Assignment #4, CSC 746, Fall 2021

Zhuozhuo Liu*
San Francisco State University

Abstract

We used OpenMP-parallel to optimize Matrix Multiplication(MM) and compared the performance between different programs and concurrency. We tested three MM programs on Cori KNL nodes and used LIKWID to collect performance data from performance counter group. We found that the problem size, the block size and the concurrency levels influenced the performance and had some cross effects between each other. We analyzed and provided possible explanation. We finally compared cache data among different programs and studied memory hierarchy utilization and data movement.

1 Introduction

In this assignment, we implemented MM and used OpenMP to optimize its performance. We tested three programs which conducted MM in different ways. The first program is a basic MM with 3 for-loops. The second program (BMMCO) implemented MM with tiling/blocking. Both programs used OpenMP-parallel library to perform paralleled programming which optimized the performance. The third program used CBLAS library to do serial MM as a benchmark. We studied 3 factors: problem size, block size, and concurrency level, and their effects and cross effects on the performance of each program. We then compared the runtime (RDTSC), MCDRAM memory read volume, and DDR memory read volume measured by LIKWID among programs and analyzed factors that may influence data movement.

2 Implementation

There were three programs that we implemented. Code block were surrounded by LIKWID MARKER API which collected performance data from performance counter group. After running each program, it printed out LIKWID metrics on MM operations code block.

All programs took problem size n, matrix A, matrix B, and matrix C as arguments and calculated C := C + A * B.

2.1 Program 1 (Basic MM)

This is a basic method of implementing MM (see Listing 1). There are 3 nested for loops in this program. i stands for i-th row and j stands for j-th column in Matrix C. We used OpenMP parallel for to parallel the first for loop (parallel computing each column). We may also use parallel for collapse to parallel both i-th and j-th but we didn't do this in this test.

2.2 Program 2 (BMMCO)

There are three outer loops (see Listing 2) that locate the index of each block and three inner loops that conducts basic MM on b*b block. Three implementations were used to optimize for performance: 1) increase FLOPs per memory read by implementing tiling/blocking, 2) load each block into cache and copying back to memory by calling copy_matrix_to_local() and copy_matrix_to_memory(), 3) use OpenMP parallel for to parallel the work of computing each block.

*email:zliu15@mail.sfsu.edu

```
#pragma omp parallel for
LIKWID_MARKER_START(MY_MARKER_REGION_NAME);
for (off_t j = 0; j < n; j++) {
  for (off_t i = 0; i < n; i++) {
     double Ccopy;
     Ccopy = C[i+j*n];
     for (off_t k = 0; k < n; k++) {
        Ccopy += + A[i+k*n] * B[k+j*n];
     }
     C[i+j*n] = Ccopy;
  }
}
LIKWID_MARKER_STOP(MY_MARKER_REGION_NAME);
```

Listing 1: record the run time of Program 1 (basic VMM)

2.3 Program 3 (CBLAS)

The program calls CBLAS library and do MM (see Listing 3). Note that this program is serial compared to other 2 paralleled programs.

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. LIKWID version is 5.2.0. GCC version was version 8.3.0. Optimization level was O3 by default. [1]

3.2 Methodology

There are 3 performance metrics from LIKWID that we focused on: run time, MCDRAM read volume and DDR read volume.

We used the maximum run time(RDTSC read-time stamp counter, in sec) among all thread because the overall run time of the program is the run time of the slowest thread which was waited by other threads.

For MCDRAM read volume and DDR read volume, we used the sum of the volume of all threads. We cared about the total volume processed because that the sum is the overall performance regardless of different concurrency level. It would also be easier to compare with serial program like CBLAS with the total volume.

We ran tests over a set of prescribed problem sizes (i.e., size of the matrix) N : 128, 512, 2048. In the paralleled computing (Basic and BMMCO), we tested the above problem size for 4 types of concurrency P (we will use P to represent concurrency in the following analysis): 1, 4, 16, 64. In program 2 (BMMCO) specifically, we tested block size B of 4 and 16.

By default, OpenMP statically assigns loop iterations to threads.

3.3 Scaling Study for Basic MM with OpenMP parallelization

Figure 1 shows the comparison on performance (measured by speedup) among different concurrency level P by using program Basic MM. The x-axis is the problem size and the y-axis is the speedup which is calculated by $T(n,1)/T(n,P)$ where T(n, 1) is the time of scheduling 1 thread (i.e., P = 1, serial computing) and T(n,

```
#pragma omp parallel for collapse(2)
LIKWID_MARKER_START(MY_MARKER_REGION_NAME);
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);
  }
}
LIKWID_MARKER_STOP(MY_MARKER_REGION_NAME);
```

Listing 2: record the run time of Program 2 (OpenMP)

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

Listing 3: record the run time of program 3 (CBLAS)

P) is the time of scheduling P threads. In another words, we used 1 thread as a benchmark to see how other concurrency work.

