Comparison of Vector-Matrix Multiplication methods CP #3, CSC 746, Fall 2023

Shubh Pachchigar[*]
SFSU

ABSTRACT

This study investigates the relationship between various methods for vector matrix multiplication on both single-core and multi-core systems, and their impact on performance and potential speedup. The focus of this study centers on the traditional task of vector matrix multiplication, performed across increasingly larger problem sizes. Throughout this study, four distinct implementations of vector matrix multiplication are thoroughly analyzed. The first implementation represents the elementary approach to vector matrix multiplication, while the second involves automatic vectorization performed by the compiler on a single core, aiming to utilize complex instructions for higher data throughput. The third implementation is a parallel version in which the same work, is distributed across multiple cores, thereby reducing program runtime and efficiently utilizing all available cores. The fourth implementation utilizes a vendor-provided CBLAS code, serving as a benchmark reference for comparison. Performance assessment is conducted using MFLOPS as the primary metric and speedup as a secondary metric for the parallel version of vector matrix multiplication. This approach allows for a precise and quantitative evaluation and comparison of the four implementations. Our findings reveal that the utilization of parallelization and vectorization techniques does indeed yield performance improvements. However, it is noteworthy that the performance falls short of the efficiency achieved by the CBLAS code for some of the smaller problem sizes but for larger sizes we observe parallel version dominating.

1 INTRODUCTION

In the realm of computer algorithms, achieving speedup often arises from the exploration of a more efficient algorithms with lower time complexities. While this form of speedup is highly valued, another avenue for enhancing performance lies in the conscious utilization of underlying hardware resources, resulting in overall speed improvements, even while maintaining the original time complexity. A classic problem that has been studied thoroughly for an extended duration is vector matrix multiplication. This interest can be attributed to its direct relevance in various modern computing systems and its dependence on the underlying hardware. Speedup in this context emerges when we understand and leverage all available hardware resources. This study aims to gain a comprehensive understanding of how vector matrix multiplication performs under different methodologies.

The initial implementation is basic vector matrix multiplication, employing a polynomial-time algorithm that utilizes a double nested loop structure to compute the final result. In the second implementation, we transform the code into a vectorized version. Here, we make slight adjustments to facilitate compiler-driven vectorization, particularly within the inner loop of the nested block so that the compiler uses complex instructions as described in Section 2.2. The third implementation harnesses OpenMP to distribute the workload across an increasing number of threads. It's noteworthy that the amount of computations remains unchanged; our objective here is to improve the overall execution speed by breaking the problem into smaller, independent chunks that can be proccessed independently. The final implementation utilizes the CBLAS version, which relies on a pre-existing library code and serves as our reference benchmark. This code incorporates various techniques, including cache blocking and loop optimizations. A comprehensive explanation of this approach is provided in Section 2.3.

Subsequent sections will highlight the significant differences in performance among these four implementations. Notably, the naive algorithm exhibits a computational bottleneck, leading to a pronounced reduction in MFLOPS. In contrast, both the vectorized and OpenMP-based parallel code display a remarkable capacity to fully harness computational resources. While the reference CBLAS code outperforms the others for smaller problem sizes, as the problem size increases, we observe the OpenMP version surpassing all, underscoring the importance of parallelism on multicore systems. A takeaway from this study is the efficacy of parallelism in enhancing performance relative to alternative algorithms for larger problem sizes. This investigation delves into the trends associated with various methods of matrix multiplication and algorithmic execution, offering valuable insights into computational efficiency and the potential for speedup.

[*]email:spachchigar@sfsu.edu

2 IMPLEMENTATION

This section delves into the C++ implementation to solve this specific problem. It's worth noting that in three out of the four scenarios, we utilize the same underlying code structure, but they diverge in how the computations are executed and how memory is accessed. The first implementation employs a basic vector matrix multiplication approach, utilizing a straightforward, naive algorithm to obtain the final matrix. In the second implementation, we employ the same code structure but leverage certain compiler optimizations/magic, to vectorize the inner loop. This vectorization significantly enhances overall program efficiency by reducing data access overhead. The third code iteration once again utilizes the same underlying code structure but introduces an OpenMP directive. This directive effectively parallelizes the outer loop, distributing the computational workload among a specified number of threads. Lastly, the fourth subsection serves as a reference benchmark against which we will compare the other three implementations.

