Your Title Goes Here
Assignment #1, CSC 746, Fall 2024

Shun Usami*
SFSU

Abstract

The focus of this study is to explore the effectiveness of various parallelization techniques in enhancing computational efficiency and memory bandwidth utilization for matrix-vector multiplication (VMM). We implemented four versions of VMM: a Basic (serial) implementation as an unoptimized baseline, CBLAS (serial) as a highly optimized baseline, a Vectorized (serial) implementation to leverage SIMD instructions, and an OpenMP (parallel) implementation to explore multi-threading. These implementations were evaluated on the Perlmutter supercomputer to understand the impact of different parallelization strategies. The Vectorized implementation achieved approximately 4x the performance of the Basic version and performed similarly to the CBLAS version, thanks to SIMD instructions. On the other hand, the results demonstrate that while the OpenMP parallel implementation achieved better performance, it fell short of the ideal speedup due to memory bandwidth limitations and data locality challenges.

1 Introduction

The objective of this study is to evaluate the effectiveness of different parallelization techniques for enhancing computational efficiency and memory bandwidth utilization in matrix-vector multiplication (VMM). The techniques explored include instruction-level parallelism, SIMD instructions, vectorization, multi-threading with OpenMP, and considerations for NUMA architecture. By examining these methods, we aim to understand their respective benefits and limitations in the context of high-performance computing.

We implemented four versions of VMM: a basic serial version as an unoptimized baseline, a highly optimized serial version using the CBLAS library, an automatically vectorized serial version leveraging SIMD instructions, and a parallel version utilizing OpenMP for multi-threading. These implementations provide a comprehensive basis for evaluating the impact of different optimizations on performance. All experiments were conducted on the Perlmutter supercomputer at NERSC, using C++ for implementation. Detailed descriptions of these methods are provided in Section 3.2.

The results indicate that the Vectorized implementation achieved four times the performance of the Basic version due to SIMD instructions and performed similarly to CBLAS for larger matrix sizes. However, CBLAS significantly outperformed the Vectorized version for smaller matrices, likely due to techniques like instruction-level parallelism (ILP). The OpenMP implementation, while outperforming CBLAS for larger matrices, had excessive overhead for smaller sizes, making it worse than both CBLAS and even the Vectorized version. Despite outperforming CBLAS for larger sizes, the OpenMP performance was still far from ideal due to memory data layout inefficiencies and limited bandwidth, which acted as bottlenecks.

*email:susami@sfsu.edu

2 Implementation

This section presents four matrix-vector multiplication (VMM) methods implemented to evaluate performance. The first two methods, VMM Basic (Serial) and VMM Vectorized (Serial), share the same source code; however, the VMM Vectorized version is compiled with flags to enable automatic vectorization. The third method, VMM OpenMP (Parallel), introduces parallelization using OpenMP to accelerate computation. Lastly, VMM CBLAS serves as a reference implementation using the highly optimized CBLAS library. Each subsection provides details on the objectives and specific implementations for these methods.

2.1 VMM Basic (Serial)

The objective of the VMM Basic (Serial) implementation is to establish a baseline performance metric for matrix-vector multiplication without any optimizations or parallelization. This implementation is used as a reference point to evaluate the performance gains from vectorization and parallelization in the subsequent methods.

The VMM Basic (Serial) and VMM Vectorized (Serial) implementations both use a naive double-nested loop to compute each element $y[i]$ of the resulting vector y, by adding the dot product of the i-th row of matrix A and the input vector x. The inner loop is carefully structured to only use the loop index variable j, ensuring proper vectorization when compiler flags are applied. The code for this implementation is shown in Listing 1.

```
void my_dgemv(int n, double *A, double *x, double
    *y) {
  for (int i = 0; i < n; i++) {
    double *a = &A[i * n];
    double dot = 0;
    for (int j = 0; j < n; j++) {
      dot += a[j] * x[j];
    }
    y[i] += dot;
  }
}
```

Listing 1: Naive implementation of matrix-vector multiplication using a double-nested loop. The VMM Basic (Serial) and VMM Vectorized (Serial) implementations use the same source code. The inner loop is carefully structured to only use the loop index variable j, ensuring proper vectorization when compiler flags are applied.

