ME 759
High Performance Computing for Engineering Applications
Assignment 7
Due Thursday 3/12/2020 at 9:00 PM

Submit responses to all tasks which don't specify a file name to Canvas in a file called assignment7.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 `HW07` 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 `HW07` of the ME759-2020 repo.

1. In HW05, you have implemented a reduction using the sequential addressing approach. In this task, you will compare the performance of `Thrust` and `CUB` with the previous GPU implementation by performing a scaling analysis for a reduction problem.

   a) Implement in a file called `task1_thrust.cu` the `Thrust` version of reduction. It's expected to do the following (some details about copying between host and device are not included here but should be implemented in your code when necessary):

   - Create and fill (with `int` type numbers) however you like a `thrust::host_vector` of length `n` where `n` is the first command line argument as below.
   - Use the built-in function in `Thrust` to copy the `thrust::host_vector` into a `thrust::device_vector`.
   - Call the `thrust::reduce` function to do a reduction.
   - Print the result of reduction.
   - Print the time taken to run the `thrust::reduce` function in *milliseconds* using CUDA events.
   - Compile: `nvcc task1_thrust.cu -Xcompiler -O3 -Xcompiler -Wall -Xptxas -O3 -o task1_thrust`
   - Run (where `n` is a positive integer): `./task1_thrust n`
   - Example expected output:
     `3141`
     `0.012`

   b) Implement in a file called `task1_cub.cu` the `CUB` version of the reduction based on the code example sent by Dan in an email (`deviceReduce.cu`)[1].

   - Stick with the same device memory allocation pattern as the code example (`DeviceAllocate()` and `cudaMemcpy()`). Do not use unified memory.
   - Modify the example program so that the host array `h_in` has length `n` where `n` is the first command line argument as below, then fill in `h_in` with `int` type numbers however you like.
   - Call the `DeviceReduce::Sum` function that outputs the reduction result to the output array.
   - Print the reduction result
   - Print the time taken to run the `DeviceReduce::Sum` function (the actual one, not the one that's used to find the size of temporary storage needed) in *milliseconds* using CUDA events.
   - Compile: `nvcc task1_cub.cu -Xcompiler -O3 -Xcompiler -Wall -Xptxas -O3 -o task1_cub`
   - Run (where `n` is a positive integer): `./task1_cub n`
   - Example expected output:
     `3141`
     `0.012`

   c) On an Euler *compute node*:

[1] Use `module load cuda/10` to compile `CUB` related files

- Run task1_thrust for value n $= 2^{10}, 2^{11}, \cdots, 2^{30}$ and generate a pattern of time vs. n in $\log_2 - \log_2$ scale.
- Run task1_cub for value n $= 2^{10}, 2^{11}, \cdots, 2^{30}$ and generate a pattern of time vs. n in $\log_2 - \log_2$ scale.
- Overlay the above two patterns on top of the plot you generated for HW05 task1 in task1.pdf.

2. In HW06, you've implemented the `scan` function using Hillis-Steele algorithm. In this task, you will implement the exclusive scan using the `Thrust` library and the `CUB` library and compare their efficiency with the previous implementation through a scaling analysis.

   a) Implement in a file called `task2_thrust.cu` the `Thrust` version of exclusive scan. It's expected to do the following (some details about copying between host and device are not included here but should be implemented in your code when necessary):

      - Create and fills (with `float` type numbers) however you like a `thrust::host_vector` of length `n` where `n` is the first command line argument as below.
      - Use the built-in function in `Thrust` to copy the `thrust::host_vector` into a `thrust::device_vector`.
      - Call the `thrust::exclusive_scan` function to fill another `thrust::device_vector` with the result of the exclusive scan.
      - Print the last element of the output array
      - Print the time taken to run the `thrust::exclusive_scan` function in *milliseconds* using CUDA events.
      - Compile: `nvcc task2_thrust.cu -Xcompiler -O3 -Xcompiler -Wall -Xptxas -O3 -o task2_thrust`
      - Run (where `n` is a positive integer): `./task2_thrust n`
      - Example expected output:
        `1560.3`
        `0.012`

   b) Implement in a file called `task2_cub.cu` the `CUB` version of the exclusive scan based on the code example sent by Dan in an email[1].

