**Assignment 4: Due – Sunday, July 21 at 9 PM**

Euler testing environment: >> module load gcc cmake cuda

Deliverables:

* In your team’s GitHub git repo, right in the HW04 subdirectory (no other subdirectories), push:
  + CMakeLists.txt
    - Running in the HW04 directory >> cmake .; make should build all executables so that they are created in the HW04 directory
  + **C/C++ code files** (named logically as you like unless named in the problem) necessary for us to build the following executables:
    - problem1
    - problem2
  + problem1.pdf
  + problem2.pdf

Notes:

* Use the [randoms.c](https://uwmadison.box.com/s/zx8nwmcs6p7clqnxtc67x3iwirac58ra) and [randoms.h](https://uwmadison.box.com/s/nnpkw2ivkynr5vft8mbqm5q0fb0byd9w) files to generate arrays of random numbers whenever necessary. Do not alter these provided files.
* A template for your CMakeLists.txt file is available [here](https://uwmadison.box.com/s/mamlfxello670rhc6kr5cxvh305rz740).
* Use Slurm and follow cluster policy.
  + Remember that GPU jobs with the exception of cuda-gdb should be run with sbatch.
* Do **not** submit your executable files
* IMPORTANT: Do not copy solutions off the internet, from colleagues, etc. Please be honest.

-----------------------------------------------------------------------------------------------------

**Problem 1**. (50 pts) This problem concerns a 1D stencil operation. Feel free to use and edit any code available in the slides we covered during the lecture. Write two kernels to carry out stencil operations: one kernel will use shared memory to (hopefully) speed up the execution; the other kernel will not use shared memory. You should run your executable like this (via Slurm):

>> ./problem1 N R seed s

* N is the length of the input array, which should contain random values between -1 and 1 (doubles)
* R is the radius of the stencil (see Lecture 16 notes)
* seed is a positive integer value between 1 and 1000 used to seed the random array of length N (use the random functions provided)
* If s is 0, use no shared memory. Otherwise use an implementation that relies on shared memory.

**Remarks:**

* Check in your code that and exit otherwise with an appropriate message
* R will be no larger than 21
* Compute the weights in your stencil as , where .
* Your program should print to standard output on separate lines:
  + The value of the [L2 norm](http://mathworld.wolfram.com/L2-Norm.html) of the output array obtained after you apply the stencil operation on the input array.
  + The value of or the value of (see below) depending on the given value of s

*Nothing* else should be printed.

* Let be the time necessary to execute the kernel (only) that carries out the stencil operation without use of shared memory. Let be the time necessary to execute the kernel (only) that carries out the stencil operation using shared memory. Generate a 3D plot that on the x-axis has the value of , on the y-axis has the value of , and on the z-axis the value of . Take and . Feel free to upload this plot on Piazza. Submit this graph as problem1.pdf on Canvas.

**Problem 2**. (50 pts) Write a CUDA program that computes the dot product of two arrays each containing 16,000,000 double values. Use the appropriate function provided in the randoms directory to generate values in the range -2 to +2.

Your program should run like (via Slurm):

>> ./problem2 seed target

Remarks:

* seed is the integer seed used to obtain the random entries in arrays
* target is an integer. If 0, the dot product is to be run on the host; otherwise on the device.
* Your program should print to standard output the value of dot product of arrays A and B and *nothing* else.
* Finally, provide timing results of the code to see where the code runs faster: on the host or on the device. Use nvprof to get timing results. Explain your timing results and how you got them in problem2.pdf to be submitted on Canvas. Given that you want the CPU to GPU comparison to be fair, in your pdf, explain which times you compared (inclusive, exclusive, wall-time, etc.) when you investigated the performance of the GPU and CPU implementations.
  + Run the codes several times to get averages rather than outlier data