In figure 1, we can find that both problem size N and concurrency levels P improved the performance speedup to some level, although there were some cross effects among these two factors.

For the effect of N, the performance was bounded by the memory bandwidth at first (e.g., performance went up) and then bounded by the CPU computational speed (e.g., performance went steady later).

For the effect of P, it's also easy to understand why concurrency level improved the performance, since there were multi-core performed the computing. The speedup didn't reached the ideal level (which is P) since 1) there were still some serial part of the code (e.g., write to C) 2) faster threads had to wait for the slowest thread to finish.

Now let's talk about the cross effect. The effect of N worked the best on P = 64, the largest concurrency level. Although N was positively correlated to speedup for both P = 16 and P = 64, the effect of N stopped at N = 2048 for P = 16. The worst is P = 4 level, the effect of N didn't take place at this level.

It might because that the computational speed had reached the peak for each thread with less concurrency (i.e., P = 4), so that the speed didn't change very much with larger N. However, for larger concurrency levels, there were still some room for each thread to improve the computational speed, therefore as problem size went

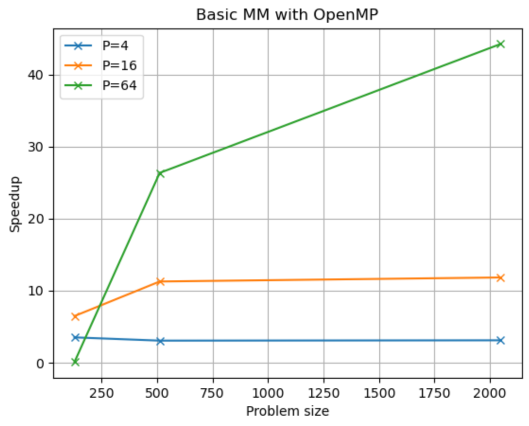

Figure 1: Scaling Study for Basic MM with OpenMP parallelization. Both problem size N and concurrency levels P improved the performance speedup to some level, although there were some cross effects among these two factors. This figure was produced by the program `plot.py` file.

up, each thread performed more computation which increased the speed. This may also explain the bad performance of P = 64 and N = 250, which was almost the similar to P = 1.

3.4 Evaluation of BMMCO with OpenMP parallelization

Figure 2 shows the comparison on performance (measured by speedup) among different concurrency level P and different block size B by using program BMMCO. Figure 3 is the same except that the speedup is in log(base of e) to show slight differences. The x-axis is the problem size and the y-axis is the speedup which is calculated by $T(n,4,1)/T(n,B,P)$ or $T(n,16,1)/T(n,B,P)$. For all configurations whose block size B = 4, The baseline or benchmark is $T(n,4,1)$. For all configurations whose block size B = 16, The baseline or benchmark is $T(n,16,1)$.

In this test, there are three factors that we studied: N, P, and B. We will discuss each of them and the cross effect if any.

For the effect of N and the effect of P, We found something similar to program 1. We have analyzed discussed the same effect in section 3.3.

For the effect of B, we observed an effect of B on P = 16 level where B positively correlated to the performance. Compared to B = 4, B = 16 was slightly faster since it had less slow memory access to original matrix with the assistance of local copy of matrices (the $copyMatrixToLocal()$ function in listing 2).

Now let's talk about the cross effect between N and P. Similarly to program 1 in section 3.3, we found the cross effect between N and P: as P went up, the effect of N went large (or as N went up, the effect of P went larger). In another word, large problem size works better on large concurrency than small concurrency.

Additionally, we observed a cross effect between N and B at P = 64 level. We found that large block size didn't necessarily improved the performance. When N = 512, the B = 4's performance was almost 1.5 times of B = 16's performance. However, when N = 2048, the B = 4's performance was slightly smaller than B = 16's performance. We guess that for each concurrency level, there should be a best-performed B size attach to it. For P = 64, the best performed B size should between 1 to 16.

The reason why B = 16 didn't outperform may be that there was at lest a negative factor that decreased the speed. Since B = 16 had larger block size, it had slower memory access to the local copy of matrices, compared to B = 4. The cache hit rate (or spatial

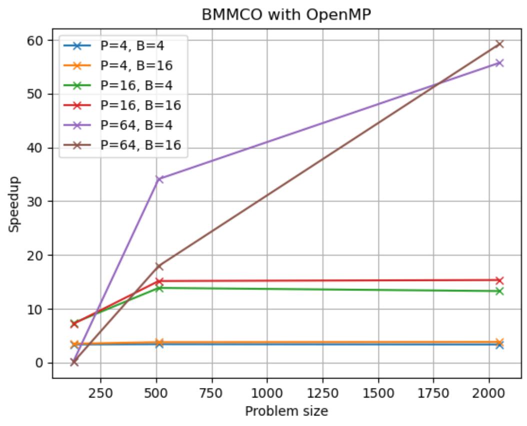

Figure 2: Evaluation of BMMCO with OpenMP parallelization. We observed a cross effect between N and B at P = 64 level. We found that large block size didn't necessarily improved the performance. This figure was produced by the program `plot.py` file.

