Performance evaluation

Performance evaluation of a single core

In this project, we studied the effect on the processor performance of the memory hierarchy when accessing large amounts of data. The product of two matrices will be used for this study.

Algorithms explanation and Performance metrics

To assess the processor's performance in a given scenario, we're testing two programming languages: C++ and Java.

For C++, we're utilizing the Performance API (PAPI) to gather data cache miss metrics from the Level 1 and Level 2 caches.

In Java, we're focusing on time-based performance comparisons with C++.

To ensure reliability, we're conducting multiple trials, 5 to be precise. Additionally, we're varying the sizes of matrices as outlined in the project requirements.

1. Basic Multiplication

The fundamental multiplication algorithm in linear algebra generates a matrix C from two matrices A and B. This process involves multiplying the elements of the ith row in matrix A with the corresponding elements of the jth column in matrix B and summing these products.

PseudoCode:

```
For each row i in the matrix A
    For each column j in the matrix B
        For each element k in the row i of A
            Multiply k by the elements of j
            Add the product to the corresponding element in the result matrix C[i]
[j]
```

Results

C++

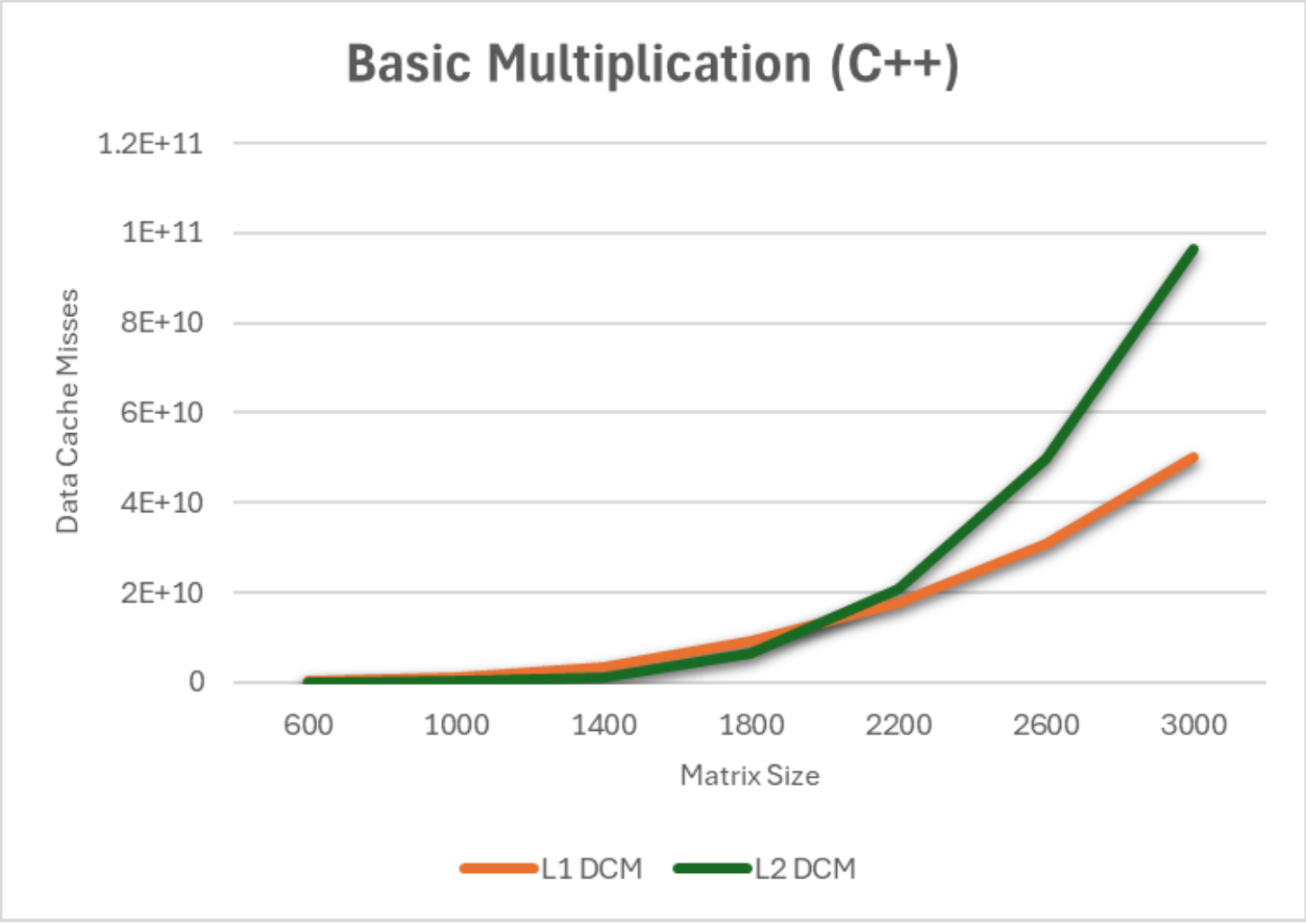
Size	Time(s)	L1 DCM	L2 DCM
600x600	0.193	244747545	39978950
1000x1000	1.196	1215313152	248319156
1400x1400	3.429	3523398807	1253135650
1800x1800	17.836	9062706413	6541764536

Size	Time(s)	L1 DCM	L2 DCM
2200x2200	38.358	17629584035	20673202138
2600x2600	68.457	30874977148	49650891442
3000x3000	116.453	50291188228	96497809106

JAVA

Size	Time(s)
600x600	0.277
1000x1000	3.553
1400x1400	14.564
1800x1800	38.209
2200x2200	69.848
2600x2600	147.208
3000x3000	207.720

Comparison



Analysis

We can observe a predictable rise in the time taken as the matrix size increases. Similarly, the occurrences of data cache misses align with this trend, indicating that larger matrices lead to more frequent memory accesses (with L2 DCM increasing faster than the L1 DCM). Also, we can observe that Java takes more time than C++ because of differences in compilation, memory management, and optimization, with C++ offering lower overhead and more control over low-level optimizations.

2. Line Multiplication

The row multiplication method is a modified version of the standard multiplication technique, in which elements from the ith row of matrix A are multiplied by matching elements from the jth row of matrix B, with the results being compiled in the corresponding position of matrix C.

PseudoCode:

```
For each row i in the matrix A
    For each element j in row i
        For each line k in matrix b
            Multiply j by the elements of k
            Add the product to the corresponding element in the result matrix C[i]
[k]
```