2.2 VMM Vectorized (Serial)

The objective of the VMM Vectorized (Serial) implementation is to evaluate the performance improvement achieved through automatic vectorization by the compiler. This implementation uses the same code as shown in Listing 1, but is compiled with specific optimization flags. These flags, *-O3 -DNDEBUG -fomit-frame-pointer -ftree-vectorize -funroll-loops -ffast-math -fopt-info-vec-all=report.txt*, ensure that the code is vectorized and generate a report detailing how the vectorization is applied.

2.3 VMM OpenMP (Parallel)

The objective of the VMM OpenMP (Parallel) implementation is to evaluate the performance improvements achieved through parallelization by distributing the workload across multiple threads using OpenMP compiler directives.

This implementation retains the same loop structure as the VMM Basic (Serial) and VMM Vectorized (Serial) versions. The only modification is the inclusion of the compiler directive *#pragma omp parallel for*, which instructs the compiler to parallelize the subsequent for loop. This ensures that each operation $y[i] = y[i] + A[i] * x$ is executed in parallel for all rows i of matrix A. The source code is compiled with the flags *-fopenmp -march=native -O1* to enable OpenMP parallelization and optimize for the native architecture. The code for this implementation is shown in Listing 2.

```
void my_dgemv(int n, double *A, double *x, double
    *y) {
#pragma omp parallel for
  for (int i = 0; i < n; i++) {
    double *a = &A[i * n];
    double dot = 0;
    for (int j = 0; j < n; j++) {
      dot += a[j] * x[j];
    }
    y[i] += dot;
  }
}
```

Listing 2: OpenMP-parallel implementation for matrix-vector multiplication. This implementation retains the loop structure of the VMM Basic (Serial) and VMM Vectorized (Serial) versions, with the addition of the `#pragma omp parallel for` directive to enable parallel execution across multiple threads.

2.4 VMM CBLAS

The objective of the VMM CBLAS implementation is to establish a baseline performance for highly optimized matrix-vector multiplication without manual parallelization or vectorization. This implementation serves as a reference point to compare with the vectorization and parallelization techniques applied in the other methods.

As shown in Listing 3, the CBLAS implementation wraps a call to the highly optimized CBLAS routine *cblas_dgemv*, which performs matrix vector multiplication.

```
#include <cblas.h>

void my_dgemv(int n, double *A, double *x, double
    *y) {
  double alpha = 1.0, beta = 1.0;
  int lda = n, incx = 1, incy = 1;
  cblas_dgemv(CblasRowMajor, CblasNoTrans, n, n,
    alpha, A, lda, x, incx, beta,
              y, incy);
}
```

Listing 3: CBLAS implementation using the cblas_dgemv routine for optimized matrix vector multiplication

3 Results

This section presents the results of our experiments evaluating various matrix-vector multiplication (VMM) methods. We conducted a series of experiments to measure and compare the computational throughput, memory bandwidth utilization, and speedup of different VMM implementations, including VMM Basic, VMM Vectorized, VMM CBLAS, and VMM OpenMP. The goal was to understand the performance benefits and limitations of various parallelization techniques.

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 equipped with two AMD EPYC 7763 (Milan) processors, each with 64 cores running at a clock rate of 2.45 GHz. Each core supports Simultaneous Multi-threading (SMT), which allows 2 threads per core to run simultaneously when enabled. 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. Each AMD EPYC 7763 (Milan) processor contains 8 memory channels per socket, 2 DIMMs per memory channel, and has 4 NUMA domains per socket (NPS=4) [1].

The system is supported by 512 GB of DDR4 DRAM, providing a memory bandwidth of 204.8 GB/s per CPU. The processor utilizes the AVX2 instruction set for vector processing, and each core offers a peak computational throughput of 39.2 GFLOPS [3].

