Parallel MMUL and LIKWID Performance Counters
Assignment #4, CSC 746, Fall 2024

Shun Usami*
SFSU

ABSTRACT

This assignment investigates two performance optimization techniques—parallelism and cache utilization—by evaluating the performance of three matrix multiplication (MM) methods: Basic Matrix Multiplication with OpenMP (Basic OMP), Blocked Matrix Multiply with Copy Optimization (BMMCO OMP), and CBLAS, using hardware counters. The implementations were tested across varying matrix sizes (128×128, 512×512, and 2048×2048) and thread counts (1, 4, 16, and 64) to assess speedup and efficiency. Results indicate that while Basic OMP scales effectively with increasing thread count, BMMCO OMP exhibits limited scalability for smaller matrices and larger block sizes due to thread underutilization. The BMMCO OMP method significantly reduced L2 and L3 cache accesses, particularly with larger block sizes, resulting in superior performance compared to Basic OMP. These findings suggest that achieving optimal performance requires balancing between increasing cache utilization through larger block sizes and maximizing thread utilization with smaller block sizes.

$$C = C + A \times B$$

$$C_{ij} = C_{ij} + \sum_{k=1}^{n} A_{ik} \times B_{kj}$$

Where:

- A is an $n \times n$ matrix.
- B is an $n \times n$ matrix.
- C is an $n \times n$ matrix.
- i, j, and k are the indices of the matrix elements.

1 INTRODUCTION

This assignment explores the problem of optimizing matrix multiplication (MM) on modern multi-core CPU architectures through two key techniques: parallelism and cache utilization. Efficient MM is critical in high-performance computing, as improvements in execution time can significantly impact scientific and engineering applications. The challenge lies in designing implementations that both maximize multi-threaded parallel execution and optimize memory access patterns to fully utilize the hardware's cache and processing capabilities.

To address this, we implemented and evaluated three MM methods: Basic Matrix Multiplication with OpenMP (Basic OMP), Blocked Matrix Multiply with Copy Optimization (BMMCO OMP), and the highly optimized CBLAS implementation. Basic OMP serves as a baseline for comparison, while BMMCO OMP introduces blocking and copy optimization techniques to improve data locality and cache efficiency. Both Basic OMP and BMMCO OMP were parallelized using OpenMP, and hardware counters were used to measure cache usage and runtime performance. The detailed implementation of these methods is presented in §2.

Our results demonstrate that Basic OMP scales effectively with increasing thread counts, but its cache utilization is suboptimal compared to BMMCO OMP and CBLAS. BMMCO OMP shows significant improvements in cache efficiency, particularly with larger block sizes, but its scalability is limited for smaller matrices due to thread underutilization. While in our implementation there was a tradeoff between maximizing cache utilization and parallelism, this isn't inherently the case. With careful design, it is possible to fully utilize both cache and parallelism simultaneously. The insights gained from these experiments emphasize the importance of balancing these optimizations to achieve the best performance.

2 IMPLEMENTATION

This section presents three MM implementations aimed at evaluating performance improvements through parallelization and optimization techniques. The first implementation, Basic OMP, evaluates the impact of adding shared-memory parallelism to a simple MM approach. The second, Blocked Matrix Multiplication with Copy Optimization (BMMCO), combines cache optimization techniques with parallelization to further enhance computational efficiency. Finally, the CBLAS MM serves as a baseline, using a highly optimized but serial implementation.

Each subsection details the objectives, describes the applied techniques, and presents compact pseudocode or listings to illustrate the use of OpenMP and optimization strategies.

2.1 Basic OMP

The objective of the Basic OMP implementation is to evaluate the effectiveness of parallelizing a MM operation using OpenMP. By implementing parallelization with OpenMP, this version aims to demonstrate performance improvements over the basic serial implementation, serving as a benchmark to understand scaling characteristics in shared-memory parallel environments.

The Basic OMP implementation employs a triply-nested loop to perform MM, where OpenMP is used to parallelize the outermost loop. This allows each iteration of the loop, representing a row computation, to be handled by a different thread. We placed LIKWID markers before and after the main MM operations, ensuring that the collected performance data accurately reflects the parallelized portion of the computation. The source code for this implementation is presented in Listing 1.

