Faculdade de Engenharia da Universidade do Porto

Performance Evaluation of Single Core and Multi-Core Implementations

First Practical Project Report

CPD 2024/2024 - LEIC

Authors:

Domingos Neto - up202108728@up.pt
Eduardo Baltazar - up202206313@up.pt
Joana Noites - up202206284@up.pt

Introduction

This work was developed as part of the curricular unit of CPD - Computação Paralela e Distribuída - and was based on the teachings of this unit.
In this report, we will analyze the effect on the processor performance of the memory hierarchy when accessing large amounts of data.

Problem Description

In this project, the main objective is to analyze the performance of a processor during the execution of a matrix multiplication operation. Performance metrics, such as execution time and data cache misses for Level 1 (L1) and Level 2 (L2) caches, will be evaluated to assess single-core performance. These metrics will also be extended to a multi-core evaluation, with the addition of measuring the millions of floating-point operations per second (MFLOPS) to further measure computational efficiency. By examining these factors, the aim is to study how cache utilization and parallel processing impact overall system performance.

Algorithms

To evaluate single-threaded performance, three distinct matrix multiplication algorithms were implemented and tested. The first algorithm (**Simple Multiplication**), provided in the work statement, performs a standard matrix multiplication. The second algorithm (**Line Multiplication**) multiplies an element from the first matrix with the corresponding row of the second matrix. The third algorithm (**Block Multiplication**) introduces a block-oriented approach, where the matrices are divided into smaller sub-matrices, and the multiplication is performed on them. **Simple Multiplication and Line Multiplication** were also implemented in Java so as to compare the performance of different languages for the same algorithm.

For multi-threaded performance evaluation, the **Line Multiplication** algorithm was modified according to two different solutions provided to us. The first solution parallelizes the outermost *for* loop, while the second one does the same to the innermost *for* loop. These modifications allow us to analyze how the placement of parallelism impacts the algorithm's performance

Simple Multiplication

As mentioned earlier, this algorithm, provided in the work statement, iterates over the rows of the first matrix and multiplies them by the corresponding columns of the second matrix to compute the resulting matrix product. (Refer to Annex A1.1)

Line Multiplication

This algorithm iterates over the elements of the first matrix and multiplies each element with the corresponding row of the second matrix, accumulating the results to produce the final matrix product. (Annex A1.2)

Block Multiplication

This algorithm divides the matrices into smaller blocks of size *bkSize* x *bkSize*. It iterates over these blocks, multiplying corresponding blocks from the two matrices and storing the results into the output matrix.(Annex A1.3)

Line Multiplication with parallel threads - First Solution

In this solution OpenMP's `#pragma omp parallel for` is used on the outermost for loop, this way, each thread independently processes a portion of the iterations of the `i` loop, while the inner loops execute sequentially within each thread. (Annex A1.4)

Line Multiplication with parallel threads - Second Solution

In this solution OpenMP's `#pragma omp parallel for` is used on the innermost *for* loop, this way, each thread processes a portion of the `j` loop iterations independently, while the outer loops (`i` and `k`) run sequentially. (Annex A1.5)

Performance Metrics

To evaluate the performance of the implemented matrix multiplication algorithms, several metrics were used. The main and most illustrative metric is **execution time**, which was measured for every algorithm considering differently sized matrices. This provided a direct comparison of the computational efficiency of every algorithm.

In addition to the execution time, performance counters such as the **Level 1 and Level 2 Data Cache Misses** (referred as L1 DCM and L2 DCM, respectively) were also accounted for, they were measured using the Performance API tool (PAPI). Measuring Data Cache Misses allows us to evaluate how efficiently each algorithm accesses the memory.

For the block-oriented algorithm, the impact of **block size** on performance was analyzed by testing different block sizes.

For the parallel implementations, **speedup** and **efficiency** were calculated to quantify the performance gains achieved through parallelization. Speedup was computed as the ratio of the sequential execution time to the parallel execution time. Efficiency, on the other hand, was derived by dividing the speedup by the number of cores used, which in our case was 8. Additionally, the computational throughput of the algorithms was measured in terms of Millions of Floating-Point Operations per Second (**MFLOPS**). This metric was calculated using the formula:

MFLOPS = (2 × MatrixSize^3) / (Time × 10^6)

where *MatrixSize* represents the dimension of the matrix and *Time* is the execution time in seconds.

All measurements were performed on the same machine at the computer rooms at FEUP, with an Intel Core i7 processor and the Ubuntu 22.04 operating system. The optimization flag *-O2* was used on C++ algorithms. It should also be noted that each measurement was performed 3 times so as to avoid deviation of results.

Results and Analysis

Simple Multiplication

The execution time increases exponentially as the matrix size grows. This is expected due to the cubic complexity of the algorithm (Annex A2.2).
When comparing the execution time between Java and C++, we can conclude that C++ outperforms Java at a consistent rate, this is likely due to the use of the optimization flag *-O2* and the lower level of memory management of C++, as it is faster than Java's garbage collector.
In this implementation, the innermost loop iterates over the columns of the second matrix (`phb`), accessing memory in a non-sequential manner. This results in poor cache utilization, as each access to `phb[k*m_br + j]` may cause a cache miss, especially for large matrices. This explains the high number of L1 and L2 data cache misses (DCM) observed.

Line Multiplication

Lower execution times are observed for both Java and C++ when using the Line Multiplication algorithm compared to Simple Multiplication (Annex A3.2). For instance, for a 3000x3000 matrix, Line Multiplication took 16.48 seconds in C++, as opposed to 118.07 seconds for Simple Multiplication. Similarly, in Java, the execution time was reduced from 238.63 seconds to just 17.78 seconds. Notably, in Line Multiplication, the execution times for Java and C++ are very similar (Annex A3.4), unlike in Simple Multiplication, where C++ significantly outperformed Java. This improvement is likely due to Line Multiplication's optimized memory access patterns, which reduce the overhead of the Java Virtual Machine (JVM) and make Java's performance more comparable to C++.

In Line Multiplication, the order of the loops is rearranged so that the innermost loop iterates over the columns of the result matrix (`phc`) and the second matrix (`phb`). This ensures sequential memory access to `phb[k*m_br + j]`, improving cache utilization. As a result, the number of cache misses is significantly reduced (Annex A3.3). For example, Line Multiplication results in 6.78 billion L1 DCM and 6.30 billion L2 DCM for a 3000x3000 matrix, compared to Simple Multiplication's 50.3 billion L1 DCM and 95.8 billion L2 DCM for the same matrix size.

Since cache misses force the CPU to fetch data from the slower main memory, Line Multiplication's cache-friendly design demonstrates that reducing cache misses directly decreases execution time.

