Matrix Multiplication Optimization Study
Assignment #2, CSC 746, Fall 2024

Shun Usami*
SFSU

Abstract

The inefficiency of matrix multiplication is due to poor memory and cache utilization, especially as matrix sizes grow. In this assignment, we implemented Basic Matrix Multiplication (Basic) as an unoptimized baseline, Matrix Multiplication using CBLAS as a highly optimized reference, and Blocked Matrix Multiplication with Copy Optimization (BMMCO) to explore techniques for enhancing spatial and temporal locality. These implementations were evaluated on the Perlmutter supercomputer at NERSC, with performance measured in MFLOP/s. The results demonstrate that BMMCO significantly improves performance over Basic MM by effectively leveraging both spatial and temporal locality, though a performance gap remains compared to CBLAS, suggesting further opportunities for optimization, such as loop reordering and improved in-memory data layout strategies.

1 Introduction

The objective of this assignment is to optimize matrix multiplication by improving cache usage to enhance both spatial and temporal locality. The naive approach to matrix multiplication is inefficient in terms of memory access and locality. To explore how we can optimize the cache usage of matrix multiplication, we implemented three different matrix multiplication approaches: Basic Matrix Multiplication (Basic), Matrix Multiplication using C Basic Linear Algebra Subprograms (CBLAS), and Blocked Matrix Multiplication with Copy Optimization (BMMCO). These implementations were chosen to represent minimal optimization, a state-of-the-art reference, and our own implementation using blocking and copy optimization. The performance of these implementations was evaluated on the Perlmutter supercomputer at NERSC, using C++ for the implementations. The key performance metric, MFLOP/s, was calculated based on the elapsed time of each method. Detailed implementations are provided in Section 2.

The results demonstrate that blocking and copy optimization techniques can significantly enhance the computational throughput of matrix multiplication compared to Basic MM by successfully utilizing both spatial and temporal locality. However, despite these gains, there remains a considerable performance gap compared to the CBLAS implementation. This performance disparity suggests that further optimizations are possible, such as loop reordering, improved in-memory data layout strategies, and the incorporation of advanced matrix multiplication algorithms like Strassen's algorithm.

2 Implementation

This section describes the two matrix multiplication methods implemented to measure performance: Basic Matrix Multiplication (Basic MM) without optimizations and Blocked Matrix Multiplication with Copy Optimization (BMMCO). Each subsection details the objectives and specific implementations of these methods.

*email:susami@sfsu.edu

2.1 Basic Matrix Multiplication (Basic MM)

The objective of the Basic MM implementation is to establish a baseline performance metric for matrix multiplication without any optimizations. This implementation serves as a reference point to compare the effects of the optimization techniques applied in subsequent methods.

The Basic MM implementation uses a naive triple-nested loop to compute each element $C[i, j]$ of the resulting matrix C, by adding the dot product of the i-th row of matrix A and the j-th column of matrix B. Here, A, B, and C are $n \times n$ matrices stored in row-major format, which is intentionally chosen to result in poor cache line usage. This approach ensures that the implementation is both naive and achieves the worst possible efficiency, providing a very low-performance baseline for comparison with optimized methods. The actual implementation is shown in Listing 1.

```
void square_dgemm(int n, double *A, double *B,
    double *C) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      for (int k = 0; k < n; k++) {
        C[i + j * n] += A[i + k * n] * B[k + j * n
    ];
      }
    }
  }
}
```

Listing 1: Basic Multiplication

2.2 Blocked MM with Copy Optimization (BMMCO)

The objective of the BMMCO implementation is to measure performance when the program accesses memory sequentially and revisit the same memory address . This implementation serves as a reference point to compare the effects of the optimization techniques applied in subsequent methods.

