CSC 656-01, S23
Coding Project #3: Vectorization, Blocking, Shared-memory parallelism with OpenMP
Due: Tue 2 May 2023 23:59 PDT

Overview

Using a code harness, implement vector-matrix multiply in multiple forms: basic, vectorized, and shared-memory parallel using OpenMP.

Build and run the codes on Cori@NERSC on a KNL node, record the runtimes for each of the 3 codes at varying problem sizes and, in the case of the parallel code, at 3 specific levels of concurrency.

Using the runtime data from your code runs, compute some derived performance metrics – MFLOP/s and % memory bandwidth utilized – then create a table of this data and answer some specific analysis questions.

Deliverables

The deliverables for this project include:

- Source Code and all materials needed to build and run your code (CMakeLists.txt, etc.) in a single zipfile or compressed tarfile (no RAR files)
- A single PDF containing results of analysis: table(s) and answers to questions. The simplest way to produce this PDF is to use Google Docs, which lets you combine image files and text, create tables, and then download as a PDF.
- A copy of the vectorization report generated by the Intel compiler for your dgemv-vectorized.cpp code. This is a text file that you can just upload to iLearn the same way you upload a .zip or .pdf file.
- A screenshot of the x86 assembly language code of your dgemv-vectorized.cpp as generated using the godbolt.org compiler explorer using the compiler x86-64 icc 19.0.1 with the compiler flags “-O3 -xMIC-AVX512”. Here we are looking for the presence of AVX512 x86 instructions in your assembler: these will have the string “zmm*”, compared to AVX256 instructions that have the string “ymm*”, and SSE (128bit) instructions having strings of the form “xmm*”.
    - Go to godbolt.org
    - Enter your source code (c++) in the window on the left
    - Configure the compiler and flags in the upper right
    - Voila, your assembler appears on the right

General Information

The code harness for this assignment is accessible via github:

    git clone https://github.com/SFSU-CSC746/vmmul-harness-instructional.git

There is a README.md in that distribution that contains a lot of detailed information.

Please refer to the Spring 2023 NERSC Topics google doc for information about accessing Cori, the NERSC software ecosystem, and building/running jobs on Cori KNL nodes.

For this assignment:

- Please use the Intel compilers on Cori. Once you log in, type this command:
  ```
  % module load PrgEnv-intel
  ```
  Depending on how your environment is set up, some of you might have to do this::
  ```
  % module swap PrgEnv-gnu PrgEnv-intel
  ```
- We will be running all experiments on Cori KNL nodes. To request an interactive node for 60 minutes, type this command:
  ```
  % salloc --nodes 1 --qos interactive --time 01:00:00 --constraint knl --account m3930
  ```

Reference material:

- Cori system configuration and technical specifications for different types of nodes
- Cori KNL nodes specification: clock rate, memory sizes, bandwidth, etc.
- Intel KNL specification

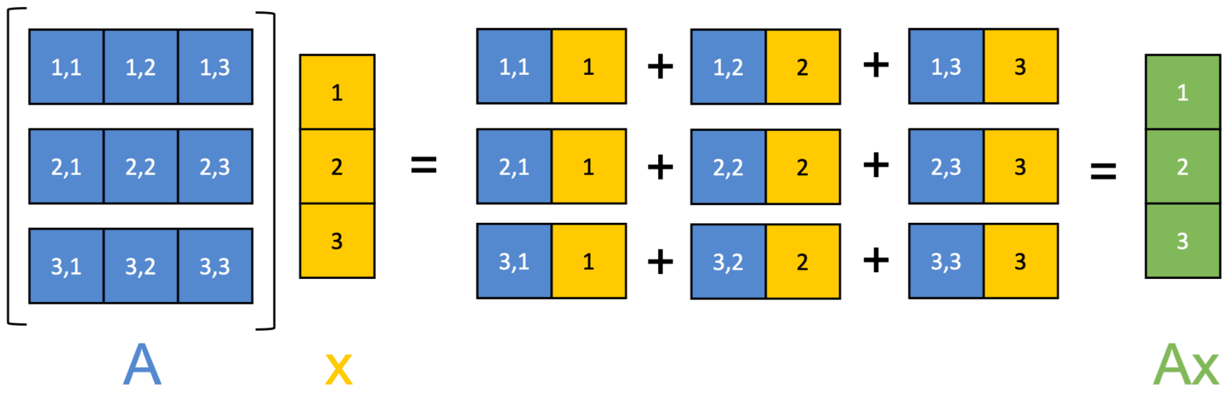
Computing various metrics:

- MFLOP/s = ops/time, where
  - ops = number of operations/1M
  - time = runtime(sec)
- % of memory bandwidth utilized = (bytes/time) / (capacity), where
  - bytes = number of memory bytes accessed by your program
  - time = runtime of your program (secs)
  - capacity = theoretical peak memory bandwidth of the system

Background: The Vector-Matrix Multiplication

The VMM Algorithm

Vector-matrix multiplication (VMM) is a common linear algebra operation where you multiply rows of an NxN square matrix A by an 1xN column vector X to produce a new Nx1 row vector Y as output.



