U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Performance Evaluation of Multi-core Matrix Multiplication Implementation

Faculdade de Engenharia da Universidade do Porto
Turma 4 - Grupo 16

Ana Carolina Coutinho: up202108685
Leonardo Monteiro Ribeiro: up202205144
José Diogo Alves Granja: up202205143

Contents

1 Introduction

This report examines the optimization of matrix multiplication, a crucial operation in high-performance computing, and its effects on processor performance and memory hierarchy. Utilizing the Performance API (PAPI), we compare serial and parallel implementations to discern the most effective strategies for maximizing computational efficiency. We evaluate the impact of different programming languages and multiprocessing techniques on processing speed and cache utilization.

2 Problem Description

This study evaluates efficient matrix multiplication strategies for handling large matrices in C++ and Java, focusing on conventional, line-by-line, and block-based methods. Additionally, we assess these methods' performance in a multi-core setting using OpenMP to implement parallel versions of the line-by-line approach. Our analysis targets MFlops, speedup, and efficiency, aiming to identify optimal computational practices for enhanced performance in varying computational environments.

3 Algorithms Explanation

This project investigates three matrix multiplication algorithms: **Standard**, **Line-by-Line**, and **Block Matrix Multiplication**. Each uses distinct memory access strategies affecting cache efficiency and computational speed. The Standard method is straightforward, Line-by-Line optimizes for cache performance, and the Block method improves data locality for enhanced parallel processing. Implementations in C++ and Java across single-core and multi-core setups highlight differences in language efficiency and parallel scalability.

3.1 Serial Algorithm

3.1.1 Standard Matrix Multiplication

This algorithm performs matrix multiplication by calculating the dot product of rows from the first matrix (A) with columns from the second matrix (B). Each element $C[i][j]$ of the result matrix (C) is computed using the formula:

$$C[i][j] = \sum_{k=1}^{n} A[i][k] \times B[k][j]$$

While straightforward, this method is computationally demanding and often suffers from inefficient memory access patterns, especially in large matrix scenarios. In C++ and Java, this method is implemented through three nested loops traversing the matrices' rows and columns, with performance in Java monitored using `System.nanoTime()` for precise time tracking, while C++ employs `clock()` alongside PAPI to gather detailed metrics such as cache misses and CPU cycles.

3.1.2 Line-by-Line Matrix Multiplication

Optimizing the standard approach, this algorithm reduces cache misses by accessing continuous memory locations. It capitalizes on the property of locality by accessing rows of the first matrix and corresponding transposed rows of the second matrix sequentially. The mathematical expression for this method is:

$$C[i][j] += A[i][k] \times B[j][k]$$

This enhancement significantly improves data cache utilization, crucial for performance gains, particularly with large datasets. In Java, the technique leverages in-place memory access to boost performance, while the C++ version can benefit from compiler optimizations like loop unrolling.

3.1.3 Block Matrix Multiplication

By segmenting the matrices into smaller blocks, this method substantially enhances cache coherence and minimizes memory access delays. It is especially effective for very large matrices that exceed cache size. The calculation for blocks is described by:

$$C[i][j] += \sum_{k=\text{blk\_start}}^{\text{blk\_end}} A[i][k] \times B[k][j]$$

where blk_start and blk_end determine the block boundaries, optimizing memory overhead by maximizing cache memory's faster access speeds. The implementation involves nested loops that operate over matrix blocks instead of entire matrices, improving both temporal and spatial locality. C++ implementations might further optimize memory allocation to align blocks with cache lines effectively.

3.2 Parallel Algorithm

3.2.1 Outer Loop Parallelism (OnMultParallel1)

This version of parallel matrix multiplication distributes the computation across multiple threads by parallelizing the outermost loop. This approach is particularly beneficial for operations involving large datasets where the division of labor across several processors can significantly reduce the total computation time.

The algorithm utilizes OpenMP, a parallel programming library, to manage the distribution of the matrix rows among the available CPU cores. By applying the `#pragma omp parallel for` directive, each thread is assigned a set of rows from the first matrix, and it computes their products with the corresponding columns of the second matrix independently of the other threads.

If matrix A has dimensions $m \times n$ and matrix B has dimensions $n \times p$, then each thread computes a subset of the resulting matrix C of dimensions $m \times p$. The distribution is such that:

$$C_{\text{segment\_i}} = A_{\text{segment\_i}} \times B$$

where $A_{\text{segment\_i}}$ is the portion of matrix A processed by thread i.

3.2.2 Inner Loop Parallelism (OnMultParallel2)

This approach enhances granularity in parallel processing by applying parallelism to the innermost loop of the matrix multiplication algorithm. It is designed to optimize workload distribution while ensuring efficient utilization of computational resources. This method is particularly suited for systems where balancing the computational load across multiple threads can significantly improve performance.

In this implementation, OpenMP is used to create a parallel region that distributes iterations of the outermost loop across available threads. Additionally, within each iteration of the second loop, a `#pragma omp for` directive is applied to the innermost loop, ensuring that different threads share the computational load of the summation operation over k. This setup allows for better distribution of computations while maintaining the sequential nature of the outer loops.

The computation of the result matrix **C** follows the standard matrix multiplication formula:

$$C[i][j] = \sum_{k=1}^{n} (A[i][k] \times B[k][j]) \tag{1}$$

In this version, each thread independently processes different rows (i) and iterates over columns (j), while the summation operation over k is explicitly parallelized. This distribution minimizes synchronization overhead while effectively utilizing computational resources. By leveraging OpenMP's parallelization strategy, this method aims to improve execution time, especially for large matrix sizes, by reducing the time required to compute each element of the result matrix.