*email:susami@sfsu.edu

```
void square_dgemm(int n, double* A, double* B,
    double* C)
{
    #pragma omp parallel for
    for (int i = 0; i < n; ++i) {
        LIKWID_MARKER_START(MY_MARKER_REGION_NAME)
    ;

        for (int j = 0; j < n; ++j) {
            double cij = C[i*n+j];
            for (int k = 0; k < n; ++k) {
                cij += A[i*n+k] * B[k*n+j];
            }
            C[i*n+j] = cij;
        }
        LIKWID_MARKER_STOP(MY_MARKER_REGION_NAME);
    }
}
```

Listing 1: Basic OpenMP implementation of MM. The implementation uses a triply-nested loop with OpenMP parallelism. LIKWID markers are added within the parallel block, specifically before and after the core matrix multiply code, to ensure performance data collection is focused only on the parallel MM operations.

2.2 CBLAS

The objective of the CBLAS implementation is to establish a baseline for the performance of highly optimized MM, without manual parallelization. This implementation uses the CBLAS library [3] to achieve efficient serial MM, serving as a reference to evaluate the benefits of parallelization using OpenMP in the other implementations.

The CBLAS implementation, as shown in Listing 2, wraps a call to the highly optimized CBLAS routine *cblas_dgemm*, which performs matrix-MM. We placed the LIKWID markers to capture performance metrics, ensuring that the collected data accurately reflects the computation time for this highly optimized serial method.

```
void square_dgemm(int n, double* A, double* B,
    double* C) {
    LIKWID_MARKER_START(MY_MARKER_REGION_NAME);
    cblas_dgemm(CblasRowMajor, CblasNoTrans,
    CblasNoTrans, n, n, n, 1., A, n, B, n, 1., C,
    n);
    LIKWID_MARKER_STOP(MY_MARKER_REGION_NAME);
}
```

Listing 2: CBLAS implementation using the cblas_dgemm routine for optimized serial MM. LIKWID markers are used to measure performance data specific to the CBLAS execution.

2.3 BMMCO OMP

The objective of the BMMCO with OpenMP implementation is to evaluate how effectively parallelization, combined with cache optimization techniques like blocking and copy optimization, can enhance MM performance. This approach aims to reduce cache misses and improve data locality, leveraging these optimizations alongside OpenMP to achieve superior efficiency.

The BMMCO OMP implementation divides matrices into smaller blocks, enabling operations on submatrices to improve data locality and make better use of cache memory. Copying these blocks into local storage allows faster cache access, reducing repeated memory costs. OpenMP parallelizes the outermost loop, allowing each block row of C to be processed concurrently by different threads, which enhances computational efficiency and reduces overall runtime. The implementation is shown in Listing 3.

```
void square_dgemm_blocked(int n, int block_size,
    double* A, double* B, double* C)
{
#define MAX_BLOCK (64 * 64)
    static double aBlock[MAX_BLOCK], bBlock[
    MAX_BLOCK], cBlock[MAX_BLOCK];
    int Nb = n / block_size;
    #pragma omp parallel for private(aBlock,
    bBlock, cBlock)
    for (int i = 0; i < Nb; i++) {
        LIKWID_MARKER_START(MY_MARKER_REGION_NAME)
    ;

        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);
                // Perform matrix multiplication
    on the copied blocks
                square_dgemm(block_size, aBlock,
    bBlock, cBlock);
            }
            blockwrite(cBlock, C, i, j, n,
    block_size);
        }
        LIKWID_MARKER_STOP(MY_MARKER_REGION_NAME);
    }
}
```

Listing 3: BMMCO with OpenMP implementation of MM using blocking and copy optimization techniques. The implementation divides the matrices into smaller sub-blocks to improve data locality and minimize cache misses. OpenMP is used to parallelize the outermost loop, allowing each block row to be processed concurrently by different threads. Copying matrix blocks into local storage (faster cache) prior to performing computations helps reduce memory latency by taking advantage of the CPU cache, rather than accessing global memory repeatedly. LIKWID markers are included to measure the performance of the parallelized block operations, focusing specifically on the parallel region of the MM.

