Runtime Efficiency Gains in Matrix Multiplication with Blocking and Copy Optimizations

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

***Abstract*—This study evaluates the performance of different implementations of the double-precision general matrix multiply (DGEMM) on the Perlmutter CPU nodes at NERSC. A naive three-loop implementation, a blocked matrix multiplication with copy optimization (BMMCO), and the CBLAS reference implementation were tested across problem sizes ranging from 64 to 2048. Performance was measured in MFLOP/s, derived from runtime and the assumption of $2n^3$ operations. Results show that the naive implementation shows poor cache line usage, with performance dropping to 123 MFLOP/s at $N = 2048$. BMMCO improves performance to 4300-4600 MFLOP/s when block sizes of 16 or 32 are used, showing more efficient cache reuse and reduced average memory access time. The CBLAS reference implementation reaches over 50,000 MFLOP/s, demonstrating the impact of additional optimizations. These findings highlight the role of algorithms with efficient cache usage in achieving high performance matrix multiplication.**

I. Introduction

The problem studied in this work is the performance of matrix multiplication on modern high-end CPU hardware. Matrix multiplication is a core operation in many scientific and engineering applications, but its efficiency depends on how data is accessed and reused in the memory caches. Naive implementations often underperform because they do not utilize the small, fast caches effectively.

This study evaluates three implementations of the DGEMM operation: a simple three-loop algorithm (basic MM), a blocked version with copy optimization (BMMCO), and the CBLAS matrix multiplication implementation. The basic MM algorithm serves as a baseline that shows the limitations of poor cache use. BMMCO uses a strategy to break the full problem space into smaller blocks to investigate how different block sizes affect performance. CBLAS is used as a highly optimized reference point for comparison.

The results show that basic MM quickly loses performance as problem size increases, ending with only 123 MFLOP/s for the largest case. BMMCO performance generally improves with problem size, achieving 4300-4600 MFLOP/s with optimal block sizes. CBLAS performs significantly better than either implementation, hinting that the implementation takes advantage of many different strategies to optimize it's performance. These findings demonstrate that reducing average memory access time through blocking is necessary, but achieving peak performance requires more than simply using blocking and copying strategies.

II. Implementation

The Double General Matrix Multiply (DGEMM) operation computes the product of two $n \times n$ matrices. This general form is defined by

$$C \leftarrow \alpha AB + \beta C, \tag{1}$$

where α and β represent an optional scalar value. For the purposes of this study, α and β are set to 1. This simplified form is defined as

$$C = AB, \qquad C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}. \tag{2}$$

This section explains the DGEMM implementations used in this paper: the basic three-loop algorithm, the blocked matrix multiplication with copy optimization (BMMCO), and the Basic Linear Algebra Subprograms (CBLAS) reference implementation. The following 2 sections will detail the design of the three-loop algorithm and the blocked matrix multiplication algorithm, but the CBLAS implementation is out of the scope of this analysis and is left as a black box for this study.

A. Basic Matrix Multiplication

This basic implementation directly performs matrix multiplication using three nested loops. There are no notable optimizations in this straightforward approach as shown in Listing 1. Each element $C_{i,j}$ is computed with the dot product of row i of A with column j of B, then adding the result to $C_{i,j}$.

```
void square_dgemm(int n, const double* A,
    const double* B, double* C)
{
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            double sum = 0.0;
            for (int k = 0; k < n; k++) {
                sum += A[i*n + k] * B[k*n + j
    ];  // row of A, column of B
            }
            C[i*n + j] += sum;
        }
    }
}
```

Listing 1. Basic matrix multiply (three-loop) computing C += $A \times B$ in row-major, with 64 bit floats.