4 Performance Metrics

In evaluating matrix multiplication implementations, we employ several performance metrics to provide quantitative insights into the efficiency and effectiveness of different algorithms. These metrics are essential for benchmarking the baseline performance of single-core implementations and assessing the improvements achieved through optimizations and parallel computing:

- **Execution Time:** This metric measures the total time taken from the start to the completion of a program or a specific algorithm execution, serving as a direct measure of performance. A lower execution time indicates a faster algorithm, which is fundamental for identifying bottlenecks and potential areas for optimization within the code.

- **MFlops (Mega Floating-Point Operations Per Second):** MFlops stands for million floating-point operations per second and measures the computational performance of a system. The formula used is

$$\text{MFlops} = \frac{2 \times N^3}{\text{Execution Time in seconds} \times 10^6}$$

  where N represents the matrix dimension, and the factor of 2 accounts for one multiplication and one addition per matrix element. This metric is crucial as it provides a normalized measure of computational speed that is independent of system hardware, facilitating comparisons across different setups.

- **Cache Misses (L1 DCM and L2 DCM):** These metrics refer to instances when the required data is not found in the cache memory, necessitating access to slower main memory. Monitoring cache misses is vital for analyzing memory access patterns and optimizing data locality, as high cache misses can significantly impede performance due to increased main memory access times.
- **Speedup:** This metric is used to measure the relative performance gain of a parallel implementation compared to its serial counterpart, calculated by

$$\text{Speedup} = \frac{\text{Execution Time of Serial Implementation}}{\text{Execution Time of Parallel Implementation}}$$

  Speedup is indicative of the effectiveness of the parallelization strategy employed, highlighting how well the algorithm scales with increased processing resources.
- **Efficiency:** Efficiency assesses the effectiveness of parallelization by evaluating how well computational resources are utilized, given by the formula

$$\text{Efficiency} = \left(\frac{\text{Speedup}}{\text{Number of Threads}} \right) \times 100\%$$

  It helps determine the point at which adding more threads yields diminishing returns and is crucial for ensuring that computational resources are not wasted.

5 Results and Analysis

This section will discuss the experimental results of matrix multiplication algorithms, focusing on the performance metrics of execution time, MFlops, and cache behavior for different matrix sizes and block sizes. We will compare serial and parallel implementations in both C++ and Java, using various approaches like block and line-by-line multiplication. The analysis will highlight key findings through graphs and tables, aiming to identify the most efficient techniques for optimizing matrix multiplication in a multi-core setting.

5.1 Experimental Setup

The experiments were conducted on a device with the following hardware and software specifications:

- **Device Name:** LAPTOP-4HJEDAV7
- **Processor:** 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz, 2.42 GHz
- **RAM:** 12.0 GB (11.8 GB usable)
- **System Type:** 64-bit operating system, x64-based processor
- **Operating System:** Windows 11 Home, Version 24H2, Installed on 01/03/2025, OS Build 26100.3476, Feature Experience Pack 1000.26100.54.0aa

5.2 Simple Matrix Multiplication

The simple matrix multiplication is the most basic and least efficient implementation. It follows the traditional approach of using three nested loops to compute each element of the result matrix, without applying any form of memory access optimization.

Main Observations

- **Execution Time:** The time complexity increases quadratically with matrix size. As shown in Figure 1, the execution time grows rapidly for larger dimensions, making this approach unsuitable for large-scale computations.
- **Cache Misses (L1 and L2):** Due to unstructured and non-local memory access, this method results in a high number of cache misses, particularly in L1 and L2 levels (see Figures 2 and 3). This leads to additional memory latency and reduced overall performance.

5.3 Row-wise Matrix Multiplication

Row-wise multiplication introduces a significant improvement by changing the order of memory access. Instead of accessing elements in a scattered fashion, the algorithm iterates row by row, aligning better with how data is stored in memory.

Main Observations

- **Execution Time:** This approach achieves a noticeable reduction in execution time compared to the simple version, especially for larger matrices (see Figure 1). It benefits from improved memory locality and fewer cache stalls.
- **Cache Misses:** The number of L1 and L2 data cache misses is significantly lower (Figures 2 and 3), as rows are stored contiguously in memory. This improves cache line reuse and reduces memory access latency.

5.4 Block-wise Matrix Multiplication

Block-wise matrix multiplication is the most efficient of the sequential approaches evaluated. It divides the matrices into smaller submatrices (blocks), allowing computations to be performed on portions of data that can fit in the cache.

Main Observations

- **Execution Time:** This technique achieves the fastest execution time among the sequential methods (Figures 7 and 8). By maximizing cache reuse, it minimizes memory overhead and accelerates computation.
- **Cache Misses:** As shown in Figures 5 and 6, both L1 and L2 data cache misses are drastically reduced. The use of small blocks keeps the relevant data in cache longer, leading to better memory performance.

5.5 Comparison Between C++ and Java

To further evaluate performance, the matrix multiplication algorithms were implemented and tested in both C++ and Java. This comparison reveals the influence of language-level characteristics and runtime environments.

Main Observations

- **Execution Time:** As illustrated in Figure 9, C++ consistently delivers better performance due to native compilation, reduced overhead, and better control over memory management.
- **Cache Misses:** Java exhibits a higher number of L1 and L2 cache misses. The automatic memory management of the Java Virtual Machine (JVM), such as garbage collection and object indirection, contributes to less efficient cache utilization.

5.6 Parallel Matrix Multiplication: Alternative 1 and 2

Parallelization was tested using two distinct strategies, each exhibiting different memory access patterns and computational efficiencies. The objective was to evaluate how effectively each method leverages multi-threading while minimizing cache-related inefficiencies.

