Scalability and Performance Analysis of OpenMP-Parallelized Matrix-Matrix Multiplication on Modern CPU Architectures
CP#4, CSC 746, Fall 2023

Ekarat Buddharuksa*
San Francisco State University

Abstract

In this study, we explore the parallelization of Matrix-Matrix Multiplication (MM) using OpenMP on modern multi-core CPU architectures.

Through employing basic and blocked multiplication methodologies alongside copy optimizations, and evaluating the implementations against a standard CBLAS reference, we aim to understand the performance dynamics under varying concurrency and problem sizes. Utilizing the LIKWID performance counters on Perlmutter's CPU nodes, the performance metrics including runtime, L2CACHE and L3CACHE memory performance counters, and instruction count were meticulously analyzed.

The findings unveil critical insights into the scalability, memory hierarchy utilization, and instruction execution efficiency of the implemented parallelized algorithms, paving a path towards optimization.

1 Introduction

Matrix-Matrix Multiplication is a cornerstone computation in many scientific and engineering applications. The demand for higher computational efficiency and lower execution time has driven the adoption of parallel computing methodologies. In this study, we aimed to parallelize MM using OpenMP, a widely recognized API for shared-memory parallel programming, and evaluate the performance on modern multi-core architectures.

Our approach encompassed implementing two versions of MM - a basic and a blocked with copy optimization version. The implementations were rigorously evaluated against a standard CBLAS reference under varying concurrency levels, block sizes, and matrix problem sizes. Utilizing the LIKWID performance counters on Perlmutter's CPU nodes, we meticulously collected and analyzed the hardware performance counter data to evaluate the runtime, memory hierarchy utilization, and instruction count across different configurations.

The analysis provides an understanding of how concurrency levels and block sizes impact the performance and scalability of our parallelized MM implementations. The comparative evaluation with the standard CBLAS reference elucidated the efficiency and areas of improvement in our implementations. Moreover, the examination of memory hierarchy utilization and instruction count provided a detailed comprehension of how well the implementations interact with the memory system hierarchy, and how instruction execution efficiency varies across different configurations.

*email:ebuddharuksa@sfsu.edu

2 Implementation

2.1 Benchmark

2.1.1 Command Line Options

In the `benchmark.cpp` file, two command line options -N and -B are introduced to allow users to specify the problem size and block size, respectively. These options provide a flexible way to control the dimensions of the matrix and the block size for blocked matrix multiplication, enabling experimentation with different sizes to observe their impact on performance.

Here's a snippet of code illustrating the parsing of these command line options:

```
    int cmdline_N = -1;
    int cmdline_B = -1;
    int c;

   while ( (c = getopt(argc, argv, "N:B:")) != -1)
     {
      switch(c) {
         case 'N':
            cmdline_N = std::atoi(optarg == NULL ?
     "-999" : optarg);
            break;
#ifdef BLOCKED
         case 'B':
            cmdline_B = std::atoi(optarg == NULL ?
     "-999" : optarg);
            break;
#endif
       }
   }
```

Listing 1: Parsing command line options in benchmark.cpp.

In this snippet:

- The variables `cmdline_N` and `cmdline_B` are initialized to store the values of the problem size and block size specified on the command line.
- The `getopt` function is used to parse the command line options -N and -B, and the values are then stored in `cmdline_N` and `cmdline_B`, respectively.

2.1.2 Timing Mechanism

The timing mechanism in `benchmark.cpp` utilizes the `chrono` library from C++ to measure the execution time of the matrix multiplication routines. This method offers high-precision timing, which is crucial for analyzing the performance of these routines accurately. Below is the segment of code responsible for timing the execution:

```
    std::chrono::time_point<std::chrono::
    high_resolution_clock> start_time = std::
    chrono::high_resolution_clock::now();
#ifdef BLOCKED
    square_dgemm_blocked(n, b, A, B, C);
#else
```

```
    square_dgemm(n, A, B, C);
#endif
    std::chrono::time_point<std::chrono::
    high_resolution_clock> end_time = std::chrono
    ::high_resolution_clock::now();
    std::chrono::duration<double> elapsed =
    end_time - start_time;
    std::cout << " Elapsed time is : " << elapsed.
    count() << " (sec) " << std::endl;
```

Listing 2: Timing mechanism implemented in benchmark.cpp.

