# Assignment #3, CSC 746, Fall 2021

Zhuozhuo Liu*
San Francisco State University

## Abstract

We used OpenMP-parallel to optimize Vector Matrix Multiplication(VMM) and compared the performance with serial programs. We tested three VMM programs on Cori and recorded the run time of each. We found that the performance of OpenMP-parallel was significantly higher than the CBLAS which is the best performed serial algorithm. The best configuration using OpenMP is the largest number of thread 64 with static scheduling. We also observed that the performance was bounded by the memory access at smaller problem size.

## 1 Introduction

In this assignment, we implemented VMM and used OpenMP to optimize its performance. We tested three programs which conducted VMM in different ways. The first program is a basic serial method without any optimization. The third program uses CBLAS library to do VMM as a benchmark which is also serial. The second program uses OpenMP-parallel library to perform paralleled programming which optimized the performance. We then compared the run time among different methods and among different problem size. The performance of OpenMP-parallel was significantly higher than the CBLAS, thanks to the parallelism. The performance of the basic serial method is the lowest. The best configuration is the 64 threads with static scheduling.

## 2 Implementation

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

All programs take problem size n, matrix A, vector x, and vector y as arguments, calculate y = y + A * x and finally return void.

### 2.1 Program 1 (basic VMM)

This is the most basic method of implementing VMM (see Listing 1). There are two nested for loops in this program. i stands for i-th row and j stands for j-th column in Matrix A.

```
for (off_t i = 0; i < n; i++) {
  for (off_t j = 0; j < n; j++) {
     y[i] += A[i*n+j] * x[j];
  }
}
```

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

*email:zliu15@mail.sfsu.edu

### 2.2 Program 2 (OpenMP)

The calculation is almost the same with Program 1 except that we applied OpenMP to parallel the outer for loop (see Listing 2). In another word, each row i of matrix A was calculated at the same time. Consequently, we copied y[i] from memory to the local in order to write only one time per row.

```
#pragma omp parallel for
// Calculating each row i of matrix A at the same
    time
for (off_t i = 0; i < n; i++) {
  // Create a local copy of y[i] to minimize
     memory access for write
  double ycopy;
  ycopy = y[i];
  for (off_t j = 0; j < n; j++) {
     // Calculation
     ycopy += A[i*n+j] * x[j];
  }
  y[i] = ycopy;
}
```

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

### 2.3 Part 3: program 3 (CBLAS)

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

```
cblas_dgemv(CblasRowMajor, CblasNoTrans, n, n,
    alpha, A, lda, x, incx, beta, y, incy);
```

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

## 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. [1]

### 3.2 Methodology

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

We ran tests over a set of prescribed problem sizes (size of the matrix/vector): 1024, 2048, 4096, 8192, 16384. Since we observed that the time of the first problem size is not very stable which might because that the initial cache was not warmed up, we added a dummy problem size of 1024 at the very beginning to warm up the cache. So the actual problem size we implemented is 1024, 1024, 2048, 4096, 8192, 16384. In the paralleled computing, we tested the above problem size for 7 types of concurrency: threads of 1, 2, 4, 8, 16, 32, 64.

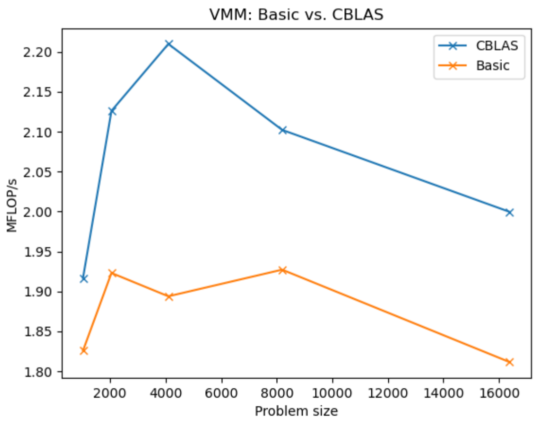

Figure 1: Comparison of Basic vs. CBLAS with increasing problem sizes. The MFLOP/s of both increased at some peak and then decreased. This figure was produced by the program `plot.py` file.

