Study Of Parallel Matrix Multiplication with OpenMP and LIKWID Hardware Performance Counters
Assignment #4, CSC 746, Fall 2024

Ruxue Jin*
SFSU

ABSTRACT

This research explores the impact of parallelization on the performance of basic and blocked matrix-matrix multiplication (MMUL) algorithms. By employing OpenMP for parallelization and using LIKWID to measure key metrics such as runtime across various problem sizes and thread counts, the study demonstrates that parallelization significantly enhances the efficiency of both the basic and blocked algorithms, particularly for larger problem sizes. The CBLAS algorithm is used as a reference point to compare performance based on L2 cache accesses, L3 cache requests, and retired instructions. The performance ranking from best to worst is block 16, block 4, and basic.

1 INTRODUCTION

This paper explores how parallel programming enhances the performance of basic and blocked matrix-matrix multiplication (MMUL) algorithms on modern multi-core processors. It also compares the performance of the serial versions of the basic, blocked size 4, and blocked size 16 algorithms with the CBLAS algorithm using data from hardware performance counters.

In §2,this research employs OpenMP to parallelize the basic and block matrix multiplication algorithms, using LIKWID to record performance metrics such as runtime (RDTSC) for multithreaded executions, along with L2 accesses, L3 cache requests, and retired instructions for single-thread execution. The performance of these parallelized algorithms is then compared to the single-threaded versions of the same problem sizes to evaluate speedup. Additionally, the serial algorithms are compared with the BLAS algorithm to assess their effectiveness.

The results indicate that using 4 and 16 threads improves the performance of the basic and blocked algorithms by approximately 3.5x and 13x, respectively, across all problem sizes. However, with 64 threads, there is overhead for smaller problem sizes, leading to a speedup of only 13x for size 128, which increases to over 50x for the larger problem size of 2048. For the single-thread execution, Block4 demonstrates more efficient L1 and L2 cache usage compared to Block16 and the basic algorithm, as evidenced by lower L2 cache accesses, L3 cache requests, and fewer retired instructions.

The remainder of this paper presents our implementation, evaluates the performance, and finding and discussions.

2 IMPLEMENTATION

The three algorithms in this paper address the matrix multiplication problem defined by the equation:

$$C = C + A \cdot B$$

where A, B, and C are square matrices of size N × N.

*email:rjin@sfsu.edu

This study employs OpenMP to parallelize the outer two loops of both the basic algorithm (§2.1) and the blocked algorithm (§2.2), utilizing multiple threads for enhanced performance. To gather hardware performance metrics, the LIKWID tool is employed, enabling the analysis of these algorithms across varying thread counts of [1, 4, 16, 64] and problem sizes of [128, 512, 2048]. Additionally, the CBLAS library (§2.3) serves as a reference point for evaluating the results obtained from these implementations.

2.1 Basic Parallel MMUL

The `basic_parallel` function implements a parallelized version of the basic matrix multiplication algorithm using OpenMP. The outer two loops iterate over the matrix dimensions, while the innermost loop computes the dot product of the corresponding row of A and the column of B to update the value of $C[i][j]$. The `collapse(2)` clause in the OpenMP directive allows for the parallel execution of both outer loops, enabling efficient utilization of multiple threads. Performance metrics are recorded using LIKWID markers to evaluate the algorithm's efficiency.The LIKWID marker is placed inside the `#pragma omp parallel` block but outside of `#pragma omp for` to measure the performance of the entire parallel region 1.

This function operates on problem sizes of 128, 512, and 2048, using thread counts of 1, 4, 16, and 64.

```
1 void basic_parallel(int n, double* A, double* B,
      double* C)  {
2   #pragma omp parallel
3   {
4       LIKWID_MARKER_START(MY_MARKER_REGION_NAME);
5       #pragma omp for collapse(2)
6       for (int i = 0; i < n; i++) {
7           for (int j = 0; j < n; j++) {
8               double sum = C[i * n + j];
9               for (int k = 0; k < n; k++) {
10                  sum += A[i * n + k] * B[k * n +
        j];
11              }
12              C[i * n + j] = sum;
13          }
14      }
15      LIKWID_MARKER_STOP(MY_MARKER_REGION_NAME);
16  }
17 }
```

Listing 1: Basic parallel matrix-matrix multiplication

2.2 Blocked Parallel MMUL

The `blocked_parallel` function implements a parallelized version of the blocked matrix multiplication algorithm using OpenMP. It partitions the $n \times n$ matrices A, B, and C into smaller blocks of size `block_size`, with each thread allocating local storage for these blocks to minimize memory contention. The outer 2 loops

over the matrix blocks are parallelized with `#pragma omp for collapse(2)`, optimizing load balancing by allowing threads to process multiple blocks concurrently. Performance tracking is facilitated by LIKWID markers, enabling detailed analysis of computational performance during execution.