In this segment:

- The `std::chrono::high_resolution_clock::now` function captures the current time before and after the execution of the matrix multiplication routine.
- The duration of the matrix multiplication is then computed by subtracting the start time from the end time.
- The `elapsed.count()` function call converts the duration to seconds, which is then printed to the console.

This timing tool not only measures pure execution times but also serves as a foundation for calculating other metrics, thereby enriching the overall analysis of the matrix multiplication performance.

2.2 Basic Matrix Multiplication with OpenMP

The Basic Matrix Multiplication method multiplies two matrices using nested loops. Given matrices A and B, the product matrix C is computed such that each element C_{ij} is the dot product of the i^{th} row of A and the j^{th} column of B. The matrices are stored in column-major format.

To enhance the efficiency of the multiplication process, this method is optimized using OpenMP to parallelize the computation. Moreover, the LIKWID performance monitoring library is employed to gather performance metrics during the execution. LIKWID markers are utilized to delineate the region of code whose performance is to be measured. These markers allow for the collection of performance data specifically for the matrix multiplication portion of the code.

Here's the code representation of this method with OpenMP optimizations and LIKWID marker usage:

```
void square_dgemm(int n, double *A, double *B,
    double *C)
{
#pragma omp parallel
   {
      LIKWID_MARKER_START(MY_MARKER_REGION_NAME);
#pragma omp for collapse(2)
      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];
            }
         }
      LIKWID_MARKER_STOP(MY_MARKER_REGION_NAME);
   }
}
```

Listing 3: OpenMP-optimized code implementation for Basic Matrix Multiplication

In this code snippet, LIKWID markers are employed to demarcate the start and the end of the region of interest. The macro `LIKWID_MARKER_START` signifies the commencement of the region to be monitored, while `LIKWID_MARKER_STOP` denotes its termination. These markers instruct LIKWID to collect performance data during the execution of the matrix multiplication, enabling a detailed analysis of its performance.

2.3 Blocked Matrix Multiplication with Copy Optimization

Blocked Matrix Multiplication with Copy Optimization partitions matrices into smaller blocks, facilitating optimized cache memory usage during multiplication. The "copy optimization" feature is vital as it copies blocks of matrices A and B into local storage, ensuring contiguous memory access and enhanced cache utilization.

2.3.1 Algorithm Overview

The algorithm iterates over the matrices in blocks, copying blocks of matrices A, B, and C into local storage. The multiplication of individual blocks is then performed using the data in local storage, optimizing memory access. The results are then copied back to matrix C.

Algorithm 1 Blocked Matrix Multiplication with Copy Optimization Algorithm

for each block si, sj in matrices A and B **do**
  Copy blocks of C into C_block
  for each block sk in matrix A and B **do**
    Copy blocks of A and B into A_block and B_block
    for each element in A_block and B_block **do**
      Multiply and accumulate to update C_block
    end for
  end for
  Copy C_block back to C
end for

2.3.2 Code Implementation

The code implementation is OpenMP-enabled for parallel execution and employs LIKWID markers to monitor performance metrics.

```
void square_dgemm_blocked(int n, int block_size,
    double *A, double *B, double *C)
{
   // Parallel region begins
#pragma omp parallel
   {
      LIKWID_MARKER_START(MY_MARKER_REGION_NAME);
#pragma omp for
      //Initialize Blocks A, B, C
      //Implementation from pseudocode

      LIKWID_MARKER_STOP(MY_MARKER_REGION_NAME);
   }
}
```

Listing 4: Blocked Matrix Multiplication with Copy Optimization and OpenMP Parallelism

This method is optimized using OpenMP to parallelize the computation. Moreover, the LIKWID performance monitoring library is employed to gather performance metrics during the execution, similar to the basic implementation.

2.4 CBLAS Implementation

The CBLAS (C interface to the Basic Linear Algebra Subprograms) provides a set of routines to perform common linear algebra operations. In this study, we utilize the CBLAS implementation for matrix multiplication. The primary advantage of using CBLAS is its optimized performance for matrix operations, leveraging

underlying hardware capabilities. The routine 'cblas_dgemm' is a double-precision general matrix-matrix multiplication function, which computes $C := C + A \times B$, where A, B, and C are matrices stored in column-major format. We employ LIKWID to monitor the algorithm's performance. For the scope of this assignment, we solely extract the time metrics from a serial execution for this algorithm.