All experiments were performed 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 for each implementation:

- **VMM Basic (Serial)**: Compiled with `-march=native -O1` for basic optimizations without vectorization or parallelization, ensuring native architecture optimizations.
- **VMM Vectorized (Serial)**: Compiled with `-O3 -DNDEBUG -fomit-frame-pointer -ftree-vectorize -funroll-loops -ffast-math -fopt-info-vec-all=report.txt`. These flags enable vectorization, loop unrolling, and fast math optimizations to achieve higher performance through vectorized operations.
- **VMM OpenMP (Parallel)**: Compiled with `-fopenmp -march=native -O1` to enable OpenMP parallelization in multiple threads. The `-march=native` flag ensures optimization for the native architecture, while `-O1` is used for moderate optimization, avoiding overly aggressive optimizations that might conflict with parallelization.
- **VMM CBLAS**: Compiled with `-march=native -O0` to ensure native architecture optimization without further compiler optimizations, relying on the performance of the CBLAS library itself.

The following OpenMP environmental variables were used when running VMM OpenMP, with `OMP_NUM_THREADS` set according to the number of threads:

- `OMP_PLACES=threads`: This variable specifies how OpenMP should assign threads to processing units. Setting it to `threads` ensures that each OpenMP thread is assigned to a separate hardware thread.
- `OMP_PROC_BIND=spread`: This variable controls how threads are bound to processors. Setting it to `spread` aims to distribute threads as evenly as possible across the available CPUs.
- `OMP_SCHEDULE=static` (default): This variable controls how iterations of a parallel loop are divided among threads. The `static` schedule assigns chunks of iterations to threads in a predetermined manner.

3.2 Methodology

We evaluated the performance of various matrix-vector multiplication (VMM) methods using matrix sizes of 1024×1024, 2048×2048, 4096×4096, 8192×8192, and 16384×16384. For the VMM OpenMP (Parallel) implementation, we tested with thread counts of 1, 4, 16, and 64.

To avoid skewed results caused by the known issue of slow performance during the first run of VMM CBLAS (due to dynamic library loading), we first performed a warm-up run using the 1024×1024 matrix size. This initial run was discarded to ensure that subsequent performance measurements were representative.

Performance was measured by calculating the elapsed time using instrumentation placed around the core matrix-vector multiplication code, as shown in Listing 4. Based on the elapsed time, we derived three key performance metrics:

- Computational Throughput
- Memory Bandwidth Utilization
- Speedup at varying levels of concurrency for the VMM OpenMP (Parallel) implementation.

```
setup(n, A, X, Y);
start_time = get_high_resolution_clock_now();
my_dgemv(n, A, X, Y);
end_time = get_high_resolution_clock_now();
elapsed_time = end_time - start_time;
```

Listing 4: Instrumentation code for measuring the elapsed time of matrix-vector multiplication

3.2.1 Computational Throughput

$$MFLOP/s = \frac{ops}{runtime \times 10^6}$$

$$GFLOP/s = \frac{ops}{runtime \times 10^9}$$

$$ops = 2N^2$$

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

3.2.2 Memory Bandwidth Utilization

$$\text{Memory Bandwidth Utilization (\%)} = \frac{bytes}{runtime \times capacity}$$

$$bytes = \frac{(2N + 2N^2) \times 8}{10^9}$$

$$capacity = 204.8 \text{ GB/s} \times 2$$

Here, *bytes* represents the number of memory bytes accessed to multiply an $N \times N$ matrix and an N-size vector, divided by one billion to convert to gigabytes. The *capacity* is the theoretical peak memory bandwidth in gigabytes per second (see Sec. 3.1).[2]

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

[2] For the matrix-vector multiplication, memory accesses are composed of accessing each element of Y twice (read and write), accessing all elements of A once (read-only), and accessing all elements of X once (read-only), leading to the formula $2N + 2N^2$.

3.2.3 Speedup

We define speedup as the ratio of a parallel program's speed to a sequential program's speed. For a problem size n and p parallel ranks (where rank refers to the number of parallel processing units used):

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

Here, $T^*(n)$ is the time taken by the best serial algorithm on a problem of size n, which in our case is the VMM Vectorized implementation, and $T(n,p)$ is the time taken to run the program on a problem of size n using p parallel ranks.

3.3 Comparison of VMM CBLAS with VMM Basic and VMM Vectorized

In this experiment, the objective is to establish a baseline for evaluating the computational throughput (MFLOP/s) of three different VMM implementations: VMM Basic (without vectorization or parallelization), VMM Vectorized (automatically vectorized by the compiler), and VMM CBLAS (a highly optimized serial implementation). All three implementations are serial, providing a foundation for comparison against the OpenMP parallel implementation that will be evaluated later. The configuration details for this experiment are provided in Sec. 3.2.