B. Blocked Matrix Multiplication with Copy Optimization (BMMCO)

The BMMCO implementation improves memory access by operating on sub-blocks of the matrices. Blocks of A, B, and C are copied into buffers, operated on, and then written back. This reduces the average memory access time by keeping active data in the smaller, faster cache levels. The implementation is shown in Listing 2.

```
// Multiply C <- C + A * B using b x b tiles (
    row-major)
void dgemm_blocked(n, b, A, B, C) {
  allocate Ablk[b][b], Bblk[b][b], Cblk[b][b];

  for (row = 0; row < n; row += b)          //
     block rows of C/A
    for (col = 0; col < n; col += b) {
      // block cols of C/B
      // copy C(i:j) vals into the block
      copy_tile(C, Cblk);

      for (k = 0; k < n; k += b) {         //
    tile along K dimension
        // copy A(i:k) and B(k:j) vals into
    blocks
        copy_tile(A, Ablk);
        copy_tile(B, Bblk);

        // Matrix Multiplication: Cblk += Ablk
     * Bblk
        for (i = 0; i < b; i++) {
          for (j = 0; j < b; j++) {
            sum = 0;
            for (k = 0; k < b; k++) {
              sum += Ablk[i][k] * Bblk[k][j];
            }
            Cblk[i][j] += sum;
          }
        }

      // write C tile back to memory
      store_tile(Cblk, C);
    }

  free Ablk, Bblk, Cblk;
}
```

Listing 2. Blocked Matrix Multiply with Copy Optimization (BMMCO): copies b×b tiles of A, B, and C into buffers, multiplies/accumulates, then writes C back.

III. Evaluation

Provide an introductory paragraph that summarizes what's in this section: a list of runs/experiments intended to test your implementation and ideas. Describe each of these experiments in a few words/a sentence.

A. Computational Platform and Software Environment

The performance studies were conducted on a CPU node of the Perlmutter system at NERSC. 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. All codes were compiled with the `-O3` optimization level and the `-march=native` flag. The build system used `cmake` version 3.20.4 and `make` version 4.2.1.

B. Methodology

All implementations operate on square, row–major $n \times n$ matrices. For each problem size n, matrices A, B, and C are created with values between -1 and 1. Clock time is measured with `std::chrono::high_resolution_clock` on each round of testing. After timing, correctness is verified by computing a reference result with `cblas_dgemm`, and checking that all values in both results match.

Performance is reported as MFLOP/s derived from elapsed time. Double-precision matrix multiply performs

approximately $2n^3$ floating-point operations (one multiply and one add per inner loop iteration). Given elapsed time t (seconds), the performance metric is

$$\text{MFLOP/s} = \frac{2n^3}{t \cdot 10^6}. \quad (3)$$

This metric is used for all charts and tables.

Experiments are conducted on the problem sizes 64, 128, 256, 512, 1024, and 2048. To avoid library initialization delays, the benchmark runs the first size ($n = 64$) twice. The first timing is treated as a warm-up and is discarded for analysis.

For the BMMCO implementation, the block size parameters used for testing are 2, 16, 32, and 64. For each block size and problem size pair, one timed trial is recorded.

All graphs in this analysis display the problem size on the horizontal axis and MFLOP/s on the vertical axis.

C. Comparison of Basic MM and CBLAS

This experiment shows the performance gap between the three loop matrix multiplication (Basic MM) and the CBLAS library (`cblas_dgemm`). The main question addressed is how inefficient use of caches in the Basic MM affects overall performance. Both implementations were tested on the same problem sizes as described in Sec. III-B, with performance measured in MFLOP/s.

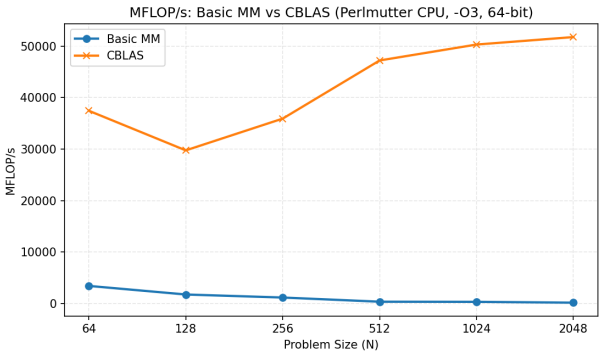

Fig. 1. Performance of Basic MM and CBLAS on different problem sizes. The horizontal axis is the problem size N, and the vertical axis is performance in MFLOP/s.

The Basic MM implementation shows a decrease in performance as N increases: from around 3400 MFLOP/s at $N = 64$ down to about 123 MFLOP/s at $N = 2048$. This decrease is likely due to loading too much data into the smaller, faster caches. Elements of B are accessed in a columnwise pattern, which causes repeated loading and unloading from the L1 cache once the matrix no longer fits.