Here's the code representation of this method with LIKWID marker usage:

```
void square_dgemm(int n, double* A, double* B,
    double* C) {
#ifdef LIKWID_PERFMON
      LIKWID_MARKER_START(MY_MARKER_REGION_NAME);
#endif
    cblas_dgemm(CblasColMajor, CblasNoTrans,
    CblasNoTrans, n, n, n, 1., A, n, B, n, 1., C,
    n);
#ifdef LIKWID_PERFMON
      LIKWID_MARKER_STOP(MY_MARKER_REGION_NAME);
#endif
```

Listing 5: CBLAS Implementation for Matrix Multiplication

3 Evaluation

3.1 Computational Platform and Software Environment

Our computational experiments were conducted on the Perlmutter system [1] at NERSC. For our experiments, we specifically utilized the CPU nodes of the Perlmutter system. Each of these nodes is equipped with two AMD EPYC 7763 (Milan) CPUs, with each CPU boasting 64 cores. These processors support the AVX2 instruction set and have an L3 cache of 256MB. The nodes have a total of 512 GB DDR4 memory, providing a memory bandwidth of 204.8 GB/s per CPU. In terms of performance, each core can achieve 39.2 GFlops, leading to a total of 2.51 TFlops per socket. The Perlmutter system operates on a specialized version of the Cray Linux Environment.

For performance monitoring and analysis, we employed the LIKWID performance monitoring library. LIKWID, which stands for "Like I Knew What I'm Doing," facilitates easy access to hardware performance counters in CPUs. It provides a set of command line utilities and a library for adding performance counter support to applications. This tool was instrumental in monitoring various performance metrics during the algorithm's execution on the Perlmutter system, which aided in the performance tuning and optimization of our implementations. Through the use of LIKWID, we were able to monitor and analyze the performance of our algorithms, gaining insightful data on the behavior of our algorithms and the utilization of system resources, which in turn informed further optimizations.

3.2 Methodology

The principal aim of our testing methodology is to meticulously evaluate the performance and efficiency of our matrix multiplication implementations. The measurements focus on runtime, cache memory performance, and instruction count. Below is an elaboration of the metrics we considered and the procedure followed to gather and analyze the data.

3.2.1 Runtime

Runtime is a crucial metric to understand the execution efficiency of our algorithms. We utilized the Runtime (RDTSC) metric from the LIKWID output to measure the time taken by our implementations to execute. In the case of parallel runs, the maximum value of Runtime (RDTSC) across all threads is considered. This choice is based on the rationale that in parallel processing, the total runtime is dictated by the slowest thread. Hence, taking the maximum runtime provides a more realistic measure of the algorithm's performance as opposed to the minimum or sum.

3.2.2 Problem Sizes and Concurrency Levels

The experiments were conducted over a range of problem sizes $\{128, 512, 2048\}$ to understand how the performance scales with the size of the problem. Additionally, different concurrency levels $\{1, 4, 16, 64\}$ were explored to study the impact of parallelism on the performance of our Basic Matrix Multiplication implementation with OpenMP.

3.2.3 L2 and L3 Cache Request Table

We evaluated the L2CACHE and L3CACHE memory performance counters, as they are critical in understanding the memory access patterns and the efficiency of our implementations concerning cache usage. The data for these counters was collected during serial, single-threaded runs to provide a baseline for comparison between our serial BasicMM-omp and BlockedMM-omp matrix multiplication implementations against the serial CBLAS implementation.

3.2.4 Retired Instruction Table

We examined the instruction count by running the `likwid-perfctr` with the `FLOPS_DP` group. The sum of `RETIRED_INSTRUCTIONS` and `RETIRED_SSE_AVX_FLOPS_ALL` provided us with the total count of instructions executed. This metric is significant in understanding the computational intensity of our algorithms. The sum of the instruction count for each Basic+omp and Blocked-omp of various problem sizes and concurrency levels will then be normalized with a serial run from Cblas implementation to create a Retired Instruction table.

3.2.5 Speedup Plots

For speedup, the horizontal axis represents the problem size, while the vertical axis denotes speedup. Separate datasets were maintained for each concurrency level, yielding three datasets in the speedup plot for the BasicMM+OpenMP plot. The reference time $T^*(n)$ for computing speedup is obtained from the runtime of the best serial algorithm on a problem size of N, at $t = 1$. For instance, for BasicMM+omp, $T^*(n)$ is the time recorded when running BasicMM+omp with `OMP_NUM_THREADS=1`. This approach ensures that the speedup computations are referenced to the serial time for the respective code.