2.1 Basic Vector Matrix Multiplication

This implementation details the basic vector matrix multiplication using two nested loops without any optimizations. In this approach, as illustrated in Listing 1, we use a one-dimensional array with a size of N^2, where N denotes the problem size, and another array of size N. All matrices are stored in row-major order, and the multiplication is executed accordingly. The process involves extracting a row from the 2D matrix A and performing multiplication with the corresponding 1D matrix X. This operation is iterated for all rows, resulting in a dot product where N values are multiplied, and N values are subsequently added. The use of the expression $i*n+j$ for the A array pointer allows it to traverse row-wise, where each column for a specific row is accessed first. However, even when N is relatively small, this approach may not fully leverage the available computational resources, therefore we notice that with increasing problem size we observe a gradual decline in performance.

```c
void my_dgemv(int n, double *A, double *x, double
     *y)
{
   for (int i = 0; i < n; i++)
   {
      double dot = 0.0;
      int k = i * n;
      for (int j = 0; j < n; j++)
      {
         dot += A[k + j] * x[j];
      }
      y[i] += dot;
   }
}
```

Listing 1: Basic Vector Matrix Multiplication: performs dot product of each row of A with all of x

2.2 Vectorized Matrix Multiplication

This implementation elaborates on vector matrix multiplication utilizing two nested loops, while also benefiting from compiler-applied optimizations. In this implementation, the code structure remains the same as shown in Listing 1. However, we've enabled a series of vectorization flags. Consequently, when the compiler translates this code into assembly, it transforms it into complex vector instructions known as AVX instructions. These AVX instructions are specialized and operate on vector registers, which are typically larger than ordinary registers. To facilitate effective vectorization, the compiler requires the loop to be structured in a specific way. For instance, in this case, the use of a single loop variable for all matrix access causes the inner loop to vectorize. This means that during the matrix multiplication computation, only one loop variable j is used while K (which remains constant throughout the inner loop execution), is allowed. In the outer loop, computation exclusively utilizes the loop variable i, with the more intensive computations taking place internally. Through vectorization, complex assembly instructions can load multiple data elements in a single fetch operation and perform operations on all of them concurrently. This results in a substantial reduction in overall runtime. However, it's important to note that this approach still falls short of fully harnessing the available computational power, as it primarily relies on complex instructions to minimize elapsed time.

2.3 Parallel Matrix Multiplication

This implementation shows the parallel version of matrix multiplication, as presented in Listing 2. It follows the same code structure previously described in Listing 1, with a key difference being the introduction of a pragma directive to parallelize the outer loop, provided by OpenMP. OpenMP is an API designed to support shared memory multiprocessing programming across various hardware platforms, including multi-core processors and multi-processor systems.

In this implementation, we employ a specific pragma directive, namely '`omp parallel for`', which informs the compiler that the subsequent section, constituting a 'for' loop, should be parallelized. This results in the distribution of work evenly among the available threads. We execute this implementation across different concurrency levels, as elaborated upon in Section 3, yielding interesting outcomes. Notably, we use '`omp parallel for`' exclusively on the outer loop, meaning that all rows of matrix A are evenly divided among a specified number of threads. Each thread then independently executes a copy of the inner loop code sequentially within its own context and updates the final array y without any race conditions. Alternative methods explored involved incorporating the '$collapse(2)$' clause and the '$reduction$' clause, preliminary analysis indicated that both had a negative impact on performance and speedup. Further insights into these experiments, including a comparison of different concurrency levels corresponding to varying numbers of threads and their resulting speedup, are detailed in Section 3.4.

```c
void my_dgemv(int n, double *A, double *x, double
     *y)
{
#pragma omp parallel for
   for (int i = 0; i < n; i++)
   {
      double dot = 0.0;
      int k = i * n;
      for (int j = 0; j < n; j++)
      {
         dot += A[k + j] * x[j];
      }
      y[i] += dot;
   }
}
```

Listing 2: Parallel vector matrix multiplication: divides the rows of matrix A among certain number of threads