Block Multiplication

This algorithm takes the performance of both Simple Multiplication and Line Multiplication up a notch (Annex A4.2 and A4.4). For instance, for a 4096x4096 matrix with a block size of 256, Block Multiplication took 27.26 seconds, compared to 41.23 seconds for Line Multiplication and an estimated even longer time for Simple Multiplication, since this last algorithm takes about 118.07 seconds to compute the product of 3000x3000 sized matrices. Similarly, for a 10240x10240 matrix with a block size of 512, Block Multiplication completed in 428.92 seconds, while Line Multiplication took 655.97 seconds).

By dividing the matrices into blocks, the algorithm ensures that the working data *fits in* the cache, lowering the number of L1 and L2 data cache misses (DCM) (Annex A4.3). For instance, for a 4096x4096 matrix with a block size of 256, Block Multiplication resulted in 9.06 billion L1 DCM and 23.02 billion L2 DCM, compared to Line Multiplication's 17.55 billion L1 DCM and 16.25 billion L2 DCM for the same matrix size. This reduction in cache misses directly translates to faster execution times.

The choice of block size also plays a crucial role in performance. Smaller block sizes often result in higher cache misses and longer execution times, while larger block sizes strike a balance between cache utilization and computational efficiency. For example, for a 10240x10240 matrix, a block size of 512 resulted in 136.92 billion L1 DCM and 311.98 billion L2 DCM, compared to 150.47 billion L1 DCM and 512.66 billion L2 DCM for a block size of 128.

In conclusion, Block Matrix Multiplication outperforms both Simple and Line Multiplication by optimizing cache utilization through its block-oriented design. This approach reduces cache misses and execution times, demonstrating the importance of memory access patterns in high-performance computing. The results also highlight the impact of block size on performance, with larger blocks generally providing better efficiency for large matrices, as less blocks are required to cover the entire matrix and more data is loaded to the cache at once.

Line Multiplication with parallel threads - First Solution

According to the results, parallelizing the *i* loop iterations (Annex A1.4) leads to a much less acute exponential growth in both cache misses and execution time (Annexes A5.2 and A5.3).

This can be trivially explained by the fact that the work is being divided into more threads and, therefore, being executed at the same time. A speedup, in relation to the unparallelized line multiplication, of more than 6 times can be observed as up to 8 threads are being used.

However, this decrease in calculation time falls shorter as the matrix size increases, which is due to several factors, such as the increase in the overhead, the load distribution and cache limitations.

The overhead, which handles the division of the workload and the joining of the calculated values, cannot itself be parallelized. As the operations become increasingly larger, dividing the work becomes a more resource intensive task, leading to less time spent on the matrix calculations.

Another factor is the increase in variability of the duration of execution for each thread - even though the load distribution is usually done fairly, as the execution time in each thread increases, so does the time that other threads may need to wait for.

Finally, all of the threads are sharing the same caches and therefore, when more cache accesses are needed, more collisions may occur which makes the threads have to wait longer for those accesses.

With all of this in mind, we can conclude that, even though the parallelization of the calculation clearly leads to an improved performance, the results demonstrate some details in the way it functions that result in some losses in speedup in exponential complexity work.

Line Multiplication with parallel threads - Second Solution

In this solution, the *j* iterations are executed in parallel, i.e., less work is being parallelized more often. As the results show, this is clearly the less optimal scenario of the two when it comes to using multiple threads. The speedup in relation to non parallel line multiplication (Annex A6.1), even though positive in most cases, far underperforms when compared to the first solution. The number of threads may be the same, but the MFLOPS are noticeably less, which leads to a lower efficiency (Annexes A6.6 and A6.7).

This is a poor setup for parallelizing work as the factors that take part in reducing the speedup in the first solution are even more evident. The overhead work is much more complex as the execution is being distributed much more frequently, which increases the amount of sequential work that needs to be done. This can be observed as, in low matrix sizes, the speedup is negative (Annex A6.1) in comparison to the single thread algorithm. If we take into consideration the use of several threads, this process seems even less helpful as a lot of cores are being kept busy for minimal gains.

Conclusions

This project helped us achieve a better understanding of how several cores may be used to improve a process's performance and how to better decide what sections of the code should be parallelized. The overhead was shown to affect execution time significantly and reducing it is important to develop a good multi-thread program. It has also shown us that the way we design the matrix multiplication algorithm, even when using a single thread, may affect the performance, as evidenced by the greater performance of the block multiplication.

Annexes

A1 - Code Snippets

A1.1 - Simple Multiplication

```
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;
       }
   }
```

A1.2 - Line Multiplication

```
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];
            }
        }
    }
```

A1.3 - Block Multiplication

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

A1.4 - Line Multiplication with parallel threads - First Solution

```
#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];
            }
         }
      }
```

A1.5 - Line Multiplication with parallel threads - Second Solution

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

A2 - Simple Multiplication - Measurements and Graphs

A2.1 - Measurements Table

Simple Multiplication Measurements - C++ vs Java

Size	Average Time (s) - C++	Average Time (s) - Java	Average L1 DCM	Average L2 DCM
600	0.19	0.27	244,695,917	39,368,746
1000	1.35	13.34	1,224,484,542	295,071,267
1400	3.64	14.96	3,465,951,803	1,292,634,886
1800	17.66	36.07	9,085,822,351	4,727,840,269
2200	38.37	77.08	17,627,555,173	18,284,610,671
2600	70.76	121.08	30,904,351,409	50,155,807,448
3000	118.07	238.63	50,289,207,683	95,779,490,469