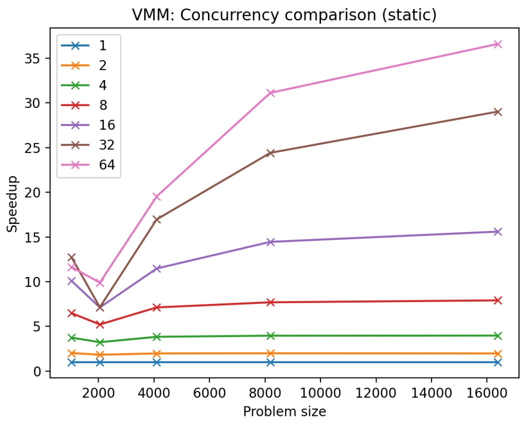

Figure 2: Static scheduling: comparison of concurrency with increasing problem sizes. P = 64 performed the best. This figure was produced by the program `plot.py` file.

### 3.3 Comparison of Basic VMM and CBLAS

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

The CBLAS version performed significantly better than Basic version. Although both of them are serial computation, CBLAS optimized the algorithm greatly since it uses hand-written kernels in assembler. The peak for both are slightly different but the increasing trend before n = 2048 and the decreasing trend after n = 8192 are the same.

There are two peaks for the Basic VMM: n = 2048 and n = 8192. Both of them are around 1.93 MFLOP/s. The occurrence of the peak may because of the memory bound (i.e., L1 cache and L2 cache). From n = 1024 to n = 2048, the performance increased because it could be bounded by the L1 memory. Then the performance decreased from n = 2048 to n = 4096 and increased again to another peak at n = 8192 because it could be bounded by L2 cache. Then it dropped from n = 8192 and up because of the increasing memory access.

### 3.4 Evaluation of OpenMP-parallel VMM

In this section, I will discuss two scheduling strategies: static scheduling and dynamic scheduling. Static scheduling means that outer loops (e.g., all rows of matrix A) are divided evenly to P threads (e.g., row 1-10 to thread 1, 11-20 to thread 2, etc.). Dynamic scheduling means that outer loops are divided by ”first come first serve” basis since the work is not divided at the very beginning and will be assigned without order.

Figure 2 shows the comparison on different concurrency for static scheduling. Figure 3 shows the comparison on different concurrency for dynamic scheduling. In both figures, 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., serial computing) and T(n, P) is the time of scheduling P threads. In another words, we used 1 thread as a benchmark to see how other concurrency work compared to it.

Comparing different concurrency using static scheduling in Figure 2, the speedup increased as the concurrency got larger. Theoretically each concurrency should double the performance from its previous level's performance. In another words, the ideal speedup for concurrency P should be P. It didn't reach the ideal P since there were some serial section that can't be paralleled such as writing to the same y vector.

Also we observed that almost all concurrency levels had a drop on n = 2048. It may because that the data needed by n = 2048 didn't fit into faster cache (e.g., L1 cache) so that it needed to use slower cache. The speedup went higher after n = 2048 because the arithmetic intensity (FLOP/bytes) increases. In another words, for each bytes read from memory, we need to perform more FLOP so that the CPU utilization increased and FLOP/sec also increased before it reaches the computational peak.

In the above observation, there's another thing that attracted our attention - at n = 2048 the speedup were almost the same of concurrency level 16, 32 and 64. This may because that there are at least 2 effects competing against each other when the number of threads increased. The positive effect is that when the number of threads went up, the parallelism improve the performance. The negative effect is that when the number of threads went up, the contention for the same cache line/memory increased. As a results, the number of threads are not linearly correlated to the performance. The performance will be dominated by dominant effect. At n = 2048, it's dominated by the contention for the same cache line so that the speedup of higher concurrency didn't increase.

Another observation is that, the more the number of threads, the more significantly the speedup correlated with the problem size. For example, for threads of 64, the speedup started from approximately 10 at n = 2048 and increased largely to approximately 37 at n = 16384. It may because that the computational performance at smaller problem size was bound by the memory bandwidth.

