Parallelization of vector-matrix multiply (VMM) using OpenMP on modern multi-core CPU architectures
Assignment #3, CSC 746, Fall 2022

Sanket Naik*
SFSU

ABSTRACT

Vector Matrix Multiplication is an operation that is useful in various fields such as Machine Learning and is even used in game engines. In this assignment, we study the performance of a basic vector matrix multiplication algorithm and how we can optimize it by using the OpenMP library to parallelize the program. We also try different thread scheduling techniques and run the program with different levels of concurrency. Finally, we compare our parallel implementation to the CBLAS dgemm implementation.

1 INTRODUCTION

The goal of this assignment is to understand how the use of the OpenMP library can help us to make our code run in parallel on modern multi-core CPU architectures. The OpenMP library makes it easy to utilize multiple cores and threads found on modern CPUs. We also test varying levels of concurrency and different thread scheduling techniques to understand which combination provides the best performance. Finally, we compare the performance of our program with that of the CBLAS reference implementation which is the C translation of the FORTRAN Basic Linear Algebra Subprograms (BLAS).

The problem being studied

- We are trying to study the impact of parallel processing on the runtime of our program for the vector matrix multiplication task.
- We look at different levels of concurrency and different thread scheduling methods to understand the speedup that we can achieve by running a task in parallel.
- We compare the parallel program with the CBLAS reference implementation which is highly optimized.

The approach used for studying this problem

- We implement 2 Vector Matrix Multiplication. The first program is the basic implementation of vector matrix multiplication. The second program uses OpenMP to run the process on multiple threads.
- We run the parallel program using 2 different thread scheduling methods and on 7 different levels of concurrency and note the runtime for different problem sizes.
- We look at the speedup that can be achieved by running the task on multiple threads and how increasing the number of threads affects the speedup.

*email:snaik@sfsu.edu

Main Results:

- We find that the basic vector matrix multiplication has a relatively stable performance and the CBLAS implementation easily outperforms it for large problem sizes.
- We also find that the parallel implementation performs extremely well but for higher levels of concurrency, there is a ramp up which impacts the performance for the smaller problem sizes in our dataset.
- We also observe that we do not achieve true linear speedup and the speedup generally ramps up as the problem size increases.

2 IMPLEMENTATION

Two programs were implemented in this assignment in order to study how OpenMP can be used to convert a serial process into a parallel process. The programs that were implemented were - (1) Basic Vector Matrix Multiplication and (2) Vector Matrix Multiplication with OpenMP. We also compare these programs to (3) the cblas_dgemm function which is the matrix multiplication implementation found in the CBLAS library - a C translation of the FORTRAN Basic Linear Algebra Subprograms (BLAS).

2.1 Part 1 - Basic Vector Matrix Multiplication

This program implements basic vector matrix multiplication of an N-by-N matrix with a N-by-1 vector and we store this in a N-by-1 vector. The equation below shows the basic operation that is performed.

$$y[i] := y[i] + A[i,j] * x[j] \tag{1}$$

The logic of the program was as shown below in Listing 1.

```
void my_dgemv(int n, double* A, double* x, double*
     y) {
  // Implementation of Basic Vector Matrix
   Multiplication
  for (int i = 0; i < n; i++){
     for (int j = 0; j < n; j++){
        // Perform dgemv operation
        y[i] += A[ (i*n) + j ] * x[j];
     }
  }
}
```

Listing 1: Basic Vector Matrix Multiplication

2.2 Part 2 - Parallel Vector Matrix Multiplication with OpenMP

This program follows the same logic as that of the basic vector matrix multiplication but here we make use of the OpenMP library to run the program in parallel. We make use of OpenMP pragmas to set up loop parallelism for with dgemv operation.

The logic of the program was as shown below in Listing 2.

```
10 void my_dgemv(int n, double* A, double* x, double*
       y) {
11   // Enable OpenMP parallelization over rows
12   #pragma omp parallel for
13   for (int i = 0; i < n; i++){
14      // Enable OpenMP parallelization with
      reduction for y[i]
15      #pragma omp parallel for reduction (+:y[i])
16      for (int j = 0; j < n; j++){
17         // Perform dgemv operation
18         y[i] += A[ (i*n) + j ] * x[j];
19      }
20   }
21 }
```

Listing 2: Parallel Vector Matrix Multiplication: Uses OpenMP to run the dgemv operation in parallel

2.3 Part 3 - CBLAS

This program simply calls the cblass_dgemm function which calculates the result of vector matrix multiplication and stores it in the relevant pointer locations.