(Image source: https://mbernste.github.io/posts/matrix_vector_mult/)

You are implementing a special version of VMM that replicates a routine *dgemv* from the CBLAS standard linear algebra library. *Dgemv* implements the following operation:

Y = Y + Ax

Where A is an NxN matrix, x is a 1xN column vector, and Y is a 1xN row vector.

The VMM Implementation: Number of Operations

In order to compute MFLOP/s, you need to divide the number of operations the code performs by the elapsed runtime to come up with a FLOP/s figure. How many operations does VMM perform?

We have/will discuss this issue in class, so please refer to the lecture slides.

The VMM Implementation: Number of Memory Accesses and Bytes Moved

In order to compute % of peak memory bandwidth utilization, you will need to compute the number of memory references and the number of bytes of each memory reference.

We have/will discuss this issue in class, so please refer to the lecture slides.

Part 0 – Instrument benchmark.cpp to measure elapsed time

You will need to add timer instrumentation to the benchmark.cpp file to measure and report elapsed time. Refer to the chrono_timer code distribution for more details. This is the same type of instrumentation you had to do for our earlier coding project.

Correctness test: inside benchmark.cpp, you will see that the answer from your implementation of my_dgemv() is being compared to the reference implementation of dgemv() inside the CBLAS library. If the benchmark.cpp code complains that your answer is not the same as that computed by CBLAS, then you have made a computational error in your implementation. You need to fix your code so that it produces the same result as the reference implementation.

Part 1 - CBLAS reference dgemv Vector-Matrix Multiply

For this part of the assignment, you will be running the benchmark-blas code to establish a performance baseline.

Due to peculiarities in how CBLAS operates, we have set up benchmark.cpp so that it runs the smallest problem size twice to “condition” CBLAS.

<u>When the benchmark-cblas code runs, you should use the runtime of the second of the two smallest problem sizes. Don’t use the runtime from the first problem size.</u>

For each problem size, compute MFLOP/s from runtime and number of operations
For each problem size, compute the % of memory bandwidth this code utilizes

Note: while you don’t know exactly how many operations or number of memory accesses the CBLAS implementation of dgemv performs, you can estimate the number of operations based on your knowledge of the VMM algorithm presented in the class lecture.

Part 2 - Basic dgemv Vector-Matrix Multiply

Do your own implementation of basic vector-matrix multiply in the dgemv-basic.cpp file in the code harness.

On a Cori KNL compute node, run the benchmark-basic program, which calls your implementation inside dgemv-basic.cpp at varying problem sizes, and make note of the runtime for each problem size in a text file or spreadsheet.

For each problem size, compute MFLOP/s from runtime and number of operations
For each problem size, compute the % of memory bandwidth your code utilizes.

Part 3 - Vectorized dgemv Vector-Matrix multiply

Here, you will transplant your working version of basic vector-matrix multiply from the dgemv-basic.cpp file and put it into dgemv-vectorized.cpp.

When you build the code, the compiler options are set on your behalf to invoke automatic vectorization in the compiler.

On a Cori KNL compute node, run the benchmark-vectorized program, which calls your implementation inside dgemv-basic.cpp at varying problem sizes, and make note of the runtime for each problem size in a text file or spreadsheet.

For each problem size, compute MFLOP/s from runtime and number of operations
For each problem size, compute the % of memory bandwidth your code utilizes.

Vectorization Report

In the cmake/make files, the compiler is configured to vectorize your code and produce a vectorization report, which you need to submit as part of this assignment. The name and location of this vectorization report file varies depending on the compiler.

<u>On Cori</u>: we are using the Intel compiler, so the vectorization report ends up being placed by the compiler into a subdirectory under your build directory. To find this file, type this command:

```
% find . -name "*.optrpt" -print
```

And you will see a directory path to the vectorization report file, which will be named something like "dgemv-vectorized.cpp.optrpt".

<u>On Perlmutter</u>: we are using the GCC compiler, which in this case will generate a vectorization report named "report.txt" that will be located in the build directory where you run make. After you run cmake then make, do a directory listing in the build directory, you should see the file named "report.txt".

During development, you should look at the contents of that file. It provides you with feedback about which portions of the code were vectorized and which were not.

Once your code is working, please submit the vectorization report associated with the good, working file as part of your iLearn submission.