3.3 Results

3.3.1 Scaling Study for Basic MM with OpenMP parallelization

In our scaling study, we aim to analyze the impact of parallelization on the performance of Basic Matrix Multiplication (BasicMM) across various problem sizes and levels of concurrency. The speedup chart elucidates the scaling performance of BasicMM when executed with OpenMP parallelization at concurrency levels $p = 4, 16, 64$.

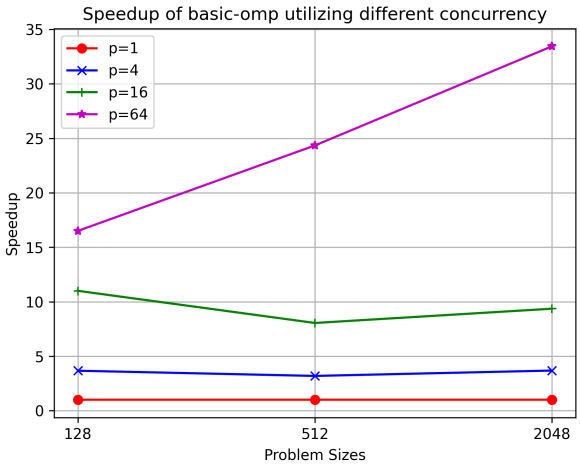

Figure 1: Speedup comparison of BasicMM when executed with OpenMP parallelization at concurrency levels

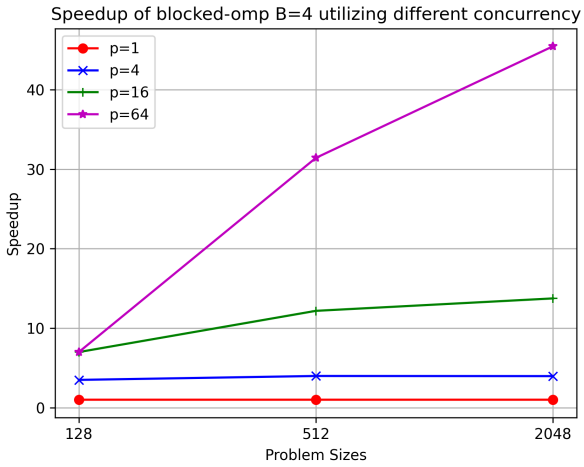

Figure 2: Speedup comparison of BlockedMM when executed with OpenMP parallelization at concurrency levels with block size is 4

The speedup chart reveals the following observations:

- At a lower problem size of $N = 128$, the speedup for $p = 4$ is 3.67, for $p = 16$ is 11, and for $p = 64$ is 16.5. These values indicate effective parallelization, especially as the concurrency level increases.
- As the problem size increases to $N = 512$ and $N = 2048$, the speedup does not scale linearly with the concurrency level. This is evident as the speedup values at $p = 16$ and $p = 64$ do not exhibit a linear increase compared to the speedup at $p = 4$ across different problem sizes.
- The highest level of concurrency $p = 64$ yields a higher speedup at the largest problem size $N = 2048$ as compared to smaller problem sizes. This suggests that larger problems benefit more from higher concurrency levels, possibly due to better utilization of the available parallel resources.

The data suggests that while OpenMP parallelization significantly enhances the performance of BasicMM, the efficiency of parallelization at lower concurrency levels ($p = 4$ and $p = 16$) seems to diminish as the problem size grows. However, at a higher concurrency level of $p = 64$, the efficiency of parallelization appears to improve with the problem size, possibly due to better utilization of parallel resources or decreased relative overhead of thread management and synchronization. The insights from this scaling study are instrumental in understanding the behavior of BasicMM with OpenMP parallelization across varying problem sizes and concurrency levels, which is crucial for optimizing the performance of parallel algorithms.

The deviation from linear speedup with the increase in problem size can be attributed to several factors. One probable cause is the overhead associated with managing multiple threads, which becomes more pronounced as the number of threads increases. Additionally, as the problem size grows, the data may exceed the cache size, leading to increased cache misses and, consequently, a degradation in performance.

3.3.2 Evaluation of BMMCO with OpenMP parallelization

The scaling performance of Blocked-omp code was evaluated across three levels of concurrency ($t = 4, 16, 64$), two block sizes ($b = 4, 16$), and various problem sizes. The data analysis reveals several notable trends in speedup, as detailed below.