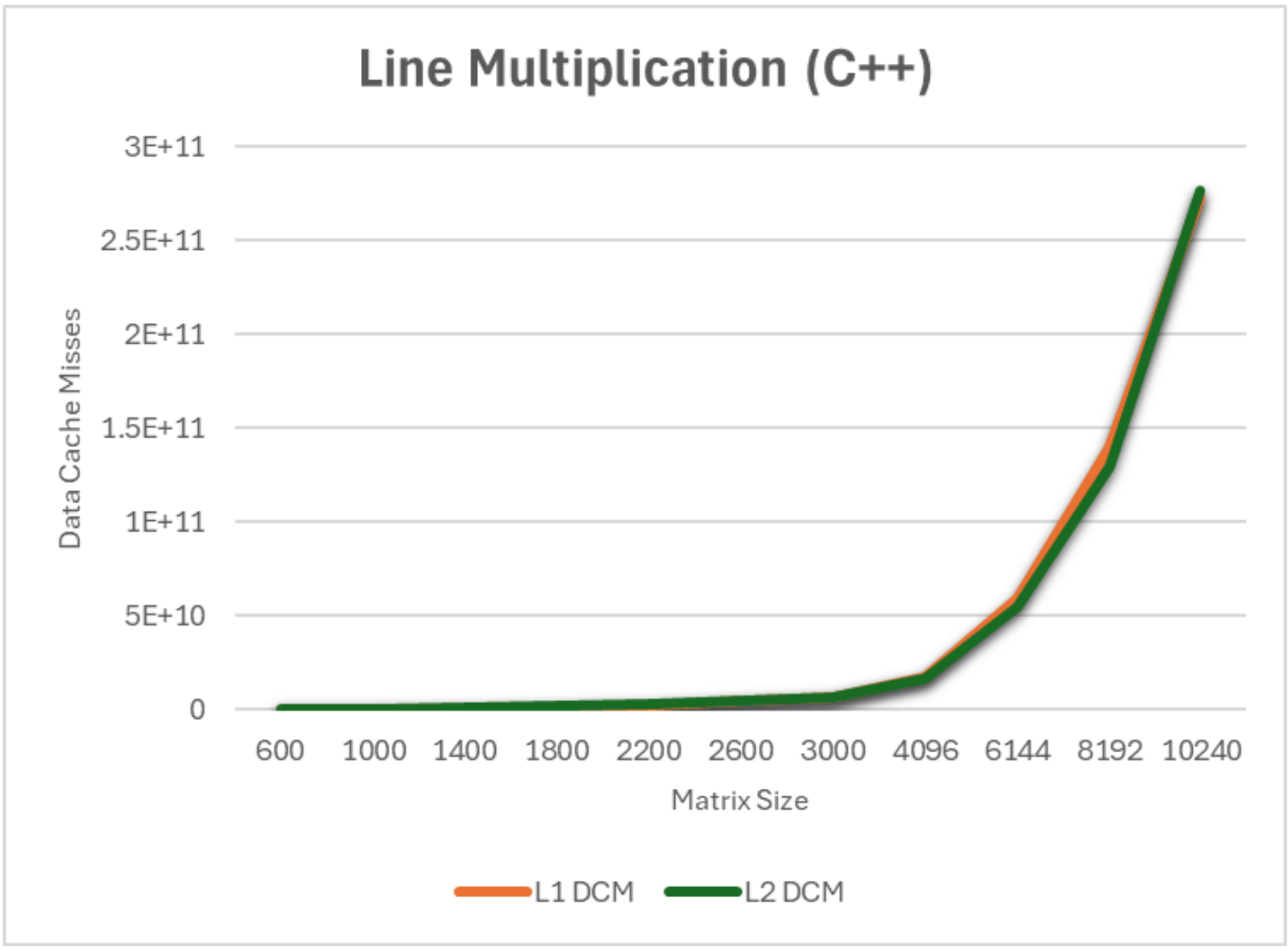
C++

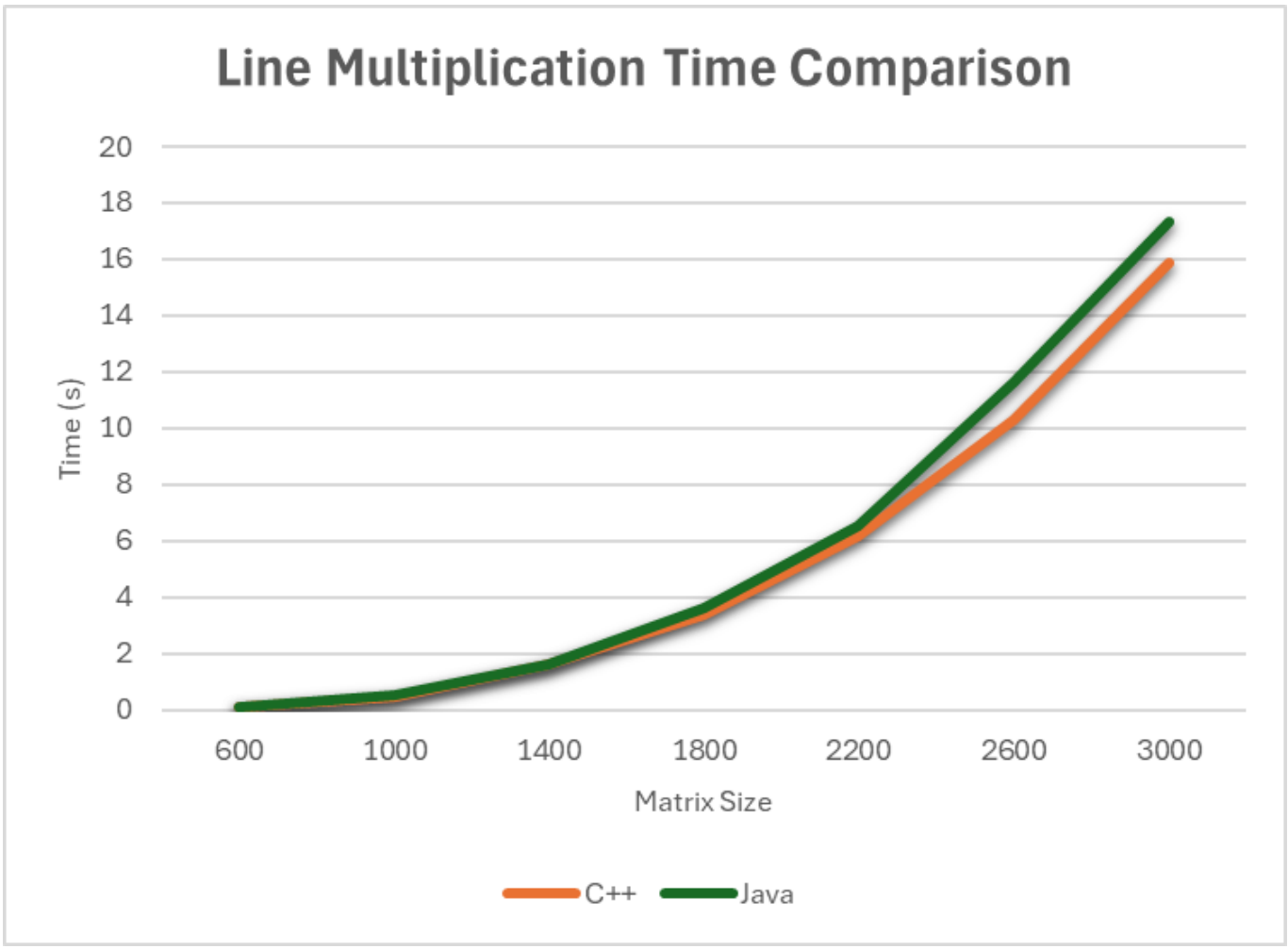
Size	Time(s)	L1 DCM	L2 DCM
600x600	0.106	27111173	58059476
1000x1000	0.495	125642250	260780188
1400x1400	1.598	345960470	702550929
1800x1800	3.394	745461447	1426447348
2200x2200	6.170	2075548635	2527297324
2600x2600	10.313	4413146923	4179519948
3000x3000	15.843	6780661035	6290745735

