EC527: High Performance Programming with Multicore and GPUs
Programming Assignment 8

Objectives
Use standard GPU features such a shared memory to write a useful GPU program.
Continue exploration of MMM (Matrix-Matrix Multiplication).

Prerequisites (covered in class or through examples in on-line documentation)
HW – Standard GPU features: shared memory
SW – Explicitly managing data transfers in the memory hierarchy
Programming – CUDA support for shared memory, tiling, more complex kernels

You can, and should, look through most of the lecture notes (slides) that have "GPU" in the title (lectures 11 through 15b) to find the MMM examples. These appear several times throughout the GPU lectures.

Assignment

General overview:
- The point of this assignment is to implement, test, and verify MMM.
- You will use single precision floating point.
- In each version you should try at least 2 different matrix sizes: 1024×1024 and 2048×2048
- In each version you should validate with MMM on the host. That means:
  - do the matrix multiply with normal CPU code
  - write code that compares the correct (CPU-computer) result with whatever comes out of your GPU code.
  - once you are sure that your GPU code is working, report the error in some quantitative way — for example, when comparing the GPU result to the CPU result, what is the largest difference in any single element?
  - When reporting this, also tell how you generated the input matrix elements (random or not? integerrs or real? over what range? all positive or mixed signs?)
- For the GPU execution time measurement, measure and report two times:
  - complete start-to-finish including data transfers, and
  - just the MMM (kernel) execution time.
- Also compare the performance with your best CPU MMM time from earlier in the semester.

Part 1: MMM using global memory only.
Using the lecture notes (slides) as guidance, create a new program **`mmm_global.cu`** that performs Matrix-Matrix Multiply with everything in global memory.

Part 2: MMM using shared memory.
Create another program **`mmm_shared.cu`** with shared memory (kernels take `__shared__` parameters)

Part 3: MMM using shared memory and loop unrolling.

Create another program `mmm_sh_unroll.cu` based on the best of your first two programs, in which you've unrolled the loop(s) inside your kernel(s).

Part 4: *(OPTIONAL)* Extra credit:

- Experiment with different ThreadBlock sizes.
- Try a few more matrix sizes (both larger and smaller).
- Find the matrix size(s), if any, where the GPU runs into problems.
- Find some GPU MMM code on the web. Get its performance. If the source code is available, explain its features.
- Now that you know about coalescing memory accesses (global) and how memory banks work (shared), change your code so that the access pattern is no longer favorable. How does this change performance?