3 Results

In this section, we present a series of experiments conducted to evaluate the performance of three matrix multiplication (MM) methods: Basic, BMMCO, and CBLAS. The experiments used matrix sizes of 128×128, 512×512, and 2048×2048. For the Basic and Blocked OpenMP (Parallel) implementations, we tested with thread counts of 1, 4, 16, and 64 to assess the impact of parallelism and thread utilization. These experiments aim to measure the performance gains from cache optimization and parallelism. By analyzing hardware counter values for L2 and L3 cache accesses, we explore the cache efficiency of each method. Additionally, by evaluating the speedup achieved through parallelism for Basic and BMMCO, we investigate potential issues related to thread underutilization.

3.1 Computational platform and Software Environment

The experiments were conducted on a CPU node of the Perlmutter supercomputer at NERSC. Each node is equipped with two AMD

EPYC 7763 (Milan) processors, each featuring 64 cores running at a clock rate of 2.45 GHz. These cores support Simultaneous Multi-threading (SMT), enabling two threads per core. Each core has 32 KiB of L1 cache and 512 KiB of L2 cache, while 8 cores share a 32 MiB L3 cache. The AMD EPYC 7763 processor also features 8 memory channels per socket, with 2 DIMMs per channel, and 4 NUMA domains per socket (NPS=4) [1].

The system is supported by 512 GiB of DDR4 DRAM, offering a memory bandwidth of 204.8 GiB/s per CPU. The processors utilize the AVX2 instruction set for vector processing, and each core has a peak computational throughput of 39.2 GFLOPS [4].

All experiments were conducted on a single CPU node running *SUSE Linux Enterprise Server 15 SP4*, with kernel version *5.14.21-150400.24.81_12.0.87-cray_shasta_c* [5]. The C++ code was compiled using *g++-12 (SUSE Linux) 12.3.0* with the following optimization flags: `-fopenmp -Wall -pedantic -march=native`. Note that the `-fopenmp` flag was not used for compiling the CBLAS implementation.

The following OpenMP environmental variables were used for the VMM OpenMP implementation. The variable `OMP_NUM_THREADS` was left unset as it conflicts with `likwid-perfctr` on Perlmutter:

- `OMP_PLACES=threads`: Assigns each OpenMP thread to a separate hardware thread, ensuring efficient mapping to processing units.
- `OMP_PROC_BIND=spread`: Distributes threads evenly across the available CPUs, aiming for optimal workload balancing.

The execution command for each program was:

```
likwid-perfctr -m -g FLOPS_DP -C
N:0-$num_threads-1 ./benchmark -N $N
```

The command runs the benchmark executable while collecting performance metrics using `likwid-perfctr`. The options used in the command are as follows:

- `-g FLOPS_DP`: Specifies the performance group to measure, in this case, the *FLOPS_DP* group, which tracks double-precision floating-point operations to evaluate computational throughput.
- `-C N:0-$num_threads-1`: Binds the execution to a set of cores. The option `N:0-$num_threads-1` allows specification of core IDs from 0 to `num_threads-1`, where `num_threads` represents the nuMiBer of threads being used. This ensures that only the desired cores are used during the run.

3.2 Methodology

We evaluated the performance of various MM methods using matrix sizes of 128×128, 512×512, and 2048×2048. For the Basic and Blocked OpenMP implementations, we tested with thread counts of 1, 4, 16, and 64 to observe the impact of varying concurrency levels on performance.

Performance was measured using LIKWID markers placed around the core MM code, as shown in Listing 1, Listing 2, and Listing 3. We used three performance groups to collect performance counter data: *FLOPS_DP*, *L2CACHE*, and *L3CACHE*. Specifically:

- *FLOPS_DP* was used for collecting runtime and instruction count data.
- *L2CACHE* provided information about the number of requests that were not satisfied by the L1 cache.
- *L3CACHE* indicated the number of requests that were not satisfied by the L2 cache.