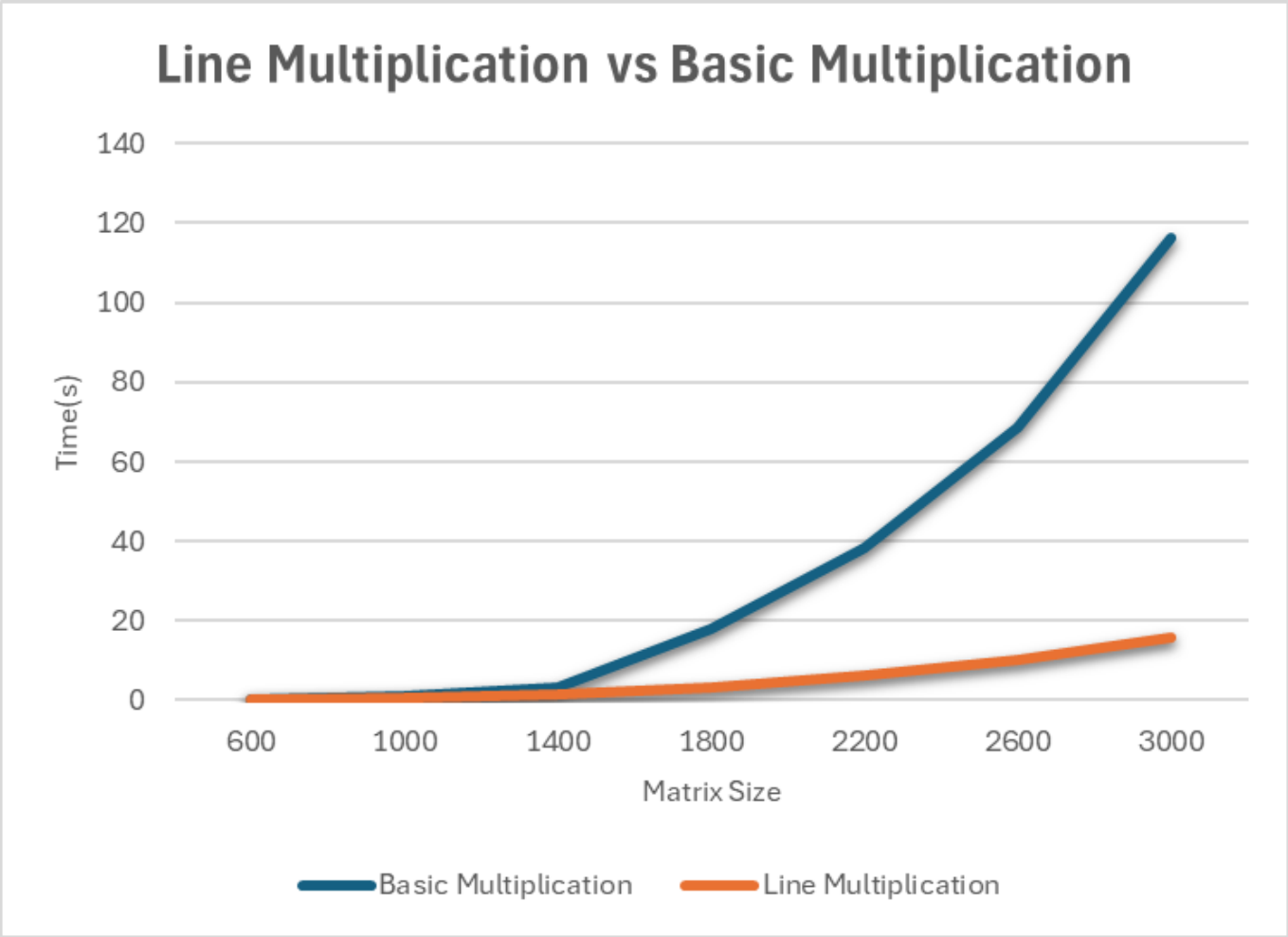
JAVA

Size	Time(s)
600x600	0.115
1000x1000	0.510
1400x1400	1.645
1800x1800	3.649
2200x2200	6.522
2600x2600	11.625
3000x3000	17.300

Comparison


Line Multiplication (C++)
3E+11
2.5E+11
2E+11
1.5E+11
1E+11
5E+10
0
Data Cache Misses
600
1000
1400
1800
2200
2600
3000
4096
6144
8192
10240
Matrix Size
L1 DCM
L2 DCM


Line Multiplication Time Comparison
20
18
16
14
12
10
8
6
4
2
0
Time (s)
600
1000
1400
1800
2200
2600
3000
Matrix Size
C++
Java



Analysis

The performance metrics show a improvement in line multiplication when comparing it to basic multiplication. This improvement stems from the line multiplication algorithm leveraging C++ memory allocation. Essentially, when accessing a particular position in the matrix, the compiler reads not only that value but also the entire block where the value resides. To minimize costly disk memory accesses, which consume time, the line multiplication algorithm aims to utilize all elements read from the memory block effectively. We can also observe that the L1 DCM and L2 DCM are quite the same, and that there is almost no difference between Java and C++ in this algorithm.

3. Block Multiplication

Block multiplication is an algorithm that segments matrices A and B into smaller blocks, leveraging row multiplication to compute the values for matrix C efficiently.

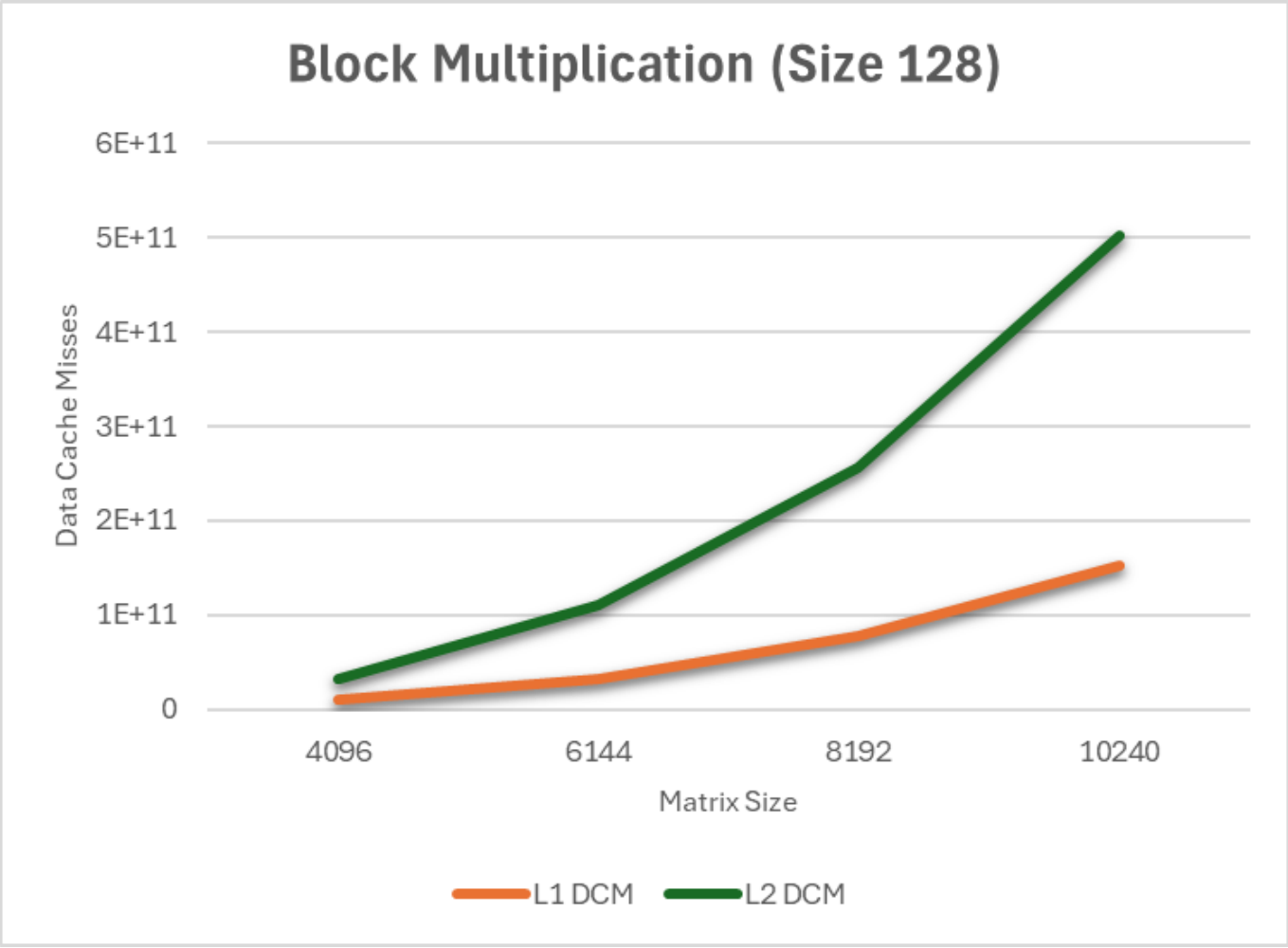
PseudoCode:

```
For each block row i in matrix A
    For each block column j in matrix B
        For each block k in both matrix A and matrix B
            For each row l in the block i of matrix A and block k of matrix B
                For each column m in the block k of matrix A and block j of matrix
B
                    For each element n in the block j of matrix B
```

```
                        Multiply the elements A[l][m] and B[m][n]
                        Add the product to the corresponding element in the result
matrix C[l][n]
```