This function operates on block sizes of 4 and 16, with problem sizes of 128, 512, and 2048, and utilizes thread counts of 1, 4, 16, and 64.

The pseudocode is shown in Listing 2.

```
void blocked_parallel(int n, int block_size,
    double* A, double* B, double* C){
    int N = n / block_size;
    #pragma omp parallel
   {
       // local storage for each thread
       double* matrixCopyA = new double[block_size
     * block_size];
       double* matrixCopyB = new double[block_size
     * block_size];
       double* matrixCopyC = new double[block_size
     * block_size];

       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++) {

               copy_matrixCopyC()
               // Multiply blocks of A and B,
    accumulate into yC
               for (int k = 0; k < N; k++) {
                   copy_matrixCopyA()
                   copy_matrixCopyB()
                   basic_MMUL()
               }
               write_matrixCopyC_back_to_memory()
               }
           }
       }
       LIKWID_MARKER_STOP(MY_MARKER_REGION_NAME);

       delete[] matrixCopyA;
       delete[] matrixCopyB;
       delete[] matrixCopyC;
   }
}
```

Listing 2: Block parallel matrix-matrix multiplication

2.3 CBLAS Algorithm

The `cblas_dgemm` function, part of the CBLAS [2] library, offers optimized routines for performing matrix-matrix multiplication. It operates in a serial manner, executing the multiplication sequentially. This paper references this algorithm to evaluate both the accuracy and efficiency of the previous two algorithms in terms of their computational results.