Alternative 1: Row-Wise Parallelization In this method, the outermost loop is parallelized by assigning different rows to separate threads. This preserves the classic row-by-column computation scheme, making it easy to implement. Since each thread works on independent rows, the workload distribution is predictable and synchronization overhead is minimal. However, the performance is limited by memory bandwidth, leading to a higher number of cache misses.

Despite this drawback, **Alternative 1 achieves superior execution times** compared to Alternative 2 (see Figure 10), likely due to the lower synchronization cost and more balanced workload.

Alternative 2: Block-Wise Parallelization with Transposition This approach begins by transposing matrix B to improve spatial locality. Then, the computation proceeds using small blocks, which enhances cache reuse by increasing temporal locality and reducing memory access frequency.

The expectation was that this optimization would lead to better overall performance. Indeed, **Alternative 2 results in fewer L1 and L2 cache misses** (see Figures 11 and 12). However, it does not outperform Alternative 1 in terms of execution time, as the overhead introduced by matrix transposition and block-wise synchronization cancels out some of the gains in memory efficiency.

Results Analysis

Execution Time Figure 10 demonstrates that Alternative 1 consistently delivers better execution times across all tested matrix sizes. The simplicity of its parallelization strategy, combined with reduced synchronization complexity, allows it to better exploit available processing cores. This makes it the preferred method when minimizing runtime is the primary concern.

L1 and L2 Cache Misses Figures 11 and 12 indicate that Alternative 2 significantly reduces L1 and L2 cache misses. Its block-based memory access pattern improves data locality, which is beneficial for large-scale computations. This result highlights Alternative 2 as a more *memory-friendly* strategy, even if it comes at the cost of slightly increased execution time.

Summary

Overall, Parallel Multiplication 1 (Alternative 1) is the best approach in terms of raw execution time, making it the most practical solution for performance-critical applications. However, Parallel Multiplication 2 (Alternative 2) achieves better cache efficiency, which may be advantageous for workloads with very large matrices where memory performance is a bottleneck. The trade-off between execution speed and memory access efficiency should be considered when choosing between these two methods.

6 Conclusion

This project has provided valuable insights into memory management and its crucial role in program efficiency. Through different matrix multiplication approaches, we observed how algorithms with the same theoretical complexity can exhibit vastly different performances due to memory access patterns. Optimizing cache usage and structuring memory-friendly computations significantly improves execution time, even in sequential implementations.

In the second phase, parallelization was explored using OpenMP, revealing that, when properly implemented, it can drastically improve performance. However, inefficient parallelization can lead to minimal or even negative gains, emphasizing the importance of workload distribution and memory locality. Additionally, we analyzed the impact of varying the number of threads, demonstrating how resource allocation affects computational efficiency.

Ultimately, the execution time improvements across different implementations highlight how critical it is to design algorithms that maximize low-level memory reuse (L1 and L2 caches) while minimizing unnecessary accesses to higher-latency memory. These findings reinforce the importance of both algorithmic optimizations and hardware-aware programming in achieving high-performance computing.