Based on the runtime data, we derived the speedup at varying levels of concurrency for the MM OpenMP (Parallel) implementations. We collected instruction count, L2 accesses, and L3 accesses only for serial runs.

3.2.1 Runtime

To collect runtime data, we used the *FLOPS_DP* performance group, specifically focusing on the `Runtime (RDTSC) [s]` metric from the LIKWID output. For parallel runs, we took the maximum runtime across all threads. Using the maximum value ensures that the measurement reflects any imbalance in load distribution, as the runtime of a parallel program is ultimately bounded by the slowest thread.

3.2.2 Instruction Count

Instruction count data was collected using the *FLOPS_DP* performance group, specifically the `RETIRED_INSTRUCTIONS` event from the LIKWID output. This metric represents the number of instructions retired during the execution of the core MM code.

3.2.3 L2 Accesses

We used the *L2CACHE* performance group to collect L2 cache access data. The `L2 accesses` metric from the LIKWID output represents the total number of L2 cache requests, calculated by summing the following counters: `REQUESTS_TO_L2_GRP1_ALL_NO_PF`, `L2_PF_HIT_IN_L2`, `L2_PF_HIT_IN_L3`, and `L2_PF_MISS_IN_L3`.

3.2.4 L3 Accesses

We used the *L3CACHE* performance group to collect L3 cache access data, specifically utilizing the `L3_CACHE_REQ` event from the LIKWID output. This metric indicates the number of memory requests that were not fulfilled by the L2 cache and thus required L3 access.

3.2.5 Speedup

Speedup was defined as the ratio of a parallel program's execution time compared to a sequential program's execution time. For a problem size n and p parallel threads, the speedup $S(n,p)$ was calculated as follows:

$$S(n,p) = \frac{T^*(n)}{T(n,p)}$$

Here, $T^*(n)$ represents the runtime of the best serial algorithm for a problem of size n. In our case, this was the runtime recorded for $OMP\_NUM\_THREADS = 1$. Thus, for Basic OMP, $T^*(n)$ is the time obtained when running Basic OMP with a single thread. $T(n,p)$ represents the time taken to run the program on a problem of size n using p parallel threads.

3.3 Scaling Study for Basic OMP parallelization

The scaling performance of the Basic OMP implementation was analyzed across different concurrency levels ($p = 4, 16, 64$) and matrix sizes ($N = 128, 512, 2048$). Figure 1 shows the speedup chart, illustrating the relationship between concurrency levels and problem sizes, allowing us to evaluate the efficiency of the parallelization approach.

For thread counts of 4 and 16, the speedup demonstrated a steady increase as the matrix size N increased, approaching the theoretical ideal speedup of 4 and 16, respectively. This behavior aligns well with expectations based on Amdahl's Law [2], which suggests that for larger problem sizes, the impact of the serial portion of the code becomes less significant, thereby reducing overhead and enhancing the effectiveness of parallel execution. In other words, larger matrix sizes increase the computational workload sufficiently to minimize the negative impact of serial portions, thus leading to near-ideal speedup values.

For a thread count of 64, the results were somewhat mixed. The highest speedup was observed for $N = 2048$, which was expected given the increased computational workload and better utilization of the available cores. However, an unexpected observation was made for $N = 512$, where the speedup was lower compared to $N = 128$. This deviation requires further investigation.

