Comparing LIKWID Hardware Metrics to Performance across DGEMM Implementations

Ben Klein
Computer Science Department
San Francisco State University
San Francisco, CA, USA
Email: bklein@sfsu.edu

***Abstract*—This study investigates how metrics gathered from the performance monitoring tool, LIKWID- such as cache activity and retired instruction counts- explain the runtime and speedup of DGEMM (Double Precision General Matrix Multiply) implementations. OpenMP is used to compare performance with different thread counts. Results show that there are far more L3 accesses in naive implementations in comparison to optimized implementations like CBLAS, in some cases, reaching nearly 600 times the number of accesses. Among the blocked and copy-optimized versions, the larger block size shows a far shorter runtime, fewer cache accesses, and fewer instructions. These findings show that choosing the right block size can be critical for performance in parallel codes.**

I. Introduction

This study explores how hardware performance metrics relate to runtime and speedup in parallel matrix multiplication. Using the LIKWID [1] performance monitoring tool, the analysis compares L2 and L3 cache accesses and retired instruction counts with different DGEMM (Double-Precision General Matrix Multiply) implementations to show how these metrics correlate with performance.

The evaluation covers three DGEMM implementations: a basic OpenMP parallelized version, a block and copy optimized parallel version, and the Basic Linear Algebra Subprograms BLAS [2] DGEMM, ran serially. Each implementation is evaluated on multiple thread counts to see how optimizations and parallelization affect hardware usage and runtime performance. The implementations are described in Section II.

Results show that naive implementations require far more L3 cache accesses - up to 600× more than CBLAS - showing inefficient memory management. In contrast, the block and copy versions have shorter runtime through improved memory access patterns, even while executing more total instructions. Among the blocked versions, the larger block size has far better performance: showing lower runtime, fewer cache accesses, and fewer instructions. These findings show that choosing the correct block size can be critical for parallel performance.

II. Implementation

Double-Precision, General Matrix Multiplication (DGEMM) computes $C := C + A \times B$ for square matrices A, B, C. The algorithm performs $2n^3$ floating-point operations (FLOPs), where a FLOP is one floating-point addition or multiplication. In this study, three implementations are compared: a basic OpenMP [3] parallel version, a block and copy-optimized version, and a reference serial implementation using the Basic Linear Algebra Subprograms (CBLAS) [2] Library.

A. Basic DGEMM with OpenMP Parallelization

This implementation performs straight forward, three loop DGEMM. The OpenMP parallelism starts on the outer loop over matrix rows using `#pragma omp parallel` and `#pragma omp for`, so each thread operates on it's own row in C. The implementation details are listed in Listing 1.

```
basic_DGEMM(n, A, B, C):
#pragma omp parallel
#pragma omp for
    for row in range(n):
      for col in range(n):
        sum = 0
# inner product
        for inner in range(n):
          sum += A[row, inner] * B[inner, col]
        C[row, col] += sum
```

Listing 1. Basic DGEMM (C += A * B) Pseudocode with OpenMP.

B. Block and Copy Optimized DGEMM with OpenMP Parallelization

The Blocked Matrix Multiplication with Copy Optimizations (BMMCO) version breaks the matrices into $b \times b$ tiles, where b is the block size, and gives each thread it's own block to work on. The `#pragma omp parallel` and `#pragma omp for collapse(2)` lines force the threads to work independently and concurrently on blocks of C. For each output tile, the thread copies data into thread private buffers, then performs a serial `basic_DGEMM` on those blocks. The block sizes used for this algorithm are $\{4, 16\}$.

```
blocked_DGEMM(n, b, A, B, C):
#pragma omp parallel
    Ablock = zeros(b, b)    # thread-local
    Bblock = zeros(b, b)    # thread-local
    Cblock = zeros(b, b)    # thread-local

#pragma omp for collapse(2)
    for row in range(0, n, b):
        for col in range(0, n, b):
#copy the input values into blocks
            copy_tile_vals(C, Cblock)

            for depth in range(0, n, b):
                copy_tile_vals(A, Ablock)
                copy_tile_vals(B, Bblock)

#basic DGEMM, but serial concurrency
                basic_DGEMM(b, Ablock, Bblock,
     Cblock)

#copy block values out to input matrix
            copy_tile_vals(Cblock, C)
```

Listing 2. Blocked DGEMM (C += A * B) with copy optimizations. OpenMP threads operate over entire blocks private to each thread.

