Assignment #2, CSC 746, Fall 2021

Zhuozhuo Liu*
San Francisco State University

ABSTRACT

In this assignment, I tested the performance of three methods that were used to implement matrix multiplication(Mmul): Basic Mmul, Blocked Mmul with copy multiplication, and CBLAS Mmul. The result showed that CBLAS outperformed the other two methods significantly. Blocked Mmul performed more FLOP/s than basic Mmul so it's faster because it required less memory access by optimizing the spatial locality. Both Blocked Mmul and Basic Mmul were bounded by the memory access across all problem sizes. CBLAS were bounded by the memory access at larger problem sizes.

1 INTRODUCTION

In this assignment, I implemented matrix multiplication(Mmul) and studied its performance. I tested three programs which conduct Mmul in different ways. The first program is the traditional way of doing Mmul. The third program uses CBLAS library to do Mmul. The second program uses tiling and blocking optimizations which increases locality and FLOPs per memory access. I then compared the performance between different methods. The performance ranking from fast to slowest is: CBLAS, Blocked Mmul, Basic Mmul. Among different block sizes, the best performed one is block size = 16. I will discuss results in details and possible explanation about these findings in the section 3 and 4.

2 IMPLEMENTATION

There are three programs that implements Mmul. After running each program, it will print out the total run time spent on Mmul operations code block. With the run time recorded, I tested out different problem sizes of matrix and study how problem size and block size are relating to the performance.

Program 1 and 3 take problem size N, matrix A, matrix B, and matrix C as arguments. Program 2 takes the same arguments and an additional argument block size. All of them calculates C + A * B and the run time was recorded.

2.1 Part 1: program 1 (basic Mmul)

This is the basic way to implement Mmul (see Listing 1). There are three nested loops in this program. j stands for j-th column in C/A and i stands for i-th row in C/B. In all 3 programs, i is nested in i since matrices are stored in column-major format and it can be faster to fix column and iterate over rows.

2.2 Part 2: program 2 (blocked Mmul)

There are three outer loops (see Listing 2) that locate the index of each block and three inner loops that conducts basic Mmul on b*b block. Compared to program 1 and 3, three implementations were used to optimize for performance: 1) increase FLOPs per memory read by implementing blocking, 2) load each block into cache and copying back to memory (by calling copy_matrix_to_local() and copy_matrix_to_memory()), 3) nest row i inside of column j to adapt to column-major format.

*email:zliu15@mail.sfsu.edu

```
for (off_t j = 0; j < n; i++) {
  for (off_t i = 0; i < n; j++) {
     for (off_t k = 0; k < n; k++) {
        // C[i][j] += A[i][k] + B[k][j]
        C[i+j*n] += A[i+k*n] * B[k+j*n];
     }
  }
}
```

Listing 1: record the run time of running the above loop

2.3 Part 3: program 3 (CBLAS)

The program (see Listing 3) calls CBLAS library and do Mmul.

3 RESULTS

3.1 Computational platform and Software Environment

All tests were running on Cori KNL node. The processor was Intel Xeon Phi Processor 7250. The clock rate was 1.4 GHz. The size of L1 cache was 64 KB. The size of L2 cache was 1MB. The memory size was 96 GB for DDR4 and 16 GB for MCDRAM. Cmake version was version 3.14.4. GCC version was version 8.3.0. Optimization level was O3 by default.

3.2 Methodology

The performance metrics is the elapsed time from instrumentation code added around the Mmul code block. I used chrono to record start time and end time and then did a subtraction to get the run time.

I ran tests over a set of prescribed problem sizes (size of the matrix): 64, 128, 256, 512, 1024. Matrix A, B, C share the same size. For program 2, I ran block size among 2, 16, 32, 64 for each problem size.

3.3 Comparison of Basic MMul and CBLAS

Figure 1 shows the comparison on performance (measured by MFLOP/sec) between Basic MMul and CBLAS. It's calculated by dividing total number of MFLOPs by run time for each problem size. In my algorithm, there are $2 * problem\_size^3$ FLOPs in total since there are 2 FLOPs (addition and multiplication between matrix elements) per MMul and 3 nested loops $problem\_size^3$ MMuls.

