DUE DATE: **By the end of your lab session**

CSE-443/543: High Performance Computing

Exercise #17

Max Points: 20

You should save/rename this document using the naming convention `MUid_Exercise17.docx` (example: `raodm_Exercise17.docx`).

Objective: The objective of this exercise is to:

- Use OpenMP target to offload to GPU
- Compare performance of single threaded, OpenMP (CPU), and OpenMP (GPU) implementations.

Submission: Save this MS-Word document using the naming convention *`MUid`*`_Exercise17.docx` prior to proceeding with this exercise. Upload the following at the end of the lab exercise:

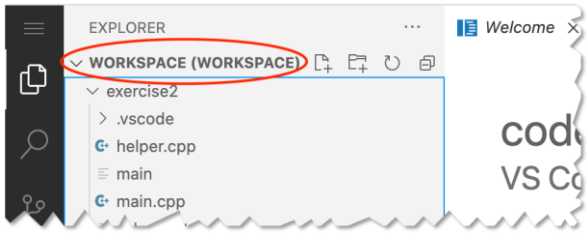
1. This document saved as a PDF file named with the convention *`MUid`*`_Exercise17.pdf`.

You may discuss the questions with your instructor.

Part #0: Setting up VS-Code project

Estimated time to complete: 5 minutes

1. Log into OSC's OnDemand portal via https://ondemand.osc.edu/. Login with your OSC id and password.
2. Startup a VS-Code server and connect to VS-Code. Ensure you switch to your workspace. Your VS-Code window should appear as shown in the adjacent screenshot.



3. Next, create a new VS-Code project in the following manner:
   a. Start a new terminal in VS-Code
   b. In the VS-Code terminal use the following commands:

```
$ # First change to your workspace directory
$ cd ~/cse443
$ # Use ls to check if workspace.code-workspace file is in pwd
$ # Next copy the basic template for a C++ project
$ cp -r /fs/ess/PMIU0184/cse443/templates/openmp exercise17
$ # Copy the starter code for this exercise
$ cp /fs/ess/PMIU0184/cse443/exercises/exercise17/* exercise17
```

4. Study the starter code supplied to you. Note how the matrix used in this example uses a 1-D `std::vector` for backing storage. Since the matrix representation is flat, an entry at (`row`, `col`) is accessed via the formula `row` * `width` + `col`.

DUE DATE: **By the end of your lab session**

Task #1: Exploring effectiveness of OpenMP

Estimated time to complete: 20 minutes

Background: Recollect that parallel program have total overheads (To = p·Tp – To) due starting, stopping, and coordinating threads. Hence, if threads are started and stopped too often, then the parallel program will not be performant. Therefore, to have an effective and scalable solution, sufficient work (*W*) has to be done by the threads before the multithreaded version outperforms its single-threaded versions. The question then becomes how much work (*W*) needs to done in order to have *p*-threads perform at least as fast as the single-threaded version.

Exercise: The objective of this part of the exercise is to determine the matrix size that is needed to see performance improvement from a 4-threaded version (versus a single-threaded version) of a given matrix multiplication benchmark. Perform this experiment via the following procedure:

1. For convenience, first create a single-threaded executable of the given starter code via the following command:

```
$ g++ -g -Wall -std=c++17 -O3 main.cpp -o ex17_seq
```

2. Next, modify the program to use OpenMP by adding the OpenMP pragma before the first 1st `for`-loop in the `SimpleMatrix::dot` method shown below:

```
#pragma omp parallel for
  for (int row = 0; (row < m1Height); row++) {
```

3. For convenient testing, create a multi-threaded executable (that uses only CPU-cores) via the following command:

```
$ g++ -g -Wall -std=c++17 -fopenmp -O3 main.cpp -o ex17_mt_cpu
```

4. Now start an interactive SLURM job:

```
$ sinteractive -n 4
```

5. Once the job starts, run the sequential and multithreaded versions with different matrix sizes (in a systematic manner) to determine the size when the 4-threaded version. Record the timings (from 3-runs and the average) in the table further below. Don't forget to `exit` out of the job once you have collected data.

```
$ export OMP_NUM_THREADS=4
$ # Example command: /usr/bin/time ./ex17_seq 10
$ # Example command: /usr/bin/time ./ex17_mt_cpu 10
```

