EE155/COMP122: Parallel Computing
Lab 5: Computing a CNN with CUDA

In this lab, you will code a convolutional neural network using C++ and CUDA. This is your second CUDA lab, so you will get to focus on a tricky part: dealing with edge effects. The main CNN evaluation loops are fairly straightforward, but making everything work around the edges (both the edges of the entire matrices and the edges of each GPU tile) can get more complex.

CUDA (and especially these edge effects) can be difficult, so we're worried less about getting the best performance and more about just getting everything to work. In fact, this lab has no discussion questions or lab report at all – but it does have three levels of coding:

- Getting the lab to work, using both shared memory and also cudaMallocPitch/CudaMemcpy2D, is worth the full 100 points
- Getting the lab to work, using shared memory but not cudaMallocPitch/CudaMemcpy2D, is worth 95 points
- Getting the lab to work at all is worth 90 points

As we discussed in class, there are many ways to do this; you are reasonably free to choose whichever strategy you like. As always, you should try to implement an algorithm that does not require a critical section.

The code files for this assignment are similar to the ones you used for the CUDA matrix multiply:

- matrix.hxx and matrix.cxx are identical to the earlier versions.
- matrix_mpy.cu has now been replaced with cnn.cu.
    - Similar to before, it contains a *main*() that goes through various matrix and filter sizes, and calls a *run*() function to do the work.
    - It contains Matrix::*CNN_dumb*(), which is a simple single-threaded reference function to compute a correct answer and thus check your code.
    - As before, you can alter your *main*() to use different matrix and filter sizes, or *run*() to initialize matrices differently so as to help with debugging.
- cnn_user.cu has the skeleton of your code, just as before.

As before, you should flesh out cnn_user.cu and turn in your version (as well as the report); the other code files should not be touched, and need not be handed in.

Hints:

- If you look at matrix::matrix() in matrix.cxx, you'll see that the matrices on the host are also pitched. So if you ask for a 12x12 filter, you actually get a 12x16 matrix; the same of course applies for the output matrix. And (1<<_log2NColsAlc) gives you the number of floats in a single row.
- If your code works for filters of 2x2, 4x4 and 8x8 but not 12x12, then perhaps it's because 12 is not a power of 2 (and see the hint just above).

The infrastructure will collect timing data for matrices of size 4Kx4K and 8Kx8K, with filter sizes of 4x4, 8x8 and 12x12. Note that computing the reference solution on the host will take far longer than the GPU version.

Debugging on a GPU:

The same general GPU-debug hints work for this lab as for the CUDA matrix-multiply lab. See the GPU-debug hints on the course web page for general hints. For this lab specifically:

- Don't try to debug an 8Kx8K input matrix. Edit **cnn.cu** to use 32x32 or 64x64 instead. You can even reduce BS (the block size) to, e.g., 8x8 so that you can debug with 4x4 or 8x8 matrices. As noted elsewhere, you can modify BS in **cnn_user.cu** without bothering to keep the value of BS in **cnn.cu** consistent.
- You can also simplify the filter matrix. Specifically, debugging with the filter matrix being, e.g., a 2x2 identity matrix should be fairly easy to debug. A 1x1 filter matrix is even easier – and results in the output matrix being the same size as the input matrix, which removes most of the edge effects.

Logistics:

- The lab assignment is due at midnight on the date specified by the class calendar.
- Submit your project via **provide**. You need submit only your version of cnn_user.cu.

Resources:

- As noted, this lab must be done on the Tufts High-Performance cluster. Instructions on how to use this are on the class web page.