A2.2 - C++ vs Java Execution Time vs Matrix Size

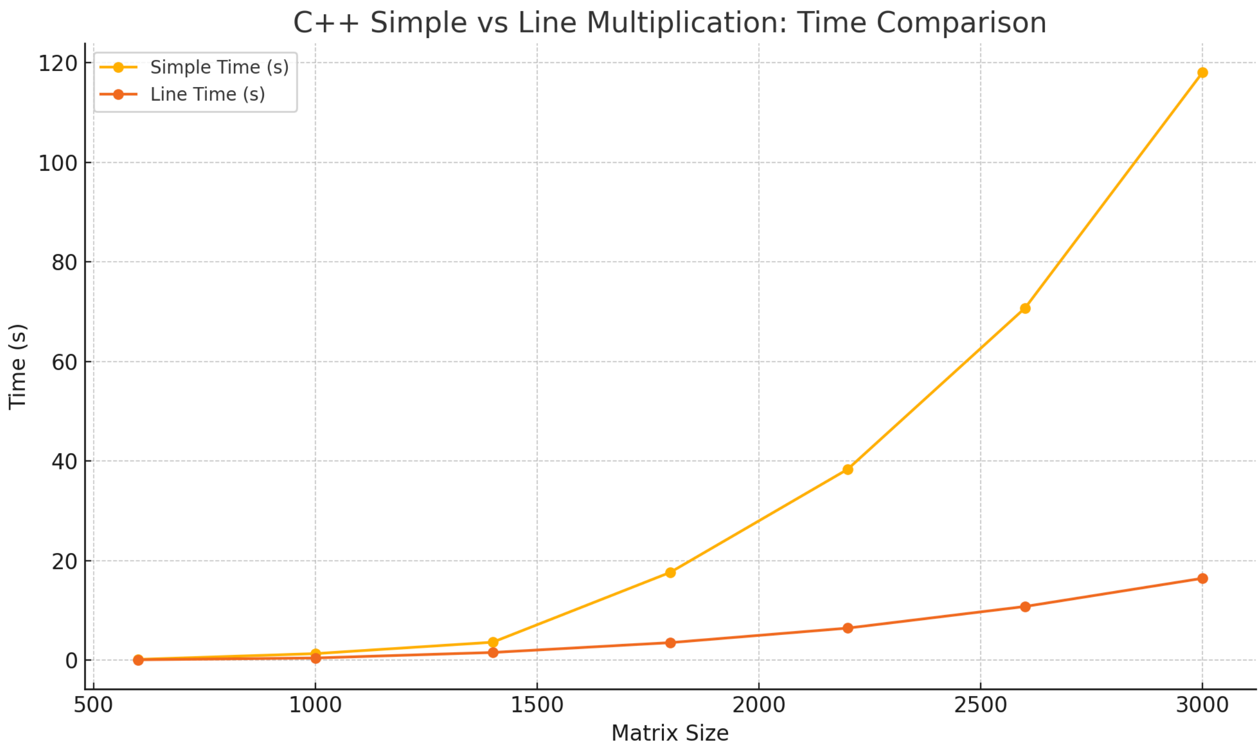
A3 - Line Multiplication - Measurements and Graphs

A3.1 - Measurements Table

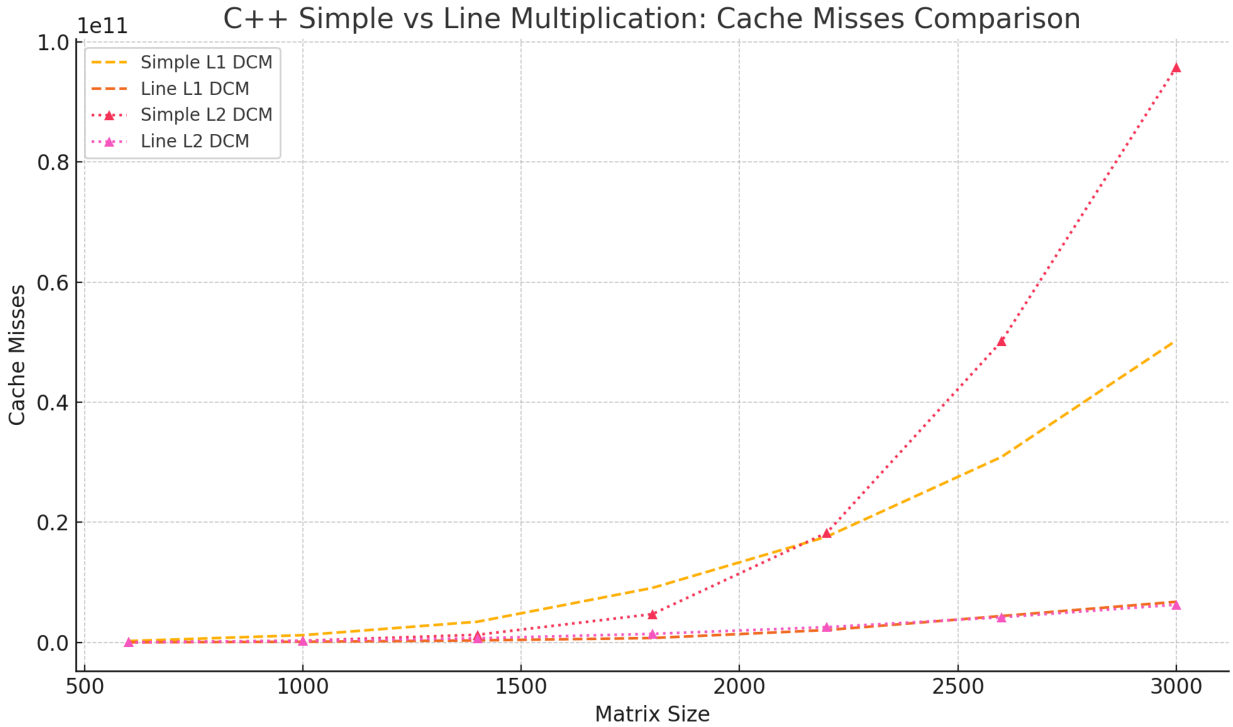
Line Multiplication Measurements - C++ vs Java

Size	Average Time (s) - C++	Average L1 DCM - C++	Average L2 DCM - C++	Average Time (s) - Java
600	0.115	27,111,997	57,787,192	0.12
1000	0.475	125,734,561	267,747,076	0.55
1400	1.583	346,116,682	708,816,444	1.65
1800	3.557	745,271,243	1,439,900,904	3.57
2200	6.473	2,073,886,158	2,561,167,064	6.57
2600	10.819	4,412,845,122	4,198,690,629	11.07
3000	16.478	6,780,698,787	6,296,071,144	17.78
4096	41.229	17,553,265,709	16,249,690,528	nan
6144	139.299	59,151,207,256	54,420,319,247	nan
8192	336.908	140,290,893,728	130,816,392,740	nan
10240	655.973	273,727,875,965	256,199,650,326	nan