Block Size: 128

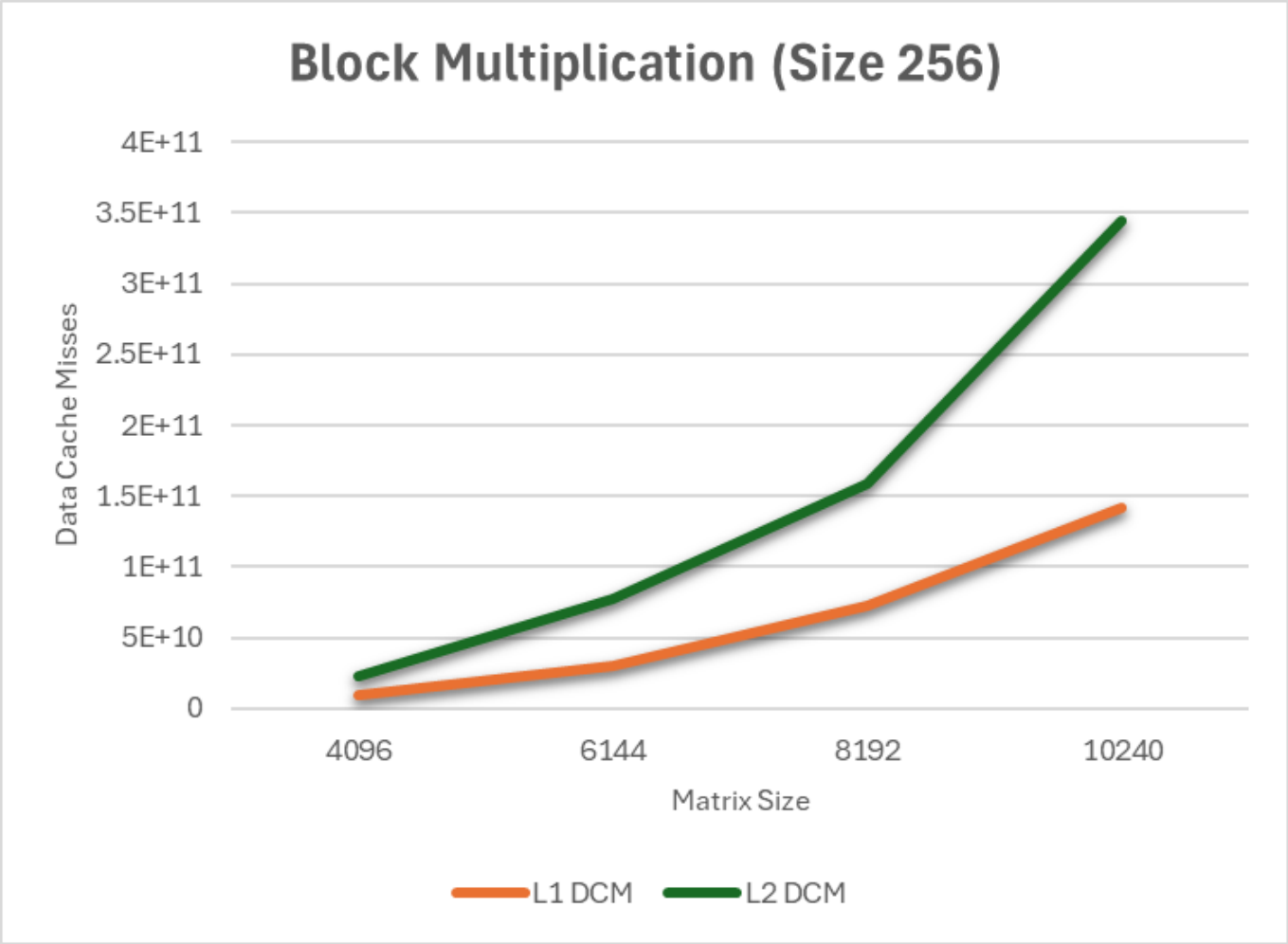
Size	Time(s)	L1 DCM	L2 DCM
4096	32.420	9726634684	32336039327
6144	113.143	32816297366	1.11489E+11
8192	268.534	77838387177	2.56855E+11
10240	577.762	1.51963E+11	5.01128E+11



