ME 759
High Performance Computing for Engineering Applications
Assignment 6
Due Thursday 3/5/2020 at 9:00 PM

Submit responses to all tasks which don't specify a file name to Canvas in a file called assignment6.txt, docx, pdf, rtf, odt (choose one of the formats). Also, all plots should be submitted on Canvas. All *source files* should be submitted in the `HW06` subdirectory on the `master` branch of your homework git repo with no subdirectories.

All commands or code must work on *Euler* with only the `cuda` module loaded unless specified otherwise. They may behave differently on your computer, so be sure to test on Euler before you submit.

Please submit clean code. Consider using a formatter like clang-format.
* Before you begin, copy the provided files from `HW06` of the ME759-2020 repo.

1. Linear algebra is ubiquitous in engineering applications. BLAS (Basic Linear Algebra Subprograms) libraries, implements a myriad of common linear algebra operations and are optimized for high performance. Some of these libraries target HPC hardware. We will use cuBLAS, which targets Nvidia GPUs. See here for documentation.

   a) BLAS libraries group functions into three levels. What do all Level 1 functions have in common? Level 2? Level 3? In other words, how did they decide how to group these functions?

   b) Some functions are specialized for performing their operations when the structure of the input matrix or vector is known. List and briefly explain two such functions which assume something about their input structure in order to optimize the computation.

   c) Implement the `mmul` function as declared and described in `mmul.h` in a file called `mmul.cu`. You should use a single call to the cuBLAS library to perform the entire matrix-matrix multiplication (gemm).

   d) Write a test file `task1.cu` which does the following:

      - Creates three `n`×`n` matrices, `A`, `B`, and `C`, stored in **column-major** order in managed memory with whatever values you like, where `n` is the first command line argument as below.
      - Calls your `mmul` function `n_tests` times, where `n_tests` is the second command line argument as below.
      - Prints the **average** time taken by a single call to `mmul` in *milliseconds* using CUDA events.
      - Compile: `nvcc task1.cu mmul.cu -Xcompiler -O3 -Xcompiler -Wall -Xptxas -O3 -lcublas -ccbin $CC -o task1`
      - Run (where `n` is a positive integer): `./task1 n n_ntests`
      - Example expected output:
        `11.0`

   e) On an Euler Tesla V100[1], run `task1` for each value `n` $= 2^5, 2^6, \cdots, 2^{15}$ and generate a plot of the time taken by `mmul` as a function of `n`.

   f) Enable tensor core math in cuBLAS by adding the line
      `cublasSetMathMode(handle, CUBLAS_TENSOR_OP_MATH);`
      before calling your `mmul` function. Run the same scaling analysis on an Euler Tesla V100, and overlay the resulting plot on top of your result in part e). Submit the resulting plot as `task1.pdf`.

   g) Comment on the differences that you see in the scaling analyses and explain briefly what the `cublasSetMathMode` call changed about the computation.

[1] `module load cuda/10.1 clang/7.0.0`
`#SBATCH -p ppc --gres=gpu:v100:1 -t 0-00:20:00`
* You are only allowed to use these settings this week and you need to follow the time limit. The reference solution takes about 20 seconds to multiply $2^{15} \times 2^{15}$ matrices, so if your program takes much longer than that, you might have a mistake.
* You will need to include the compile command in your slurm script so that it compiles on the compute node.

2. a) Implement in a file called `scan.cu` the function `scan` as declared and described in `scan.cuh`. Your `scan` should call a kernel called `hillis_steele`, which you write to implement the Hillis-Steele exclusive scan given in Lecture 15. `scan` may also call other kernel functions that you write in `scan.cu`. *None* of the work should be done on host, only in the kernel calls. Note that it is important that your `scan` is able to handle general values of `n` (for example, values which are not multiples 32 (or your block size)). You have some freedom when writing the `hillis_steele` kernel to add a small amount of work that may help complete the scan. Feel free to use any code that was provided in the ME759 slides, if at all useful.

   b) Write a test program `task2.cu` which does the following.
      - Creates and fills however you like an array of length `n` where `n` is the first command line argument as below.
      - Uses your `scan` to fill another array with the result of the exclusive scan.
      - Prints the last element of the output array.
      - Prints the time taken to run the full `scan` function in *milliseconds* using CUDA events.
      - Compile: `nvcc task2.cu scan.cu -Xcompiler -O3 -Xcompiler -Wall -Xptxas -O3 -o task2`
      - Run (where `n` is a positive integer): `./task2 n`
      - Exampled expected output:
        `1065.3`
        `1.12`

   c) On an Euler *compute node*, run `task2` for each value `n` $= 2^{10}, 2^{11}, \cdots, 2^{20}$ with `threads_per_block` of 1024 and generate a plot `task2.pdf` which plots the time taken by your algorithm as a function of `n`.