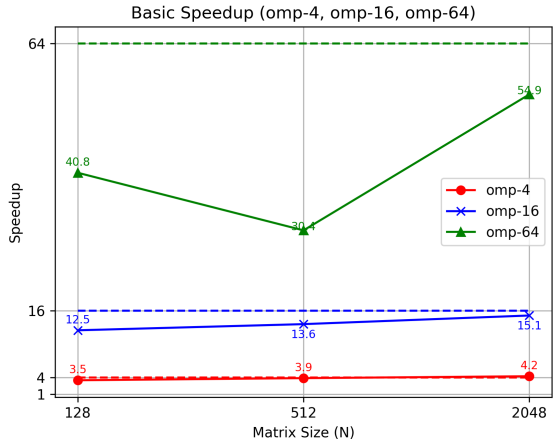

Figure 1: Speedup of Basic OMP with OpenMP Parallelization. The figure shows the speedup achieved by the Basic OMP implementation at different concurrency levels ($p = 4, 16, 64$) for matrix sizes of $N = 128, 512, 2048$. The ideal speedup for each concurrency level is also shown for comparison, highlighting the efficiency of parallelization as the matrix size increases. It is observed that for $p = 64$, the speedup for $N = 512$ is unexpectedly lower compared to $N = 128$, suggesting potential issues such as cache contention or memory bandwidth limitations.

Interestingly, when measuring runtime using `std::chrono::high_resolution_clock`, as shown in Figure 2, the results appear more consistent with the expected outcomes of Amdahl's Law. Specifically, for each thread count, $N = 128$ consistently achieves lower speedup, while larger N values yield speedup results that are closer to the theoretical ideal. This steady increase in speedup as N increases suggests that the instrumentation code used in Listing 4 provides a more accurate representation of the expected scaling behavior.

```
setup(n, A, B, C);
start_time = get_high_resolution_clock_now();
square_dgemm(n, A, B, C);
end_time = get_high_resolution_clock_now();
elapsed_time = end_time - start_time;
```

Listing 4: Instrumentation code for measuring the elapsed time of MM

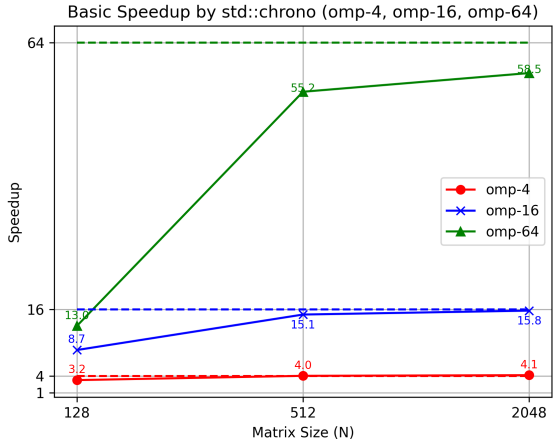

Figure 2: Speedup of Basic OMP with OpenMP Parallelization measured by std::chrono::high_resolution_clock. This figure shows the speedup results when using `std::chrono::high_resolution_clock` for measuring elapsed time. Unlike the previous measurements, the speedup values exhibit more predictable scaling behavior, consistent with Amdahl's Law, where larger matrix sizes ($N = 512$, $N = 2048$) result in improved speedup, while $N = 128$ demonstrates lower efficiency.

3.4 Evaluation of BMMCO OMP parallelization

The scaling performance of the BMMCO OMP implementation was analyzed across different concurrency levels ($p = 4, 16, 64$), block sizes ($b = 4, 16$), and matrix sizes ($N = 128, 512, 2048$). Figure 3 presents the speedup chart for six different configurations, illustrating the interplay between concurrency levels, block sizes, and problem sizes.

Overall, the scaling behavior aligns well with Amdahl's Law [2], with larger matrix sizes yielding near-ideal speedup as parallelism increases. Larger problem sizes effectively reduce the overhead from serial portions of the code and allow for more efficient workload distribution across threads.

One notable observation is that a block size of 4 achieved better speedup for smaller matrix sizes ($N = 128, 512$) compared to a block size of 16. Specifically, Blocked B16 at concurrency levels of $p = 16, 64$ for $N = 128$ and $p = 64$ for $N = 512$ exhibited limited speedup. Additionally, Blocked B4 for $N = 128$ showed that increasing the thread count from $p = 16$ to $p = 64$ actually resulted in worse performance. These observations can be attributed to how parallelism was applied in the BMMCO implementation, where the outermost loop was parallelized (as shown in Listing 3). The number of iterations in this outermost loop corresponds to the number of blocks along a row, which directly affects thread utilization.

