UNIVERSITY OF WASHINGTON
Paul G. Allen School of Computer Science and Engineering

CSE P 524 **Assigned:** 3/5/2022
Problem Set #9 **Due:** 3/12/2022

1 Preliminaries

For this assignment you will need access to an Nvidia GPU and CUDA development environment, available on the machine mcu.cs.washington.edu. As a reminder, to connect to mcu, you need to first connect to attu:

```
$ ssh <your netid>@attu.cs.washington.edu
```

Then from attu you can connect to mcu

```
$ ssh mcu.cs.washington.edu
```

You can do this in one command:

```
$ ssh -J <your netid>@attu.cs.washington.edu <your netid>@mcu.cs.washington.edu
```

To make things even more convenient, you can add something like the following to your .ssh/config:

```
Host attu
     HostName attu.cs.washington.edu
     User <your netid>
     IdentityFile ~/.ssh/<your private key>

Host mcu
     HostName mcu.cs.washington.edu
     ProxyJump attu
     User <your netid>
```

(Note that the ~ above will not copy-paste correctly due to typesetting issues – you will need to substitute the tilde character manually.) With this in your .ssh/config, you don't need to specify your netid explicitly when issuing the ssh command to attu. In addition, if a public key is on attu/mcu, you won't have to enter your password. Your home directory should be shared between attu and mcu so you only need to copy your ssh public key to attu. You can copy your keys to attu by using

```
  $ ssh-copy-id <uw net id>@attu.cs.washington.edu
```

(Thanks to Ken Yang for the tip.)

1.1 Connecting with VS Code

If you are accustomed to editing with an IDE such as VS Code, connecting to mcu with ssh would seem to preclude your usual development practices. While you can connect to mcu and use a screen editor such as emacs, VS Code (and, presumably, other similar tools) can connect to remote hosts through the tool so that you can use VS Code on your local machine, while editing files on the remote machine.

To do this with VS Code, install the "Remote SSH" extension. Once this is installed, you can connect to any remote host. Simply click on the icon in the lower left of the editor window and select "Connect to Host." You can also enter "Connect to Host" directly in the Command Palette. If you have mcu listed in your .ssh/config file, mcu will show up as an option when you select "Connect to Host."

The first time you connect, it will take a few minutes while VS Code sets up the remote environment. Subsequent connections will go more quickly. Once you are connected, when you create a new terminal window in VS Code, it will be on the remote host, so you can do all of your development through the IDE.

1.2 Finding Out About the GPU

Once you are logged on to mcu, you can query GPU resources using the command `nvidia-smi`.

```
$ nvidia-smi

Sat Mar  5 09:57:01 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.63.01    Driver Version: 470.63.01    CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ...  Off  | 00000000:03:00.0 Off |                  N/A |
| 30%   38C    P0    39W / 260W |      0MiB / 11019MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  NVIDIA GeForce ...  Off  | 00000000:04:00.0 Off |                  N/A |
| 21%   36C    P0    24W / 260W |      0MiB / 11019MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

This indicates the GPU is of the GeForce family, has 11GiB of memory, we have version 470.63.01 of the drivers installed, and version 11.4 of CUDA installed.

`nvidia-smi` has a number of different options that can give you more (and more specialized) information about the GPU. To see more information about `nvidia-smi`, use the `-h` option

```
$ nvidia-smi -h
```

NB: `nvidia-smi` also has commands for modifying the device. The system should be set up so that normal users do not have sufficient privileges to modify the device, but there may still be unintended consequences from using these options to `nvidia-smi`. Do not, under any circumstances, attempt to modify the GPU card settings.

Question 1: Use `nvidia-smi -h` to find out how to use `nvidia-smi` to give you more information about the GPUs on mcu. Answer the following questions.

a How many GPUs are attached to mcu?

b What is the complete product name of each GPU?

c What are the minimum, maximum, and default power limits?

d What power limit is currently set?

e What is the GPU shutdown temperature?

f What is the maximum SM clock?

1.3 CUDA Development Environment

The CUDA toolkit on mcu is installed in `/usr/local/cuda`. To use the tools (e.g., the `nvcc` compiler), you need to have `/usr/local/cuda/bin` in your `PATH`. To set this up automatically, add

```
export PATH=/usr/local/cuda/bin:${PATH}
```

to your `.bashrc` file (in your home directory). Alternatively, you can save this command in a script and source (not execute) it when you need it.

To verify that your path is set up, make sure you can invoke `nvcc`:

```
$ nvcc
nvcc fatal   : No input files specified; use option --help for more information
```

1.4 CUDA Samples

The Nvidia CUDA toolkit contains a large number of substantive examples, which can be found at

```
https://github.com/NVIDIA/cuda-samples.git
```

To Do 1: Clone the cuda-samples repository and build the deviceQuery example.