The program was as shown below in Listing 3.

```
22 void my_dgemv(int n, double* A, double* x, double*
       y) {
23   double alpha=1.0, beta=1.0;
24   int lda=n, incx=1, incy=1;
25   cblas_dgemv(CblasRowMajor, CblasNoTrans, n, n,
      alpha, A, lda, x, incx, beta, y, incy);
26 }
```

Listing 3: CBLAS Vector Matrix Multiplication: Calls the cblas_dgemm function to calculate the result

3 Evaluation

The programs were compiled and run on a KNL node on CORI and the runtimes for different problem sizes were recorded.

3.1 Computational platform and Software Environment

All tests were run on a KNL node on CORI and its specifications can be found below

- **Processor**: Intel Xeon Phi Processor 7250
- **Number of Cores**: 68
- **Number of Threads**: 272 (4 per core)
- **Clock Rate (GHz)**: 1.4GHz
- **L1 Cache**: 64 KB per core
- **L2 Cache**: 1MB (shared between 2 cores)
- **L3 Cache**: 8MB
- **Memory Size**: 96 GB DDR4 + 16 GB MCDRAM
- **Memory Bandwidth**:
  - **MCDRAM**: >460 GB/s
  - **DDR4**: 102 GiB/s
- **OS Version**: SUSE Linux Enterprise Server 15 SP2

The GNU compiler was used to compile all programs.

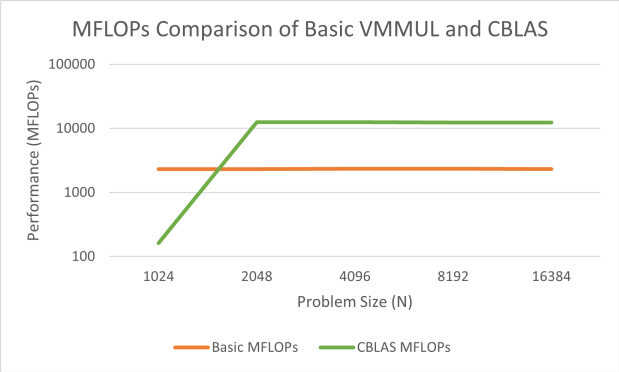

Figure 1: A plot showing the MFLOPs comparison between Basic VMM and CBLAS VMM

3.2 Methodology

All programs followed the same testing methodology. Each program was compiled and run with different problem sizes that were included in the bench-marking program.

Performance metrics: The main performance metric observed for these programs was the run-time of vector matrix multiplication. This was calculated using the chrono library by placing two time points around the function call and calculating the difference between these times.

Experimental design: All program sizes were tested by multiplying an N-by-N matrix with a N-by-1 vector where N has the following values - [1024, 2048, 4096, 8192, 16384]. For the OpenMP implementation, two different thread scheduling techniques were used as follows - Static and Dynamic. For each thread scheduling method, the program was run with the following 7 levels of concurrency - [1, 2, 4, 8, 16, 32, 64].

3.3 Part 1: Basic Vector Matrix Multiplication

The program was run using the problem sizes described in 3.2. The problem sizes were included in the bench-marking code and the run-time was noted. The CBLAS program was also run to form a comparison between the two implementations. The output of the program along with the runtime can be found in the Github repository here

Based on the runtime and the problem size, we can find the compute power in MFLOPs by using the following formula.

$$Performance(MFLOPs) = \frac{N^2}{runtime} x \frac{1}{100000} \quad (2)$$

The comparison of the performance of the two implementations was as shown in Fig. 1. As we can see from the plot, the CBLAS VMM program performs better than the basic implementation for most problem sizes and it's performance ramps up quickly from N=1024 to N=2048 and remains fairly constant for the remaining problem sizes. On the other hand, the Basic VMM's problem size remains relatively constant at 2330 MFLOPs for all problem sizes and only drops slightly as the problem size increases.