Tables 1 and 2 show the number of blocks per row for different matrix sizes and block sizes, along with the corresponding thread utilization for different concurrency levels. For $b = 16$ and $N = 128$, only 8 blocks are available, which limits the number of threads that can be effectively utilized to 8. This is significantly lower than the total available threads for configurations with $p = 16, 64$, leading to underutilization of potential parallelism. Similarly, for $N = 512$, $b = 16$ results in 32 blocks, which limits effective thread utilization to 32 when using $p = 64$.

On the other hand, $b = 4$ provides more blocks for each problem size, which enables better thread utilization, particularly for smaller matrices like $N = 128$. For $N = 128$, $b = 4$ allows for 32 blocks, which is sufficient to fully utilize threads for configurations with $p = 4, 16$. However, increasing the thread count to $p = 64$ resulted

in underutilization, as the overhead of multithreading outweighed the performance gain from adding more threads.

In conclusion, block size has a significant impact on scaling performance, particularly for smaller problem sizes. The availability of sufficient blocks to match the number of threads determines the efficiency of parallelism. For larger matrices ($N = 2048$), both block sizes showed good scalability, as there were enough blocks available to fully utilize all threads across all concurrency levels.

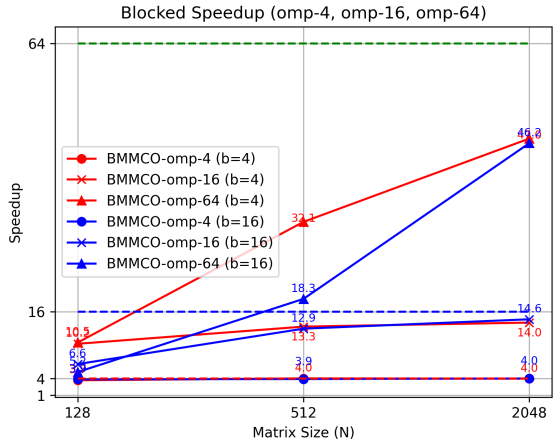

Figure 3: Speedup of BMMCO with OpenMP Parallelization for Different Block Sizes. The figure shows the speedup achieved by the BMMCO implementation at different concurrency levels ($p = 4, 16, 64$) for matrix sizes of $N = 128, 512, 2048$ with block sizes of $b = 4$ and $b = 16$. The results demonstrate that $b = 4$ generally provides better performance for smaller matrix sizes due to improved thread utilization, whereas $b = 16$ struggles to fully leverage available threads for smaller matrices.

N	Blocks	$p = 4$	$p = 16$	$p = 64$
128	32	100%	100%	50%
512	128	100%	100%	100%
2048	512	100%	100%	100%

Table 1: Number of Blocks per Row and Thread Utilization for Block Size of 4. The table shows the number of blocks per row for different matrix sizes and the resulting thread utilization percentage for different concurrency levels. Block size $b = 4$ provides sufficient blocks to maintain high thread utilization for most configurations.

N	Blocks	$p = 4$	$p = 16$	$p = 64$
128	8	100%	50%	12.5%
512	32	100%	100%	50%
2048	128	100%	100%	100%

Table 2: Number of Blocks per Row and Thread Utilization for Block Size of 16. The table shows the number of blocks per row for different matrix sizes and the resulting thread utilization percentage for different concurrency levels. Block size $b = 16$ results in fewer blocks for smaller matrices, which limits the effective thread utilization, particularly at higher concurrency levels.

3.5 Comparison of CBLAS, Basic OMP, and BMMCO OMP

3.5.1 L1 Cache Utilization

L2 accesses represent the number of requests that are not satisfied by the L1 cache and, therefore, must be handled by the L2 cache. The Basic implementation shows L2 accesses that are between 50 and 100 times greater than those for CBLAS, indicating significantly poorer cache efficiency. In contrast, Blocked B4 has L2 accesses roughly 5 to 8 times higher than CBLAS, indicating an improvement compared to Basic but still relatively far from optimal. Blocked B16, with L2 accesses between 1.2 and 1.8 times that of CBLAS, demonstrates the closest behavior to the highly optimized CBLAS implementation.