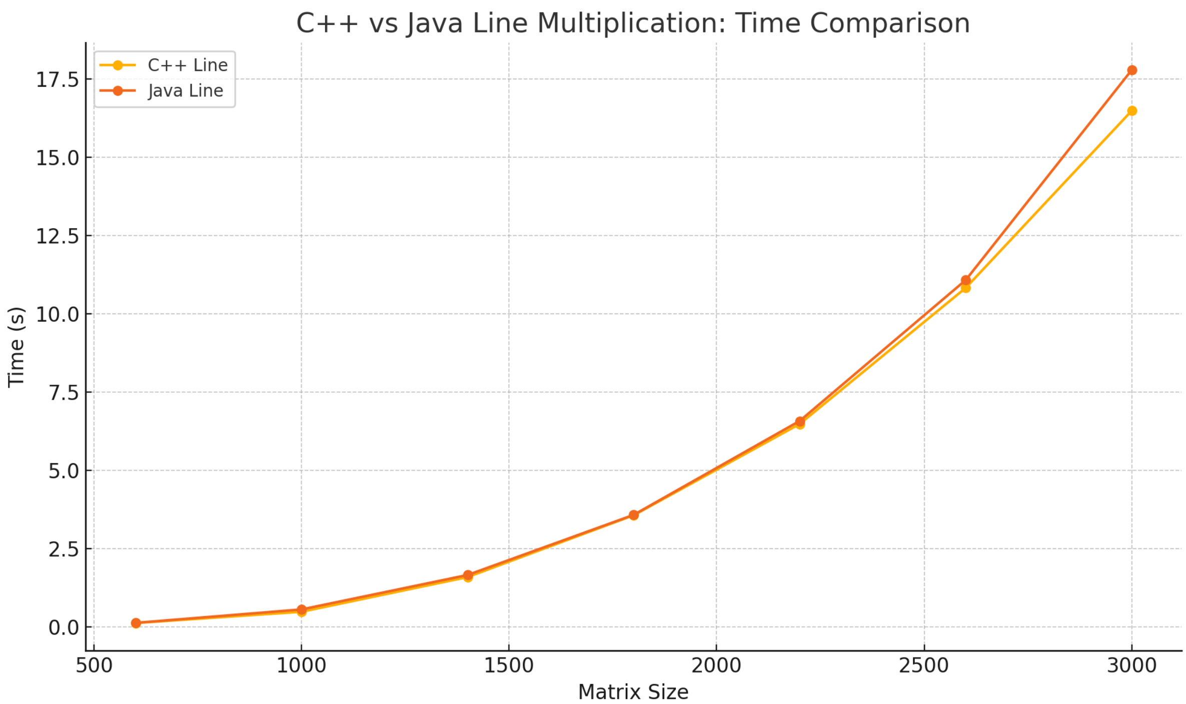
A3.2 - C++ Simple vs. Line Multiplication: Time Comparison



A3.3 - C++ Simple vs. Line Multiplication: Cache Misses Comparison



A3.4 - C++ vs. Java Line Multiplication: Time Comparison



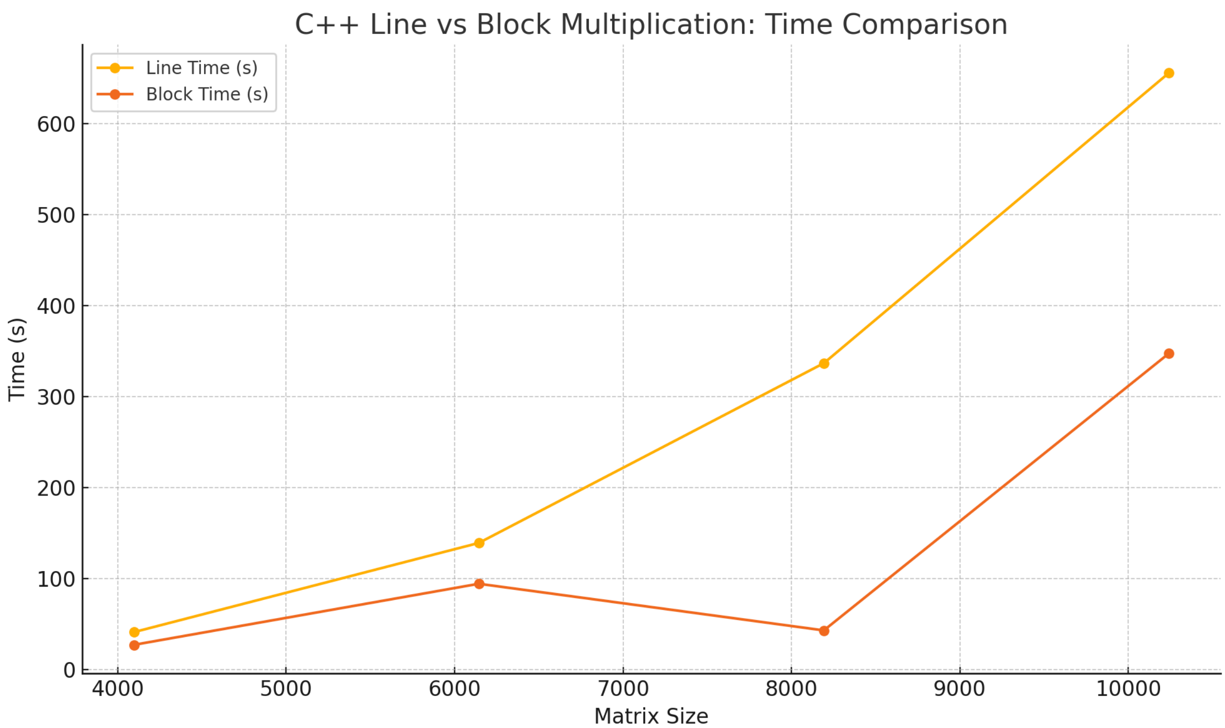
A4 - Block Multiplication - Measurements and Graphs