2.4 CBLAS reference

This implementation uses a library function from CBLAS header file and invokes the function cblas dgemm(). Because of all matrices are stored as one dimensional array in row major order we pass in string named CblasRowMajor to indicate row major order and CblasNoTrans tells that no transpose should be performed on A and B. All of this is shown in Listing 3. Here alpha and beta which indicate scaling of matrices are set to 1 and N represent the size of matrices. The cblas dgemm() function is part of the Basic Linear Algebra Subprograms (BLAS) library, which is a set of low-level routines for performing common linear algebra operations such as vector addition, scalar multiplication, dot products, linear combinations, and matrix multiplication. The BLAS library is widely used in scientific computing and machine learning applications to speed up matrix operations. The cblas dgemm() function uses a variety of techniques to speed up matrix multiplication, including loop optimization techniques and cache-blocking. Loop optimization techniques involve optimizing the order of nested loops and with the reference BLAS implementation across varying problem sizes and block sizes. This examination sheds light on the impact of different factors on performance.

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

Listing 3: CBLAS reference implementation

3 Evaluation

This section builds upon the implementations described in Section 2 and conducts a series of experiments aimed at assessing their impact on computational performance and speedup. These experiments are carried out across an array of increasing problem sizes. The initial segment outlines the essential environment prerequisites for replicating the experiments, while the following sections gives comprehensive descriptions of the experiments themselves.

The first experiment compares the basic implementation detailed in Section 2.1 with the vectorized implementation discussed in Section 2.2, employing CBLAS as a reference benchmark. The second experiment delves into the exploration of varying concurrency levels for the implementations elucidated in Section 2.3 and measures the speedup as compared to the basic serial code. Subsequently, the third experiment takes the most proficient concurrency level identified in the second experiment and draws a comparison with the reference CBLAS implementation.

3.1 Computational platform and Software Environment

The system in use is Perlmutter at NERSC, equipped with hardware specifications including a 2 MiB L1 cache, 32 MiB L2 cache, and 256 MiB L3 cache. It runs on the SUSE Linux Enterprise Server 15 SP4 operating system, version 5.14.21-150400.24.69 12.0.74-cray shasta c, and is powered by an AMD EPYC 7763 Milan CPU capable of clock rates ranging from 1.5 GHz to 2.45 GHz. The system has 256GB of DRAM, utilizes the GCC 11.2.0 compiler from Cray Inc. with default optimization levels set to O3, and employs CMake version 3.20.4 alongside GNU Make version 4.2.1, built for x86 architecture.

3.2 Methodology

In this section, we will subject the implementations discussed in Section 2 to a series of tests and comparisons with a reference implementation. Each of these implementations will be executed with varying problem sizes, [1024, 1024, 2048, 4096, 8192, 16384]. Each problem size corresponds to the dimensions of a square array, where each 'N' implies an 'N*N' matrix. All four implementations will undergo testing with randomly generated values sampled from a Gaussian distribution. The second experiment executes the implementation described in Section 2.3 on varying level of concurrency, [1, 4, 16, 64]. Following the completion of these experiments, we will evaluate and compare their performance using the MFLOPS metric, which measures millions of floating-point operations per second. This metric allows us to assess how effectively both the hardware and software leverage their capabilities to achieve optimal performance. Another metric we use is the speedup which is used to measure the performance of the parallel implementation. We will conduct these assessments through three distinct experiments. To measure the elapsed time of the matrix multiplication, we will employ the chrono timer library. To calculate MFLOPS, we require two essential components: the total number of arithmetic operations, which depends upon the problem sizes, and the elapsed time of the code. To calculate the speedup, we use elapsed runtime from the serial code and divide it by the runtime of the parallel code. The formula used to calculate MFLOPS and Speedup is described below:

$$\text{MFLOPS} = \frac{\textit{num of arithmetic ops}}{\textit{elapsed time}} = \frac{2N^2}{\textit{elapsed time}}$$

$$\text{Speedup} = \frac{\textit{elapsed time serial code}}{\textit{elapsed time parallel code}}$$