3.3.1 Performance of VMM Basic and VMM Vectorized

As shown in Figure 1, the VMM Vectorized implementation consistently outperforms VMM Basic across all problem sizes. The MFLOP/s of the VMM Basic implementation is relatively stable, averaging around 2300 MFLOP/s. For the VMM Vectorized implementation, the performance peaks at $N = 1K$[3] with 9986.4 MFLOP/s and decreases slightly to 9118.1 MFLOP/s at $N = 2K$. For larger problem sizes ($N = 4K$ to $N = 16K$), the performance stabilizes around 8000 MFLOP/s.

The superior performance of the VMM Vectorized implementation for $N = 1K$ and $N = 2K$ can be attributed to the efficient utilization of the L1 cache. Specifically, as shown in Table 1, the combined memory footprint of $A[i]$ and vector x (comprising $2N$ double elements) fits within the L1 cache for these problem sizes. This ensures that vector x is not evicted from the L1 cache, as both $A[i]$ and x are accessed in each iteration of the outer loop.

For problem sizes ranging from $N = 4K$ to $N = 16K$, the performance of the VMM Vectorized implementation remains relatively stable because, while the vectors no longer fit in the L1 cache, they do fit in the L2 cache across all these problem sizes, as indicated in Table 1.

Vector Size (K=1024)	2 Vector (Bytes)	L1	L2
$1K$	16 KiB	Yes	Yes
$2K$	32 KiB	Yes	Yes
$4K$	64 KiB	No	Yes
$8K$	128 KiB	No	Yes
$16K$	256 KiB	No	Yes

Table 1: Memory footprint for different vector sizes and L1/L2 cache fit. The table shows the memory required for two vectors of each size and whether they fit within the L1 cache (32 KiB) and L2 cache (256 KiB). Vector sizes up to $2K$ fit in L1, while $4K$ does not.

[3] In this paper, K notation for vector and matrix sizes represents 1024 elements rather than the conventional 1000.

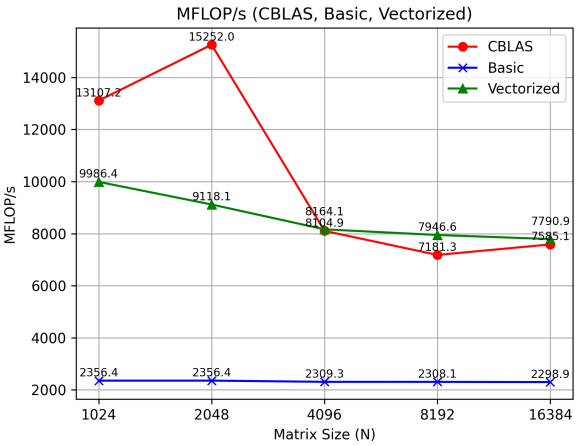

Figure 1: MFLOP/s Comparison of VMM CBLAS with VMM Basic and VMM Vectorized across different matrix sizes. The figure illustrates the computational throughput (MFLOP/s) for each implementation across varying matrix sizes, highlighting the relative performance differences between VMM Basic, VMM Vectorized, and VMM CBLAS. VMM CBLAS demonstrates superior performance for smaller matrix sizes due to efficient cache utilization, while VMM Vectorized shows competitive performance at larger sizes by leveraging automatic vectorization. VMM Basic remains stable but underperforms compared to the other two implementations due to the lack of SIMD utilization and other optimizations.

3.3.2 Comparison of VMM CBLAS and VMM Vectorized

The VMM CBLAS implementation consistently outperforms VMM Basic but shows varying results compared to VMM Vectorized depending on the problem size. For smaller problem sizes ($N = 1K$ and $N = 2K$), VMM CBLAS significantly outperforms VMM Vectorized by approximately 30% and 70%, respectively. However, for larger problem sizes ($N = 4K$, $N = 8K$, and $N = 16K$), VMM Vectorized slightly outperforms VMM CBLAS.

The AMD EPYC 7763 processor, which utilizes the AVX2 instruction set (see Sec. 3.1), supports 256-bit SIMD instructions. This enables four arithmetic operations on double-precision (64-bit) data in a single cycle, explaining the approximately 400% performance improvement observed for both VMM CBLAS and VMM Vectorized compared to VMM Basic for larger matrix sizes.