7 Annexes

7.1 Graphs

Execution Time: Normal Multiplication and Line Multiplication

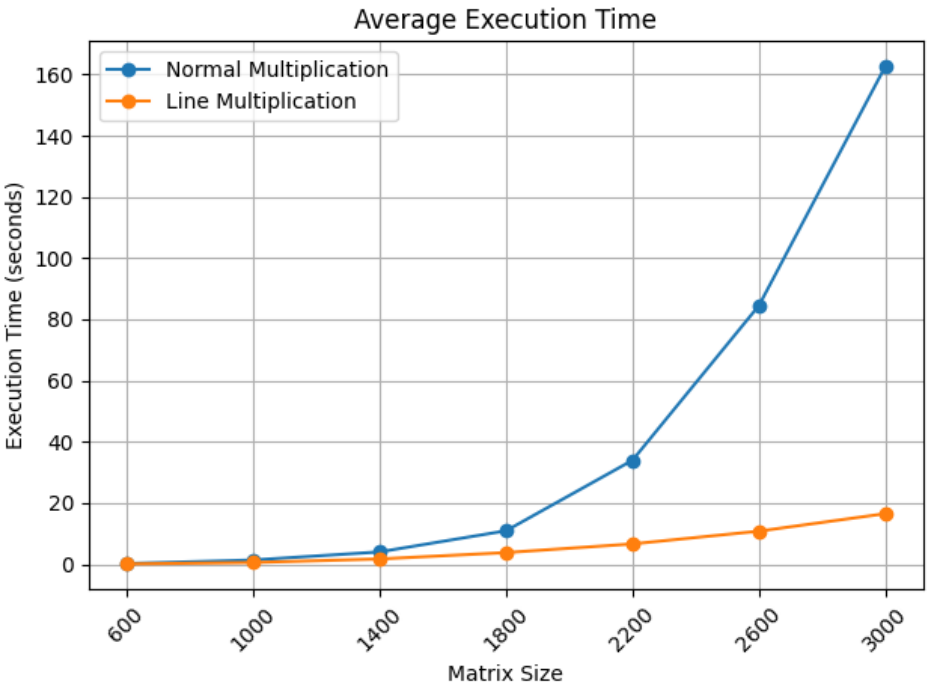

Figure 1: Execution Time for Normal vs Line Multiplication

L1 Data Cache Misses: Normal Multiplication and Line Multiplication

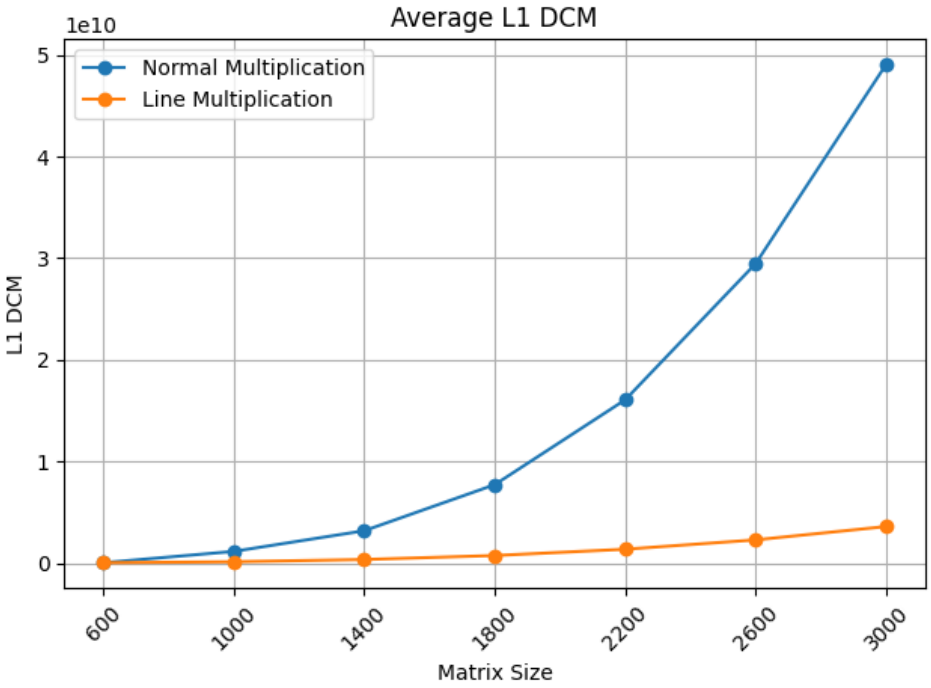

Figure 2: L1 Cache Misses for Normal vs Line Multiplication

L2 Data Cache Misses: Normal Multiplication and Line Multiplication

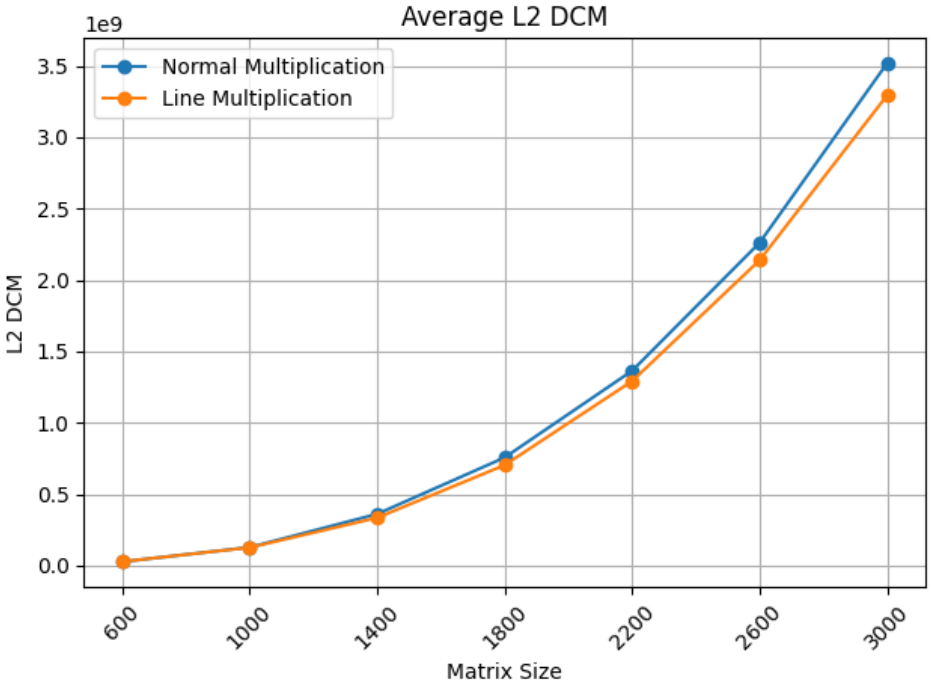

Figure 3: L2 Cache Misses for Normal vs Line Multiplication

Execution Time: Line Multiplication and Block Multiplication

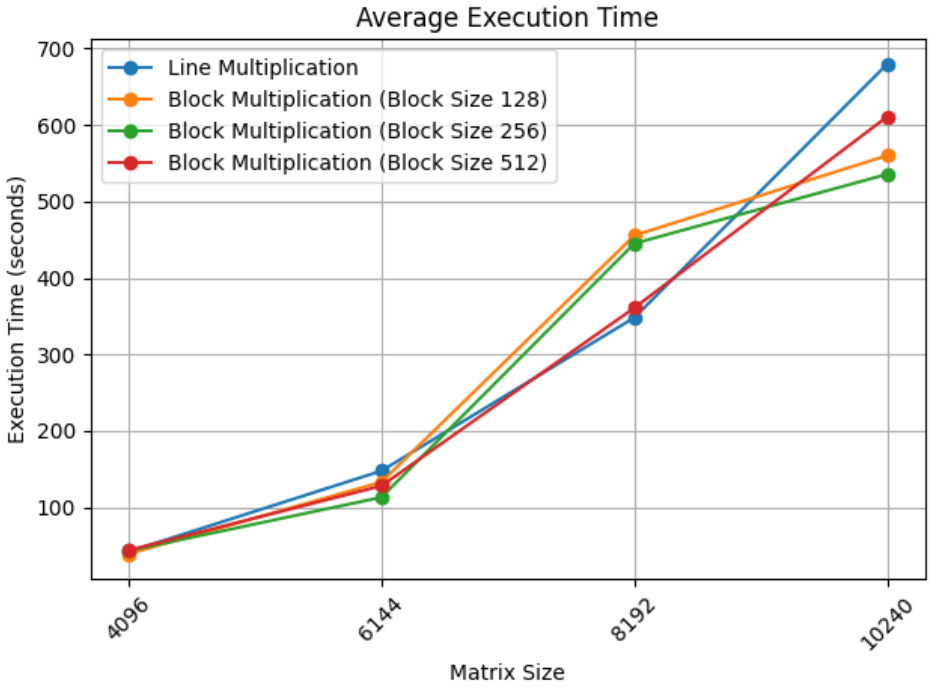

Figure 4: Comparison of Execution Time Between Line Multiplication and Block Multiplication in C++

