EE155/COMP122: Parallel Computing
Lab 4: Matrix multiplication with CUDA

In this lab, you will code matrix multiplication using C++ and CUDA. CUDA can be difficult, so we're worried less about getting the best performance and more about just getting everything to work. As we discussed in class, you should try to implement an algorithm that does not require a critical section. This requires that, for any single entry in the result matrix, only one thread writes that entry.

The code files for this assignment are similar to the ones you used for the C++-threads matrix multiply:

- matrix.hxx and matrix.cxx are identical to the earlier versions.
- matrix_mpy.cxx has now been replaced with matrix_mpy.cu, but the general idea and routines are quite similar. Note that Matrix::mpy_dumb() is now faster than the version you used before (it now uses the loop ordering *r*, *k*, *c*).
- matrix_mpy_user.cxx has now been replaced with matrix_mpy_user.cu. However, again they are fairly similar. Specifically, you have to write the host function Matrix::mpy1() and also write the CUDA kernel function *mat_mult*() that it calls.

As before, you should flesh out matrix_mpy_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.

The infrastructure will collect timing data for matrices of size 1Kx1K, 2Kx2K and 4Kx4K. Note that computing the reference solution on the host will take far longer than the GPU version; it may take roughly 15 seconds for 2Kx2K and over 2 minutes for 4Kx4K.

Debugging on a GPU:

Debug on a GPU is not always trivial. Nvidia supplies the best debug tools that they can (especially, a very useful memory debugger). However, there is no good way (at least that I'm aware of) to keep track of thousands of threads at once. We have collected some general CUDA-debugging hints in a document on the course web page. Here are some hints specific to this homework:

- Reduce the problem. Don't try to debug a 1024x1024 matrix. Edit **matrix_mpy.cu** to use 32x32 or 64x64 instead. You can even reduce BS (the block size) to, e.g., 4x4 so that you can debug with 4x4 or 8x8 matrices. Note that to make your life easier, you're allowed to modify BS in your file **matrix_mpy_user.cu** without modifying BS in **matrix_mpy.cu**; the code in each file uses that files BS, and they're allowed to disagree.
- The *run*() function in **matrix_mpy.cu** initializes the *A* and *B* matrices with *init_random*(). You can temporarily switch this to, e.g., *init_cyclic_order*() or even *init_identity*() to make the numbers easier to think about.

Reference results (seconds)

	1Kx1K	2Kx2K	4Kx4K
Dumb	2.1	17	123
GPU copy time	.009	.023	0.07
GPU compute time	.020	.148	1.20
GPU total time	.029	.171	1.27

Please answer the following questions using the reference data above. Having your code execute correctly is worth 65 points; the two questions are worth the other 35.

Question #1 (*15 points*). For the results above, did the compute times for the different matrix sizes vary in a reasonable manner as the problem size increased? How do they compare to the reference results from lab #3 (matrix multiply on a multi-core CPU)?

Question #2. The results above are for code that has both good shared-memory bank usage and good memory-coalescing access. Show a small snippet of your code and explain why its shared-memory bank usage is either good or bad (*10 points*), and ditto for its memory-coalescing access *10 points*).

Logistics:

- The lab assignment is due at midnight on the date specified by the class calendar.
- Submit your project via **provide**. You should submit a copy of your report as a PDF, as well as your version of matrix_mpy_user.cu.

Resources:

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