3.4 Part 2: OpenMP Parallel Vector Matrix Multiplication

The OpenMP Parallel Vector Matrix Multiplication program was run using two different thread scheduling and each thread scheduling technique was run with 7 levels of concurrency as described in 3.2.

Static Thread Scheduling

In the first job, the program was run with static thread scheduling on all problem sizes and for all 7 levels of concurrency. The output

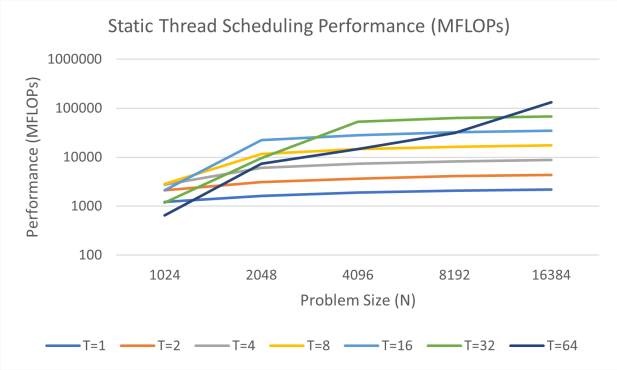

Figure 2: A plot showing the MFLOPs comparison for all 7 levels of concurrency with Static thread scheduling

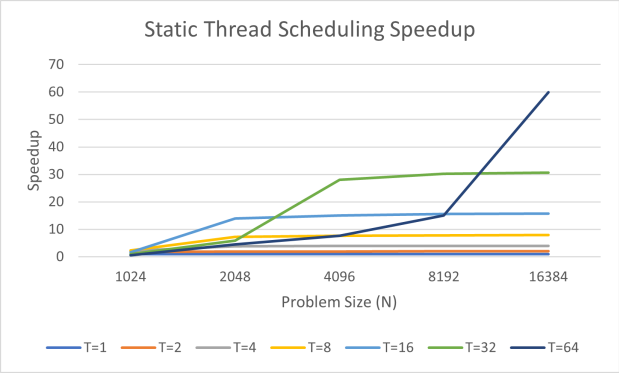

Figure 3: A plot showing the speedup for T threads when using Static thread scheduling

and runtime data for the program can be found in the Github repository here. Based on the runtime of the program, we can find the performance of the program for all 7 levels of concurrency. Fig. 2 shows the performance in MFLOPs for the program. We can also generate a speedup graph which shows how much faster the parallel code performs as compared to the serial version. Fig. 3 shows the speedup graph which is a measure of how well our program performs as the number of threads increases.

Dynamic Thread Scheduling

In the second job, the program was run with static thread scheduling on all problem sizes and for all 7 levels of concurrency. The output and runtime data for the program can be found in the Github repository here. Based on the runtime of the program, we can find the performance of the program for all 7 levels of concurrency. Fig. 4 shows the performance in MFLOPs for the program. We can also generate a speedup graph which shows how much faster the parallel code performs as compared to the serial version. Fig. 5 shows the speedup graph which is a measure of how well our program performs as the number of threads increases.

Based on both speedup charts, we can see that as we increase the number of threads, the performance increases proportionally but we never achieve the ideal speedup (in this case ideal speedup = number of threads). We can also see that as the number of threads increases, it takes longer to reach anywhere close to the ideal speedup. As we can see very clearly from the case when T=64, the speedup only gets close to the ideal speedup in the last problem size (N=16384). We can also see from the speedup and performance charts for both static and dynamic thread scheduling that the dynamic thread scheduling does perform slightly better but the performance difference is not very large.