However, the observed performance differences indicate that VMM CBLAS is significantly better optimized for cache utilization under these conditions. For smaller values of N, the performance of CBLAS is 6 to 8 times better than that of VMM Basic, which is a surprising result that cannot be fully explained by SIMD utilization alone. As shown in Table 2, for matrix sizes $N = 1K$ and $N = 2K$, the matrices fit within the L3 cache, suggesting more efficient cache usage by CBLAS for these sizes. Given that each element of matrix A is accessed only once during the matrix-vector multiplication, spatial locality does not appear to be a limiting factor. Additionally, since both x and $A[i]$ fit in the L1 cache for these sizes, temporal locality also does not seem to be a limiting factor. It is likely that CBLAS achieves superior performance through advanced optimizations such as efficient instruction scheduling, prefetching, and effective exploitation of instruction-level parallelism (ILP), which allows multiple instructions to be executed simultaneously by overlapping their execution stages. CBLAS likely leverages ILP by replicating internal components to process multiple instructions concurrently within each pipeline stage.

Matrix Size (K=1024)	Matrix size (Bytes)	L3
$1K \times 1K$	8 MiB	Yes
$2K \times 2K$	32 MiB	Yes
$4K \times 4K$	128 MiB	No
$8K \times 8K$	512 MiB	No
$16K \times 16K$	2 GiB	No

Table 2: Memory footprint for different matrix sizes and L3 cache fit. The table shows the memory required for matrices of each size and whether they fit within the L3 cache (32 MiB). Matrix sizes up to $2K \times 2K$ fit in L3, while $4K \times 4K$ does not.

3.4 Evaluation of VMM OpenMP (Parallel)

Figure 2 shows that the speedup for omp-1 is approximately 0.9, which is slightly worse than the best serial implementation across different matrix sizes. This is likely due to some overhead introduced by the OpenMP threading code, even when only one thread is used.

For larger thread counts, the speedup is only between 0.3 to 0.5 for smaller matrix sizes (N=1K, 2K). This can be attributed to the overhead of creating and joining threads, which becomes relatively significant compared to the gain in parallel computation for smaller problems.

For larger matrix sizes (N=4K, 8K, 16K), the speedup achieved by omp-4, omp-16, and omp-64 is relatively stable, ranging between 1.2 to 1.3. Among these, omp-16 shows slightly better performance for larger matrix sizes. This indicates that although some speedup is achieved, it falls far short of the ideal values due to other limiting factors.

We expected to observe a speedup close to 4x for omp-4, 16x for omp-16, and 64x for omp-64, especially for larger matrix sizes, where the overhead of the serial portion and the introduction of multithreading should be minimal according to Amdahl's law [2][4]. However, the observed results fall significantly short of these expectations, indicating unexpected and undesirable bottlenecks in the parallel execution.

The unexpectedly poor performance is due to a bottleneck not in the arithmetic computations but in other components, such as memory bandwidth. We did not allocate the memory across the NUMA nodes when setting up the matrices and vectors, which means the matrix/vector data likely reside entirely within one NUMA node. Each NUMA node has 64 GB of memory, and the total system memory is 512 GB, which is sufficient to hold all the elements for our problem sizes. As a result, even though the program is multithreaded, the available memory bandwidth is constrained, limiting overall performance. This can explain why omp-16 performs slightly better than omp-64; since the bottleneck is not resolvable by adding more threads, increasing the number of threads only introduces additional overhead without yielding further performance gains.

3.5 Comparison of VMM OpenMP with VMM CBLAS

From Figure 3, we observed distinct performance differences between the two implementations based on the matrix sizes. For smaller matrix sizes (N=1K and 2K), CBLAS significantly outperformed the OpenMP implementation with 16 threads (omp-16), achieving performance gains of 2.5x to 3.0x. This indicates that CBLAS is better optimized for smaller matrices, where efficient cache use boosts performance, and multi-threading overhead is more significant compared to the performance gain.