C. CBLAS DGEMM with OpenMP Serial Concurrency

CBLAS is the reference implementation for this study. BLAS [4] is broken into three sets of routines called "levels", which correspond to the complexity of the algorithm. At the highest level, level three, is GEMM. The BLAS GEMM algorithm decomposes the input matrices into blocks, and unlike the previous implementation, can be further broken into smaller sub blocks. BLAS is designed to work with modern cache sizes, and in the memory hierarchy utilization section in the evaluation of this study, the memory access patterns are compared.

III. Evaluation

This section presents the performance evaluation of the basic, block and copy, and CBLAS DGEMM implementations. The goal of these experiments is to explore how runtime, speedup, and hardware performance metrics correlate across implementations and numbers of threads. The evaluation includes speedup scaling analysis for both the basic and block and copy versions using OpenMP, followed by a comparison of LIKWID hardware performance metrics using the CBLAS implementation as a baseline.

A. Computational Platform and Software Environment

The performance studies were conducted on a CPU node of the Perlmutter system at NERSC. [5] Each node has two AMD EPYC 7763 processors running at 2.45 GHz. Each core has a 32 KB L1 cache (with a 64-byte cache line size) and a 512 KB L2 cache, while all eight cores in a CCD share a 32 MB L3 cache. The node has 512 GB of memory, with a measured peak memory bandwidth of 204.8 GB/s.

The software environment consisted of the GNU C++14 compiler (SUSE Linux) version 7.5.0 on SUSE Linux Enterprise Server 15 SP5. The build system used `cmake` version 3.20.4 and `make` version 4.2.1. The OpenMP code used static scheduling.

B. Test Methodology

One performance metric used in this study is runtime, measured using the `std::chrono::high_resolution_clock` timer from the C++ standard library.

The other metric used is speedup, defined as:

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

where $T^*(n)$ is the baseline serial runtime, $T(n,p)$ is the runtime using p threads.

In order to compare memory access patterns, the performance monitoring and benchmarking tool, LIKWID [1] is used. In the following sections, LIKWID is used to gather data on L2 cache accesses, L3 cache accesses, and retired instruction counts. Retired instruction counts refer to the number of completed CPU instructions and can be used a metric for determining unnecessary computation. In order to contextualize the performance numbers, CBLAS is used for normalization, by dividing each implementation's values by the corresponding CBLAS values.

The block sizes used are $\{4, 16\}$, the parallel threads measured are $\{1, 4, 16, 64\}$, and the problem sizes are $\{128, 512, 2048\}$.

C. Scaling Study for Basic MM with OpenMP parallelization

Figure 1 shows the speedup compared to problem size for different levels of parallelization for the basic DGEMM implementation. Once the problem size reaches 512, performance plateaus, reaching nearly perfect speedup: about 4x for 4 threads, 15.5x for 16 threads, and 55 for 64 threads. At 4 threads and problem size 128, the speedup is at 0.27, showing a speed decrease. At a problem size of 128, the overhead of thread management overshadows any performance gains that could be made with parallelism. This scaling study shows that even with a basic implementation, parallelization is highly efficient for matrix multiplication tasks.

D. Evaluation of BMMCO with OpenMP parallelization

The speedup comparison for the block and copy optimized version can be found in Figure 2. To show the differences in runtime between block sizes, all speedup calculations use the baseline runtime for a thread count of 1 and a block count of 16.

For the block and copy version, the speedup is more gradual as problem sizes increase. Around 2048 problem size, the 16 size block version is near peak parallel efficiency, where the speedup nears the thread count. In contrast, however, for the 4 size blocks, the speedup grows far slower, ending around 25 at 2048 problem size. This shows a large payoff in efficiency for parallel code when using properly sized blocks. In the following sections, LIKWID performance metrics show more insight into why the increased block size is running so much faster.

E. Comparison of CBLAS, BasicMM-omp, and BMMCO-omp

1) L2 Memory Hierarchy Utilization: The L2 cache access comparison can be found in Table I. For L2 cache accesses, the basic implementation is highly inefficient compared to CBLAS. At it's worst performance, it makes nearly 125 times the number of L2 cache accesses, meaning the memory is not being reused and is instead being reloaded multiple times. The blocked versions are working as intended, with the block size 4 version making about 1/10 the number of L2 accesses of the basic version. Even better, the 16 size block version is nearly on par with the CBLAS implementation.