L1 Data Cache Misses: Line Multiplication and Block Multiplication

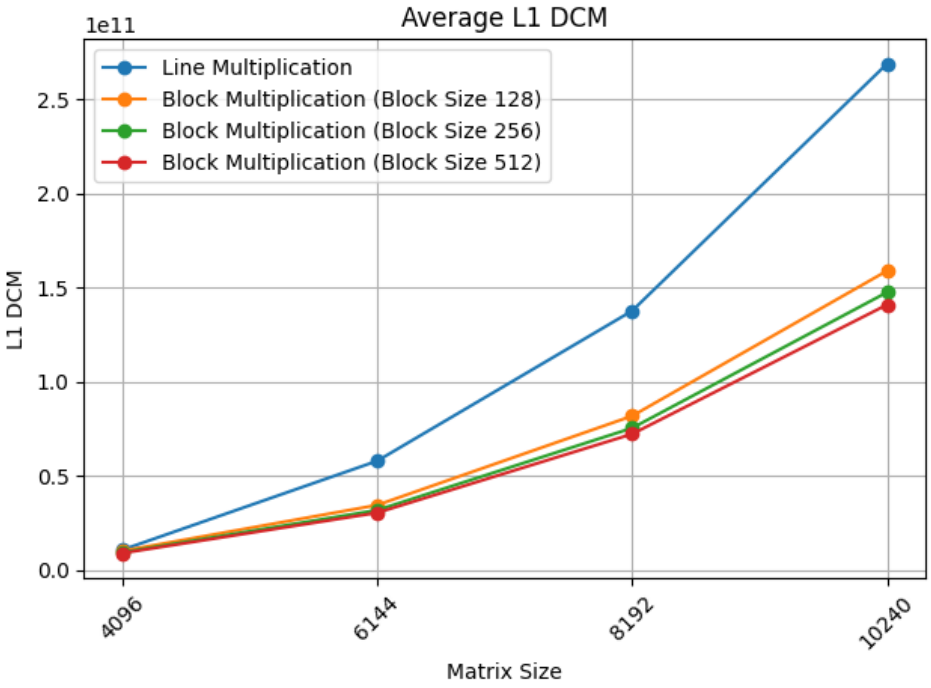

Figure 5: L1 Cache Misses for Line vs Block Multiplication

L2 Data Cache Misses: Line Multiplication and Block Multiplication

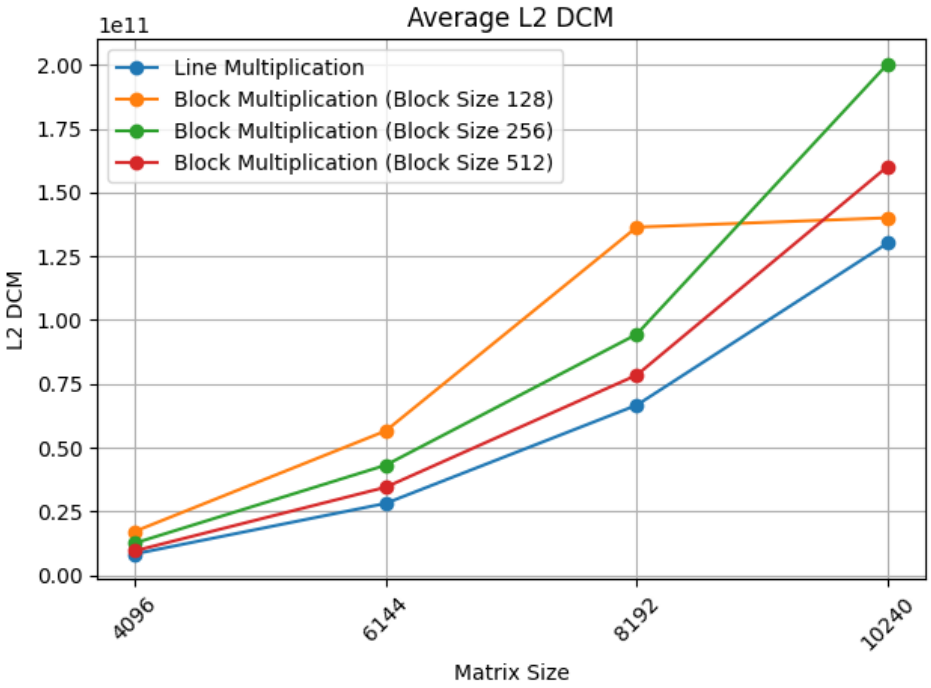

Figure 6: L2 Cache Misses for Line vs Block Multiplication

Execution Time: Block Multiplication (C++)