Matrix Size	Sequential (elapsed time)				4-threaded Version (Elapsed time)			
	Run #1	Run #2	Run #3	Average	Run #1	Run #2	Run #3	Average
10	0:00.03	0:00.03	0:00.03	0:00.03	0.04	0.01	0.02	0.02333333333
100	0.07	0.08	0.06	0.07	0.07	0.06	0.07	0.06666666667

500	**3.83**	**3.88**	**3.76**	3.823333333	**1.03**	1.03	1.02	1.026666667
1000	**33.13**	**33.21**	**31.53**	32.62333333	**8.2**	7.98	8.12	8.1

Part #2: OpenMP offloading to the GPU [10 points]

Estimated time to complete: 20 minutes

Background: OpenMP provides a convenient mechanism to offload computation to a GPU. Effectively offloading to the GPU does requires understanding the operations of the GPU. However, in this exercise, we will be using a very simple offload via OpenMP's `target` directive. In addition, instead of OpenMP's `parallel` directive (that is meant for CPU) we will use the `teams distribute` directive that is used to distribute the workload to a team of threads on a GPU. Note that offloading to the GPU incurs additional overheads to copy the matrices to-and-from the GPU.

Exercise: The objective of this part of the exercise is to determine the size of the matrix for which the GPU starts performing faster than the sequential and 4-threaded version. Perform this experiment via the following procedure:

1. Enable offloading to a GPU by adding the following OpenMP pragma before the first 1st `for`-loop in the `SimpleMatrix::dot` method shown below:

```
#pragma omp target map(to: m1Data[0:size()], m2Data[0:m2.size()]),     \
   map(to: m1Height, m2Width, m1Width), map(from: resData[0:result.size()])
#pragma omp teams distribute parallel for
   for (int row = 0; (row < m1Height); row++) {
```

2. For convenient testing, create a multithreaded executable (that uses GPU) via the following command:

```
$ g++ -g -Wall -std=c++17 -fopenmp -foffload=nvptx-none -O3 main.cpp -o ex17_mt_gpu
```

3. Now start an interactive SLURM job to use a GPU node (via the `-g 1` option):

```
$ sinteractive -g 1
```

4. Once the job starts, run the GPU version with different matrix sizes (in a systematic manner) to determine the size when the GPU implementation runs faster than the single and 4-threaded CPU version. Record the timings (from 3-runs and the average) in the table further below. Don't forget to `exit` out of the job once you have collected data.

```
$ # Example command: /usr/bin/time ./ex17_mt_gpu 10
```

Matrix Size	GPU (elapsed time)			
	Run #1	Run #2	Run #3	Average
10	**0.01**	0.01	0.01	0.01

100	**0.03**	0.02	0.02	0.02333333333
500	**1.14**	1.17	1.88	1.396666667
1000	**7.99**	8.01	8.11	8.036666667

Part #3: Inferences from the experiments

Estimated time to complete: 10 minutes

Using the data from the above experiments draw inferences on the costs (p·Tp), overheads (To), and efficiency of the different approaches.

We can see a significant decrease in time while analyzing the differences between 1 and 4 cpu cores. This difference is especially visible when we focus on bigger matrices requiring more memory. We can start to see the increase of performance already at n = 500, Tp is almost speedup (3.74x faster). The total overhead of parallel algorithms was 4.08 – 3.82 = 0.2s. What is surprising, the GPU gave similar results to the ones on 4 cores, where I was expecting much faster process (maybe I set up something wrong). Difference is even higher in size 1000 where speed up is higher that theoretical (4.02) which may be caused by some noise and probably would be less is we test it on a higher number of runs. In both cases where the matrix size increases, the efficiency is very high and (higher than 0.93) and it is definitely worth to parallelize the program.

Part #4: Submit files to Canvas

Upload just the following files:

1. This MS-Word document saved as a PDF file and named with the convention *MUid_Exercise17*.pdf.