3 Evaluation

In this section, we present the results of our experiments, aimed at evaluating the performance of the matrix multiplication implementations introduced earlier. We conducted tests on the Perlmutter supercomputer to compare the computational throughput (in MFLOP/s) of three matrix multiplication methods: Basic MM (unoptimized) and CBLAS (highly optimized) as performance baselines, and Blocked Matrix Multiplication with Copy Optimization (BMMCO) to analyze the impact of block size and matrix size on performance. The evaluation involved testing various matrix sizes and block sizes in BMMCO to investigate how different spatial and temporal memory access patterns influence computational efficiency.

3.1 Computational platform and Software Environment

The experiments were conducted on the CPU node of the Perlmutter supercomputer at NERSC. Each CPU node is powered by an AMD EPYC 7763 (Milan) processor, which has 64 cores running at a clock

```
void square_dgemm_blocked(int n, int block_size,
    double *A, double *B,
                          double *C) {
  static double aBlock[MAX_BLOCK], bBlock[
    MAX_BLOCK], cBlock[MAX_BLOCK];
  int Nb = n / block_size;
  for (int i = 0; i < Nb; i++) {
    for (int j = 0; j < Nb; j++) {
      blockread(cBlock, C, i, j, n, block_size);
      for (int k = 0; k < Nb; k++) {
        blockread(aBlock, A, i, k, n, block_size);
        blockread(bBlock, B, k, j, n, block_size);
        // C[i, j] += A[i, k] * B[k, j];
        square_dgemm(block_size, aBlock, bBlock,
    cBlock);
      }
      blockwrite(cBlock, C, i, j, n, block_size);
    }
  }
}
```

Listing 2: BMMCO Multiplication

rate of 2.45 GHz [2]. Each core is equipped with 32 KB of L1 cache and 512 KB of L2 cache, while 8 cores share a 32 MB L3 cache [1]. The system is supported by 512 GB of DDR4 DRAM, providing a memory bandwidth of 204.8 GB/s per CPU [2]. The processor utilizes the AVX2 instruction set for vector processing, and each core offers a peak computational throughput of 39.2 GFLOPS [2].

All experiments were performed on a single CPU node running *SUSE Linux Enterprise Server 15 SP4* [4], with kernel version *5.14.21-150400.24.81_12.0.87-cray_shasta_c* [4]. The C++ code was compiled using *g++-12 (SUSE Linux) 12.3.0* with the following optimization flags: *-O3 -DNDEBUG -Wall -pedantic -march=native*, to achieve maximum speed and strict compliance with standards.

3.2 Methodology

We evaluated the performance of different matrix multiplication methods using matrix sizes of 64×64, 128×128, 256×256, 512×512, 1024×1024, and 2048×2048. For the BMMCO implementation, we used block sizes of 2, 16, 32, and 64. To account for the known issue of slow performance on the first run of CBLAS MM due to dynamic library loading, we first ran the 64×64 matrix size once before conducting further evaluations. This initial run was discarded to avoid skewing the results with unreasonably bad performance.

Performance was measured by calculating the elapsed time using instrumentation code placed around the main matrix multiplication code. From this, we computed the MFLOPs (Millions of Floating Point Operations per second) using the following formula:

$$MFLOP/s = \frac{ops}{runtime}$$

$$ops = \frac{2N^3}{10^6}$$

Here, *ops* represents the number of floating-point operations required to multiply two $N \times N$ matrices, divided by one million (to convert to MFLOPs) [1]. The *runtime* is the time elapsed (in seconds) measured for each matrix multiplication method and size.

[1]When the matrix size is $N \times N$, the number of floating-point operations (FLOPs) required for matrix multiplication (MMUL) is approximately $2N^3$. This is because matrix multiplication involves N^2 dot products, each requiring $2N$ operations (one multiplication and one addition per element).

3.3 Comparison of Basic MM and CBLAS

In this experiment, the objective is to establish a baseline for our evaluation by measuring the computational throughput (MFLOP/s) of Basic MM, an unoptimized implementation serving as our baseline, and CBLAS MM, a state-of-the-art, highly optimized implementation. The configuration details for this experiment are provided in Sec. 3.2.

