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

Elliot Kang*
SFSU

ABSTRACT

The focus of the study is to parallelize VMM using OpenMP on modern multi-core CPU architectures. We use timing instrumentation to measure compute times and control the problem size to use as data points in our analysis of performance across different experiments. We found that parallelization helps with larger problem sizes, but there is a nontrivial overhead that comes with parallelized programs.

1 INTRODUCTION

Vector matrix multiplication (or VMMs) are foundational blocks of numerical analysis using linear algebra. Moreover, many mathematical problems can be reduced to matrix operations to feasibly compute approximations for pressing scientific problems. For example, probabilistic models can be lifted into feature space (and thus represented as matrices or vectors) if they can be expressed as an inner product of vectors.

When measuring computational performance, there are generally a minimum of two data axes one can easily measure and change, respectively: compute time and problem size. Especially when measuring the performance of matrix operations, one can use these measurable quantities to make inferences about underlying compute architecture. In addition, the number of FLOPs required for a VMM is provably fixed.

Parallelization is a powerful technique that helps programmers utilize their computer processors to the fullest extent. However, there is a nontrivial overhead where parallelization only becomes effective after a problem size lower bound. To put it concisely, there are only so many things to parallelize and there is some additional complexity introduced when one parallelizes programs. Lastly, we test different thread scheduling schemes for parallelized VMMs.

2 IMPLEMENTATION

We briefly go over vector matrix multiplication in the C++ language. Compact code statements are used, as well as comments to clarify points about matrix operations, as semantics may differ depending on choice of data structures for matrices. Each subsection contains the implementation details, as well as the objective of the implementation.

Additionally, for the threaded VMMs, we try two different thread scheduling schemes: static and dynamic.

2.1 Basic Vector Matrix Multiplication

The objective for this implementation was the implement a basic vector matrix multiplication in the C++ language. Using the basic vector matrix multiplication implementation, we can then find places to optimize matrix multiply with parallelization.

*email:kkang1@mail.sfsu.edu

We use two for-loops because the linear system of equations for VMM is solved as follows: For each row of A, we compute the dot product with the Nx1 vector B. Our problem size is N=[1024, 2048, 4096, 8192, 16384]. This is all done in row-major format. See Listing 1.

```
1   for (int i = 0; i < n; i++){
2       for(int k = 0; k < n; k++){
3        //Row major order (VMM):
4        //prod[i] += A[i][k] * B[k]
5        y[i] = y[i] + A[i*n + k] * x[k];
6       }
7   }
```

Listing 1: Basic Vector Matrix Multiplication.

2.2 Parallelized Vector Matrix Multipication

The objective for this implementation is to use openMP directives to use threads and use parallelism with p = [1,2,4,8,16,32,64] threads to speed up computation. Optimally, 32 threads would result in a 32x speedup and 64 threads would result in a 64x speedup.

After we specify the parallel region with the first pragma statement, we specify a directive specifically for the two for-loops in Listing 2. This tells the program to parallelize the loops within the parallel region. With modern C++, the program can infer some of the private and shared variables inside the execution code (like integer i, k).

```
8         #pragma omp parallel
9         #pragma omp for
10        for (int i = 0; i < n; i++){
11            for(int k = 0; k < n; k++){
12             //Row major order (VMM):
13             //prod[i] += A[i][k] * B[k]
14             y[i] = y[i] + A[i*n + k] * x[k];
15            }
16        }
```

Listing 2: Parallelized VMM.

3 EVALUATION

For all the experiments, we run over problem sizes N=[1024, 2048, 4096, 8192, 16384]. For the parallelized vector matrix multiplication, we run over the problem sizes and for the following thread counts for each problem size (thread maximum p=[1, 2, 4, 8, 16, 32, 64]). And for each experiment, we benchmark against the BLAS implementation of vector matrix multiplication.

So for the basic vector matrix multiplication, we just run over the problem size. For parallelized vector matrix multiplication, we run over both the problem size and thread count.

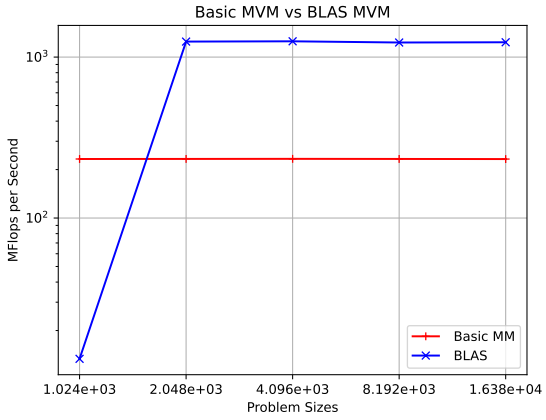

Figure 1: Comparison of Basic VMM vs BLAS VMM

3.1 Computational platform and Software Environment

Intel Xeon Phi Processor 7250, 1.4 GHz, 96GB DDR4, 16GB MCDRAM, 2.12MB/34MB L1/L2 cache, SUSE Linux Enterprise Server 15 SP2.

We used the g++ compiler for the C++ language. We use C++ version 17 and g++ version 7.5.0. Zero compiler optimizations were used.

3.2 Methodology

We measure elapsed time from instrumentation code we added around the main computational code. This is our measured performance metric. We can then derive the FLOPs per second. We can measure compute time, and we know how many FLOPs (in our case, IOPs) it takes to compute VMMs. So we can compute FLOPs per second by using elapsed time.

For all the experiments, we run over problem sizes N=[128, 256, 512, 1024]. For the blocked matrix multiplication, we run over the problem sizes and for the following thread counts for each problem size (thread count p=[2, 16, 32, 64]).