A4.1 - Measurements Table

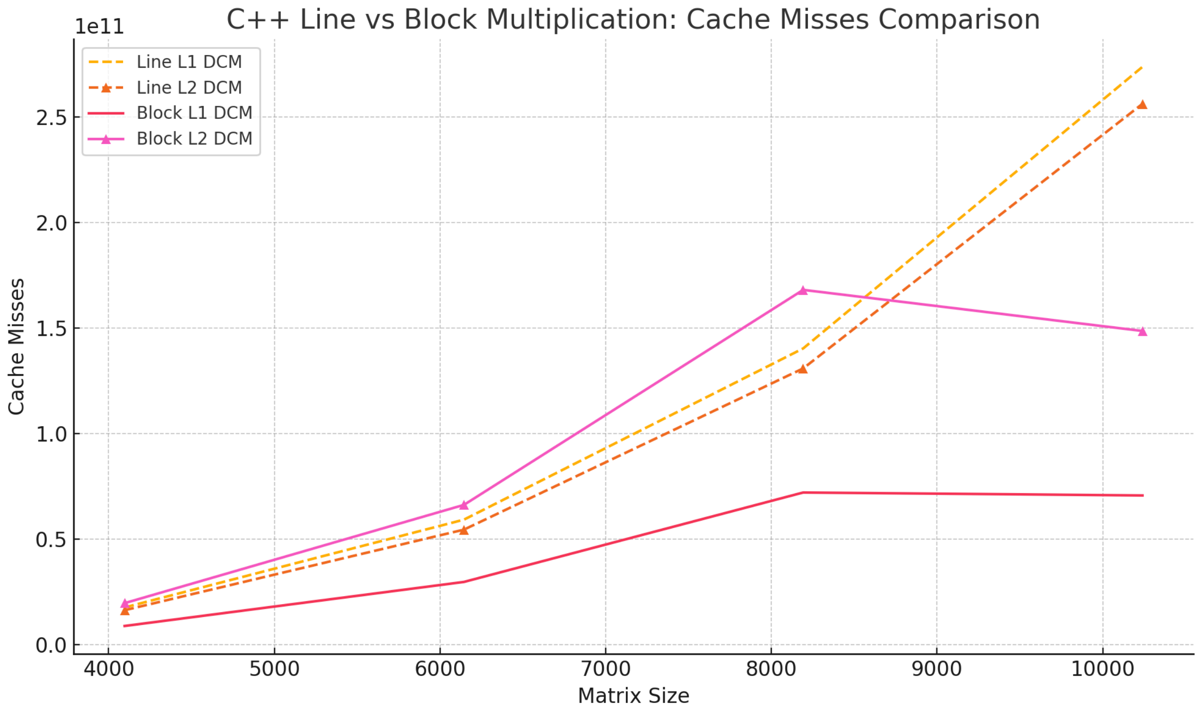
Block Multiplication Measurements - C++

Size	Block Size	Average Time (s)	Average L1 DCM	Average L2 DCM
4096	128	31.34	9,659,705,054	32,416,819,339
4096	256	27.264	9,062,584,580	23,017,786,804
4096	512	28.516	8,768,735,120	19,594,615,939
6144	128	108.801	32,486,518,058	109,657,143,152
6144	256	94.431	30,547,622,618	77,903,305,841
6144	512	96.499	29,641,740,553	66,145,750,807
8192	128	303.677	74,278,262,468	256,665,253,340
8192	256	43.166	72,020,268,392	168,037,026,742
8192	512	347.631	70,629,754,365	148,556,814,500
10240	128	505.63	150,474,644,703	512,657,128,442
10240	256	431.71	141,972,681,471	356,982,368,547
10240	512	428.924	136,921,824,089	311,978,693,469

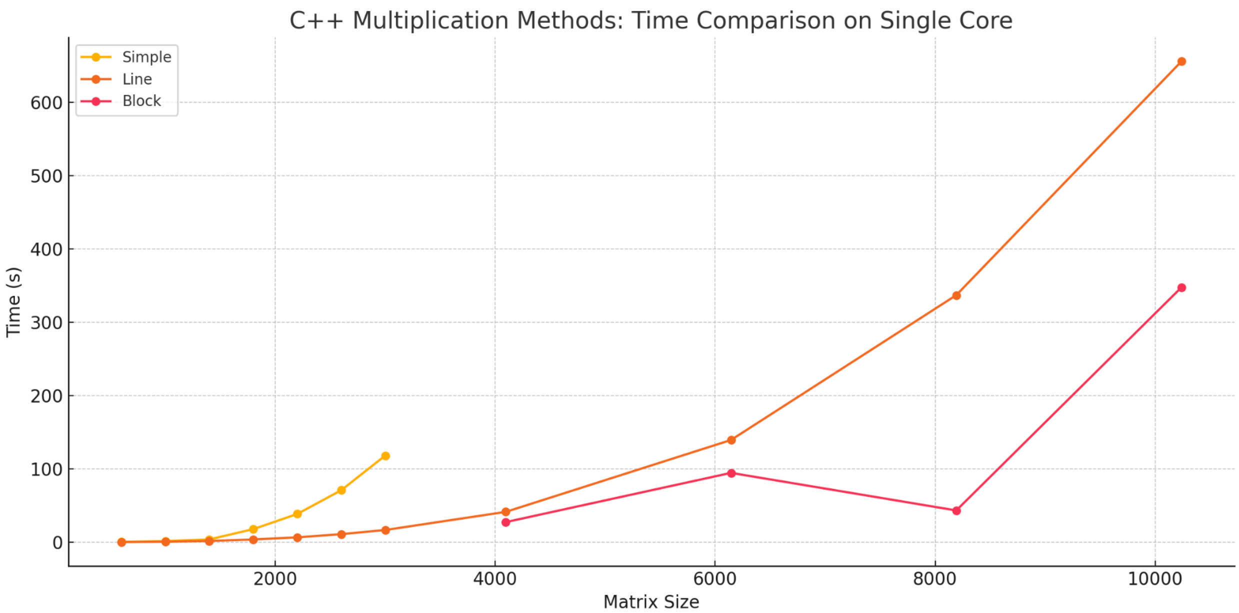
A4.2 - C++ Line vs. Block Multiplication - Time Comparison



A4.3 - C++ Line vs. Block Multiplication - Cache Misses Comparison



A4.4 - C++ Simple vs. Line vs. Block Multiplication - Time Comparison



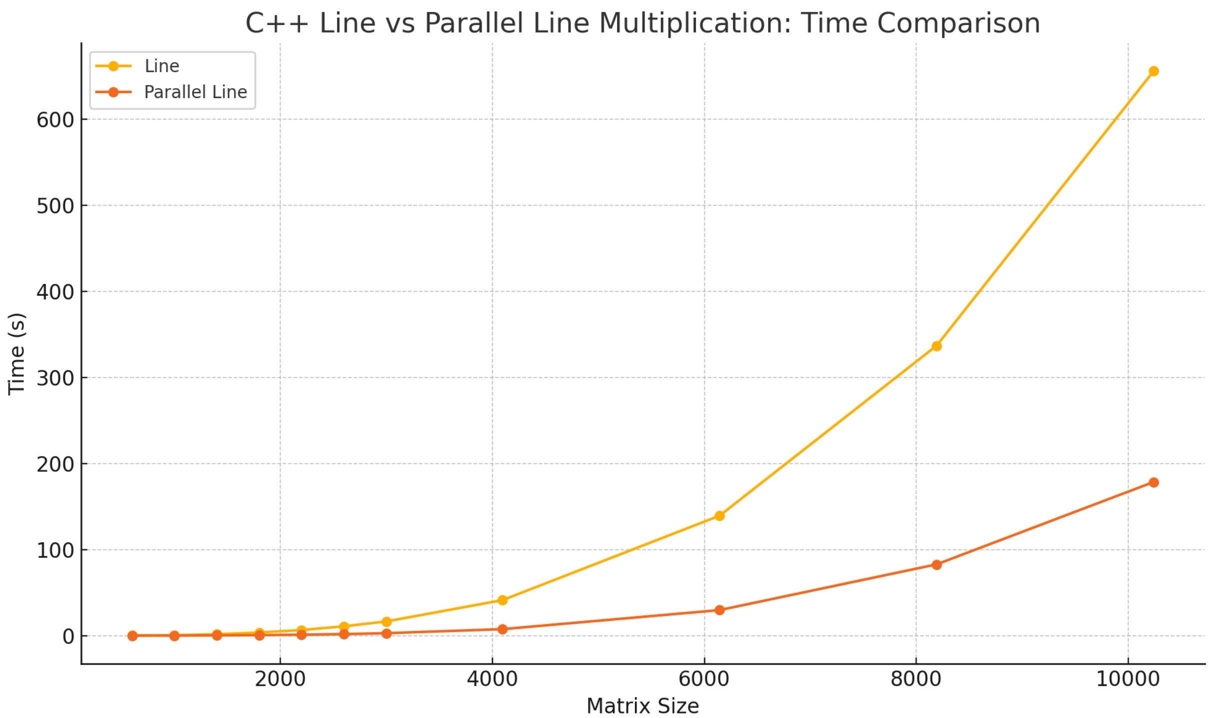
A5 - Line Multiplication with Parallel Threads - First Solution - Measurements and Graphs