Besides, the more the number of threads, the later it reached the best performance. For instance, for threads of 4, it reached the upper bound at n = 4096; for threads of 8, it reached the upper bound at n = 8192; for threads of 16, it reached the upper bound at somewhere after n = 8192. It might because that

The dynamic scheduling performs similarly to static scheduling, comparing Figure 2 and Figure 3, except that the performance of higher concurrency is slightly lower than static scheduling. Specifically, the upper bound is approximately speedup = 24 for threads of 32, which is less than the upper bound speedup = 28 for threads of 32. The upper bound is approximately speedup = 32 for threads of 64, which is less than the upper bound speedup = 36 for threads of 64. It indicates that the static scheduling may be more appropriate to solve VMM problem since the workload of calculating each iteration

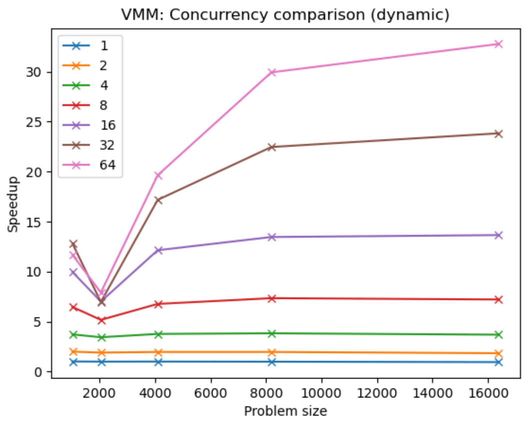

Figure 3: Dynamic scheduling: comparison of concurrency with increasing problem sizes. It's similar to the static scheduling. This figure was produced by the program `plot.py` file.

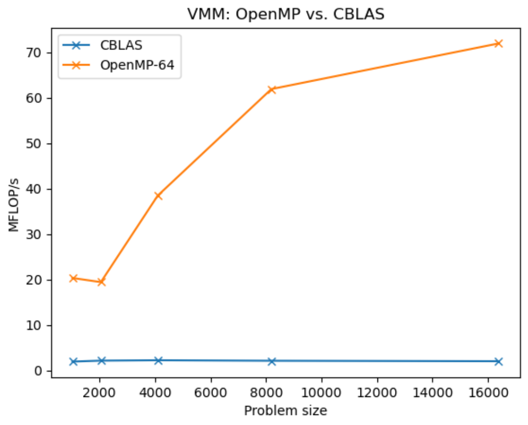

Figure 4: Comparison of OpenMP-parallel vs. CBLAS with increasing problem sizes. the MFLOP/s of OpenMP-parallel-in-64-threads was significantly higher than the CBLAS. This figure was produced by the program `plot.py` file.

of i is pretty even (e.g., inner loop of n iterations). Because of the even nature of the problem itself, static scheduling was able to divide the workload pretty evenly. While dynamic scheduling may need to allocate additional computing resources to determine the schedule which slowed the speed down slightly.

Similar to static scheduling, the speedup increased as the concurrency got larger because of parallelism. It didn't reach the ideal speedup since there were some serial section that can't be paralleled such as writing to the same y vector. Also, the performance at smaller problem size was bound by the memory bandwidth. Another similar thing is the drop at n = 2048 which we have discussed earlier.

## 3.5 Comparison of OpenMP-parallel VMM with CBLAS

The best configuration using OpenMP parallelism that I tested is P = 64 with static scheduling. Figure 4 shows the comparison between this configuration and the best performed serial algorithm CBLAS.

As we can see, the MFLOP/s of OpenMP-parallel-in-64-threads was significantly higher than the CBLAS, thanks to the parallelism. In n = 16384, the MFLOP/s of OpenMP-parallel-in-64-threads was approximately 64 times of the MFLOP/s of CBLAS because 64 threads were doing the calculation at the same time. Smaller n didn't reach that peak because the arithmetic intensity (FLOP/bytes) was lower. In another words, it's bounded by the memory bandwidth so that the CPU utilization was lower with smaller size.

## 3.6 Overall Findings and Discussion

The comparison between CBLAS and OpenMP-parallel version are not quite fair since CBLAS was using one core whereas OpenMP-parallel was using multi-cores. Therefore, the theoretical peak for different cores are not the same.

As I described in the above subsection, the ideal speedup of concurrency P should be P and the ideal run time of concurrency P should be T/P (T is the time using 1 thread). It didn't reach the ideal P since there were some serial sections that can't be paralleled. For example, different threads may write to the same y vector at the same time. Another possible reason is that the run time is determined by the run time of the slowest thread.

## References

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