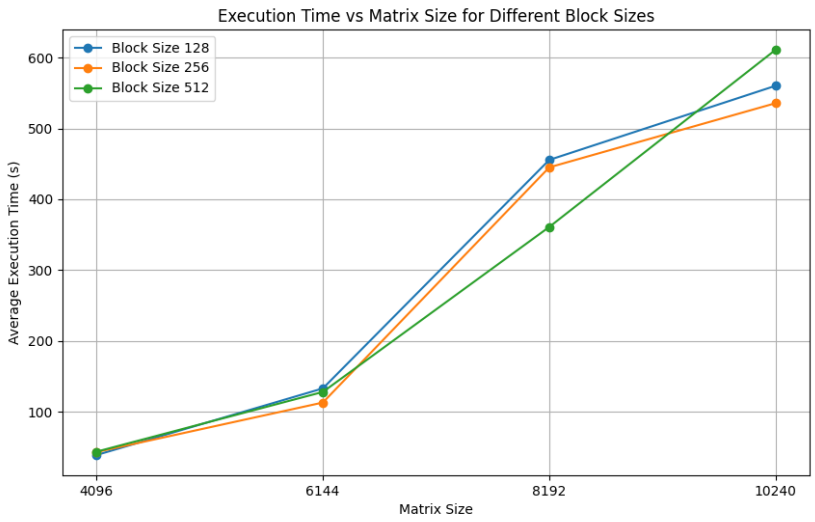

Figure 7: Execution Time vs Matrix Size for Different Block Sizes in C++

Execution Time: Block Multiplication (Java)