As shown in Fig. 1, the MFLOP/s of the Basic MM decreases significantly as the problem size increases, while the MFLOP/s of the CBLAS implementation improves with larger matrix sizes. The decrease in Basic MM performance is almost inversely proportional to the matrix size, reflecting its inefficiency in terms of cache usage and memory access. In contrast, the CBLAS implementation shows a gradual improvement as the matrix size increases, with one exception: performance drops from 64×64 to 128×128. We believe this is caused by the initial discarded run for the 64×64 problem size. It is likely that CBLAS's *cblas_dgemm* method allocates an internal buffer based on the matrix size, which can be reused for efficiency in subsequent runs. The overhead of allocating this internal buffer is not negligible for smaller problem sizes, but the allocation process is skipped for 64×64, leading to a performance drop for 128×128.

Interestingly, the MFLOP/s of CBLAS sometimes exceeds the CPU core's theoretical peak performance of 39.2 GFLOP/s (see Sect. 3.1). During the execution, we verified that the program was not utilizing multiple cores by running the *top* command, which showed a CPU usage value (%CPU) close to, but less than, 100%. We hypothesize that CBLAS is using advanced matrix multiplication techniques such as Strassen's algorithm, which can achieve a computational complexity better than $O(N^3)$[2]. Therefore, our calculation of MFLOP/s using $2N^3$ as *ops* for CBLAS may not be entirely appropriate.

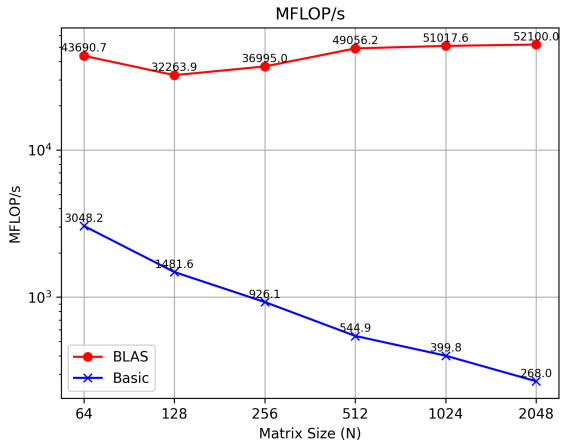

Figure 1: MFLOP/s comparison of Basic MM and CBLAS MM across different matrix sizes. The matrix is $N \times N$. Note that MFLOP/s decreases consistently on a logarithmic scale for Basic MM as the matrix size increases. In contrast, MFLOP/s gradually increases for CBLAS MM, with the exception of a performance drop from 64×64 to 128×128. This drop is likely due to the initial discarded run for the 64×64 problem size.

3.4 Blocked MM with Copy Optimization (BMMCO), and CBLAS

In this experiment, we evaluate the efficiency of the BMMCO implementation at four different block sizes by comparing its MFLOP/s

[2]Strassen's algorithm has a complexity of approximately $O(2^{2.8074})$.

with that of the CBLAS implementation. The configuration details for this experiment are provided in Sec. 3.2. Figure 2 shows how block size and matrix size affect the MFLOP/s of BMMCO compared to the CBLAS MM implementation. From this figure, three key insights can be derived.

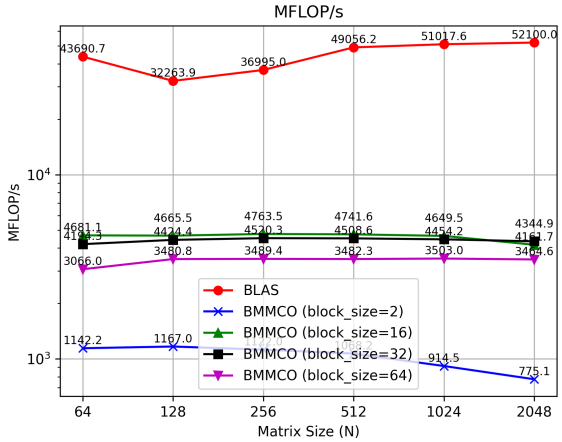

Figure 2: MFLOP/s comparison of BMMCO and CBLAS MM across different block sizes and matrix sizes. The matrix is $N \times N$. The figure illustrates how BMMCO's performance varies with block sizes of 2, 16, 32, and 64, compared to the highly optimized CBLAS MM implementation. The block size of 2 is significantly inefficient due to the limited performance gain from small block sizes. While larger block sizes generally improve performance, the block size of 64 shows performance degradation due to exceeding the L1 cache capacity, as discussed in Sec. 3.4.2.