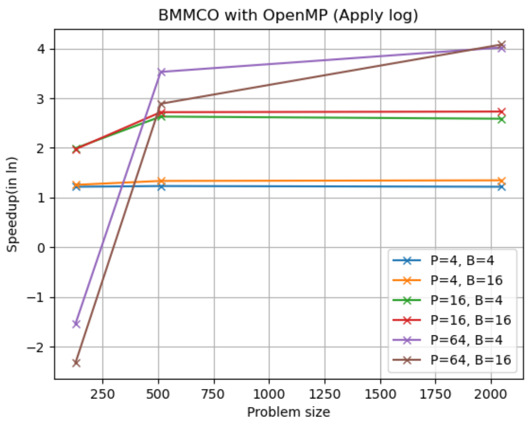

Figure 3: Same to Figure 2 with log applied. This figure was produced by the program `plot.py` file.

locality) for a larger local copy of matrices was lower. Therefore, the performance decreased as the block size was larger, combining both positive and negative factors.

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

Table 1 shows the comparison on HBMCACHE performance (measured by LIKWID) among serial computation: CBLAS, Basic MM and BMMCO (both are at N = 2040 and P = 1). The data of BMMCO is B=16 since it's the best performed configuration.

MCDRAM is the last-level cache which are shared by all cores. We focused on it because it is a proxy for L1/L2 cache utilization. The correlation between MCDRAM read volume and run time is positive. The correlation between MCDRAM read volume and cache utilization is negative. Comparing all 3 programs, basic MM has the largest MCDRAM volume (i.e., 672.01 GB), which indicated that fewer load requests from L1 and L2 were satisfied. In another words, the cache hit is lowers so that the data has to move from DRAM to L2/L1. Therefore, we observed a significant higher run time (i.e., 181.47 sec) in the basic MM program because of the additional data movement, compared to the other two programs. CBLAS performed

Program	Runtime	DDR vol (GB)	MCDRAM bw (MB/s)	MCDRAM vol (GB)
CBLAS	0.75	0.06	3892.62	2.92
Basic	181.47	13.18	3703.15	672.01
BMMCO	52.05	3.78	332.13	17.29

Table 1: Comparison of different programs on hardware performance metrics. The correlation between MCDRAM read volume and run time is positive. They are all of N = 2048 and P = 1. BMMCO is of B = 16.

Program	Runtime	DDR vol (GB)	MCDRAM bw (MB/s)	MCDRAM vol (GB)
Basic	4.10	0.31	127866.55	522.00
BMMCO	0.88	0.08	8880.81	7.54

Table 2: Comparison of different programs on hardware performance metrics. The correlation between MCDRAM read volume and run time remain positive, slightly better than serial versions. They are all of N = 2048 and P = 64. BMMCO is of B = 16.

the fastest (i.e., 0.75 sec) with lowest MCDRAM volume size (i.e., 2.92 GB).

We found that the DDR data read volume was significantly lower than MCDRAM read volume on all programs because all problem sizes fit entirely within MCDRAM (i.e., less than 16 GB). But we can still find the similar pattern of negative correlation between DDR read volume and the run time. The reason may be the same with what we mentioned above.

Table 2 shows the comparison on HBMCACHE performance (measured by LIKWID) among paralleled computation: Basic MM and BMMCO (both are at N = 2040 and P = 64). The correlation remained similar with higher concurrency. Basic MM had higher MCDRAM volume (i.e., 522 GB) and higher run time (i.e., 4.10 sec), although the run time is reduced largely due to the concurrency.

4 Overall findings and discussion

Compared Table 1 and Table 2, we observed that as the concurrency level got larger, the MCDRAM volume decreased. For Basic MM program, the MCDRAM data volume dropped slightly from 672.01 GB to 522.00 GB. For BMMCO program, the MCDRAM data volume dropped largely from 17.29 GB to 7.54 GB. It indicated that threading helped in filling in L1 and L2 memory by dividing the tasks in to multiple sections. At higher concurrency, the certain amount of the data were divided into many L1/L2 caches so that the total data in all L1/L2 are larger than the serial version.

References

[1] https://docs.nersc.gov/systems/cori/. Cori system specification.