```
$ git clone https://github.com/NVIDIA/cuda-samples.git
$ cd cuda-samples/Samples/1_Utilities/deviceQuery
$ make
$ ./deviceQuery
```

The deviceQuery program reports information about the GPUs on your system, some of which are (evidently) not available with nvidia-smi.

Question 2: Using deviceQuery, answer the following questions about the GPUs on mcu.

a How many total cores are available on each GPU?

b What is the maximum dimension size of a thread block?

c What is the maximum dimension size of a grid size?

d What is the CUDA capability level of the GPUs?

In addition to deviceQuery, I encourage you to look through, build, and run some of the other samples. The `5_Domain_Specific` subdirectory in particular has some interesting and non-trivial examples (such as three different nbody simulators).

2 Problems

2.1 AXPY CUDA

"axpy" is the base name of one of the fundamental vector-vector operations in the basic linear algebra subprograms (BLAS) library. When the library was developed, the language of choice for scientific computing was Fortran 66 – in which only the first six letters in any identifier were significant. In addition, programs were "written" on punch cards – one program statement per card, and each card was limited to 80 characters, including any indentation. Hence, naming tended to be very cryptic. The mnemonic "axpy" is for a times x plus y – axpy effected the operation $y \leftarrow a \times x + y$ for two one-dimensional arrays x and y and scalar a. There was no function overloading (nor templates) in Fortran 66 – so there was a separate function for each basic type (single precision real, double precision real, single precision complex, double precision complex), each with its own name: `saxpy`, `daxpy`, `caxpy`, and `zaxpy`.

The `cu_axpy` subdirectory of your repo contains a set of basic examples: several `cu_axpy` programs. The executables can be compiled by issuing "make". Each program takes one argument – the log (base 2)

of problem size to run. That is, if you pass in a value of, say, 20, the problem size that is run is 2^{20}. The default problem size is 2^{16}.

Again, the size argument in these examples is the $\log_2$ of the size.

The programs print some timing numbers for memory allocation (etc), as well a Gflops/sec for the axpy computation. Each program only runs the single problem size specified.

The axpy subdirectory also contains a script `script.bash` which runs all of the axpy programs over a range of problem sizes and plots the results in axpy_cuda.pdf. In addition to the basic CUDA examples, there is a sequential implementation, an OpenMP implementation, and a thrust implementation.

The examples cu_axpy_0 through cu_axpy_3 follow the development of CUDA kernels as shown in slides 67-74 in lecture.

To build and run one of the CUDA examples

```
$ make cu_axpy_1.exe
$ ./cu_axpy_1.exe 20
```

You can compare to sequential and omp versions with seq_axpy and omp_axpy:

```
$ ./seq_axpy.exe 20
$ ./omp_axpy.exe 20
```

To Do 2: Generate the scaling plots for this problem, by running script.bash

```
$ bash script.bash
```

Examine the output plot and review the slides from lecture and make sure you understand why the versions 1, 2, and 3 of cu_axpy give the results that they do. Note also the performance of the sequential, OpenMP, and Thrust cases. Make sure you can explain the difference between version 1 and 2 (partitioning).

Question 3: How many more threads are run in version 2 compared to version 1? How much speedup might you expect as a result? How much speedup do you actually see in your plot? Include your `axpy_cuda.pdf` plot with your answer to this question.

Question 4: How many more threads are run in version 3 compared to version 2? How much speedup might you expect as a result? How much speedup do you see in your plot? (Hint: Is the speedup a function of the number of threads launched or the number of available cores, or both?) Include your `axpy_cuda.pdf` plot with your answer to this question.

Question 5: The program cu_axpy_3 also accepts as a second command line argument the size of the blocks to be used. Experiment with different block sizes with, a few different problem sizes (around 2^{24} plus or minus). What block size seems to give the best performance? Are there any aspects of the GPU as reported by deviceQuery that might point to why this would make sense?

2.2 Striding

In previous problem sets we considered block and strided approaches to parallelizing norm. We found that the strided approach had unfavorable access patterns to memory and gave much lower performance than the blocked approach.

But, consider one of the kernels we launched for axpy (this is from cu_axpy_2):