These results indicate that Blocked B16 achieves the most efficient use of the L2 cache, as its L2 access values are closest to those of CBLAS, which serves as the baseline for an optimized memory access pattern. Increasing the block size from 4 to 16 leads to a significant reduction in L2 accesses, highlighting the impact of larger block sizes in enhancing data locality and reducing the need for higher-level cache accesses.

It is reasonable to assume that the runtime of these methods is proportional to the number of L2 accesses, given that the number of floating-point operations (FLOPs) remains largely consistent across implementations. Therefore, the primary factor contributing to runtime differences is the efficiency of memory access patterns and cache utilization. The reduced L2 cache accesses observed in the blocked methods, particularly Blocked B16, should contribute to lower runtimes compared to Basic OMP, as fewer requests to higher levels of the memory hierarchy result in fewer latency penalties.

N	CBLAS	Basic	Blocked B4	Blocked B16
128	1	47.89	5.37	1.19
512	1	45.53	4.41	1.30
2048	1	116.14	7.91	1.86

Table 3: L2 Cache accesses. The table shows the value of the L2 Accesses metric normalized by the corresponding value of the L2 Accesses metric for CBLAS. Blocked B4 refers to the blocked method with a block size of 4, and Blocked B16 refers to the blocked method with a block size of 16. The CBLAS column serves as a baseline with all values equal to 1, while the other columns represent the ratio of L2 accesses compared to CBLAS for each problem size. Basic OMP shows significantly higher L2 accesses compared to CBLAS, whereas Blocked B16 achieves the closest values to CBLAS, indicating more efficient cache usage and better alignment with an optimized memory access pattern.

3.5.2 L2 Cache Utilization

L3_CACHE_REQ represents the number of requests that are not satisfied by the L2 cache and, therefore, must be handled by the

L3 cache. For larger matrix sizes ($N = 512$ and $N = 2048$), the Basic implementation exhibits an order of magnitude more L3 cache requests compared to the blocked methods, indicating inefficient L2 cache utilization. Blocked B16 consistently demonstrates approximately 5 to 6 times fewer L3 cache requests than Blocked B4, suggesting that increasing the block size significantly improves data locality and cache efficiency, thus reducing the dependency on L3 cache.

Interestingly, for $N = 128$, Basic, Blocked B4, and Blocked B16 show fewer L3 cache requests than CBLAS. This suggests some overhead in the CBLAS implementation that necessitates additional L3 cache accesses.

We assume that the runtime of these methods is closely correlated to the number of L3 cache requests, as the number of FLOPs required for MM remains relatively constant across implementations. Therefore, the primary factor contributing to runtime differences is the efficiency of memory access patterns and cache utilization. The significant reduction in L3 cache requests observed in the blocked methods, particularly Blocked B16, likely translates to lower runtimes compared to Basic OMP due to fewer accesses to higher memory levels and the associated lower latency penalties.

N	CBLAS	Basic	Blocked B4	Blocked B16
128	1	0.41	0.44	0.60
512	1	283.96	18.51	3.61
2048	1	578.74	22.49	4.16

Table 4: L3 Cache accesses. The table shows the value of the L3_CACHE_REQ counter normalized to the corresponding value for CBLAS. The CBLAS column contains values of 1, serving as a baseline, while the other columns represent the ratio of L3 cache requests relative to CBLAS for each problem size. Blocked B4 refers to the blocked method with a block size of 4, and Blocked B16 refers to the blocked method with a block size of 16. For larger matrix sizes ($N = 512, 2048$), Basic OMP shows significantly higher L3 cache requests compared to CBLAS, whereas Blocked B16 achieves values closest to CBLAS, indicating more efficient L3 cache usage. However, for $N = 128$, Basic, Blocked B4, and Blocked B16 show fewer requests than CBLAS, suggesting an overhead in CBLAS that increases L3 access.