In contrast, the CBLAS implementation maintains high performance on all problem sizes, with a low of 29,747 MFLOP/s at $N = 128$ and reaching 51,789 MFLOP/s at $N = 2048$. The MFLOP/s increase is due to vectorization optimizations, and likely other optimizations that pay off as problem sizes increase. For this analysis, CBLAS is simply used as a reference to show how fast the matrix multiplication can be performed with the right optimizations.

D. Blocked MM with Copy Optimization (BMMCO) vs CBLAS

This experiment compares the blocked matrix multiplication with copy optimization (BMMCO) to the CBLAS reference implementation. The main question addressed is how block size impacts performance and whether more efficient use of caches can improve memory access time. BMMCO was run with block sizes 2, 16, 32, and 64, while CBLAS serves as a highly optimized baseline. Performance is again measured in MFLOP/s across the same problem sizes described in Section. III-B.

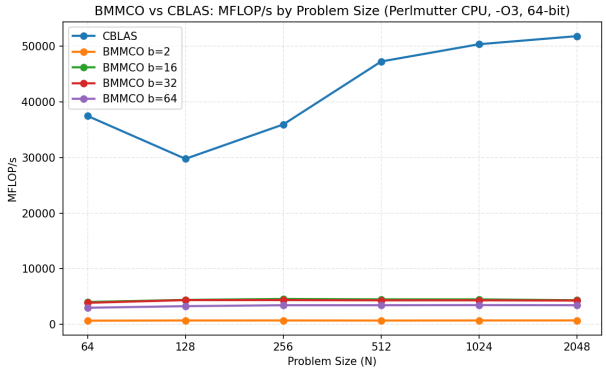

Fig. 2. Performance of BMMCO at different block sizes (b) compared with CBLAS. The horizontal axis is the problem size N, and the vertical axis is performance in MFLOP/s.

The BMMCO results vary across different block sizes. With a very small block size ($b = 2$), performance is poor at around 700 MFLOP/s across all problem sizes. This is due to inefficient data loading into cache lines and results in a high average memory access time. Increasing the block size to a sweet spot around $b = 16$ or $b = 32$ improves performance to about 4300-4600 MFLOP/s. This improvement reflects more data reuse in the smaller caches, which lowers the frequency of cache misses and as a result lowers the average memory access time. At $b = 64$, performance decreases to around 3400 MFLOP/s), which demonstrates blocks becoming

too large to fit in the caches, which in turn increases the average memory access time.

In contrast, the CBLAS implementation consistently maintains very high performance, above 35,000 MFLOP/s for $N \geq 256$ and reaching over 51,000 MFLOP/s at $N = 2048$. These results are an order of magnitude higher than the best BMMCO runs. Under the hood, CBLAS likely implements many optimizations beyond simple blocking and copy optimizations.

Overall, the BMMCO experiment shows that efficient use of cache lines is key for performance. It also highlights that tuning the block size accordingly with the cache line length is important, as smaller blocks aren't reused effectively and large blocks overflow the cache lines which results in unnecessary loading and unloading.

E. Overall Findings and Discussion

The Basic MM algorithm performs poorly, with MFLOP/s dropping as N increases. This decline comes from accessing memory of a large size B in a column major order on a row major ordered array, which causes the L1 cache lines to be overloaded and more data to be copied than necessary.

The BMMCO implementation reduces average memory access time by operating on smaller blocks and then copying back the results to the main memory. As a contrast to the basic implementation, performance increases slightly as N increases, which is likely due to more efficient usage of L1 cache lines. Block sizes of 16 and 32 are the sweet spot for the environment in this studies' experiments: showing around 4300–4600 MFLOP/s. Very small blocks ($b = 2$) waste cache line space, while large blocks ($b = 64$) overflow caches, both leading to higher average memory access time.

CBLAS performs efficiently across all problem sizes tested in this paper (peaking around 50,000 MFLOP/s), especially as problem sizes increase, hinting at vectorization optimizations in the implementation.

Overall, these results show that reducing average memory access time through blocking and copying is useful for good performance, but full use of the hardware requires the additional optimizations present in CBLAS.