Block Size: 256

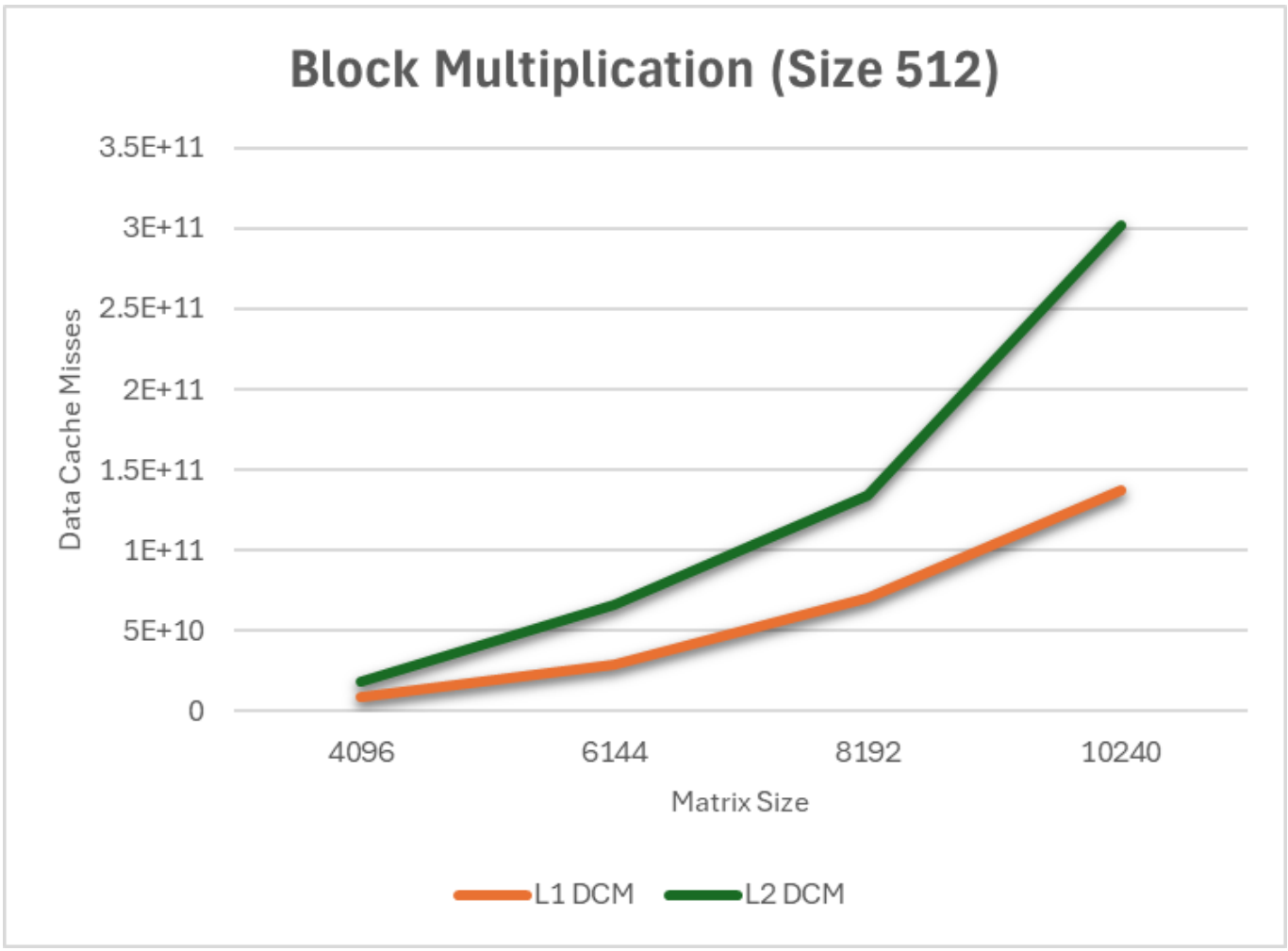
Size	Time(s)	L1 DCM	L2 DCM
4096	27.134	9077964464	23191634192
6144	96.966	30637532883	77563361945
8192	390.608	73036003266	1.58385E+11