As we can see, for Basic Mmul, the MFLOP/s slightly decreased across different problem sizes. It might because the performance is bounded by memory access. However, for CBLAS Mmul, problem size = 512 is a turning point. The MFLOP/s increases significantly from about 4 MFLOP/s in problem size = 64 to 39 MFLOP/s in problem size = 512. It doesn't change significantly from problem size = 512 and up. The performance might be bounded by the computational speed before the turning point and then it's bounded by memory access.

Comparing Basic and CBLAS, I found that CBLAS performed significantly better than Basic Mmul in every problem size, since Basic Mmul algorithm required more memory access and more computational resources than CBLAS.

```
for (off_t j = 0; j < num_blocks; j++) {
  for (off_t i = 0; i < num_blocks; i++) {
     // copy block C[i, j] into cache
     copy_matrix_to_local(Ccopy, C, block_size, n,
     i, j);
     for (off_t k = 0; k < num_blocks; k++) {
        // copy block A[i, k] and block B[k, j]
     into cache
        copy_matrix_to_local(Acopy, A, block_size,
     n, i, k);
        copy_matrix_to_local(Bcopy, B, block_size,
     n, k, j);
        // mmul on blocks
        for (off_t y = 0; y < block_size; y++) {
           for (off_t x = 0; x < block_size; x++)
     {
              for (off_t z = 0; z < block_size; z
     ++) {
                 Ccopy[x+y*block_size] += Acopy[x+
     z*block_size] * Bcopy[z+y*block_size];
              }
           }
        }
     }
     copy_matrix_to_memory(C, Ccopy, block_size, n
     , i, j);
  }
}
```

Listing 2: record the run time of running the above loop

```
cblas_dgemm(CblasColMajor, CblasNoTrans,
     CblasNoTrans, n, n, n, 1., A, n, B, n, 1., C,
     n);
```

Listing 3: record the run time of running the above loop

3.4 Comparison of Blocked MM with Copy Optimization (BMMCO) and CBLAS

Figure 2 shows the comparison on performance (measured by MFLOP/sec) between Blocked MM with Copy Optimization (BMMCO) and CBLAS. Similar to the above section, it's calculated by dividing total number of MFLOPs by run time for each problem size.

Figure 3 shows the same thing with Figure 2 except that Figure 3 has a zoom-in window on BMMCO variants. As we can see, BMMCO's performance didn't vary a lot among different problem size but varied different among different block sizes. The best performed configuration was block size 16, followed by block size 32 and 64, finally the worst was block size 2.

Given the fact that the relationship between block size and performance is not linear, there are at least 2 factors that influence the performance in that way: one positively correlated to the performance as block size increases while the other negatively correlated to the performance as block size increases.

Compared to block size = 2, block size = 16 was faster since it had less slow memory access to original matrix with the assistance of local copy of matrices (the $\text{copy}_{m}atrix_{t}o_{l}ocal()functionintheimplementationsection$).

Compared to block size = 16, block size = 32 was slower not because the above factor didn't exist. It was still existing which accelerated the performance. However, there was another factor that slowed the performance. Since block size = 32 had larger block size, it had slower memory access to the local copy of matrices, compared to block size of 16. The cache hit rate (or spatial locality) for a

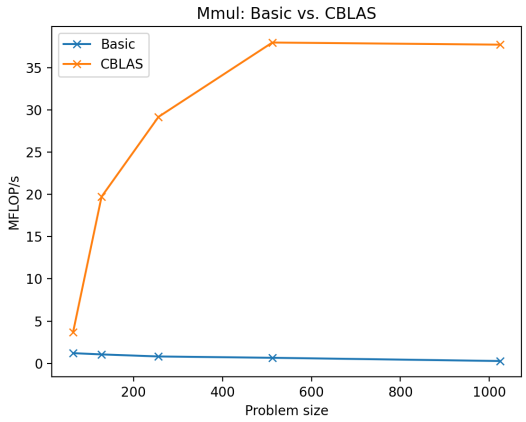

Figure 1: Comparison of BMMCO Mmul vs. CBLAS Mmul with increasing problem sizes. This figure was produced by the program `plot.py` file.