```
__global__ void dot0(int n, float a, float* x, float* y) {
  int index  = threadIdx.x;
  int stride = blockDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = a * x[i] + y[i];
}
```

It is strided!

Question 6: Think about how we do strided partitioning for task-based parallelism (e.g., OpenMP or C++ tasks) with strided partitioning for GPU. Why is it bad in the former case but good (if it is) in the latter case?

2.3 norm_cuda

In this part of the assignment we want to work through the evolution of reduction patterns that were presented in lecture (slides 83-87) but in the context of norm rather than simply reduction. (In fact, we are going to generalize slightly and actually do dot product).

Consider the implementation of the `dot0` kernel in `cu_norm_0`

```
__global__
void dot0(int n, float* a, float* x, float* y) {
  extern __shared__ float sdata[];

  int tid    = threadIdx.x;
  int index  = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;

  sdata[tid] = 0.0;
  for (int i = index; i < n; i += stride)
    sdata[tid] += x[i] * y[i];

  __syncthreads();

  if (tid == 0) {
    a[blockIdx.x] = 0.0;
    for (int i = 0; i < blockDim.x; ++i) {
      a[blockIdx.x] += sdata[i];
    }
  }
}
```

There are two phases in the kernel of this dot product. In the first phase, each thread computes the partial sums for its partition of the input data and saves the results in a shared memory array. This phase is followed by a barrier (`__syncthreads()`). Then, the partial sums are added together for all of the threads in each block by the zeroth thread, leaving still some partial sums in the `a` array (one partial sum for each block), which are then finally added together by the CPU.

As we have seen, a single GPU thread is not very powerful, and having a single GPU thread adding up the partial sums is quite wasteful. A tree-based approach is much more efficient.

A simple tree-based approach `cu_norm_1.cu` is

```
__global__
void dot0(int n, float* a, float* x, float* y) {
  extern __shared__ float sdata[];

  int tid    = threadIdx.x;
  int index  = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;

  sdata[tid] = 0.0;
  for (int i = index; i < n; i += stride)
    sdata[tid] += x[i] * y[i];

  __syncthreads();

  for (size_t s = 1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
      sdata[tid] += sdata[tid + s];
    }
```

```
    __syncthreads();
  }

  if (tid == 0) {
    a[blockIdx.x] = sdata[0];
  }
}
```

Deliverable 1: Implement the reduction schemes from slides 85, 86, and 87 in `cu_norm_2.cu`, `cu_norm_3.cu`, and `cu_norm_4.cu`, respectively.

Deliverable 2: This subdirectory has script files for queue submission and plotting, similar to those in the `axpy_cuda` subdirectory. When you have your dot products working, run the script to plot your results.

You can compare the performance of the CUDA norms to the sequential and omp versions with `norm_seq.exe` and `norm_parfor.exe`.

Question 7: What is the max number of Gflop/s that you were able to achieve from the GPU? How does the CUDA performance compare to sequential and omp? Include your `norm_cuda.pdf` in your answer to this question.

Deliverable 3: (Extra Credit.) Complete the implementation of `norm_thrust` in `norm_thrust.cu`. You should attempt to do a one-line solution, using an appropriate `thrust::` algorithm, similar to what you did with a C++ `std::` algorithm for `stl_1` in assignment 7. (You should be able to come up with a `thrust::` solution that bears a striking similarity to your previous `std::` solution.

Question 8: (Extra Credit.) The `norm_thrust` executable will run the norm function over a variety of problem sizes, for both single and double precision. What is the highest performance that you achieve? How does the single precision performance compare to the double precision performance? Include a listing of your `norm_thrust` function with this question.

3 What to Turn In

All of the code for this assignment should be checked into your course shared repository. You should also check in axpy_cuda.pdf and norm_cuda.pdf. Your repository should also include the PDF of your written answers, saved as "ps9-written.pdf." Tag your final commit as "ps9_submission".

Checklist:

- Append a list of sources / references you used for this assignment to the end of ps9-written.pdf.
- Upload your ps9-written.pdf document to GradeScope under assignment "ps9".
- Upload your `cu_norm_2.cu`, `cu_norm_3.cu`, `cu_norm_4.cu`, and, (if you attempted it) `norm_thrust.cu` to GradeScope under assignment "ps9_code".