Size	Time(s)	L1 DCM	L2 DCM
10240	497.023	1.4183E+11	3.44638E+11

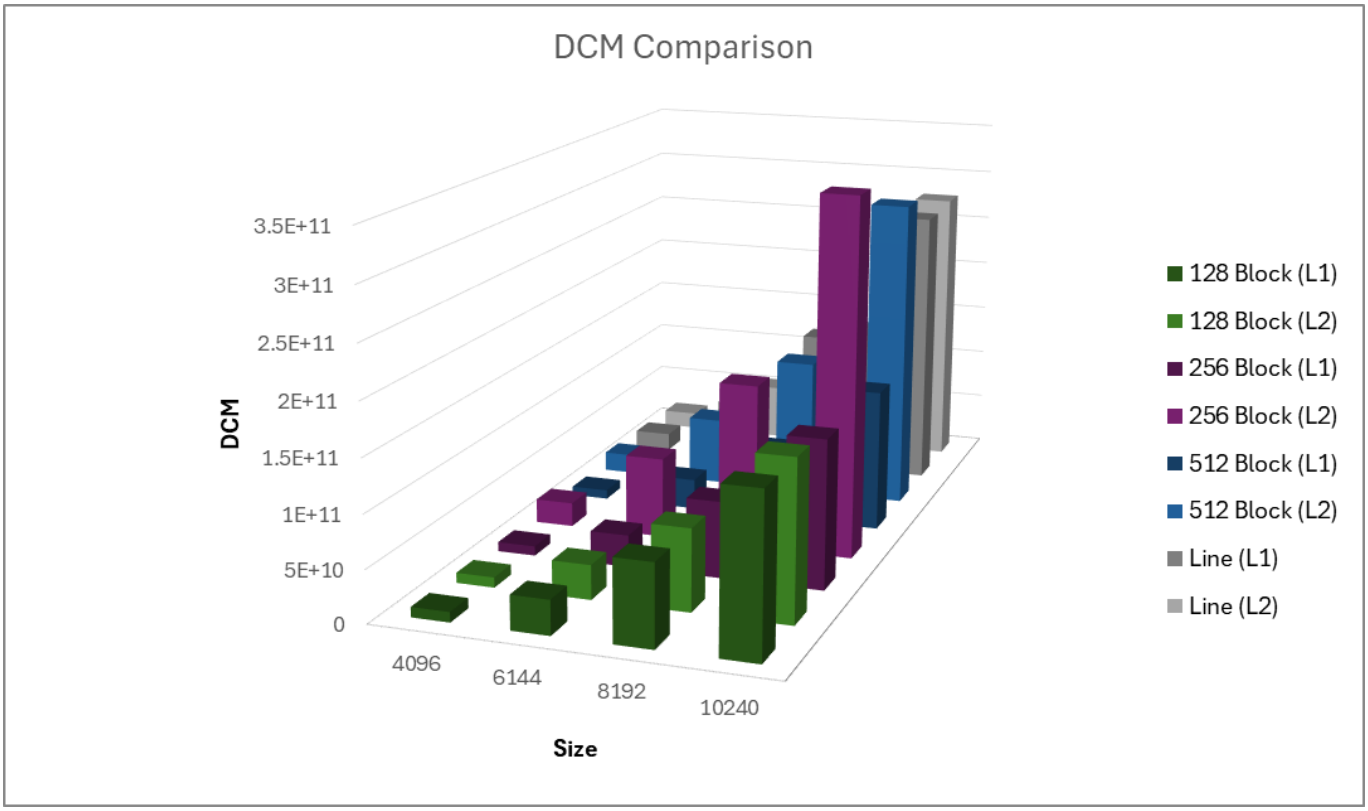


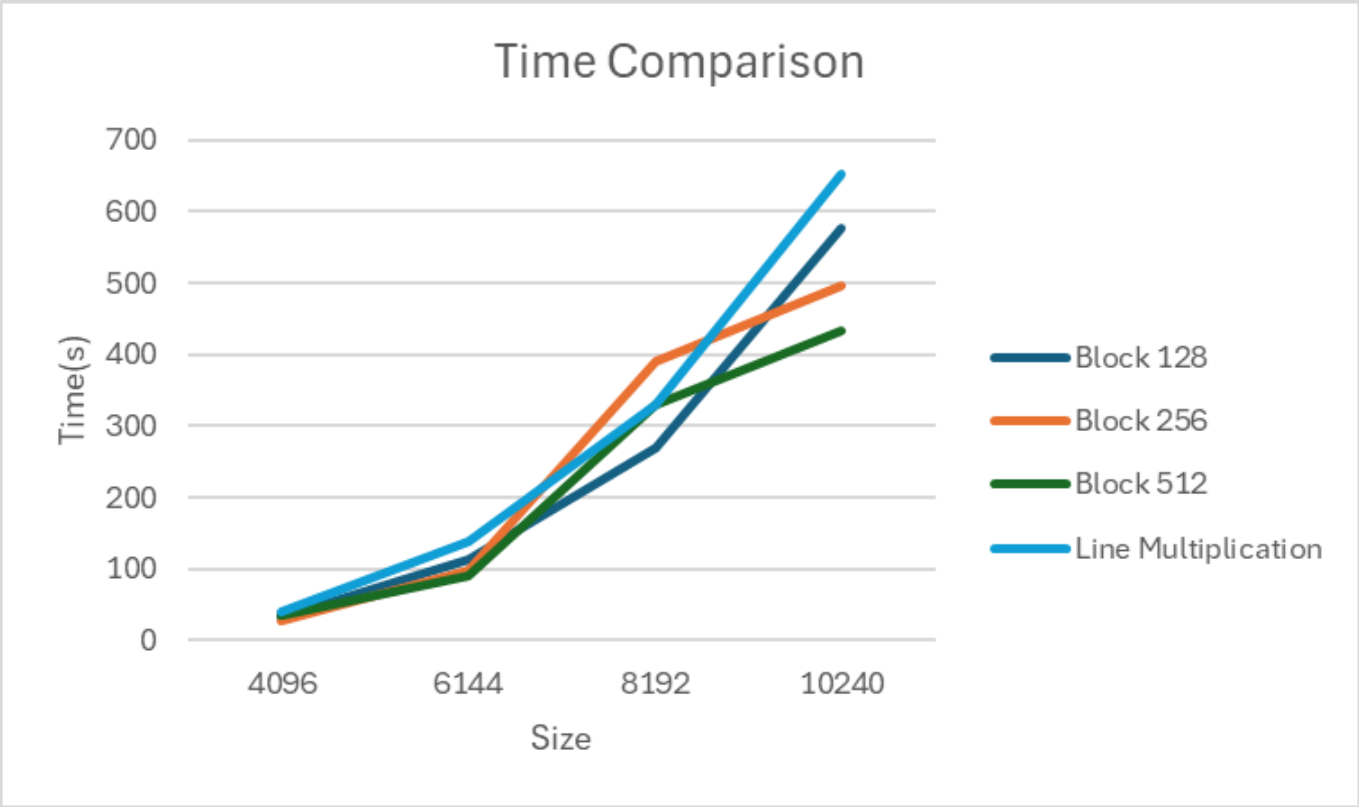
Block Size: 512

Size	Time(s)	L1 DCM	L2 DCM
4096	34.222	8766986850	19004137434
6144	89.656	29605181682	66059445685
8192	329.561	70347816557	1.34401E+11
10240	433.028	1.36924E+11	3.02231E+11


Block Multiplication (Size 512)
3.5E+11
3E+11
2.5E+11
2E+11
1.5E+11
1E+11
5E+10
0
Data Cache Misses
4096
6144
8192
10240
Matrix Size
L1 DCM
L2 DCM

Comparison


DCM Comparison
3.5E+11
3E+11
2.5E+11
2E+11
1.5E+11
1E+11
5E+10
0
DCM
4096
6144
8192
10240
Size
128 Block (L1)
128 Block (L2)
256 Block (L1)
256 Block (L2)
512 Block (L1)
512 Block (L2)
Line (L1)
Line (L2)



Analysis

Given that memory is organized into consecutive blocks, each memory access retrieves an entire block at once. When dealing with substantial datasets, minimizing these accesses becomes crucial for optimizing program performance.

In this context, the block algorithm capitalizes on the memory layout by subdividing the problem into smaller segments, thereby reducing the number of memory calls. This stands in contrast to the line multiplication algorithm, which may not efficiently exploit this memory organization, leading to increased occurrences of data cache misses.

Furthermore, the performance benefits of the block algorithm are evident when increasing the block size, up to the point where the size matches the Level 1 cache block size, which is typically the smallest among various cache levels. By examining the data provided in the tables, it becomes evident that the aforementioned observations hold true.

Performance evaluation of a multi-core implementation

In the second part of this project, we will implement parallel versions of the Line Multiplication that we studied in the first part.

We're also utilizing the Performance API (PAPI) to gather data cache miss metrics from the Level 1 and Level 2 caches, and we will calculate the MFlops, speedup and efficiency, comparing to the line implementation we had before.

To ensure reliability, we're also conducting 5 trials as before, and changing the size of the matrices. In this part we are using 8 threads.

Algorithms

```
#pragma omp parallel for
for (int i=0; i<n; i++)
     for (int k=0; k<n; k++)
          for (int j=0; j<n; j++)
          {              }
```

```
#pragma omp parallel
for (int i=0; i<n; i++)
     for (int k=0; k<n; k++)
          #pragma omp for
          for (int j=0; j<n; j++)
          {              }
```

Results

First Algorithm

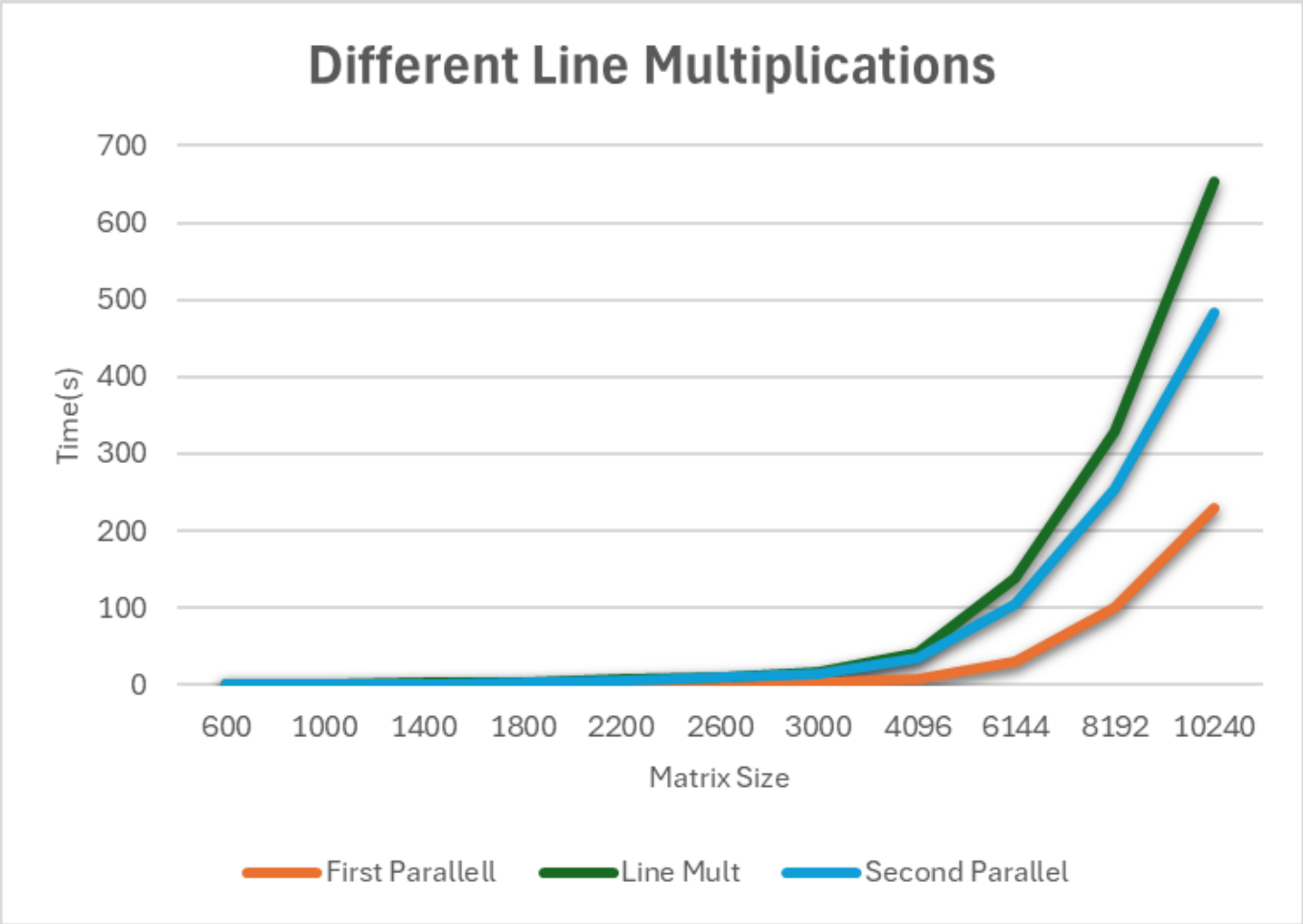
Size	Time(s)	L1	L2	Mflops	SpeedUp	EFF
600	0.019	3171451	7507834	22736842105	5.578947	0.697368
1000	0.093	16597225	34483540	21505376344	5.322581	0.665323
1400	0.283	45588306	89956679	19392226148	5.646643	0.70583
1800	0.532	66573841	185903907	21924812030	6.379699	0.797462
2200	1.223	273088515	338601329	17412919052	5.044971	0.630621
2600	1.811	529437620	549352489	19410270569	5.694644	0.71183
3000	2.812	903793963	855944512	19203413940	5.634068	0.704259
4096	7.301	2093249104	2156863466	18824675178	5.642789	0.705349
6144	29.311	7120818815	7363238942	15825337517	4.744055	0.593007
8192	100.21	16458577482	16134253813	10972074920	3.286858	0.410857
10240	230.172	35767557623	32837807188	9329908277	2.83445	0.354306