2) L3 Memory Hierarchy Utilization: The L3 cache access comparison can be found in Table II.

This table shows a dramatic difference in L3 accesses between basic and BLAS, especially as problem sizes increase. This points at memory being loaded and unloaded many times into a larger, slower cache, which contributes to the poor performance results for the basic implementation.

N	CBLAS	Basic	Blocked b=4	Blocked b=16
128	1	47.970	4.848	1.099
512	1	43.641	4.332	1.395
2048	1	124.266	7.695	1.587

TABLE I
L2 CACHE ACCESSES VS. CBLAS BASELINE (SERIAL). EACH NUMBER OF ACCESSES IS DIVIDED BY THE CBLAS L2 ACCESSES TO OBTAIN THE NORMALIZED COMPARISON VALUE.

N	CBLAS	Basic	Blocked b=4	Blocked b=16
128	1	0.734	0.802	0.824
512	1	282.856	16.159	3.149
2048	1	598.657	21.581	4.222

TABLE II
L3 CACHE ACCESSES VS. CBLAS BASELINE (SERIAL). EACH NUMBER OF ACCESSES IS DIVIDED BY THE CBLAS L3 ACCESSES TO OBTAIN THE NORMALIZED COMPARISON VALUE.

3) Instruction Count Analysis: The instruction count comparison can be found in Table III. Interestingly, despite runtime being longer in the basic version, both blocked versions use more instructions. This hints at the bottleneck for the basic version is in poor memory access patterns, rather than inefficient instruction counts. As block sizes increase from 4 to 16, there is a reduction in instruction counts by about half. This hints at the instruction count increases being caused by unnecessarily copying data into the caches.

N	CBLAS	Basic	Blocked b=4	Blocked b=16
128	1	9.608	25.672	12.065
512	1	10.570	28.565	13.449
2048	1	10.642	28.841	13.587

TABLE III
RETIRED INSTRUCTION COUNTS VS. CBLAS BASELINE (SERIAL). EACH NUMBER OF ACCESSES IS DIVIDED BY THE CBLAS RETIRED INSTRUCTION COUNTS TO OBTAIN THE NORMALIZED COMPARISON VALUE.

F. Overall Findings and Discussion

This study shows more nuance into performance characteristics of codes beyond just runtime analysis.

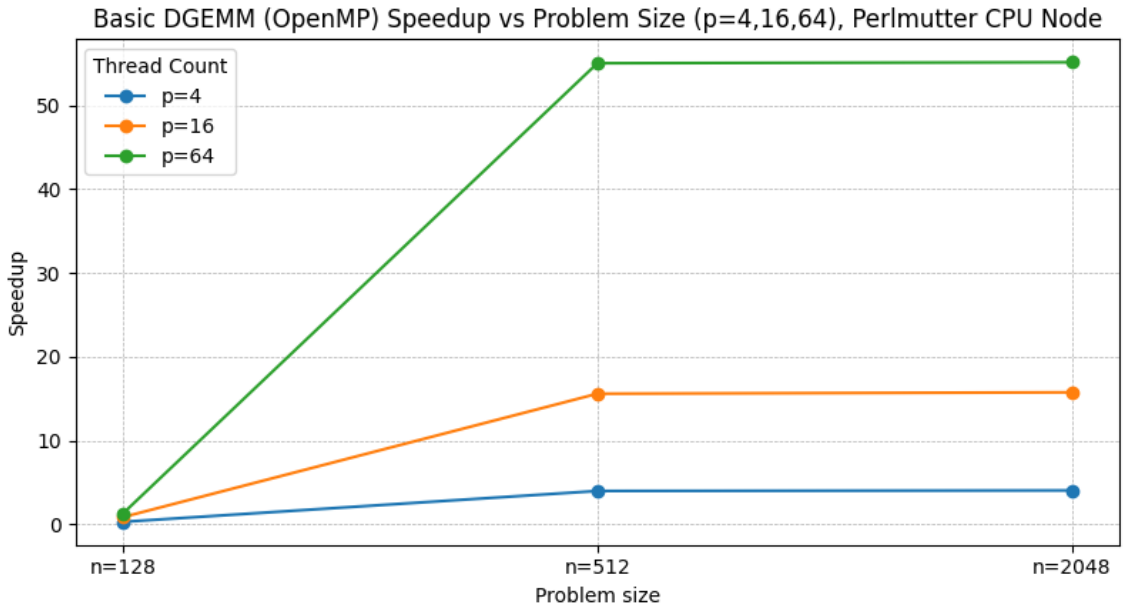

Fig. 1. Comparison of actual speedup of basic parallel DGEMM with increasing problem sizes over 4, 16, and 64 thread counts.