A5.1 - Measurements Table

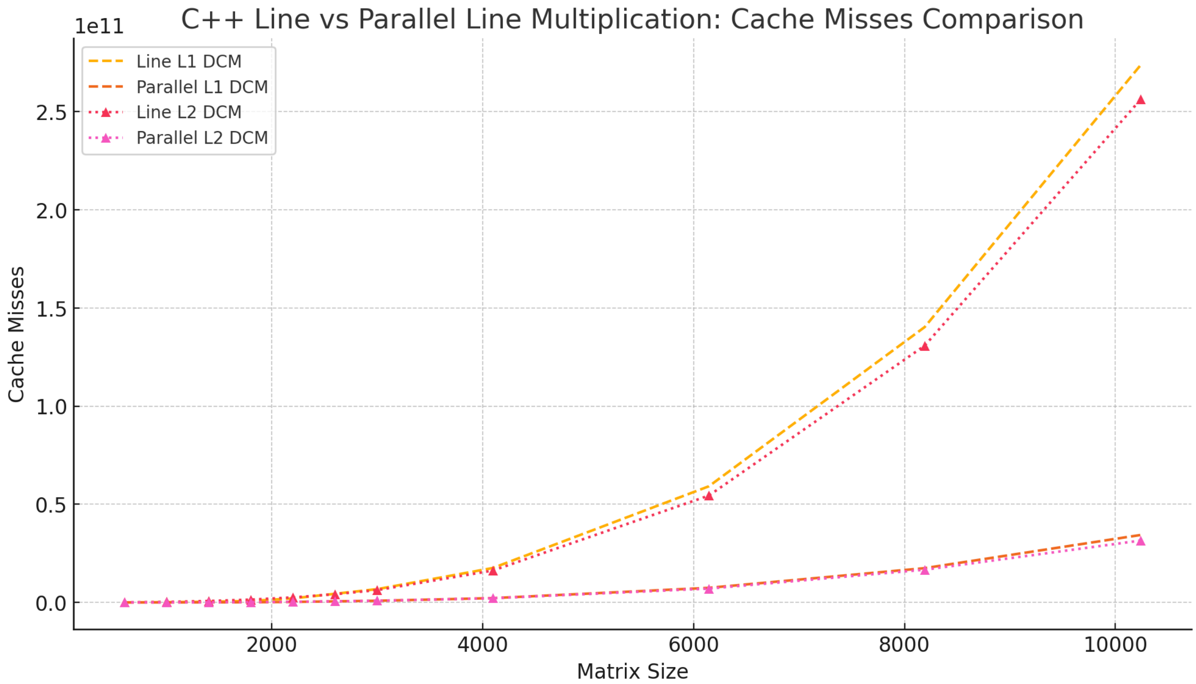
Parallel Multiplication Measurements (First Solution)

Size	Average Time (s)	Average L1 DCM	Average L2 DCM	Mflops	Speedup	Efficiency
600	0.018	3,390,655	7,174,854	24000.0	6.389	79.86%
1000	0.078	15,713,594	32,649,054	25641.03	6.09	76.12%
1400	0.244	43,437,512	87,413,699	22491.8	6.488	81.10%
1800	0.552	93,484,317	184,974,373	21130.43	6.444	80.55%
2200	1.065	258,616,185	331,133,590	19996.24	6.078	75.97%
2600	1.76	549,797,831	548,448,835	19972.73	6.147	76.84%
3000	2.816	845,412,912	836,080,513	19176.14	5.852	73.14%
4096	7.514	2,197,529,677	2,132,961,687	18291.05	5.487	68.59%
6144	29.642	7,431,951,547	7,049,828,317	15648.62	4.699	58.74%
8192	82.798	17,459,326,625	16,672,832,914	13279.45	4.069	50.86%
10240	178.491	34,362,758,953	31,519,192,325	12031.33	3.675	45.94%

A5.2 - Line vs Parallel Line 1: Time Comparison



A5.3 - Line vs Parallel Line 1: Cache Misses Comparison



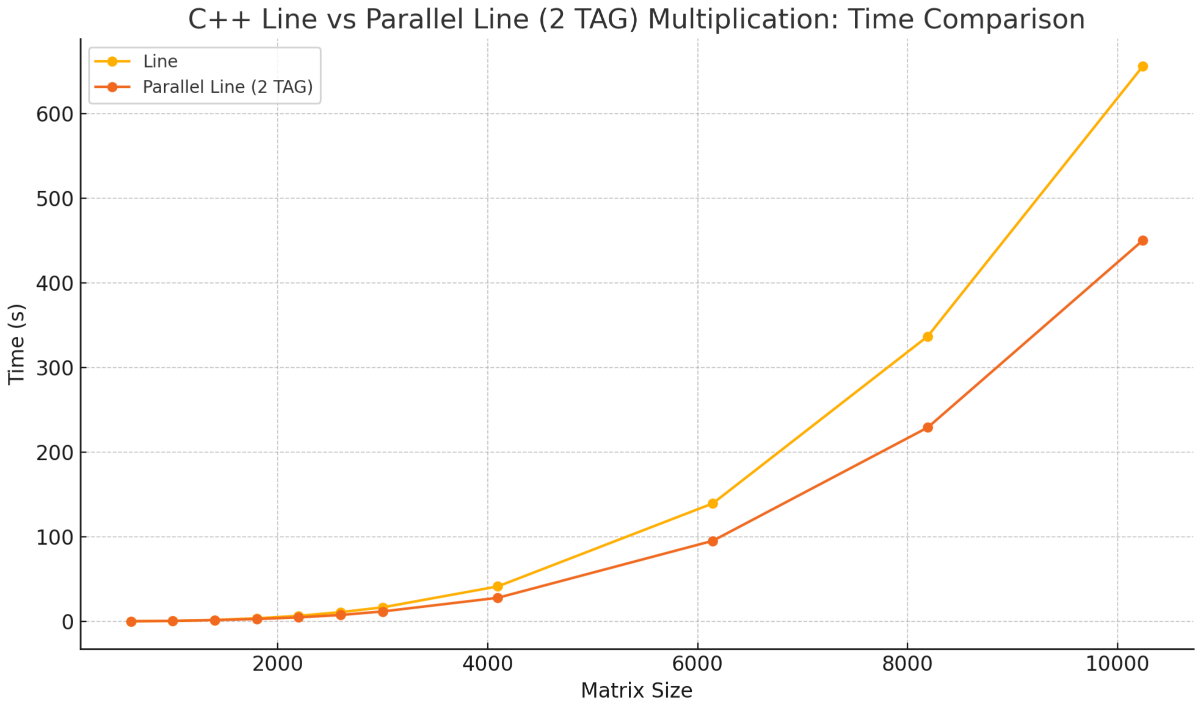
A6 - Line Multiplication with Parallel Threads - Second Solution - Measurements and Graphs

