ME 759
High Performance for Engineering Applications
Assignment 5
Due Friday 10/14/2022 at 6:00 PM

Submit responses to all tasks which don't specify a file name to Canvas in a file called assignment5.{txt, docx, pdf, rtf, odt} (choose one of these formats). Submit all plots (if any) on Canvas. Do not zip your Canvas submission.

All *source files* should be submitted in the `HW05` subdirectory on the `main` branch of your `GitLab` repo. For this assignment, your `HW05` folder should contain `task1.cu`, `task2.cu`, `reduce.cu`, and `matmul.cu`.

All commands or code must work on *Euler* with only the `nvidia/cuda` module loaded. The commands may behave differently on your computer, so be sure to test on *Euler* before you submit. Loading the module is done via
`$ module load nvidia/cuda`

Please submit clean code. Consider using a formatter like clang-format.
* Before you begin, copy the provided files from `Assignments/HW05` directory of the ME759 Resource Repo

---

1. a) Implement in a file called `reduce.cu` the functions `reduce` and `reduce_kernel` as declared and described in `reduce.cuh`. Your `reduce_kernel` should use the alteration from Reduction #4 ("First Add During Load" from Lecture 14). The `reduce_kernel` function should be called inside the `reduce` function (repeatedly if needed) until the final sum of the entire array is obtained.

   b) Write a test program `task1.cu` which will complete the following (some memory management steps are omitted for clarity, but you should implement them in your code):
   - Create and fill an array of length `N` with random numbers in the range [-1,1] on the host, where `N` is the first command line argument as below.
   - Copy this host array to device as the `input` array where the reduction will be performed on.
   - Create another `output` array on the device that has its length equal to the number of blocks required for the first call to the kernel function `reduce_kernel`.
   - Call your `reduce` function to sum all the elements in the `input` array, with the `threads_per_block` read from the second command line argument as below.
   - Print the resulting sum.
   - Print the time taken to run the `reduce` function in *milliseconds*.
   - Compile: `nvcc task1.cu reduce.cu -Xcompiler -O3 -Xcompiler -Wall -Xptxas -O3 -std c++17 -o task1`
   - Run (where `N` $\leq 2^{30}$ and `threads_per_block` are positive integers, and `N` is not necessarily a power of 2):
     `./task1 N threads_per_block`
   - Exampled expected output:
     `102536`
     `3.215`

   c) On an Euler *compute node*, run `task1` for each value `n` $= 2^{10}, 2^{11}, \cdots, 2^{30}$ and generate a plot named `task1.pdf` with the time taken by your algorithm as a function of `N` when `threads_per_block` $= 1024$. Overlay another plot which plots the same relationship with a different choice of `threads_per_block`.

2. a) Implement in a file called `matmul.cu` the template functions `matmul_*` as used and described in `matmul.cuh`. Be sure to follow the use of shared memory tiles. These should be based on the tiled matrix multiplication method presented in Lecture 11. Your implementation should work for arbitrary matrix dimension $n \leq 2^{14}$ (and `n` is not necessarily a power of 2).

   b) Write a test program `task2.cu` which does the following.

   - For each of the test functions defined in `matmul.cuh`: `matmul_1`, `matmul_2`, and `matmul_3`.
     - Creates and fills however you like row-major representations of `n`×`n` matrices `A`, `B`, and `C`, where `n` is the first command line argument as below.
     - Copies them to the device memory.
     - Calls the functions `matmul_1`, `matmul_2`, and `matmul_3` to produce `C` as the matrix product `AB`.
     - Prints the first element of the resulting `C`.
     - Prints the last element of the resulting `C`.
     - Prints the time taken to run the matrix multiplication in *milliseconds* using CUDA events.
   - Compile: `nvcc task2.cu matmul.cu -Xcompiler -O3 -Xcompiler -Wall -Xptxas -O3 -std c++17 -o task2`
   - Run (where `n` and `block_dim` are positive integers and `n` is not necessarily a multiple of `block_dim`): `./task2 n block_dim`
   - Exampled expected output:
```
1025
561
10256.2
1025.1
561.3
10256.2
1025.1
561.3
60256.2
```

   c) On an Euler *compute node*, run `task2` for each value $n = 2^5, 2^6, \cdots, 2^{14}$ and generate a plot of the time taken by your algorithm as a function of `n`.

   d) What is the best performing value of `block_dim` when $n = 2^{14}$?

   e) Does the performance change depending on the type of the data (i.e. `int`, `float`, `double`)? Why do you think that is?

   f) Present the best runtime for $n = 2^{14}$ from your HW04 matrix multiplication task. Explain why one of them (tiled vs. naive) performs better than the other. It is preferable that you use your own implementation for this comparison, but for those who dropped HW04, please refer to the performance data others had on Piazza.

   g) Present the runtime for $n = 2^{14}$ from HW02 (serial implementation `mmul1`) (or state that it goes beyond 10 minutes). Compare the performance between CPU and GPU implementations and explain why one of them is better. It is preferred that you use your own implementation for this comparison, but for those who dropped HW02, simply predict the better approach and explain why it is superior.