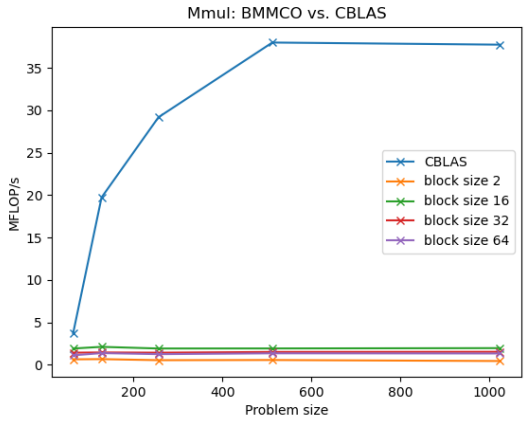

Figure 2: Comparison of BMMCO vs. CBLAS Mmul with increasing problem sizes. This figure was produced by the program `plot.py` file.

larger local copy of matrices was lower. Therefore, the performance decreased as the block size was larger, combining both factors.

It's intuitive to think about it with extreme cases: when the block size is 1 or the block size is as much as the size of the matrix, the algorithm will become the basic Mmul which less performed (I will discuss the comparison with basic Mmul in the next section).

Also, the similarity between different block sizes was that both of them slight decreased from problem size = 128 to problem size = 256 and kept stable after that.

Comparing BMMCO and CBLAS, I found that CBLAS performed significantly better than BMMCO in every problem size. It indicated that CBLAS might have some more powerful optimization methods.

On the hardware's level, the performance of BMMCO was still bounded by the memory latency since it still needed to read/write to memory.

4 Overall Findings and Discussion

Basic Mmul and BMMCO Mmul shared something in common. For example, both of them were bounded by the memory access (or memory latency) so that the problem size was not correlate to the performance very much. Also both slightly decreased from problem

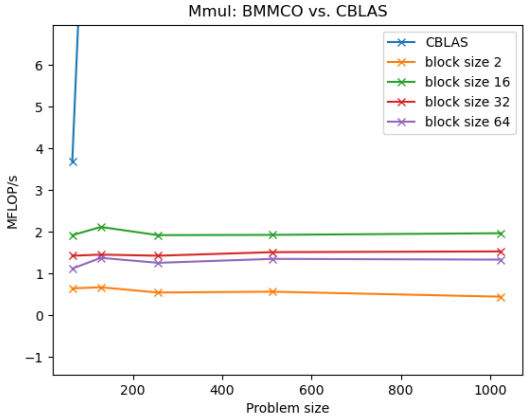

Figure 3: Comparison of BMMCO vs. CBLAS (in details). This figure was produced by the program `plot.py` file.

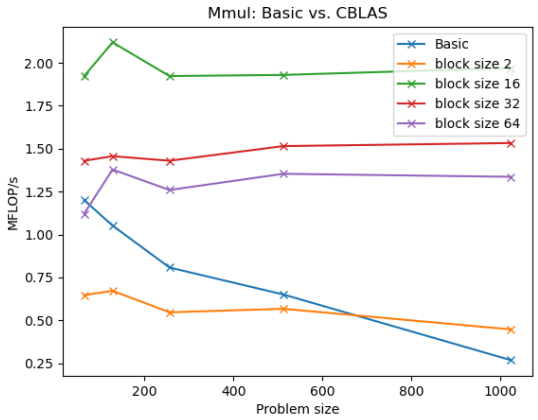

Figure 4: Comparison of Basic vs. CBLAS. This figure was produced by the program `plot.py` file.

size = 128 to problem size = 256 in terms of the performance.

The difference is that the performance of basic MMul decreased as the problem size went higher for larger problem sizes. However, the performance of BMMCO Mmul kept stable for larger problem sizes. In another word, BMMCO Mmul performed better on larger problem sizes (block size = 16 is the fastest configuration among all). Also, compared with basic MMul, BMMCO Mmul performed faster in general (see Figure 4), since it optimizes locality. BMMCO Mmul had less slow memory access to original matrix because of local copy of matrices. On hardware's level, the average memory latency was less with copy optimization than without it.

There might be some variants since I only ran problem once. In future I can ran programs multiple times and calculate the average run time to get more accurate metrics.