Homework 10

Compiling and running GPU jobs on scholar.

Instead of connecting to scholar.rcac.purdue.edu for your frontend machine, connect to *gpu*.scholar.rcac.purdue.edu. You can probably run directly on the frontend – I did without problems. Keep your jobs short. If you do an sbatch, use

```
	sbatch -A scholar --nodes=1 --gpus=1 -t 00:01:00 gpu_hello.sub
```

to submit your job. Without the -A you might end up on a scholar node without a GPU, which will get you a slurm.out file with an error message. Note that the example and the documentation at https://www.rcac.purdue.edu/knowledge/scholar/run/examples/slurm/gpu differ, and you might want to use -G instead of -A. But again, you can also run it locally.

Immediately after logging in do a *module unload intel* to ensure that you don't load intel .h files, followed by a *module load cuda* to get access to nvcc and the Nvidia libraries.

Compile the hello_world.c program using *nvcc hello_world.cu -o hello_world* and run it.

Note that you can print to stdout on the device. There is a limit of the number of characters (1,000,000?) so if you print a lot, you will exceed that and not get all of your output. Note that prints will be executed by all 172,032 threads, so guard them with ifs so that only some threads print.If you want to see the output, you need to put *cudaDeviceSynchronize( );* after the kernel launch that does the print. hello_world.cu has an example of this.

The Volta GPUs have double precision floating point. If you use float for this problem you will get different answers for the sequential and GPU versions, since the order of operations is different and Intel does not implement pure IEEE 754 floating point.

I spent an hour or more trying to debug the difference between my GPU and sequential output until I thought to try double instead of float to see what happened. Notice that in general float is often good enough, halves the bandwidth requirements of shipping data to/from the GPU and to/from global and shared memory, and doubles the amount of data the GPU and shared memory can hold. So in general, use float if it is good enough. Just don't use it for this homework!

Some parameters for the GPUs that we will be using (see table 7.1 at https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf)

Number of SMs: 84
Max threads/SM: 2048
Max number of blocks/SM: 32
Max number of warps/SM 64:
Max number of threads: 172,032
Max number of registers/SM: 64K
Max number of registers/thread: 255
Shared memory per SM: 96KB

Constants supplied by cuda that will be useful:

gridDim.x: the number of blocks in the grid along the .x dimension.
blockDim.x: the number of threads in a block along the .x dimension
blockIdx.x: the block number for this block in the grid.
threadIdx.x: the thread number in its block.

What to do:

The file dotproduct.cu will give the skeleton of a solution and guidance on what to do. You can ignore this if you want. We will do a dot product of two vectors whose length is 5X the number of threads supported by the Volta (Tesla V100) GPU. Your solution should

1. Create a dotproduct kernel that will execute on the device
    a. Each thread in the kernel produces a partial result consisting of c += d_a[idx]*d_b[idx], where idx eventually specifies five values from the input vectors. Since the input vectors are in the global device memory, we want all threads in a block to fetch and add adjacent elements of d_a and d_b, i.e. to do coalesced memory accesses.
    b. The value produced above will be place in a shared memory vector in a position corresponding to the thread producing the vector. The kernel will perform the partial dot product within each block, by performing sum reductions across the threads in a block, to form gridDim.x (the number of blocks in the grid along the x dimension, which we will use since we are solving a 1-D problem) partial reductions representing the part of the dot product done by each block. Use the "efficient" reduction described in the histogram slides.
    c. The result of each reduction will be placed in the output vector d_c. The result for each block will be placed into the appropriate position of d_c[ ], which is in global memory.
2. Create an hdotproduct function that will execute on the host and
    a. allocates resources;
    b. initializes the input array;
    c. sends data to the GPU;
    d. launches the kernel;
    e. retrieves results from the GPU;
    f. frees the device data;
    g. does the final summation and return the final dot product result.
3. The main function will
    a. call hdotproduct and malloc and initialize the host data arrays h_a, h_b and h_c;
    b. compute and print the value of a serial dot product on the host processor that uses values in h_a and h_b;
    c. print out the values from the GPU dot product;
    d. free the arrays h_a, h_b and h_c.