A6.1 - Measurements Table

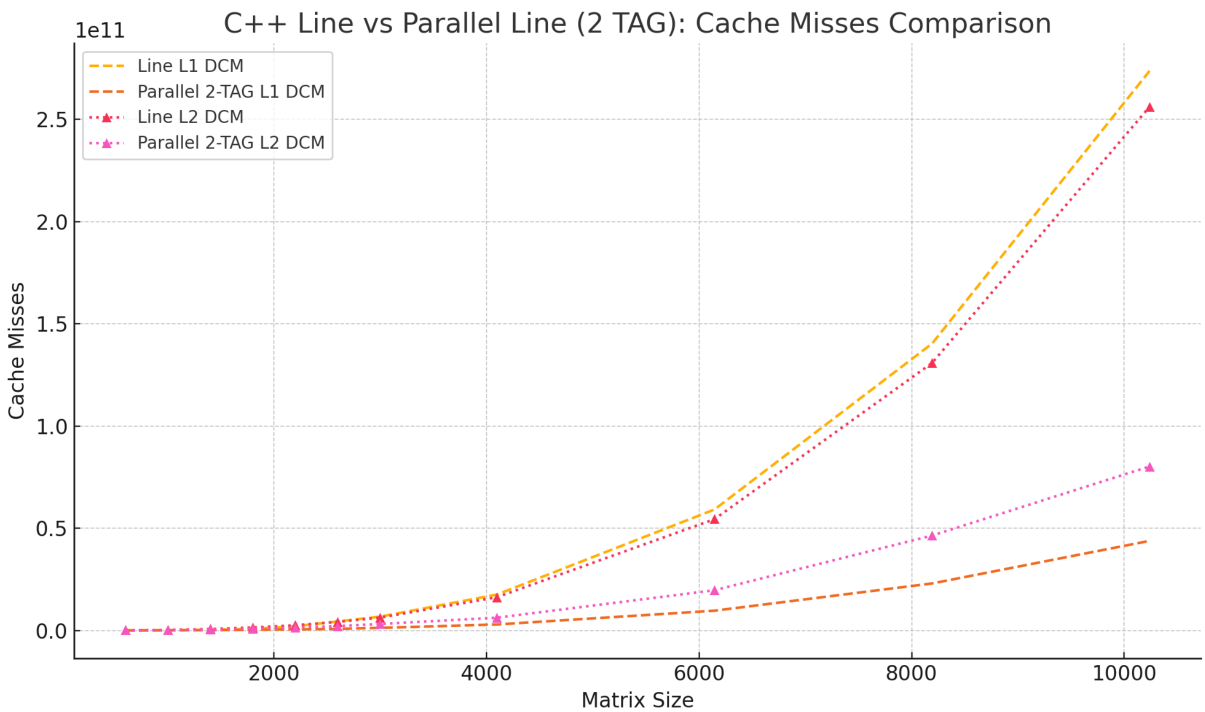
Parallel Multiplication Measurements (Second Solution)

Size	Average Time (s)	Average L1 DCM	Average L2 DCM	Mflops	Speedup	Efficiency
600	0.121	12,687,581	43,001,109	3570.25	0.95	11.88%
1000	0.513	57,322,579	186,063,316	3898.64	0.926	11.57%
1400	1.408	149,451,412	464,926,561	3897.73	1.124	14.05%
1800	2.703	309,991,421	914,588,639	4315.21	1.316	16.45%
2200	4.629	514,954,420	1,457,204,131	4600.56	1.398	17.48%
2600	7.448	782,717,803	2,138,071,919	4719.66	1.453	18.16%
3000	11.662	1,265,554,735	3,173,222,781	4630.42	1.413	17.66%
4096	27.762	2,927,306,005	6,236,507,766	4950.61	1.485	18.56%
6144	95.12	9,667,315,438	19,660,774,794	4876.54	1.464	18.31%
8192	229.101	22,984,441,740	46,412,806,000	4799.24	1.471	18.38%
10240	450.163	43,823,951,312	80,218,070,755	4770.46	1.457	18.21%

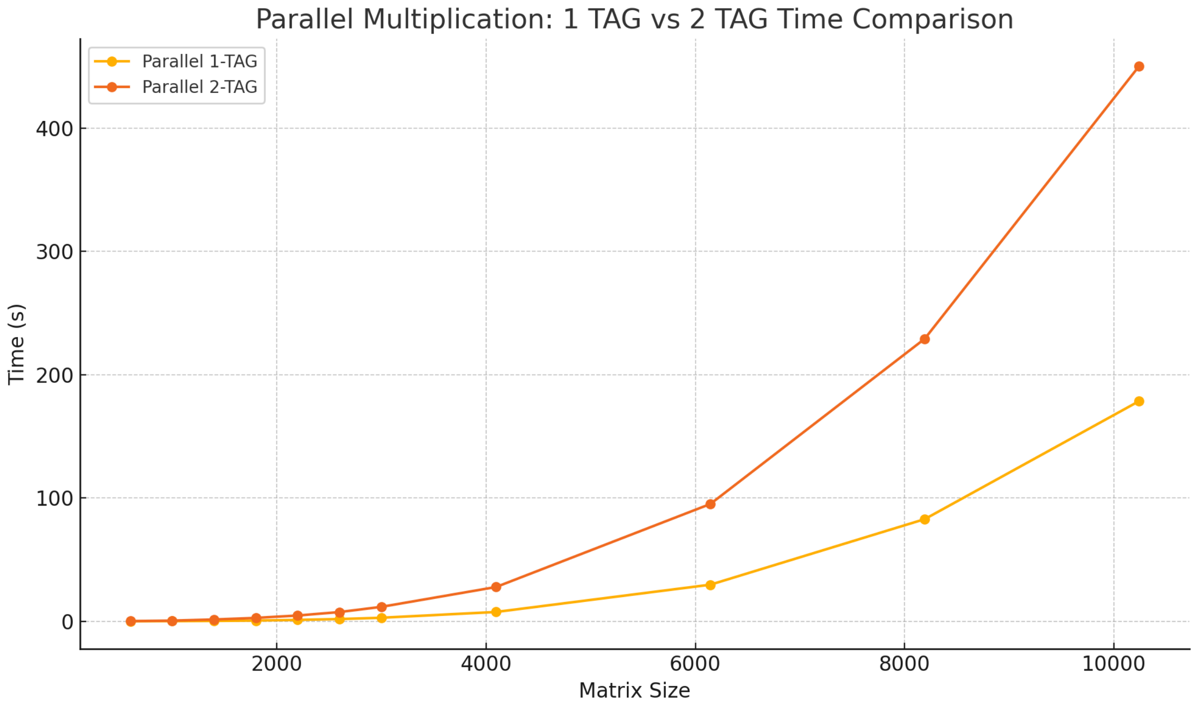
A6.2 - Line vs. Parallel Line 2: Time Comparison