3.4.1 Performance limitation of small block sizes

The first insight is that the block size of 2 is significantly inefficient compared to larger block sizes, and its MFLOP/s decreases as the matrix size increases, following a similar trend to the Basic MM implementation in Section 3.3. This inefficiency is due to the fact that the performance gain is primarily limited by the small block size[3].

3.4.2 Impact of L1 cache capacity on performance

The second insight is that larger block sizes, particularly 16 and 32, perform better. The block size of 16 produced the best results, while the block size of 32 performed slightly worse. However, the block size of 64 showed a performance degradation of approximately 20-30% compared to both 16 and 32. Initially, we expected larger block sizes to consistently provide better MFLOP/s, so these results were counterintuitive. This discrepancy is due to L1 cache utilization. As shown in Table 1, the block size of 64 requires a memory footprint of 96 KiB for three blocks, which exceeds the 32 KiB L1 cache capacity (see Sec. 3.1). While block sizes up to 32×32 fit within the L1 cache, the block size of 64 does not, resulting in performance degradation as tiling and blocking optimizations rely on fitting blocks into fast memory.

[3]The number of slow memory operations, m, depends on the block size b and the matrix size N. The total number of slow memory operations is proportional to $(2N_b + 2) \cdot N^2$, where N_b is the number of blocks. Compute intensity (CI), defined as $CI = \frac{ops}{number\ of\ slow\ memory\ accesses}$, improves with larger block sizes as CI approaches $n/N_b = b$ for large matrices.

Block Size	3 Blocks (Bytes)	L1 Cache Fit
2×2	96 B	Yes
16×16	6 KiB	Yes
32×32	24 KiB	Yes
64×64	96 KiB	No

Table 1: Memory footprint for different block sizes and L1 cache fit. The table shows the memory required for three blocks of each size and whether they fit within the L1 cache (32 KiB). Block sizes up to 32×32 fit, while 64×64 does not.

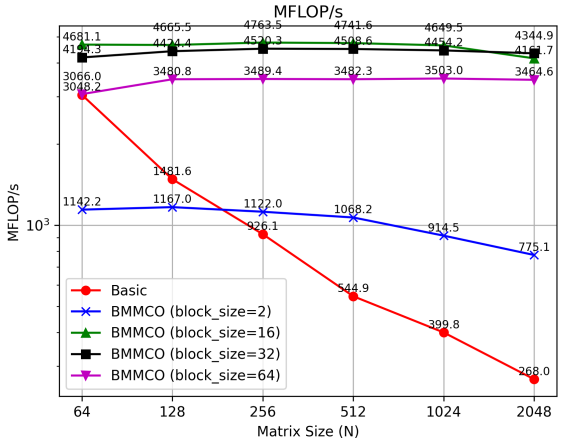

Figure 3: MFLOP/s Comparison of BMMCO and Basic MM across Matrix Sizes. This figure illustrates the performance differences between BMMCO and Basic MM across various matrix sizes. BMMCO shows more stable performance with larger matrix sizes due to optimization techniques such as blocking and copy, while Basic MM's performance degrades as the matrix size increases.

3.4.3 Remaining Performance Gap Between BMMCO and CBLAS

Even with the best BMMCO results (block size of 16), the CBLAS MM implementation still outperforms BMMCO by approximately tenfold. This significant performance gap suggests that there is still room for optimization, such as loop reordering, improved in-memory data layout strategies, and the use of advanced matrix multiplication algorithms like Strassen's algorithm.