3.5 Part 3: CBLAS Vector Matrix Multiplication

The CBLAS implementation of matrix multiplication was used as a reference baseline. The cblas_dgemm function is present in the CBLAS library which is a C translation of the FORTRAN Basic Linear Algebra Subprograms (BLAS). This implementation of vector matrix multiplication is a serial implementation and has been thoroughly optimized. The output of the program and its runtime can be found in the Github repository here. We can compare the performance of this program to our best performing parallel program (dynamic thread scheduling with 64 threads) to visualize how well the CBLAS implementation performs even if it is a serial implementation. Fig.6 shows the comparison of the performance of CBLAS and OpenMP parallel implementation of VMMUL. We can see from the performance chart that the CBLAS implementation

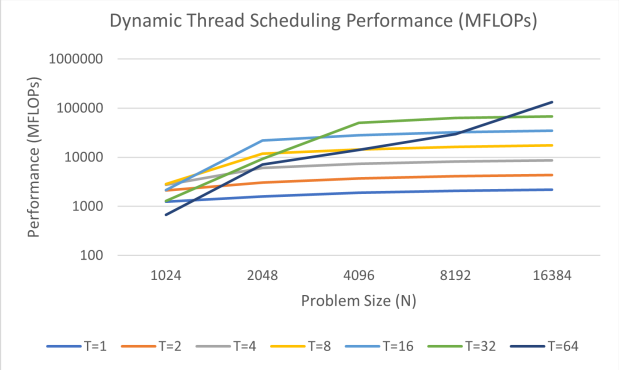

Figure 4: A plot showing the MFLOPs comparison for all 7 levels of concurrency with Dynamic thread scheduling

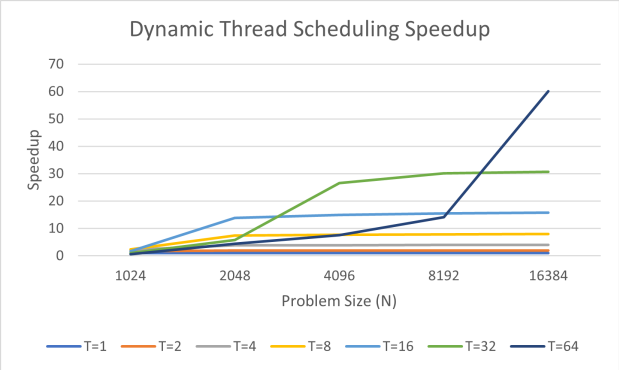

Figure 5: A plot showing the speedup for T threads when using Dynamic thread scheduling

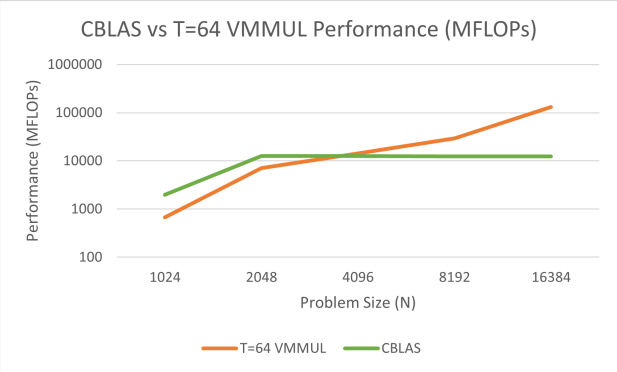

Figure 6: A comparison of the performance (MFLOPs) of CBLAS and Dynamic thread scheduling VMMUL with 64 threads

performs favorably and even performs better until the problem size (N) is 4096 and after that the parallel program performs much better. This chart also shows clearly that in spite of being a serial program, the CBLAS implementation is highly optimized and performs well against the parallel implementation.

3.6 Findings and Discussion

Serial CBLAS vs. OpenMP-parallel

From the performance comparison we can clearly see that for very large problem sizes, the OpenMP-parallel program performs much better than the serial CBLAS program. I believe that this is not a fair comparison because there are parallel implementations of CBLAS available which may perform much better than our parallel implementation. This is because CBLAS uses specialized instructions to perform matrix multiplications and this is evident from our performance charts.

Parallel Performance and Scalability

As we can see from the speedup chart, our program does not exhibit true linear speedup. Our program ramps up towards the ideal speedup value but we do not achieve the ideal speedup in any scenario. For example, when T=64, our initial speedup is actually slower than that of T=32 or T=16 and the program only reaches anywhere close to the ideal speedup at the largest problem size of N=16834. This is because our program may be facing other bottlenecks such as memory bandwidth or ineffective use of the cache. To better analyze why we do no achieve a true linear speedup we can look at other hardware performance counters which will give us a true reading of the actual performance metrics. There may also be a load imbalance or a synchronization cost associated with running with so many threads in parallel.