3.3 Experiment 1: Comparing Basic VMM, Vector VMM and CBLAS

This experiment involves a comparison between the basic implementation and the vectorized implementation, both in relation to the reference CBLAS as shown in Fig. 1. This comparison clearly illustrates that CBLAS surpasses both implementations, but its performance diminishes as the problem size increases. The basic implementation, owing to its simplistic approach that underutilizes underlying hardware capabilities, yields less favorable results but still we observe that MFLOPS remains relatively constant as the problem size escalates. This phenomenon may be attributed to the likelihood that, despite the absence of complex instructions or parallel code, one-dimensional vectors are brought into cache, following to the principle of locality. Consequently, the cost of data transfers diminishes, leading to reduced elapsed time. For the vectorized implementations, the manual activation of compiler flags results in significantly improved performance compared to the basic implementation. In addition to potential cache optimizations, vectorization leverages complex AVX instructions, which reduces data fetching operations by pulling multiple data elements in one fetch call. The behavior of the CBLAS reference code appears strange. Initially, it has a high MFLOPS for smaller problem sizes, but subsequently experiences a sharp decline, eventually stabilizing just above the vector VMM (Vector Matrix Multiply) line. The exact cause of this performance decay remains unknown and would require further investigation into the vendor code, which lies beyond the scope of this study.

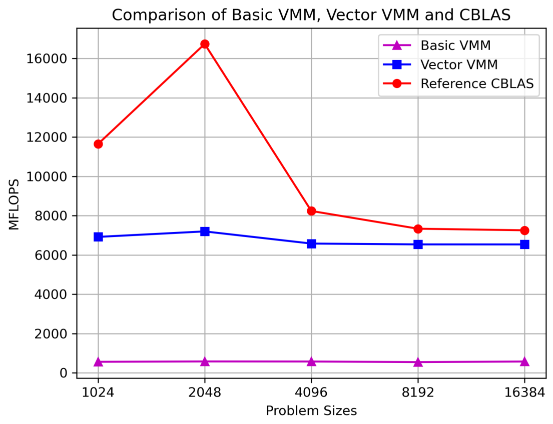

Figure 1: Comparison of MFLOPS for Basic VMM, Vector VMM and CBLAS with increasing problem sizes.

3.4 Experiment 2: Comparing OpenMP concurrency levels

This experiment as shown in Fig. 2 shows a comparison of the implementation outlined in Section 2.3 across varying levels of concurrency. In this context, concurrency is determined by the number of threads, and OpenMP permits the adjustment of the environment variable `OMP NUM THREADS`, ranging from 1 to 64 threads.

When the number of threads is set to one, we observe minimal deviation from the serial version's performance, albeit some slight improvements for certain problem sizes, indicative of the optimizations achievable through the OpenMP library. Moving to the next level, employing OpenMP with four threads, we observe nearly a twofold increase in speedup for the initial problem size, followed by a linear progression. However, as the problem size further escalates, the speedup growth decelerates, likely due to the influence of Amdahl's law, which introduces flat regions in the chart. The third level, with 16 threads, presents an unexpectedly irregular speedup chart. Multiple factors, including optimal cache utilization and load balance among threads, may contribute to this behavior. However, a peculiar drop in speedup for the problem size of 2048 remains unexplained and demands further investigation. Subsequently, we observe a sharp speedup increase that stabilizes afterward. In the final level, employing 64 threads, we witness a steady, nearly linear increase in speedup up to a problem size of 8192, followed by stabilization. This notable speedup can be attributed to effective load balancing among threads. The consistent speedup observed in this highest concurrency level makes it an ideal candidate for comparison with CBLAS in the upcoming experiment.

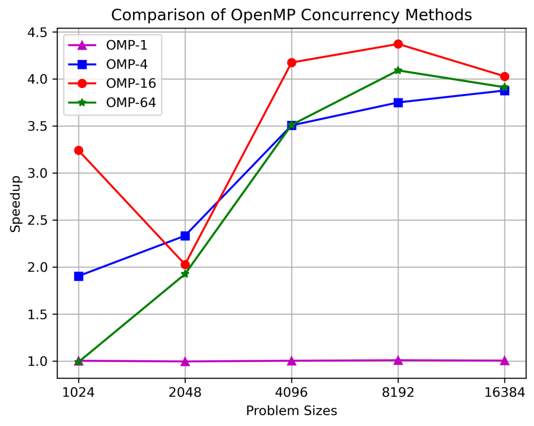

Figure 2: Comparison of speedup of different concurrency levels in OpenMP with increasing problem sizes.