3.5 Findings and Discussion

From Figure3, we can gain two key insights.

3.5.1 Poor performance of BMMCO with a block size of 2

The BMMCO implementation with a block size of 2 exhibits significantly lower performance than Basic MM for matrix sizes of 64×64 and 128×128. This can be attributed to the fact that matrices A, B, and C all fit entirely within the 512 KiB L2 cache, with a significant portion of them also fitting into the 32 KiB L1 cache for these sizes (see Table 2). As a result, the benefits of blocking and copy optimization are outweighed by the overhead introduced (i.e., reading and writing blocks to internal buffer).

The MFLOP/s of BMMCO with a block size of 2 decreases as the problem size increases, following a similar trend observed in Basic MM, though the decline is more gradual. This is because, for larger matrix sizes that do not fit into the L2 cache, the overhead becomes relatively smaller, and the gains from optimization become more substantial.

Matrix Size	3 Matrices (Bytes)	L1	L2	L3
64×64	96 KiB	No	Yes	Yes
128×128	384 KiB	No	Yes	Yes
256×256	1.5 MiB	No	No	Yes
512×512	6 MiB	No	No	Yes
1024×1024	24 MiB	No	No	Yes
2048×2048	96 MiB	No	No	No

Table 2: Memory footprint for different matrix sizes and cache fit. The table shows the memory footprint required to hold three matrices of varying sizes and whether they fit within the L1 cache (32 KiB), L2 cache (512 KiB), and L3 cache (32 MiB). Matrices up to 128×128 fit in the L2 cache, and those up to 512×512 fit within the L3 cache.

3.5.2 Large Block Sizes Outperform Basic MM

For a matrix size of 64×64, the performance of Basic MM is comparable to that of BMMCO with a block size of 64. This is because the matrix size and block size are equal, resulting in no significant performance gain from the BMMCO implementation.

However, BMMCO with larger block sizes consistently outperforms Basic MM, with performance remaining relatively stable across different matrix sizes, while Basic MM's MFLOP/s decreases almost inversely as the matrix size increases. This improvement in BMMCO is due to performance gains that scale with the block size (see the footnote[3]) when the blocks fit into fast memory. In fact, the three blocks used in the optimization mostly fit within the L1 cache and fully fit within the L2 cache (see Table 3), allowing matrix multiplication to proceed with minimal slow memory access, thereby increasing efficiency.

In summary, without blocking and copy optimization, performance degrades significantly due to inefficient memory access. However, by leveraging these optimizations, both spatial and temporal locality are effectively utilized, leading to substantial improvements in overall efficiency.

Block Size	3 Blocks (Bytes)	L1 Cache	L2 Cache
2×2	96 B	Yes	Yes
16×16	6 KiB	Yes	Yes
32×32	24 KiB	Yes	Yes
64×64	96 KiB	No	Yes

Table 3: Memory Footprint for different block sizes and L1 and L2 cache fit. This table shows the memory footprint required to hold three blocks of different sizes and whether they fit within the L1 and L2 cache levels. Block sizes up to 32×32 fit within the L1 cache, while larger block sizes fit within the L2 cache.

Acknowledgments

This paper was edited with the assistance of *ChatGPT-4o* (accessed September 2024), which was primarily used for correcting grammatical errors and rephrasing. Additionally, *ChatGPT-o1-preview* was utilized to help generate visualizations of blocked matrix multiplication, as shown in Appendix B. This research was supported by resources from the National Energy Research Scientific Computing Center (NERSC), a Department of Energy Office of Science User Facility, under NERSC award DDR-ERCAP m3930 for 2024.

References

[1] AMD. Hpc tuning guide for amd epyc 7003 series processors. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/tuning-guides/high-performance-computing-tuning-guide-amd-epyc7003-series-processors.pdf, 2022. Accessed: September 2024.

[2] NERSC. Perlmutter architecture. https://docs.nersc.gov/systems/perlmutter/architecture/, 2024. Accessed: September 2024.