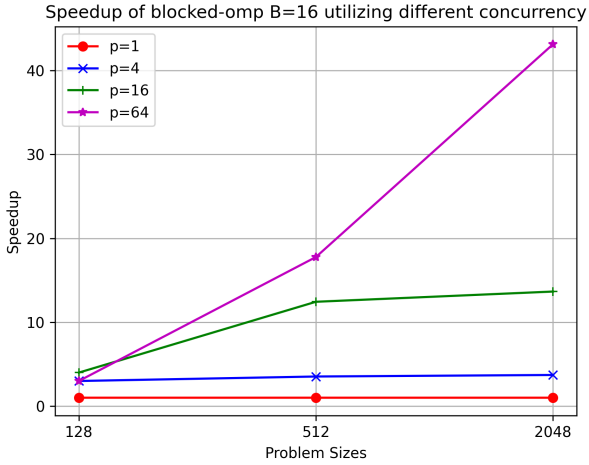

Figure 3: Speedup comparison of BlockedMM when executed with OpenMP parallelization at concurrency levels with block size is 16

The analysis unveils the pronounced effect of concurrency levels on speedup, especially for larger problem sizes. For instance, with a problem size of 2048, the speedup significantly rises as the concurrency level escalates from 4 to 64, particularly for block size 4, underscoring the benefits of higher concurrency for larger problem sizes.

Block size also notably influences speedup, especially at higher concurrency levels. For instance, at a problem size of 512 and concurrency level of 64, a higher speedup is observed for block size 4 compared to block size 16, a trend that persists at a problem size of 2048, revealing a nuanced interplay between block size, concurrency level, and speedup.

Furthermore, the data elucidates a positive correlation between problem size and speedup, particularly at higher concurrency levels, suggesting that larger problem sizes better leverage the available parallel resources, thereby amplifying parallelization effectiveness.

The data also unveils the interplay between block sizes and concurrency levels. At higher concurrency levels, a smaller block size tends to yield better speedup, possibly due to the reduced over-

head associated with managing smaller blocks in a highly parallel environment.

Interestingly, at a smaller problem size of $N = 128$, the speedup does not scale well with increasing concurrency, especially for block size 16, likely due to the ineffective utilization of the increased parallel resources attributable to the small problem size.

In conclusion, the block size choice, optimized based on the concurrency level and problem size, could augment performance. Larger problem sizes seem to benefit more from higher concurrency levels, and smaller block sizes appear to yield better speedup at these higher concurrency levels, providing a vital insight for optimizing the performance of parallel algorithms in Blocked-omp code.

3.3.3 L3 Cache Requests Analysis

Table 1 show the L2 Cache Accesses from Basic, BlockedMM B4, and BlockedMM B16 normalized with respect to the L2 Cache Accesses of CBLAS for different problem sizes (128, 512, 2048).

N	CBLAS	Basic	Blocked B4	Blocked B16
128	1	42.43	5.24	0.93
512	1	61.04	5.67	1.15
2048	1	309.99	7.10	1.23

Table 1: Normalized L2 Cache Accesses comparison

The BasicMM significantly exceeds CBLAS in L2 Cache accesses, especially as the problem size increases, extending runtime due to increased cache misses and higher-level memory hierarchy fetching overhead.

Both blocked implementations (B4 and B16) markedly reduce L2 Cache accesses compared to the basic implementation, suggesting better cache utilization which contributes to improved runtime performance by exploiting data locality through block data accessing, thereby reducing cache misses.

Comparatively, Blocked B16 exhibits slightly lower L2 Cache accesses than Blocked B4, especially as problem size increases, hinting that larger block sizes might enhance cache utilization, further reducing cache accesses, and potentially improving runtime performance.

With increasing problem size, L2 Cache accesses for the BasicMM implementation surge dramatically, while BlockedMM implementations maintain relatively low and stable cache accesses, highlighting the scalability advantage of blocked implementations in terms of cache utilization, likely translating to better runtime performance for larger problem sizes.

In summary, the substantial reduction in L2 Cache accesses in blocked implementations underscores the potential runtime benefits of optimized memory hierarchy utilization, especially as problem size scales up.

3.3.4 L3 Cache Requests Analysis

The table below represents the L3 cache requests (`L3_CACHE_REQ`) normalized by the corresponding values of CBLAS for different problem sizes and implementations.