We benchmark basic VMM against the BLAS implementation. For parallelized VMMs, we chart the realized speedup, and static vs. dynamic thread scheduling is charted separately.

3.3 Basic VMM vs CBLAS

See Section 2 for implementation details. Refer to Fig. 1 for results.

The performance for basic vector matrix multiplication stayed constant as the problem size increased. However, for BLAS, the performance slightly increases with an increase in problem size. There is no decrease in basic VMM performance over larger problem sizes, probably because VMM is easier to allocate memory-wise than full matrix multiplication.

The performance of basic matrix multiplication is much worse than BLAS in terms of MFLOPs per second.

3.4 OpenMP-parallel VMM

See Section 2 for implementation details. Results are in Figure 2 for OpenMP-parallel VMM with static scheduling.

We see that for static thread scheduling, speedup is not linear but gets better over time (never truly gets linear speedup though). However, there is enough speedup to suggest that parallelization is worth doing in this situation (for VMMs, that is). The non-linear speedups (64x speedup for 64 threads) suggest that there is some overhead for initializing the threads and parallelizing the program.

For dynamic thread scheduling, we see that for smaller problem sizes, the speedup is not as good. Also, the performance of

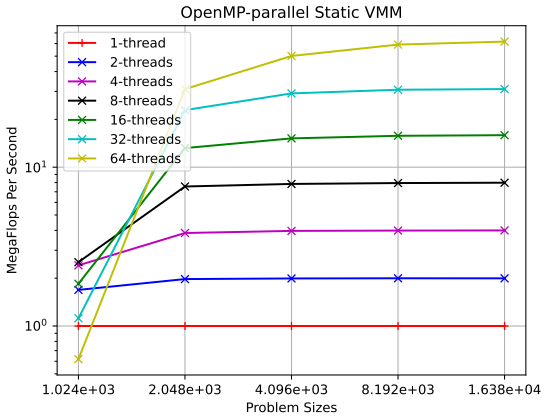

Figure 2: Comparison of parallelized VMM with Static Scheduling

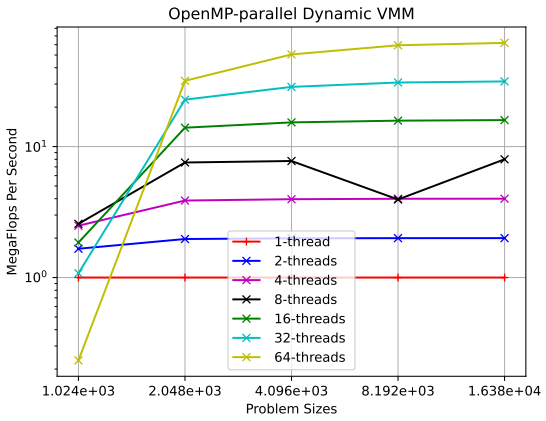

Figure 3: Comparison of parallelized VMM with Dynamic Scheduling

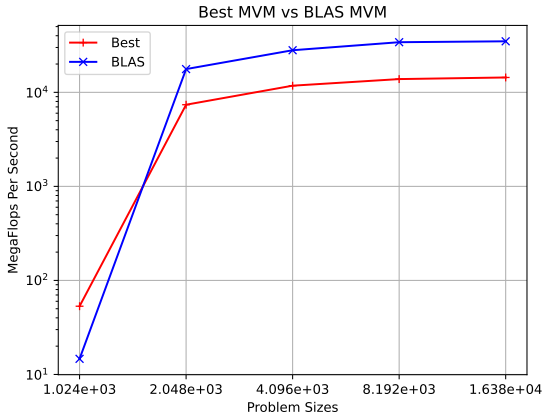

Figure 4: Best OpenMP-parallel VMM vs BLAS VMM

a dynamic thread scheduler can be erratic. Assuming there is no significant difference between static and dynamic thread scheduling, one should consider using static thread scheduling in practice. Comparing both static and dynamic thread scheduling suggests that while dynamic thread scheduling does provide some optimization benefit for threads, there is additional overhead that is noticeably large in (small problem size, many threads) scenarios.

3.5 Best OpenMP-parallel VMM vs BLAS VMM

See Section 2 for implementation details. Refer to Fig. 4 for results.

We identified that for large problem sizes, 64-thread parallelization with dynamic thread scheduling was best for VMM. Comparing this against the vectorized BLAS VMM, concurrency helps make VMMs faster to calculate, but they still do not compare to the BLAS vectorized implementation.

Concurrency seems to make computation many multiples faster, while vectorization makes computation orders of magnitude faster.

3.6 Findings and Discussion

Our hypothesis was that parallelized vector matrix multiplication would help reduce the computation time of vector matrix multiplication and make it comparable to serial CBLAS. Our hypothesis was wrong. However, we did not get close to the performance of the BLAS optimization routine. So there must be something that BLAS does under the hood to make computation orders of magnitude faster. In a way, concurrency is only multiples faster. Ground-breaking performance gains are in the orders of magnitude.

BLAS vector matrix multiplication is not the same as paralllelized vector matrix multiplication. One uses vectorization while the other uses concurrency to speed up computation, so it is not a fair comparison.

Looking back at the speedup charts for parallelized VMMs (both static and dynamic thread scheduling), we can conclude that we never got perfectly linear speedup. The reason why true linear speedup is impossible is because there is a certain amount of overhead involved with initializing and dividing work among threads, then coordinating efforts among threads. However, for infinitely large problem sizes, we can probably get almost true linear speedup (since each workload is so large).