[3] S. Usami. Blocked matrix multiplication visualization code. https://github.com/usatie/bmmco-visualization, 2024.

[4] S. Usami. Operating System and Kernel Information of Perlmutter CPU Node. `hostnamectl` command output on Perlmutter CPU node, 2024. Retrieved on September 17, 2024.

Appendices

A The Effect of Discarding the First Run

In performance benchmarking, particularly when using libraries like CBLAS, the first execution of a matrix multiplication algorithm often demonstrates significantly lower performance compared to subsequent runs. This reduction in performance is primarily due to overhead introduced by factors such as dynamic library loading and memory allocation, which can skew the results of the initial run. Figure 4 presents the results after discarding the first run for each matrix size, showing a more consistent and gradual increase in performance for CBLAS MM.

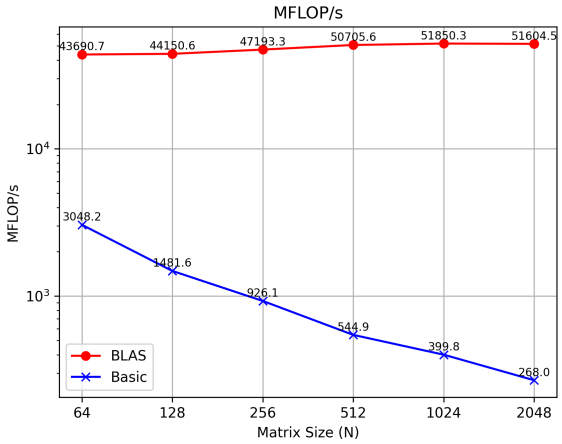

Figure 4: MFLOP/s comparison of Basic MM and CBLAS MM for each matrix size when the first run for each size is discarded. This figure shows a more consistent and gradual increase in performance for CBLAS MM.

B Visualizing Blocked Matrix Multiplication

The following visualizations represent the process of blocked matrix multiplication using a block size of 16×16 for a matrix of size 64×64. Each figure corresponds to a specific stage in the computation, illustrating the progression of matrix multiplication. The blocks currently used for computation are highlighted in red, while the green and blue blocks represent memory cache usage (L1 and L2 caches, respectively).

This visualization scripts for blocked matrix multiplication, available at [3].

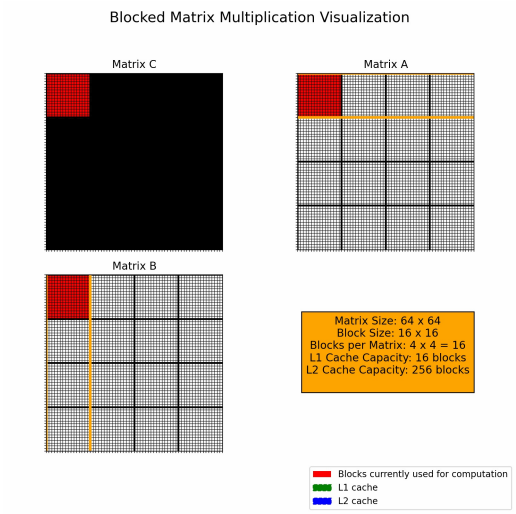

Figure 5: Blocked matrix multiplication at Frame 0. Initial frame showing the first block currently used for computation, highlighted in red. The green blocks indicate the sections that fit into the L1 cache, while blue shows the portions stored in L2 cache.

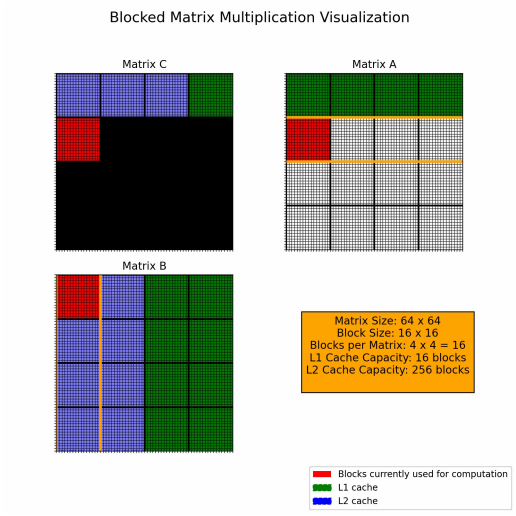

Figure 6: Blocked matrix multiplication at Frame 17. In this frame, computation progresses as more blocks are processed. Notice that different blocks of Matrix A and Matrix B are loaded into the L1 cache, and new sections are moved to the L2 cache.