N	CBLAS	Basic	Blocked B4	Blocked B16	L2
128	384 KiB	384 KiB	384 KiB	390 KiB	Yes
512	6 MiB	6 MiB	6 MiB	6 MiB	No
2048	96 MiB	96 MiB	96 MiB	96 MiB	No

Table 5: Memory Footprint for Different Matrix Sizes and Methods. The table shows the memory required for matrices of each size and whether they fit within the L2 cache (512 KiB). Each method requires memory for three matrices. In addition, the blocked methods require three blocks, which are relatively small except for Blocked B16 at $N = 128$.

3.5.3 Instruction count analysis: RETIRED_INSTRUCTIONS

The relative number of instructions executed compared to CBLAS is approximately 10 times greater for Basic, 24 to 28 times for Blocked B4, and 12 to 13 times for Blocked B16. In the case of Basic, the increased instruction count does not significantly impact runtime since the primary bottleneck lies in memory access rather than arithmetic operations. This is indicative of Basic's poor cache utilization and frequent memory accesses.

For the blocked methods, however, memory access is significantly optimized, which makes the number of retired instructions more directly proportional to runtime. The larger instruction counts for Blocked B4 compared to Blocked B16 suggest inefficiencies that could be linked to fewer vectorization opportunities or less efficient use of fused multiply-add (FMA) instructions. Blocked B16, with a lower instruction count than Blocked B4, indicates more efficient execution likely due to better cache locality and greater potential for hardware-level optimizations such as vectorization and FMA usage. As such, the reduced instruction count in Blocked B16 can contribute to improved runtime performance.

N	CBLAS	Basic	Blocked B4	Blocked B16
128	1	9.99	24.66	11.88
512	1	10.60	27.48	13.28
2048	1	10.64	27.74	13.42

Table 6: Instruction Count Analysis. The table shows the RETIRED_INSTRUCTIONS counter values normalized by the corresponding value for CBLAS, with CBLAS values set to 1 as a baseline. The relative instruction counts for Basic, Blocked B4, and Blocked B16 provide insights into the efficiency of different implementations. Basic OMP has significantly higher instruction counts compared to CBLAS, while Blocked B16 shows lower instruction counts than Blocked B4, indicating more efficient execution due to better optimization techniques.

3.6 Findings and Discussion

Overall, Basic OMP scaled well as the level of parallelism increased, particularly due to the large number of iterations in the outermost loop. However, for BMMCO, particularly when the matrix size was small and the block size large, the performance gains from parallelism were limited due to underutilization of threads. In these cases, the number of blocks per row was insufficient to fully leverage the available threads. To mitigate this limitation, we could modify the code by collapsing two outermost loops, increasing the effective number of iterations and thus improving thread utilization.

As shown in Table 3 and Table 4, Basic OMP exhibits poor cache utilization, particularly for L1 and L2 caches, when compared to CBLAS. Blocked B16 demonstrates much better L1 cache utilization, approaching the efficiency of CBLAS (only 1.2x to 1.8x more L2 accesses). However, Blocked B16 is still not fully utilizing the L2 cache, as its L3 requests are 3.6x to 4.2x those of CBLAS for larger matrix sizes. Blocked B4 falls between Basic OMP and Blocked B16 in terms of cache efficiency, showing moderate improvement over Basic OMP but still trailing behind Blocked B16 in terms of L1 and L2 cache utilization.

To improve memory system utilization for the blocked methods, increasing the block size helps reduce the number of higher-level cache accesses, but we need to balance this against the number of available blocks and their impact on parallel thread utilization. An optimal balance between block size and parallelism can significantly reduce latency penalties from memory hierarchy inefficiencies, leading to better overall performance.

ACKNOWLEDGMENTS

This paper was edited with the assistance of *ChatGPT (GPT-4o and GPT-4o with canvas)* (accessed October 2024), which was primarily used for correcting grammatical errors and rephrasing. 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] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), p. 483–485. Association for Computing Machinery, New York, NY, USA, 1967. doi: 10.1145/1465482.1465560

[3] B. W. Group. *CBLAS: C Interface to the Basic Linear Algebra Subprograms*. Netlib, 1993.

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

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