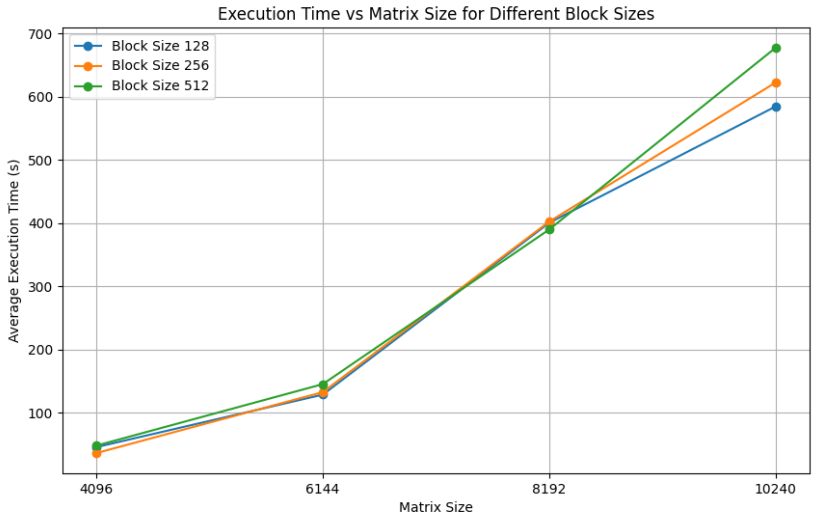

Figure 8: Execution Time vs Matrix Size for Different Block Sizes in Java

Comparison Between Java and C++

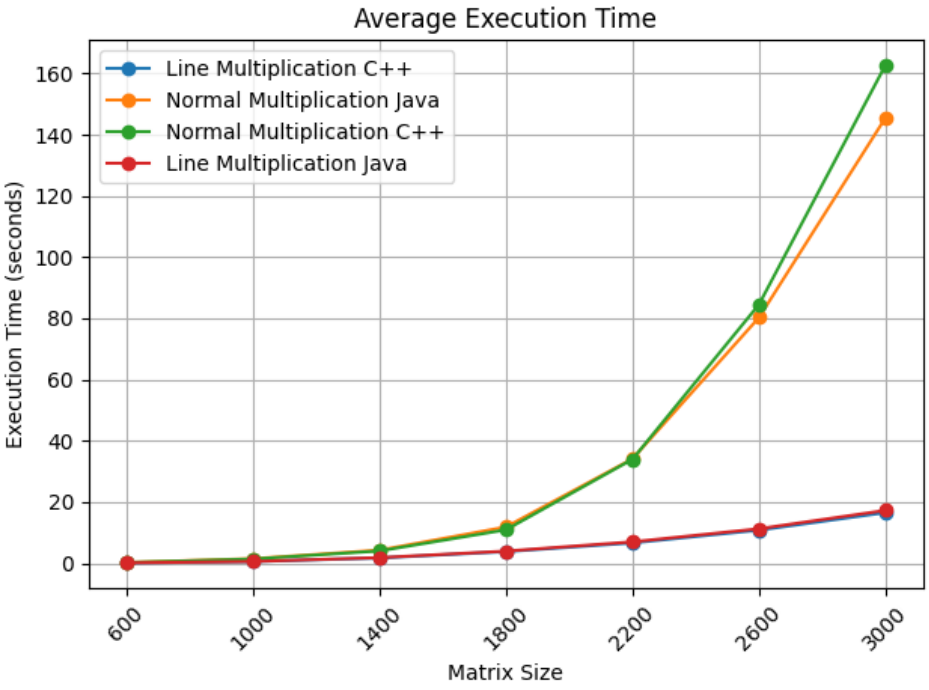

Figure 9: Comparison of Execution Time Between Java and C++

Execution Time: Parallel Multiplication

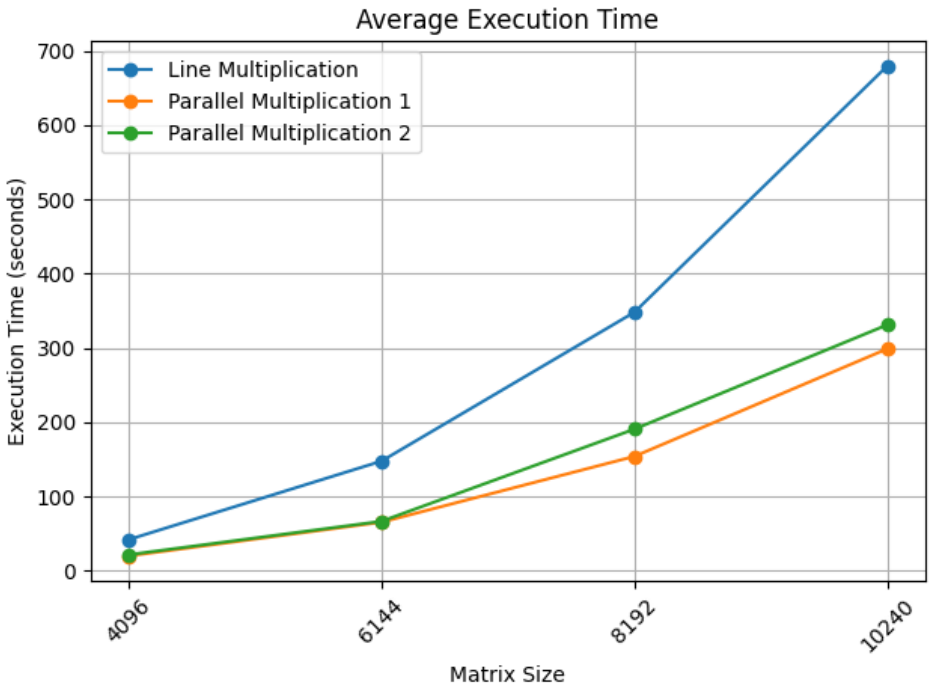

Figure 10: Execution Time for Parallel Multiplication

L1 Data Cache Misses: Parallel Multiplication

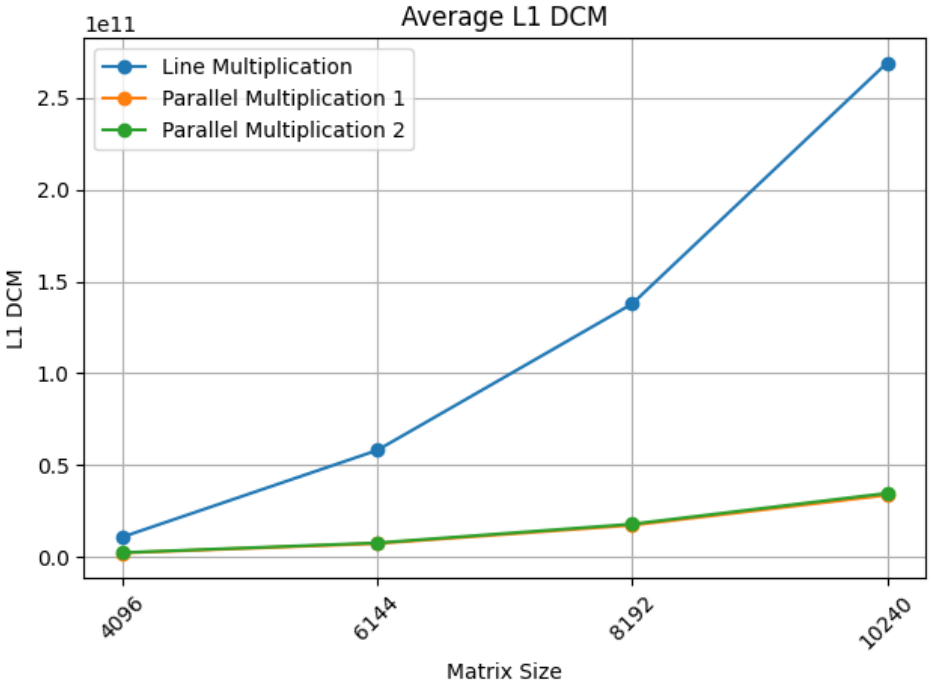

Figure 11: L1 Cache Misses for Parallel Multiplication

L2 Data Cache Misses: Parallel Multiplication

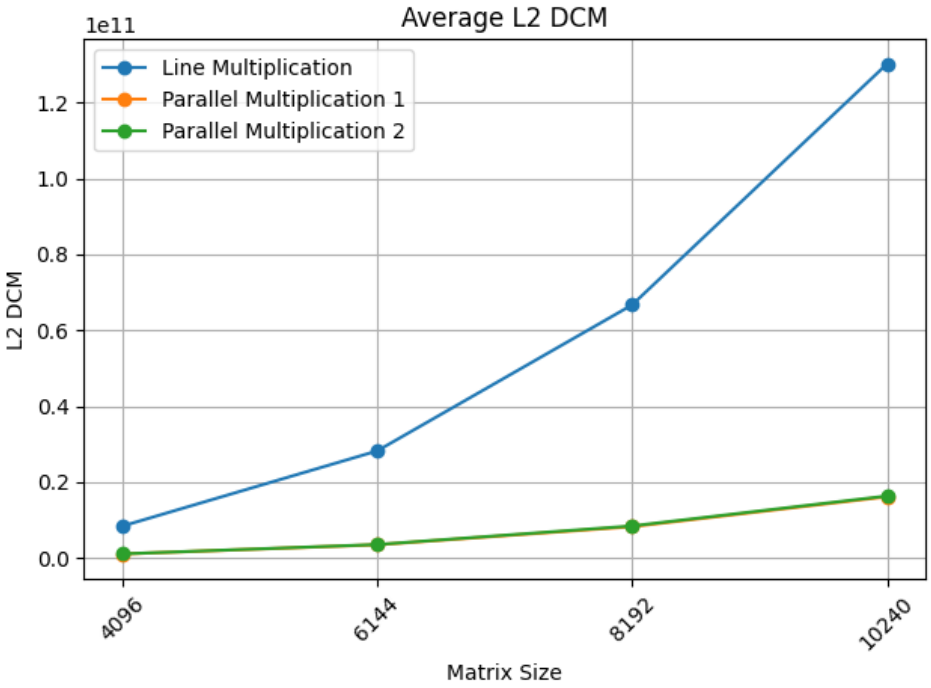

Figure 12: L2 Cache Misses for Parallel Multiplication

7.2 Final Results

Matrix Multiplication in C++

Dimension	Time	L1_DCM	L2_DCM	Mflops
600	0.2263	29480083	27093020	951.37
1000	1.3811	1145805735	128298298	2,060.38
1400	4.0302	3177824639	361399476	3,951.42
1800	10.9297	7712058741	756916664	6,713.69
2200	33.9705	16068720477	1364627722	6,296.91
2600	84.5309	29478824820	2261353129	5,112.74
3000	162.7255	49092714989	3518072010	5,591.11