3 Evaluation

This section describes the testing environment for these three programs, presents the results of the three algorithms for different problem sizes and different threads, and outlines the performance metrics used for evaluation: runtime speedup and L2CACHE, L3CACHE and Retired Instructions.

```
void cblas_dgemm(int n, double* A, double* B,
    double* C) {

#ifdef LIKWID_PERFMON
      LIKWID_MARKER_START(MY_MARKER_REGION_NAME);
#endif
    cblas_dgemm(CblasRowMajor, 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 3: CBLAS matrix-matrix multiplication

3.1 Computational platform and Software Environment

The computational experiments were conducted using a Oracle Virutal Machine VirtualBox to connect remotely to NERSC's Perlmutter supercomputer. Perlmutter CPU node operates on Linux kernel version 5.14.21, featuring 2x AMD EPYC 7763 (Milan) CPUs, running at 2.45 GHz(boost up to 3.5 GHz). As described in the Perlmutter system overview [1], the system had 512 GB of DDR4 memory, with a cache hierarchy of 32 KB L1, 512 KB L2 and 256 MB shared L3 cache per core. Peak GFLOP/s is 39.4 GFLOP/s per core. Peak memory bandwidth is 204.8 GB/s. The CXX compiler identification is GNU 7.5.0.The compilation flag used is `set(CMAKE_CXX_FLAGS_RELEASE "${CMAKE_CXX_FLAGS_RELEASE} -O3")`.

3.2 Methodology

This paper employs LIKWID [3] to collect performance data from the FLOPS_DP, L2CACHE, and L3CACHE counter groups across all problem sizes. The Runtime (RDTSC) metric is gathered from the FLOPS_DP group for the basic and blocked algorithms using 1, 4, 16, and 64 threads. For parallel executions, the Runtime (RDTSC) value is taken as the maximum across all threads. The speedup for each problem size in multithreaded execution is calculated compared to single-thread execution using the following formula:

$$Speedup = \frac{T_{\text{serial}}}{T_{\text{parallel}}} \tag{1}$$

Additionally, performance metrics such as RETIRED_INSTRUCTIONS from the FLOPS_DP group, L2 Accesses from the L2CACHE group, and L3_CACHE_REQ from the L3CACHE group are collected for the basic, block4, block16, and BLAS [2] algorithms, evaluated in a single-threaded context across all problem sizes.

L2 cache accesses indicate the effectiveness of L1 cache utilization. A higher number of L2 accesses reflects more L1 cache misses, signifying lower L1 cache utilization.

L3 cache requests reflect the efficiency of L2 cache utilization, with higher L3 requests indicating more L2 cache misses and therefore lower overall efficiency.

3.3 Scaling Study for Basic MM with OpenMP parallelization

In Fig. 1, the speedup for 4 threads compared to 1 thread ranges from 3 to 4 across all problem sizes. For 16 threads, the speedup increases to between 12 and 14. Notably, for 64 threads, the speedup exhibits a significant improvement, ranging from 12 to 53 as the problem size increases.

The results indicate a substantial performance enhancement when utilizing multiple threads for matrix multiplication. Specifically,

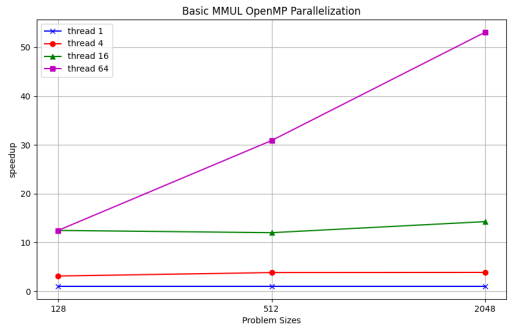

Figure 1: Evaluation of Basic MMUL with OpenMP parallelization

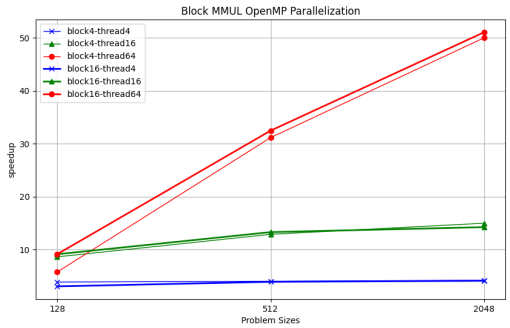

Figure 2: Evaluation of Blocked MMUL with OpenMP parallelization

using 4 threads yields a modest speedup of 3 to 4 times compared to single-thread execution, suggesting that some parallel efficiency is gained. The increase in speedup to 12 to 14 with 16 threads demonstrates more effective resource utilization as the number of threads increases.

The most significant improvement occurs with 64 threads, where the speedup escalates dramatically from 12 to 53, highlighting the algorithm's scalability with larger problem sizes. This behavior indicates that the overhead of managing threads diminishes as the workload increases, allowing for greater parallelization benefits. In contrast, when a higher number of threads operates on a smaller dataset, contention for shared resources such as memory and cache can increase. In these scenarios, threads may frequently access the same cache lines or memory locations, leading to bottlenecks that can slow down overall performance. Therefore, the findings suggest that for large-scale problems, leveraging a higher number of threads can lead to substantial reductions in execution time, while careful consideration is needed for smaller problem sizes to avoid resource contention issues.

3.4 Evaluation of Blocked MMUL with OpenMP parallelization

In Fig. 2,the speedup achieved with 4 threads for both block4 and block16 ranges from 3 to 4 across all problem sizes. For 16 threads, the speedup for both algorithms increases slightly from 9 to 14 as the problem size grows. In contrast, the performance with 64 threads shows a dramatic increase, rising from below 10 to over 50 with larger problem sizes. Notably, the performance difference between block4 and block16 is minimal.