[4]Amdahl's law describes the theoretical maximum speedup that can be achieved in a parallel system based on the portion of a task that must be executed serially. The theoretical speedup limit for a problem size n and p parallel processors is given by: $S(n,p) = \frac{1}{f+\frac{1-f}{p}}$, where f is the portion of the program that must remain serial, and $(1-f)$ is the parallelizable portion. For smaller values of f, the closer the speedup is to the ideal scaling with p.

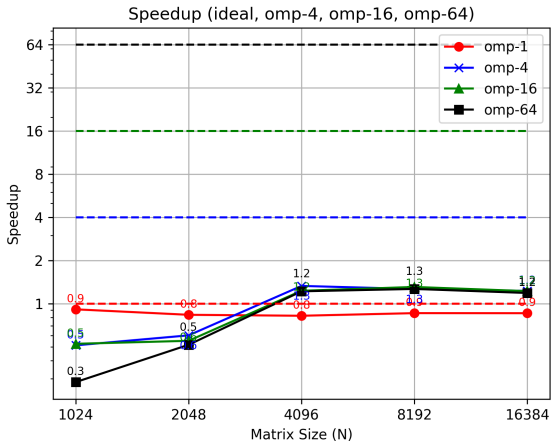

Figure 2: Speedup comparison of OpenMP parallel VMM across different matrix sizes and thread counts. The figure shows the speedup achieved with different numbers of threads (1, 4, 16, 64) across various matrix sizes. The figure shows that omp-1 performs slightly worse than the best serial implementation due to threading overhead. For smaller matrix sizes, higher thread counts achieve limited speedup due to significant overhead in thread management. For larger matrix sizes, speedup stabilizes around 1.2 to 1.3, indicating bottlenecks in memory bandwidth.

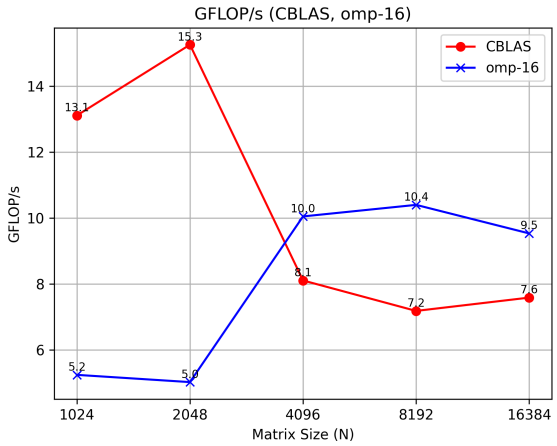

Figure 3: Comparison of MFLOP/s between VMM OpenMP (omp-16) and VMM CBLAS across different matrix sizes. The figure shows that for smaller matrix sizes (N=1K, 2K), CBLAS significantly outperforms omp-16, likely due to better cache optimization and lower threading overhead. For larger matrix sizes (N=4K, 8K, 16K), omp-16 outperforms CBLAS, benefiting from parallelism when the data no longer fits in the cache.

For larger matrix sizes (N=4K, 8K, and 16K), omp-16 outperformed CBLAS by 1.2x to 1.4x. As matrix sizes grow beyond the L3 cache capacity, the parallelism advantage in omp-16 becomes more pronounced, surpassing the serial CBLAS implementation. As shown in Table 2 and Table 1, when the matrix fits within the L3 cache and both vectors fit in the L1 cache, CBLAS outperforms omp-16, implying more efficient cache use by CBLAS.

Another important factor is the NUMA architecture of the system. Each CPU node has 8 NUMA nodes, with 2 memory channels per NUMA node (see Sec. 3.1). Multi-threading can utilize both memory channels in the NUMA node where the data resides, whereas a single-threaded program, like CBLAS, can only utilize a single memory channel. However, the overhead introduced by distributing the threads across NUMA nodes can be significant, approximately doubling the memory access time. This overhead is partly countered by the increased memory bandwidth available to multiple threads[5].

3.6 Overall Findings and Discussion

The analysis of memory bandwidth utilization across different configurations reveals several insights. For the smallest matrix size (N=1K), CBLAS demonstrates the highest utilization, showing that it efficiently uses available memory resources when the problem fits well in the cache. On the other hand, the basic implementation for the largest matrix size (N=16K) shows the worst memory bandwidth utilization, indicating poor scalability and limited optimization in handling larger data.

