Hints for debugging GPU programs

- The CUDA calls return error status. Remember to check the error status and report any errors after every call – this will help you localize where an error occurred. The kernel call itself does not return error status; you nonetheless should check for errors on this too (as shown in the skeleton code for the labs).
- Use *printf*() statements. CUDA does not directly support C++ printing with **cout**, but does support *printf*(). Note that printing is done from the CPU rather than the GPU; thus, CUDA buffers the *printf*() output and returns it to the CPU for printing. Also, we've given you a simple function *print_array*() that may be helpful.
- One simple (but useful) way to use *printf*() is as follows:
  - If your program fails with a mismatch, note the array location that mismatches. Say it's [5,7]
  - Most people assemble *blockIdx* and *threadIdx* into variables with names like *rb*, *cb*, etc. Then you might add a line like "if ((*ri*==*5*) && (*ci*==7) && …) printf (all of the *blockIdx* and *threadIdx* variables, as well as anything else that's useful).

  Why is this useful? First, it's a great way to limit output to the one location that's interesting. Second, it helps you isolate whether your code is operating on the correct input data (if not, then probably your copy to the GPU is broken) and is producing the correct output sum (if not, then your GPU thread func is likely wrong). Finally, simply noting how many times the printf statement is invoked can be quite useful (in the CNN lab, edge effects can easily result in it happening not at all, or too often). Remember, though, that your code gets called up to four times by the main routines.

- As noted in other places, you can decrease the block size BS from 32 to, e.g., 4 to make calculations easier or give you more debug clues. To make your life easier, the code is structured so that while different files may each define BS, they are allowed to disagree – i.e., you can modify BS in one file without worrying about the other files.
- Some GPUs are quite slow at double-precision arithmetic. Be sure to use **float** rather than **double**.
- CUDA has a program called *compute_sanitizer* that is meant to catch run-time errors; it doesn't catch everything, but can save you lots of debug time when it does work. For each memory error, it will tell you whether the access was a read or write, and whether it involved GPU DRAM, shared memory or both. This detail is usually a pretty good clue as to where in your code the problem is. In addition to memory errors, it can also catch races and initialization errors.
  You can run it with *run_debug.sh*:

  ```
  source /etc/profile.d/modules.sh
  module load cuda/11.0
  module load gcc/11.2.0
  compute-sanitizer --tool memcheck ./a.out > results_memcheck.out
  compute-sanitizer --tool initcheck ./a.out > results_initcheck.out
  compute-sanitizer --tool synccheck ./a.out > results_synccheck.out
  ```

  Like any debugging tool, it can give you more information with the right compiler flags. For maximally-precise debug messages, you should now compile with

  ```
  $ nvcc -Xcompiler -rdynamic -lineinfo -O2 -std=c++11
  ```

- If your program runs way too slowly, the job manager might evict it for exceeding its CPU allocation. This might be because of your code. However, it is more often simply because you've forgotten to compile with **-O2**.