Part 4 - Shared-memory parallel dgemv Vector-Matrix Multiply

Here, you will transplant your working version of basic vector-matrix multiply from the dgemv-basic.cpp file and put it into dgemv-openmp.cpp.

Then, modify this implementation to add loop-level parallelism using OpenMP.

On a Cori KNL compute node, run the "job-openmp" script, which will execute this code three times at different levels of concurrency (t=1,4,8), and record the run time from each problem size in a text file or spreadsheet.

For each problem size, compute MFLOP/s from runtime and number of operations
For each problem size, compute the % of memory bandwidth your code utilizes

Notes on running this code

Assuming that:

- You have successfully built the benchmark-openmp code, and
- Your current working directory is the *build* directory, and
- You are on a Cori interactive compute node, then

You have two options for running the shared-memory parallel code.

Option 1: Manual

OpenMP concurrency is controlled/specified via an environment variable, OMP_NUM_THREADS

To run at 1-way concurrency, first set the environment variable then run the code:

```
% export OMP_NUM_THREADS=1
% srun ./benchmark-openmp
```

To run at 4-way concurrency, first set the environment variable then run the code:

```
% export OMP_NUM_THREADS=4
% srun ./benchmark-openmp
```

Option 2: Scripted

When you run cmake, it creates a new file in your *build* subdirectory named "job-openmp". This file is a bash script containing a loop over the CP#2 concurrency levels.

To run this script, execute it as a bash shell script:

```
% bash ./job-openmp
```

Part 5 – Analyzing Results

From each of the codes in Parts 1 – 3, you now have the following information:

- MFLOP/s at each problem size
- % of memory bandwidth utilized at each problem size

Table 1. Create a table (e.g, use Google Sheets or Excel) where the rows are the problem sizes (1024, 2048, 4096, 8192, 16384) and the columns are: blas, basic, vectorized, omp-1, omp-4, omp-8. <u>Each table cell contains the runtime in seconds</u> (or milliseconds) at each of these configurations. Please be sure to use the same number of decimal points on all numbers in the table.

Table 2. Create a table (e.g, use Google Sheets or Excel) where the rows are the problem sizes (1024, 2048, 4096, 8192, 16384) and the columns are: blas, basic, vectorized, omp-1, omp-4, omp-8. Each table cell is your computed MFLOP/s of each code at each problem size. You will need to use the entries from Table 1, along with information about the number of operations the code performs at different problem sizes, to derive the MFLOP/s entries.

Table 3. Create a table (e.g, use Google Sheets or Excel) where the rows are the problem sizes (1024, 2048, 4096, 8192, 16384) and the columns are: blas, basic, vectorized, omp-1, omp-4, omp-8. Each table cell is your computed % of peak memory bandwidth utilization of each code at each problem size. Since these numbers are all percentages, they are all in the range 0% … 100%. You will need to use the entries from Table 1, along with information about the number of operations the code performs at different problem sizes and Cori's peak main memory bandwidth to derive % of peak memory bandwidth utilization.

Analysis questions. Please provide a brief (2-3 sentences maximum) answer to each of the following questions.

1. Comparing the results of your basic and vectorized implementations at N=16384, which code has better performance in terms of MFLOP/s, and by how much? Which code has better memory system utilization, and by how much?
2. Comparing the results of your basic and OpenMP 8-way parallel implementation at N=16384, which code has better performance in terms of MFLOP/s and by how much? Which code has better memory system utilization, and by how much?
3. Looking at the results of your OpenMP implementation at N=16384, what is the ***speedup*** of this code going from 1 to 4 threads, and from 1 to 8 threads? Use your runtime data to compute these speedup metrics.

Important Dates

- Submissions open: 18 Apr 2023
- Submissions due: Tue 2 May 2023 23:59 PDT
- Submissions close: Fri 5 May 2023 23:59 PDT

Note: Cori will be down for scheduled maintenance from 07:00–20:00 on Weds 4/19/2023. Plan your work accordingly.

Grading

This assignment is worth 100 points (counts for about 1/8th of your total grade)

Late Submissions

Per the CSC 656 Syllabus:

Advice: do not wait until the “last minute” to get started on homeworks. Homeworks can require a significant amount of effort, and it is inevitable that unexpected things happen that will slow you down.

- Late submissions will be subject to a 5%-per-day penalty assessment: 0-1 days late, 5% deduction; 1-2 days late, 10% deduction; 2-3 days late, 15% deduction, etc.
- Submissions are not accepted more than 3 days late, except in unusual cases that are (1) outside the students control (e.g., medical) and that (2) can be verified with objective documentation (e.g, such as a doctor’s note, but preferably a formal accommodation from the DPRC office).