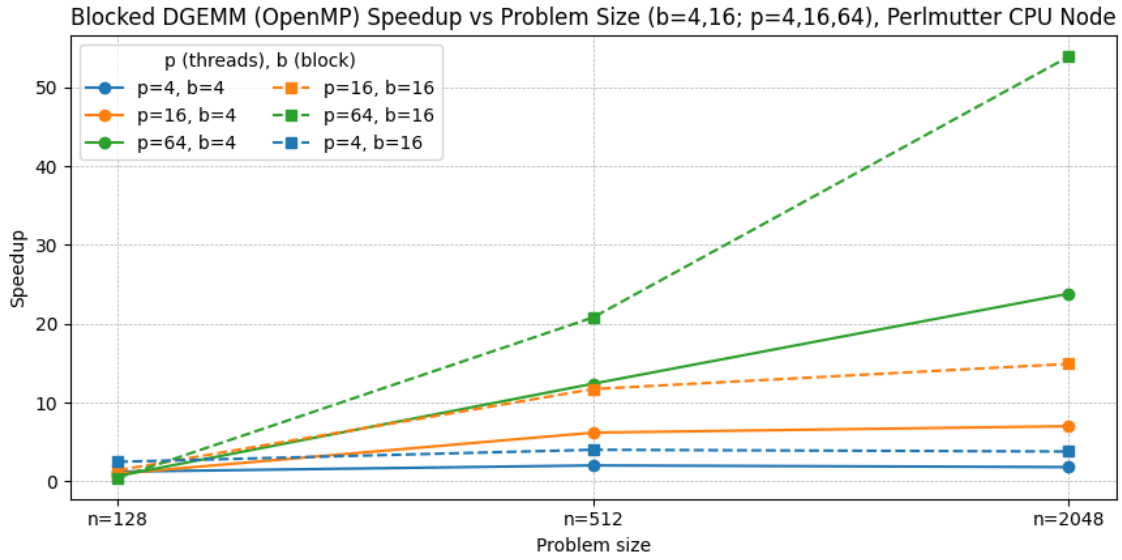

Fig. 2. Comparison of actual speedup of the block and copy optimized parallel DGEMM with increasing problem sizes over 4, 16, and 64 thread counts and 4 and 16 block sizes. For all speedup calculations, p=1, b=16 is used as a common serial baseline so that the differences between block sizes can be clearly seen.

The biggest surprise in the results is just how big the difference between CBLAS and basic DGEMM cache accesses are. At it's worst performance, on problem size 2048, the basic implementation had 600 times the number of L3 accesses. The results imply that good usage of the memory hierarchy means avoiding L3 accesses, thereby improving runtime.

Overall, all codes scale well with parallelization on large problem sizes, though the block and copy versions speedup scales slower as problem sizes increase. The benefits of a larger block size can be seen in the lower number of L2 and L3 cache accesses and fewer instruction count, and this is reflected in speedup, as seen in Figure 2.

The block size is a major factor in performance in parallel code. A well chosen block size can reduce the number of L2 and L3 accesses dramatically. To achieve even better performance in future studies, the block size is a great parameter to look at optimizing for improvement.

ACKNOWLEDGEMENT

OpenAI's ChatGPT 5 was used for assistance with graphing and formatting this document.

REFERENCES

[1] RRZE-HPC, "Likwid: Performance monitoring and benchmarking suite," https://github.com/RRZE-HPC/likwid, 2025, accessed: February 28, 2026.

[2] "BLAS (Basic Linear Algebra Subprograms)." [Online]. Available: https://www.netlib.org/blas/

[3] "Home - OpenMP." [Online]. Available: https://www.openmp.org/

[4] "Basic Linear Algebra Subprograms," Sep. 2025, page Version ID: 1314158568. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Basic_Linear_Algebra_Subprograms&oldid=1314158568

[5] "Architecture - NERSC Documentation." [Online]. Available: https://docs.nersc.gov/systems/perlmutter/architecture/