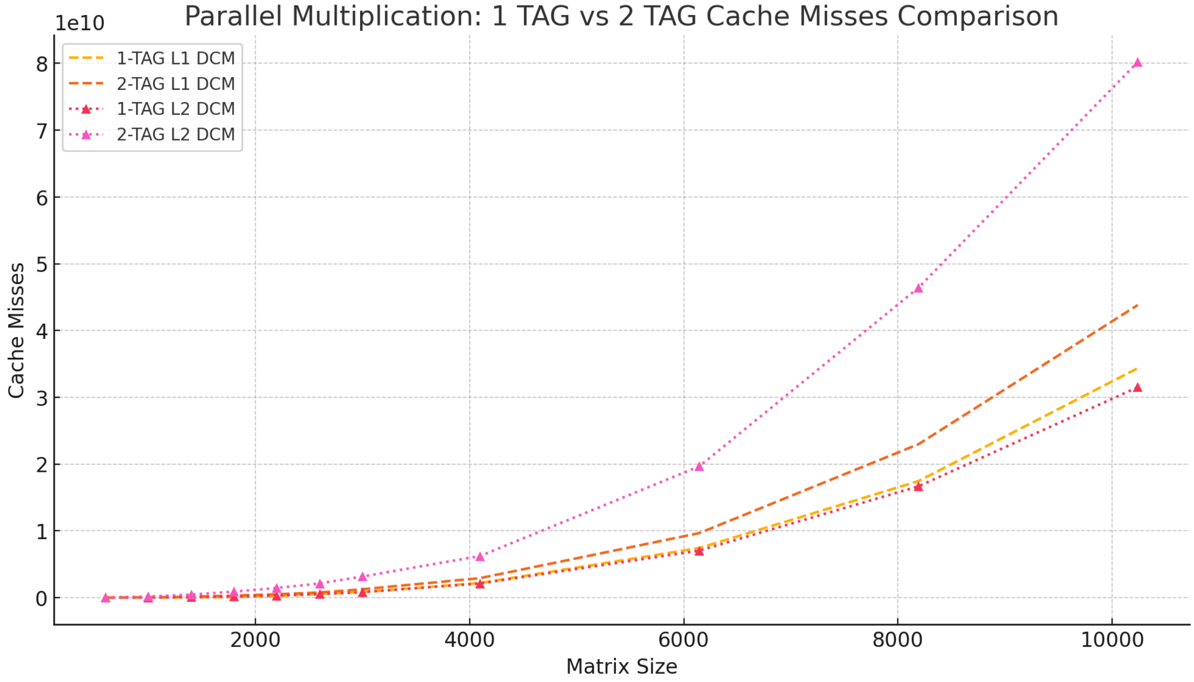
A6.3 - Line vs. Parallel Line 2: Cache Misses Comparison



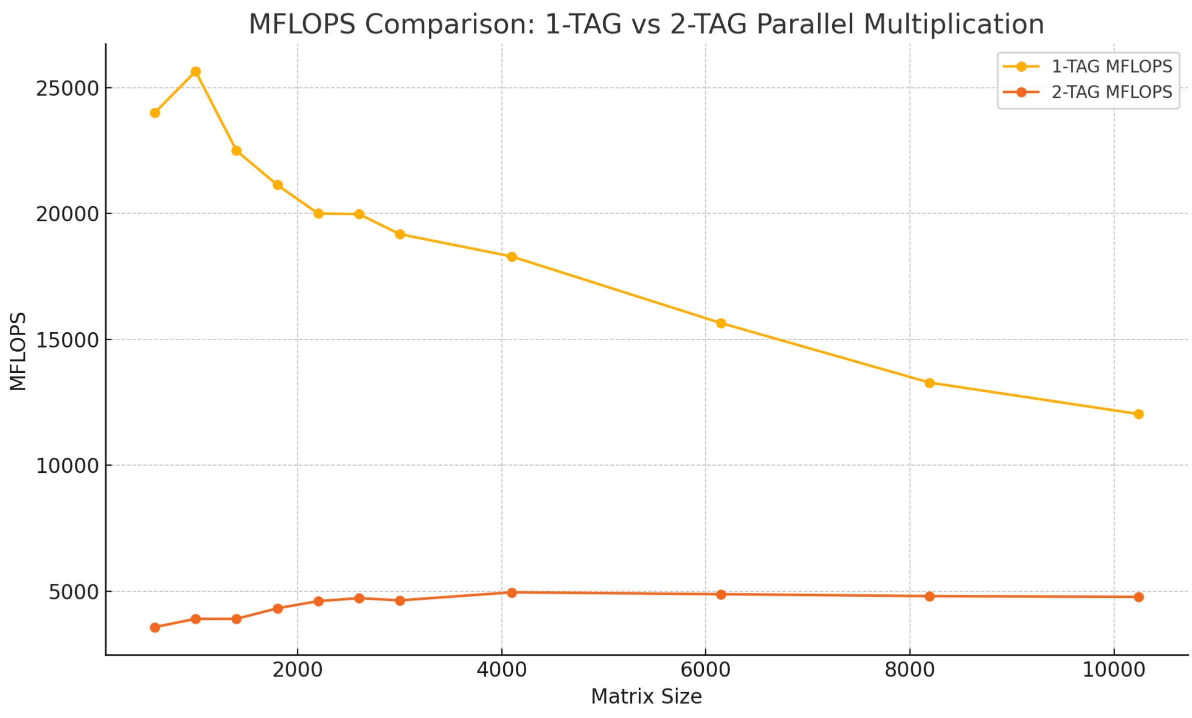
A6.4 - Parallel 1 vs. Parallel 2 - Time Comparison



A6.5 - Parallel 1 vs. Parallel 2 - Cache Misses Comparison



A6.6 - Parallel 1 vs. Parallel 2 - MFLOPS Comparison



A6.7 - Parallel 1 vs. Parallel 2 - Efficiency Comparison