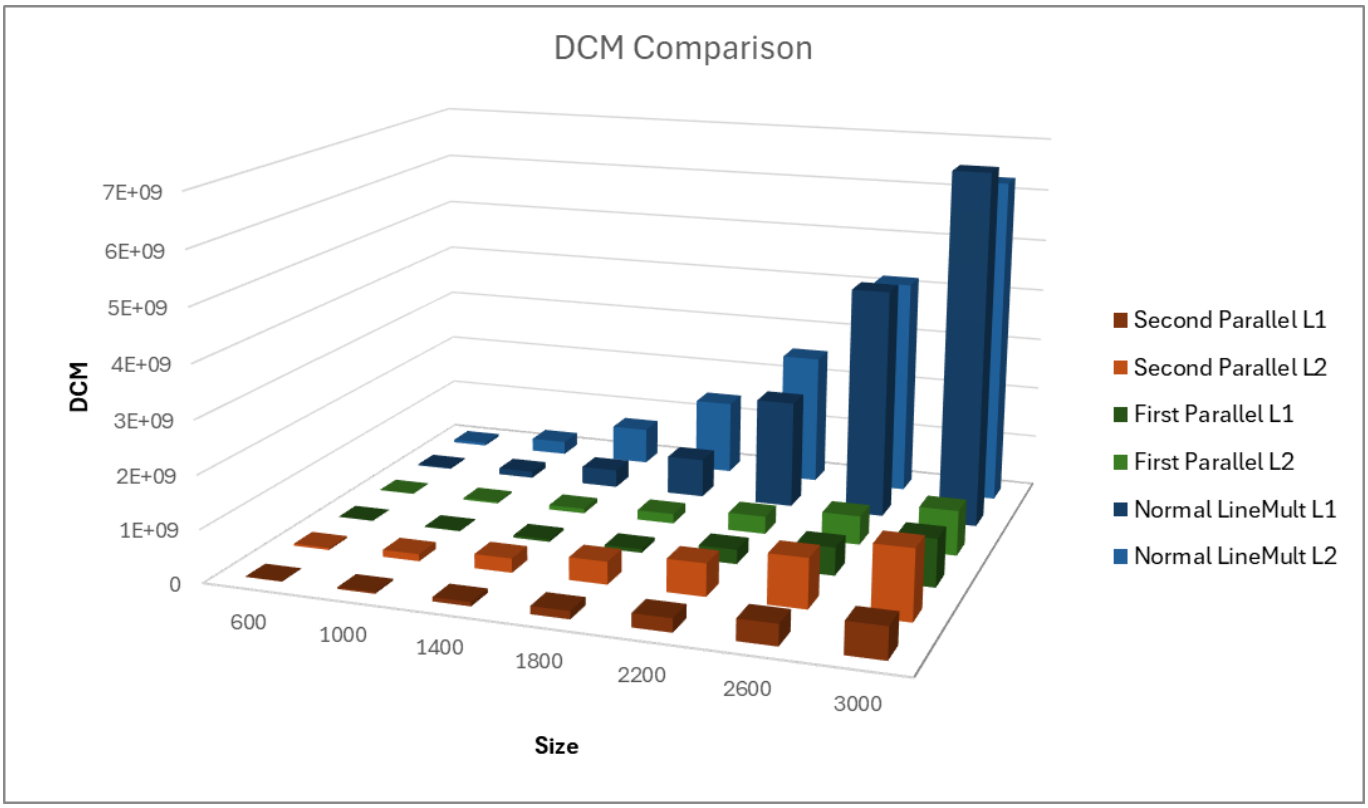
Second Algorithm

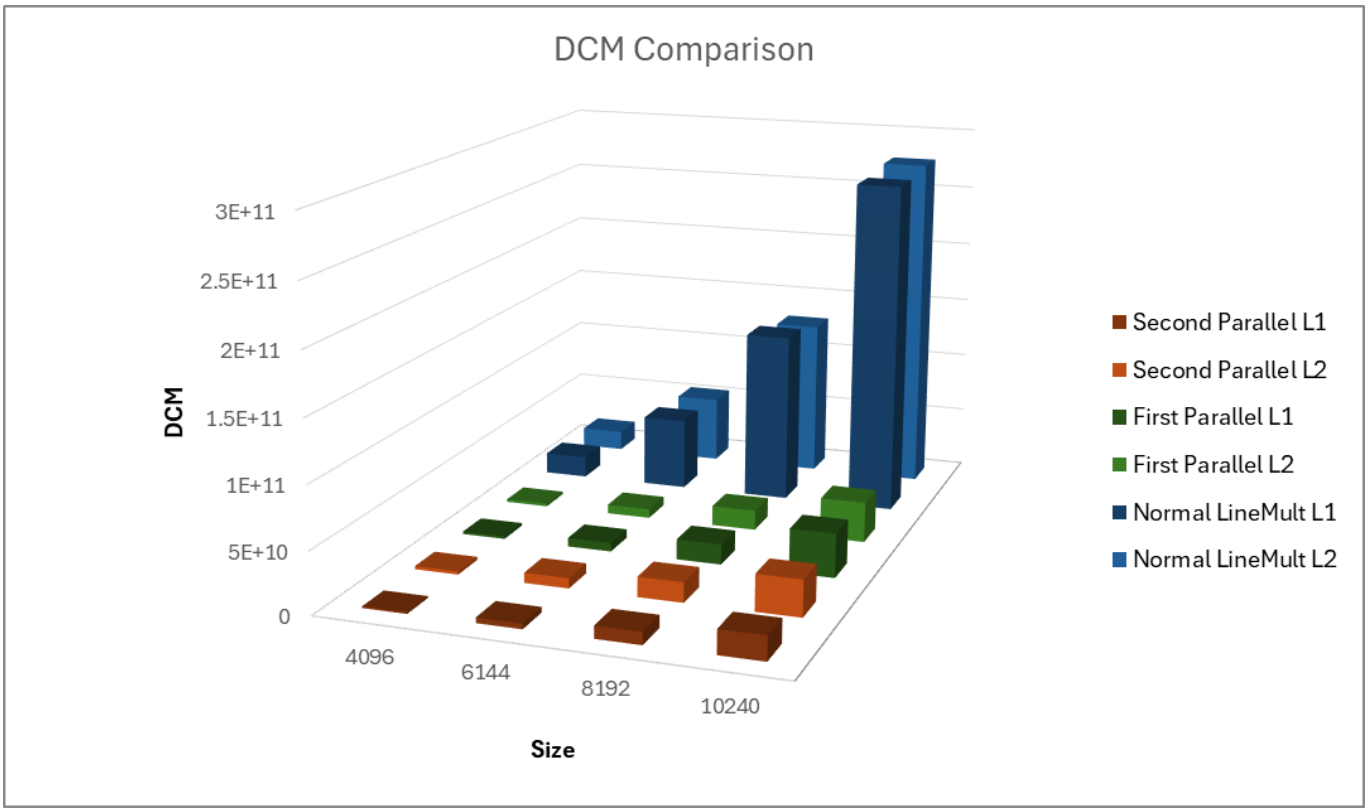
Size	Time(s)	L1	L2	Mflops	SpeedUp	EFF
600	0.192	10228869	40510051	2250000000	0.552083	0.06901
1000	0.521	34013380	134165027	3838771593	0.950096	0.118762
1400	1.399	77989297	268861297	3922802001	1.142244	0.142781