Matrix Multiplication in Java

Dimension	Time (s)	GFlops
600	0.2377	6,802.00
1000	1.4897	4,797.00
1400	4.2823	3,527.00
1800	11.8640	3,664.00
2200	34.1160	3,718.00
2600	80.2593	3,378.00
3000	145.5080	3,127.00

Line Matrix Multiplication in C++

Dimension	Time	L1_DCM	L2_DCM	MFlops
600	0.096305	27,888,702	28,016,269	14,090.00
1000	0.576820	126,919,541	126,604,355	7,820.00
1400	1.723755	347,396,488	334,946,464	6,180.00
1800	3.805966	739,948,617	703,844,240	4,340.00
2200	6.640872	1,359,694,600	1,292,308,573	3,420.00
2600	10.812590	2,279,892,426	2,138,895,045	2,670.00
3000	16.534746	3,595,057,260	3,294,040,760	2,190.00

Line Matrix Multiplication in Java

Dimension	Time (s)	GFlops
600	0.113000	14,330.00
1000	0.579667	7,775.00
1400	1.829667	4,957.00
1800	3.961667	3,923.00
2200	6.990667	3,208.00
2600	11.264333	2,550.00
3000	17.263000	1,992.00

Block Matrix Multiplication in C++

Dimension	BlockSize	Time	L1_DCM	L2_DCM	MFlops
4096	128	39.140	10,246,154,447	17,161,481,135	4,314.38
4096	256	43.286	9,446,091,027	12,604,269,896	3,168.78
4096	512	43.866	9,055,014,700	9,573,603,480	3,125.41
6144	128	133.158	34,590,077,039	56,591,590,120	1,853.94
6144	512	128.159	30,497,166,658	34,510,431,899	3,623.30
8192	128	327.961	81,472,762,419	106,444,383,736	1,116.13
8192	256	393.428	75,483,343,462	88,167,307,067	2,869.50
8192	512	315.641	72,341,314,950	79,035,738,786	3,519.88
10240	128	560.187	159,055,276,386	140,078,091,629	654.10
10240	256	535.640	147,653,312,642	200,426,319,960	4,024.23
10240	512	610.349	141,054,544,255	160,251,354,601	3,523.86

Block Matrix Multiplication in Java

Dimension	Block Size	Time (s)	MFlops
4096	128	45.673	3393
4096	256	36.272	4378
4096	512	48.075	3226
6144	128	128.717	3625
6144	256	132.709	3541
6144	512	145.370	3221
8192	128	400.678	2928
8192	256	402.545	2915
8192	512	390.069	2990
10240	128	584.721	1969
10240	256	622.653	1850
10240	512	677.663	1699

Parallel Matrix Multiplication V1 in C++

Dimension	Time (s)	L1_DCM	L2_DCM	MFlops	Speedup	Efficiency
600	0.042555	3,260,602	2,818,936	10,777.00	2.26	56.58%
1000	0.259648	14,154,464	13,626,966	7,959.00	2.22	55.54%
1400	0.575584	45,225,922	39,602,062	9,556.00	2.99	74.87%
1800	1.570190	154,239,706	84,583,553	7,434.00	2.42	60.60%
2200	3.207660	310,972,960	157,217,741	6,669.00	2.07	51.76%
2600	5.026067	524,794,784	260,771,839	6,996.00	2.15	53.78%
3000	7.637303	825,514,812	402,130,368	7,072.00	2.16	54.12%
4096	20.101200	2,128,797,877	1,031,701,956	6,838.33	1.95	48.68%
6144	65.531500	7,268,372,099	3,461,071,790	7,078.33	2.03	50.80%
8192	154.172000	17,230,083,523	8,228,796,098	7,132.33	2.13	53.18%
10240	299.058000	33,635,019,155	16,121,160,872	7,181.00	1.87	46.83%

Parallel Matrix Multiplication V2 in C++

Dimension	Time (s)	L1_DCM	L2_DCM	MFlops	Speedup	Efficiency
600	0.096435	3,830,807	2,754,998	4,705.76	1.00	24.97%
1000	0.362623	19,403,851	13,786,082	5,527.09	1.59	39.77%
1400	1.128270	70,491,909	38,226,052	4,943.19	1.53	38.19%
1800	2.330800	187,360,855	84,047,420	5,010.37	1.63	40.82%
2200	3.906530	361,450,826	159,294,251	5,476.31	1.70	42.50%
2600	5.828020	596,208,891	262,372,178	6,037.26	1.86	46.38%
3000	8.526610	917,624,305	405,808,337	6,333.53	1.94	48.48%
4096	20.101200	2,128,797,877	1,031,701,956	6,838.54	1.95	48.68%
6144	66.765300	7,627,911,465	3,518,766,057	6,950.72	1.99	49.86%
8192	190.844000	17,875,481,411	8,420,953,053	5,766.51	1.72	42.96%
10240	331.339000	34,681,725,299	16,313,744,190	6,482.00	1.69	42.27%