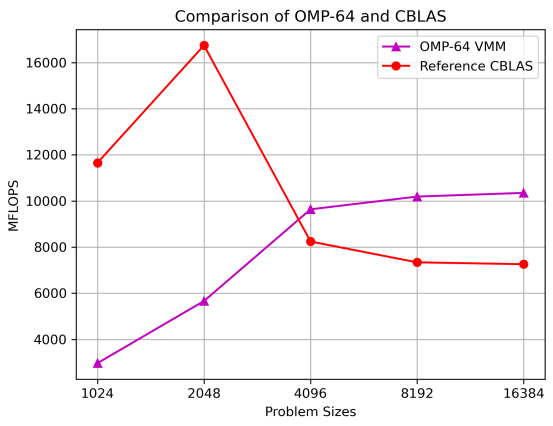

Figure 3: Comparison of MFLOPS for OpenMP with 64 threads and CBLAS.

3.5 Experiment 3

In this experiment, we compare the best-performing candidate from the previous experiment with the reference CBLAS. We have opted for the OpenMP configuration with 64 threads, primarily due to its linear speedup characteristics. When evaluating MFLOPS, we observe that it surpasses CBLAS for larger problem sizes.

The line in Fig. 3 representing OMP-64 exhibits an initial almost linear increase, scaling in proportion to the problem size growth. Even after the problem size reaches 4096, while the rate of increase slows down, it continues to gradually rise. Notably, at the 4096 problem size mark, CBLAS experiences a decline in performance, while OMP 64 maintains higher performance levels and continues to gradually increase for larger problem sizes. As for the CBLAS version, it achieves the highest MFLOPS for smaller problem sizes, with the peak occurring at a problem size of 2048. However, beyond this point, its performance declines exponentially. The linear speedup observed in OMP 64 can be attributed to the relatively linear speedup characteristics identified in the previous experiment.

3.6 Findings and Discussion

This study explores various methods for performing vector matrix multiplications and presents the results of three experiments. The first experiment demonstrates the improvement in MFLOPS

Problem Size	Basic VMM	Vector VMM	OMP-1 VMM	OMP-4 VMM	OMP-16 VMM	OMP-64 VMM	CBLAS
1024	9.07	27.06	9.09	17.26	29.39	8.96	45.56
2048	9.17	28.14	9.13	21.38	18.59	17.64	65.44
4096	8.98	25.72	9.00	31.48	37.48	31.52	32.19
8192	8.94	25.55	9.01	33.53	39.08	36.57	28.67
16384	8.93	25.54	8.96	34.63	35.98	34.94	28.35

Table 1: Percent of Peak Memory bandwidth utilization across different methods.

achieved through vectorization, although it still falls short when compared to CBLAS. The second experiment investigates the impact of different concurrency levels on speedup, while the third experiment showcases the superiority of the most stable configuration of OpenMP with 64 threads, over CBLAS for larger problem sizes.

Table **??** provides a comparison of memory bandwidth utilization as a percentage of the peak memory bandwidth of the Perlmutter system, which is 204.8 GB/sec. Assuming 8 bytes of storage for each double type, we calculate the bandwidth utilization for all implementations discussed in Section 2 . Assuming a consistent total memory access of $2N^2 + 2N$ across all implementations, we determine the percentage utilization. The highest utilization is observed with CBLAS, reaching 65.44 percent for a problem size of 2048. This is attributed to several factors, including data alignment, increased cache usage, vector instructions, and loop optimizations, although further investigations is needed. Conversely, the lowest utilization is by the basic VMM for the largest problem size, as expected due to its naive approach leading to deteriorating performance.

Here are some insights on each implementation: The basic implementation maintains a stable percentage of peak bandwidth usage and the vector implementation initially exhibits higher utilization that gradually stabilizes while the OMP-1 performs similarly to the basic VMM. OMP-4 increases steadily, ultimately surpassing CBLAS. This trend is also observed for OMP-16 and OMP-64. Notably, OMP-16 shows the highest usage among OMP configurations for a problem size of 8192.

The charts in Section 3.4 reveal that the OpenMP version, across different concurrency levels, does not exhibit linear speedup. Most levels stabilize after reaching a certain problem size, possibly due to Amdahl's law, which implies a limit to achievable speedup in the presence of serial code portions. Lastly, Section 3.5 demonstrates that OMP-64 outperforms CBLAS for larger problem sizes, highlighting the superiority of parallelism.