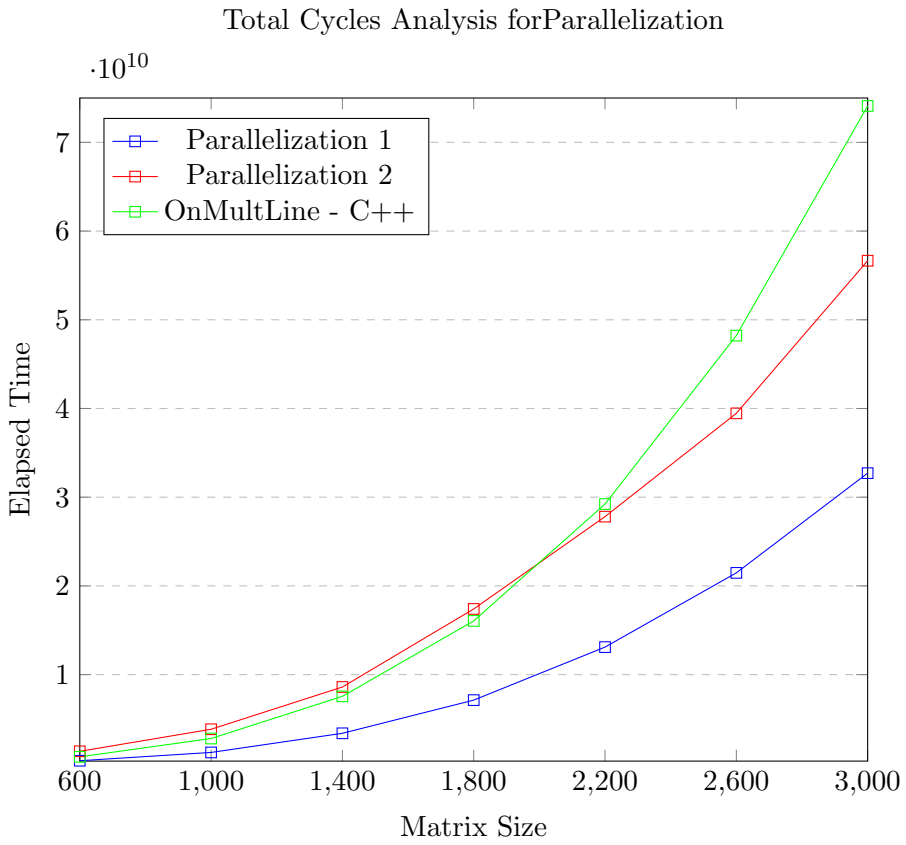


Level 2 Data Cache Misses Analysis for Parallelization



Total Instructions Completed Analysis for Parallelization




Total Cycles Analysis forParallelization
$\cdot 10^{10}$
Parallelization 1
Parallelization 2
OnMultLine - C++
Elapsed Time
7
6
5
4
3
2
1
600
1,000
1,400
1,800
2,200
2,600
3,000
Matrix Size