      - Stick with the same device memory allocation pattern as the code example (`DeviceAllocate()` and `cudaMemcpy()`). Do not use unified memory.
      - Modify the example program so that the host array `h_in` has length `n` where `n` is the first command line argument as below, then fill in `h_in` with `float` type numbers however you like.
      - Call the `DeviceScan::ExclusiveSum` function that fills the output array with the result of the exclusive scan.
      - Print the last element of the output array
      - Print the time taken to run the `DeviceScan::ExclusiveSum` function (the actual one, not the one that's used to find the size of temporary storage needed) in *milliseconds* using CUDA events.
      - Compile: `nvcc task2_cub.cu -Xcompiler -O3 -Xcompiler -Wall -Xptxas -O3 -o task2_cub`
      - Run (where `n` is a positive integer): `./task2_cub n`
      - Example expected output:
        `1560.3`
        `0.012`

   c) On an Euler *compute node*:

      - Run `task2_thrust` for value `n` $= 2^{10}, 2^{11}, \cdots, 2^{29}$ and generate a pattern of time vs. `n` in $\log_2 - \log_2$ scale.
      - Run `task2_cub` for value `n` $= 2^{10}, 2^{11}, \cdots, 2^{29}$ and generate a pattern of time vs. `n` in $\log_2 - \log_2$ scale.
      - Overlay the above two patterns on top of the plot you generated for HW06 `task2` in `task2.pdf`.

[1] Use `module load cuda/10` to compile `CUB` related files

3. (a) Implement in a file called `count.cu` the function `count` as declared and described in `count.cuh`. Your `count` function should be able to take a `thrust::device_vector`, for instance, named `d_in` (filled by integers), and fill the output `values` array with the unique integers that appear in `d_in` in ascending order, as well as the output `counts` array with the corresponding occurrences of these integers. A brief example is shown below:

   - Example input: `d_in` = [3, 5, 1, 2, 3, 1]
   - Expected output: `values` = [1, 2, 3, 5]
   - Expected output: `counts` = [2, 1, 2, 1]

   Hints:

   - Since the length of `values` and `counts` may not be equal to the length of `d_in`, you may want to use `thrust::inner_product` to find the number of "jumps" (when `a[i-1] != a[i]`) as you step through the sorted array (the input array is not sorted, so you would have to do a sort using `Thrust` built-in function). - See Lecture 18 slide 56 for an example. There are other valid options as well, for instance, `thrust::unique`.
   - `thrust::reduce_by_key` could be helpful.

   (b) Write a test program `task3.cu` which does the following:

   - Create and fill (with `int` type of numbers) however you like a `thrust::host_vector` of length `n` where `n` is the first command line argument as below.
   - Use the built-in function in `Thrust` to copy the `thrust::host_vector` into a `thrust::device_vector` as the input of your `count` function.
   - Use your `count` function to fill another two arrays with the result of `count`.
   - Print the last element of `values` array.
   - Print the last element of `counts` array.
   - Print the time taken to run the `count` function in *milliseconds* using CUDA events.
   - Compile: `nvcc task3.cu count.cu -Xcompiler -O3 -Xcompiler -Wall -Xptxas -O3 -o task3`
   - Run: (where `n` is a positive integer): `./task3 n`
   - Example expected output:
     ```
     370
     23
     0.13
     ```

   (c) On an Euler *compute node*, run `task3` for value `n` $= 2^5, 2^6, \cdots, 2^{24}$ and generate a plot of time vs. `n` in $\log_2 - \log_2$ scale in a file called `task3.pdf`.