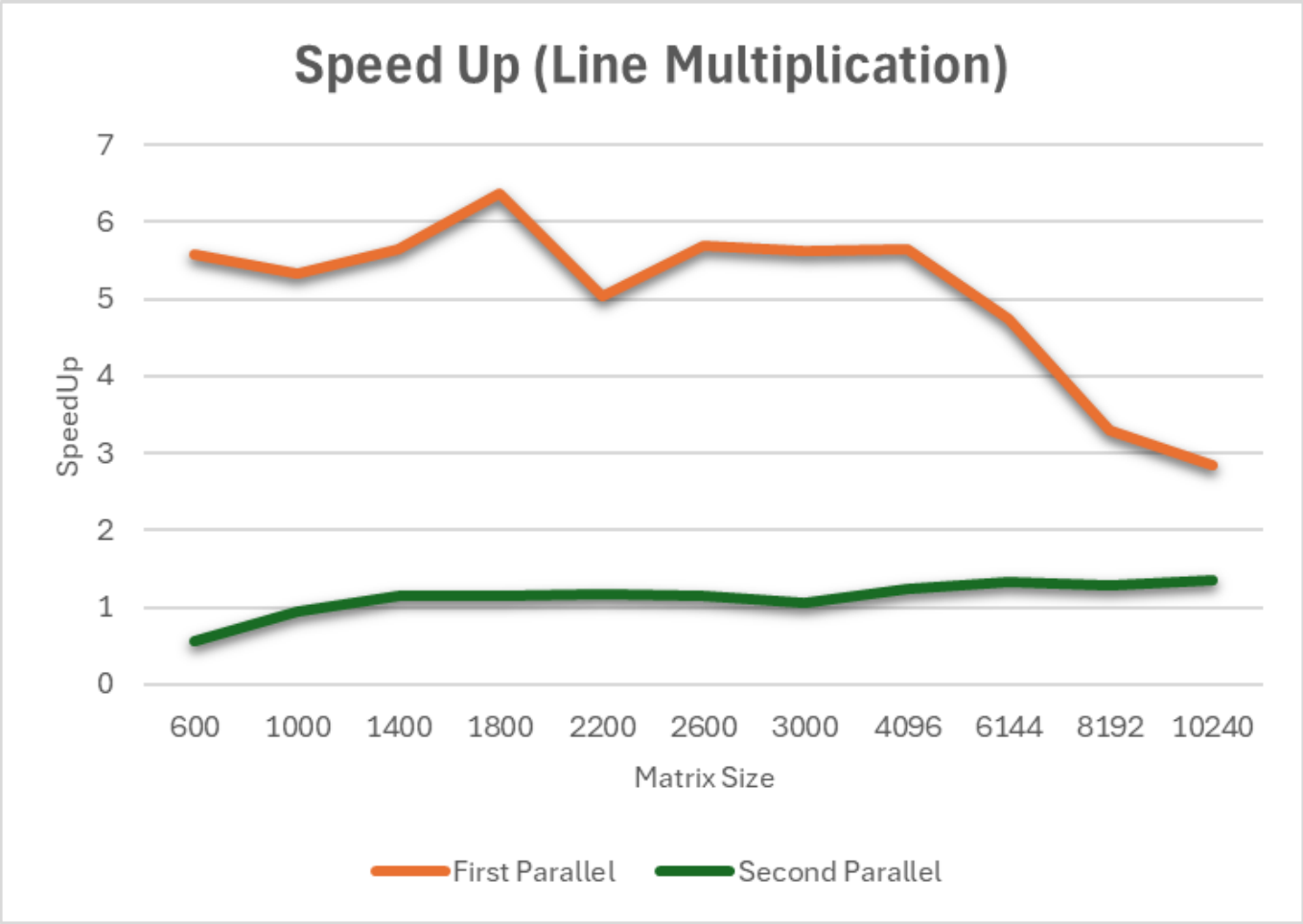
Size	Time(s)	L1	L2	Mflops	SpeedUp	EFF
1800	2.931	149037132	426259560	3979529171	1.157967	0.144746
2200	5.291	254871794	614850160	4024948025	1.166131	0.145766
2600	8.902	396643495	932303405	3948775556	1.158504	0.144813
3000	14.892	594038204	1321052994	3626107977	1.06386	0.132982
4096	33.339	1325409351	2541042382	4122467785	1.23573	0.154466
6144	104.982	4393498890	8158918709	4418438094	1.324541	0.165568
8192	255.012	9958311495	16118124179	4311607406	1.29161	0.161451
10240	483.291	19390775359	29488947822	4443458802	1.349934	0.168742

Comparison




DCM Comparison
DCM
7E+09
6E+09
5E+09
4E+09
3E+09
2E+09
1E+09
0
600
1000
1400
1800
2200
2600
3000
Size
Second Parallel L1
Second Parallel L2
First Parallel L1
First Parallel L2
Normal LineMult L1
Normal LineMult L2


DCM Comparison
DCM
3E+11
2.5E+11
2E+11
1.5E+11
1E+11
5E+10
0
4096
6144
8192
10240
Size
Second Parallel L1
Second Parallel L2
First Parallel L1
First Parallel L2
Normal LineMult L1
Normal LineMult L2



Analysis

From the statistics, it's clear that the first parallelization approach, parallelizing the outermost loop, consistently outperforms the nested parallelization approach.

The first algorithm's direct outer loop parallelization minimizes overhead and ensures better load balancing compared to the second algorithm's nested parallelization approach. This happens because the direct parallelization of the outermost loop distributes the workload evenly across threads, optimizing resource utilization and reducing the overhead associated with managing parallel tasks. Additionally, cache miss rates (L1 DCM and L2 DCM) decrease with parallelization, indicating better cache utilization.

We can also observe that both parallelized algorithms use the power of parallel processing by dividing the computational tasks among multiple threads. This utilization of parallelism results in a improved performance when compared to the sequential execution seen in the line multiplication algorithm of the first part.

Conclusion

In summary, this study broadened our understanding of program performance by considering additional metrics beyond time complexity. By optimizing how data is accessed and organized, we can achieve notable improvements in computational efficiency, highlighting the importance of these factors alongside traditional measures of algorithmic complexity.