Interestingly, the expectation that higher thread counts would yield better bandwidth utilization was not entirely met. For smaller matrix sizes, a single-threaded implementation (omp-1) performed best, likely due to lower overhead and more efficient use of cache.

[5]In the same NUMA node, the memory distance is 10, while the distance for other NUMA nodes in the same CPU is 12, and the distance for NUMA nodes in the other CPU is 32 [4]. Therefore, the expected average memory distance when data resides in a single node and threads are scattered is (10 + 12*3 + 32*4)/8 = 21.75, whereas for a single thread in a single memory node, the distance is 10.

For larger matrix sizes (N=4K, 8K, 16K), the utilization for omp-4, omp-16, and omp-64 was similar, with omp-4 showing slightly better results. This can be explained by the data being confined to a single NUMA node, preventing full utilization of all 8 NUMA nodes. To maximize memory bandwidth, data needs to be distributed across all NUMA nodes, allowing concurrent access from each local NUMA node.

The calculation of memory accesses includes redundant access to vector x, which occurs n times. If two rows of the matrix fit into the cache, effective utilization of the cache can reduce the memory accesses by approximately half. The high bandwidth utilization observed for CBLAS, reaching around 100 GB/s, is likely due to this optimization, effectively reducing the number of memory accesses. In practice, this suggests that CBLAS utilizes about 50 GB/s of memory bandwidth, which aligns with the theoretical limit of a single NUMA node's bandwidth. Given that the CPU has a peak memory bandwidth of 204.8 GB/s with 4 NUMA nodes, a single NUMA node can effectively provide about a quarter of this bandwidth, or roughly 51 GB/s, matching our observations.

In terms of scalability, we initially expected a nearly ideal speedup for larger matrix sizes, with omp-4 achieving 4x, omp-16 achieving 16x, and omp-64 achieving 64x speedup. However, the actual results showed significant bottlenecks, primarily due to limited memory bandwidth and inefficient memory allocation across NUMA nodes. The lack of memory distribution across NUMA nodes meant that the available bandwidth was restricted, leading to lower-than-expected speedups. Additionally, adding more threads, as in the case of omp-64, introduced additional overhead without resolving the bottleneck, which explains why omp-16 performed slightly better for larger problem sizes.

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

Method	Matrix Size N (GB/s)				
	1K	2K	4K	8K	16K
CBLAS	104.96	122.08	64.86	57.46	60.68
Basic	18.87	18.86	18.48	18.47	18.39
Vectorized	79.97	72.98	65.33	63.58	62.33
omp-1	73.02	61.04	53.81	54.62	53.50
omp-4	40.96	43.88	86.89	80.50	76.55
omp-16	41.98	40.20	80.39	83.18	76.25
omp-64	22.69	37.72	79.67	80.56	74.00

Table 3: Memory bandwidth in GB/s for varying matrix sizes. The matrix is $N \times N$. The table shows how different implementations use available memory bandwidth across varying matrix sizes. CBLAS exhibits high utilization for smaller matrices, while OpenMP implementations show improved utilization for larger matrices due to increased concurrency.

Method	Matrix Size N (% Peak)				
	1K	2K	4K	8K	16K
CBLAS	25.6%	29.8%	15.9%	14.0%	14.8%
Basic	4.6%	4.6%	4.5%	4.5%	4.5%
Vectorized	19.5%	17.8%	16.0%	15.5%	15.2%
omp-1	17.9%	14.9%	13.2%	13.4%	13.1%
omp-4	10.0%	10.7%	21.2%	19.7%	18.7%
omp-16	10.2%	9.8%	19.7%	20.3%	18.6%
omp-64	5.6%	9.2%	19.5%	19.7%	18.1%

Table 4: Memory bandwidth utilization (%) for varying matrix sizes. The matrix is $N \times N$. The table illustrates the percentage of peak memory bandwidth utilization for each implementation. CBLAS achieves high utilization for small matrices, while OpenMP implementations make better use of available bandwidth as matrix size increases, albeit with diminishing returns for higher thread counts.

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

[4] S. Usami. NUMA Node Information of Perlmutter CPU Node. `numactl` command output on Perlmutter CPU node, 2024. Retrieved on October 7, 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.