The observed trends suggest that moderate thread counts, such as 4 and 16, provide stable and effective speedup due to manageable parallelization overhead. However, with 64 threads, the overhead becomes more prominent for smaller problem sizes, leading to lower-than-expected performance. As the problem size increases, the workload becomes sufficient to offset the overhead, allowing for better utilization of available cores, which results in significant speedup gains. The minimal performance difference between block4 and block16 indicates that the size of the blocking factor has a lesser impact compared to the number of threads and problem size.

3.5 Comparison of CBLAS, BasicMM-omp, and BMMCO-omp

This section uses the L2 Accesses, L3 Cache Requests, and Retired Instructions of the CBLAS algorithm as a baseline reference for comparison. To standardize the results, the values for CBLAS are normalized to 1 for all metrics across all problem sizes. For the other algorithms, the values in each column represent the ratio of that algorithm's performance counters (L2 Accesses, L3 Cache Requests, or Retired Instructions) to the corresponding CBLAS value for the same problem size. This approach allows for a direct comparison of how each algorithm performs relative to the CBLAS reference across different problem sizes.

Problem Size (N)	cblas	basic	block 4	block 16
128	1	48	4.6	0.87
512	1	46	4.3	1.47
2048	1	148	8.19	1.72

Table 1: Comparison of L2 Cache Access of basic, block and cblas algorithm for different problem sizes.

Problem Size (N)	cblas	basic	block 4	block 16
128	1	0.98	0.89	0.90
512	1	293	18.20	3.75
2048	1	572	19.67	4.13

Table 2: Comparison of L3 Cache Reqeust of basic, block and cblas algorithm for different problem sizes.

3.5.1 Comparison of L2 Cache Access of CBLAS, BasicMM-omp, and BMMCO-omp

In Table 1, basic matrix multiplication shows 48 to 148 times more L2 accesses than CBLAS, indicating the poorest L1 cache utilization. Block4, on the other hand, exhibits 4 to 8 times more L2 accesses than CBLAS. Block16 performs the best, with its L2 accesses ranging from only 0.87 to 1.7 times that of CBLAS, demonstrating the highest L1 cache efficiency.

3.5.2 Comparison of L3 Cache Reqeust of CBLAS, BasicMM-omp, and BMMCO-omp

In Table 2, for smaller problem sizes such as 128, both basic and block matrix multiplication exhibit low L3 cache requests, suggesting relatively good L2 cache performance at this scale. However, as the problem size increases, the basic matrix multiplication experiences a significant rise in L3 cache requests, reaching 300 to 570 times that of CBLAS, which suggests poor L2 cache utilization. Block4 performs moderately, with around 20 times more L3 requests than CBLAS. Block16 outperforms both, with L3 cache requests only 3 to 4 times higher than CBLAS, indicating the best L2 cache efficiency among the methods.

For smaller problem sizes, all the data can fit into the L1 and L2 caches, leading to minimal differences in performance across the algorithms. However, as the problem size increases, Block16 shows better performance due to more effective cache blocking, which improves data locality and reduces cache misses, particularly in the L2 cache, allowing it to maintain higher efficiency compared to the other methods.

Problem Size (N)	cblas	basic	block 4	block 16
128	1	9.7	24.5	11.85
512	1	10.6	27.41	13.29
2048	1	10.6	27.8	13.43

Table 3: Comparison of Retired Instruction of basic, block and cblas algorithm for different problem sizes.

3.5.3 Comparison of Retired Instructions of CBLAS, BasicMM-omp, and BMMCO-omp

In Table 3, both the basic and block 16 algorithms exhibit approximately 10X more retired instructions than CBLAS, whereas block 4 shows around 25X more retired instructions. This indicates that block 4 is less efficient, requiring more computational steps to achieve the same results. This inefficiency can likely be attributed to suboptimal memory access patterns and the increased overhead associated with managing parallel execution.

3.6 Findings and Discussion

This paper demonstrates that multithreaded programming significantly improves the efficiency of both the basic and block matrix multiplication algorithms. While using many threads with small problem sizes can sometimes degrade performance due to overhead, the advantages become more evident as the problem size increases. For single-threaded execution, block 16 outperforms block 4 by exhibiting fewer L2 cache accesses, fewer L3 cache requests, and fewer retired instructions, indicating better cache utilization. In contrast, the basic algorithm performs the worst among the three, showing poor efficiency and cache usage.

ACKNOWLEDGMENTS

The author would like to acknowledge the assistance of ChatGPT, a language model developed by OpenAI, for helping to articulate the interpretation of our data and for providing valuable explanations on technical concepts, such as retired instructions and their significance in performance analysis.

REFERENCES

[1] N. E. R. S. C. C. (NERSC). Perlmutter system overview. https://www.nersc.gov/users/systems/perlmutter/, 2024. Accessed: 2024-10-07.

[2] Netlib. *CBLAS: C interface to the Basic Linear Algebra Subprograms*, 2002.

[3] W. Schreiner et al. Likwid: Like i knew what i do. https://likwid.sourceforge.net, 2024. Version 5.2.2.