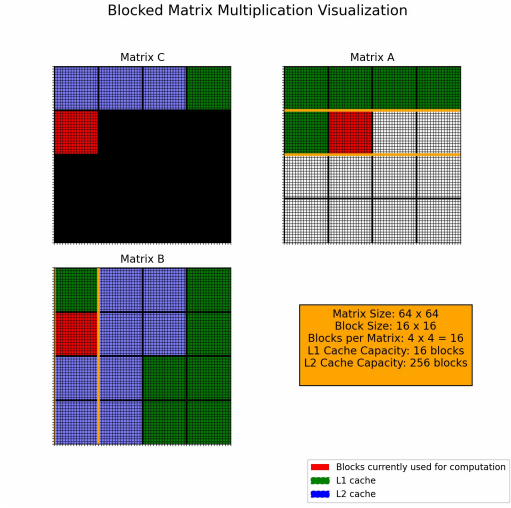

Figure 7: Blocked matrix multiplication at Frame 18. The red block in Matrix C is updated as the corresponding blocks from Matrix A and Matrix B are multiplied. L1 cache is utilized to store the blocks of the matrices being multiplied, improving the efficiency of memory access.

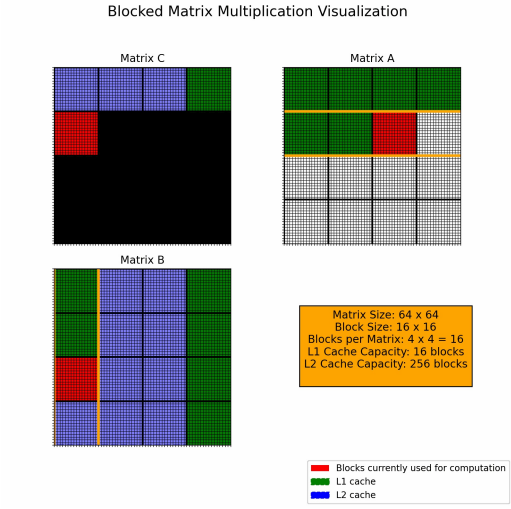

Figure 8: Blocked matrix multiplication at Frame 19. The computation moves forward with more blocks being processed. Matrix A continues to slide horizontally, while Matrix B progresses vertically, keeping the working blocks in the L1 cache for quick access.

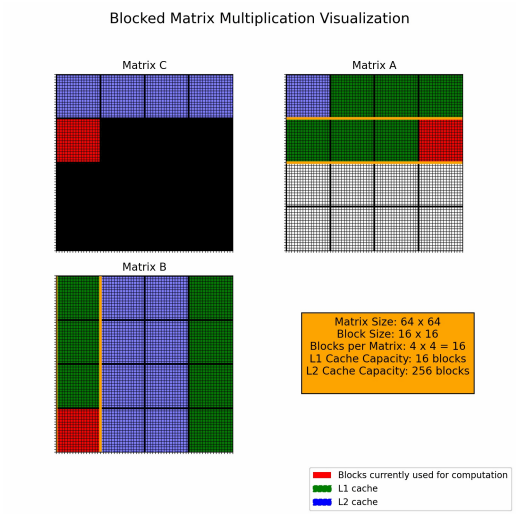

Figure 9: Blocked matrix multiplication at Frame 20. The process of multiplying corresponding blocks from Matrix A and Matrix B continues. Matrix C's red block is updated with new results. By keeping data in the cache, the performance is enhanced due to reduced slow memory access.