N	CBLAS	Basic	Blocked B4	Blocked B16
128	1	0.71	0.26	0.12
512	1	70.89	13.25	3.17
2048	1	511.05	23.02	5.71

Table 2: Normalized L3 Cache Requests comparison

At problem size $N = 128$, BasicMM implementation shows fewer L3 cache requests compared to CBLAS, with BlockedMM implementations exhibiting even lower requests. As problem size increases, L3 cache requests for BasicMM implementation surge, notably at $N = 512$ and $N = 2048$, while BlockedMM implementations show a milder increase. High L3 cache requests in BasicMM implementation extend runtime due to increased L3 cache or memory access. Conversely, BlockedMM implementations, especially Blocked B16, maintain relatively low L3 cache requests across all problem sizes, indicative of better cache utilization, leading to improved runtime performance. Between the blocked implementations, Blocked B16 consistently shows fewer L3 cache requests than Blocked B4, hinting that a larger block size may more efficiently reduce cache requests, enhancing runtime performance. Overall, the data highlights the significance of algorithmic strategies, like blocking, on memory hierarchy utilization and potential runtime performance. Efficient cache utilization, as observed in blocked implementations and CBLAS, can significantly curtail the number of L3 cache requests, likely contributing to better runtime performance as problem size scales up.

3.4 Instruction Count Analysis

The table below presents the sum of the counters `RETIRED_INSTRUCTIONS` and `RETIRED_SSE_AVX_FLOPS_ALL`, normalized to the respective values from the CBLAS implementation, for different problem sizes and implementations.

N	CBLAS	Basic	Blocked B4	Blocked B16
128	1	3.97	6.10	3.95
512	1	4.04	6.21	4.03
2048	1	4.04	6.21	4.03

Table 3: Normalized Instruction Count comparison

Upon reviewing the table, it's evident that compared to CBLAS, both Basic and Blocked implementations execute more instructions across all problem sizes, potentially extending runtimes. The instruction count in Basic and Blocked B16 remains relatively stable across varying problem sizes, indicating consistent instruction count scalability. Blocked B4 consistently has the highest instruction count, likely due to overhead from managing smaller blocks. The instruction count for Basic and Blocked B16 is quite alike across all problem sizes, hinting that a block size of 16 doesn't notably alter the instruction count compared to the Basic implementation.

The instruction count difference between Blocked B4 and Blocked B16 is notable. Blocked B4's higher count might be due to the overhead of managing smaller blocks, while Blocked B16, with larger block size, maintains a lower count closer to the Basic implementation. This suggests a well-chosen block size can affect both instruction count and potentially the overall runtime performance. Blocked B16's lower instruction leads to better performance, results in fewer cache misses, and improved data locality.

4 Conclusions

4.1 Scaling Characteristics

The relative scalability of the three codes: basic, blocked B4, and blocked B16 reveal interesting patterns. BlockedMM B4 exhibits a notable speedup at higher concurrency levels and larger problem sizes, making it the most scalable among the three. On the other hand, BlockedMM B16 demonstrates a reasonable scalability, but not as effectively as Blocked B4. The BasicMM, however, shows limited scalability, particularly as the problem size increases. The observed limitations to scalability may largely be attributed to the overhead associated with thread management and suboptimal cache utilization. To alleviate these limitations and improve scalability, potential modifications to the codes could include optimizing cache usage or tuning block sizes to better align with the memory system hierarchy.

4.2 Memory System Utilization

Examining the data in Tables 1 and 2 (L2CACHE, L3CACHE), certain observations regarding memory system utilization among the four codes (CBLAS, BasicMM, Blocked B4 serial, blocked B16 serial) can be made. CBLAS stands out for its efficient cache utilization across all problem sizes, setting a benchmark for optimal memory system utilization. Following a similar trend, Blocked B16 demonstrates efficient L2 Cache access which slightly increases with the problem size but remains notably efficient. Conversely, both Blocked B4 and basic-serial exhibit less efficient cache utilization as reflected by the higher L2 and L3 cache access values, potentially impacting their runtime performance adversely. Effective memory system utilization, as exhibited by CBLAS and Blocked B16, undeniably contributes to better runtime performance. The data underscores the potential performance benefits of employing cache-optimized algorithms or adapting block sizes to enhance memory system utilization, thereby aligning with the memory hierarchy more effectively for improved performance.

REFERENCES

[1] NERSC. Perlmutter architecture. https://docs.nersc.gov/systems/perlmutter/architecture/. Accessed: 2023-10-1.