

ME 759
High Performance Computing for Engineering Applications
Assignment 7
Due Thursday 10/27/2022 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). Submit all plots (if any) on Canvas. Do not zip your Canvas submission.

All *source files* should be submitted in the **HW07** subdirectory on the **main** branch of your homework git repo with no subdirectories. For this assignment, your **HW07** folder should contain **task1_cub.cu**, **task1_thrust.cu**, **task2.cu**, **count.cu** and **task3.cpp**.

Important note: All commands or code must work on *Euler* with the **nvidia/cuda/11.6.0.lua** module and **gcc/9.4.0** module loaded. Loading the modules is done via
\$ module load nvidia/cuda gcc/9.4.0

This is because *Euler* may be currently experiencing an environment bug that requires you to use **gcc/9.4.0** for compiling **Thrust**- or **CUB**-related code. **gcc/9.4.0** may not be a requirement in other working environments.

We encourage you to test on *Euler* before you submit your homework.

Please submit clean code. Consider using a formatter like **clang-format**.

IMPORTANT: Before you begin, copy any provided files from **Assignments/HW07** directory of the **ME759 Resource Repo**. Do not change any of the provided files since these files will be overwritten with clean, reference copies when grading.

1. (40 pts) In HW05, you have implemented a reduction using the first add during load approach. In this task, you will compare the performance of **Thrust** and **CUB** with the previous GPU implementation by performing a scaling analysis for the 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 random **float** numbers in the range $[-1.0, 1.0]$ 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 perform a reduction on the previously generated **thrust::device_vector**.
 - 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 -std c++17 -o task1_thrust**
 - Run by submitting a **sbatch** script (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 from [this link](#). Specifically, you should do the following.
 - 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 random **float** numbers in the range $[-1.0, 1.0]$.

- 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. It's recommended to remove the debug function wrapper when gauging the performance of the reduction.
 - Compile: `nvcc task1_cub.cu -Xcompiler -O3 -Xcompiler -Wall -Xptxas -O3 -std c++17 -o task1_cub`
 - Run by submitting a `sbatch` script (where `n` is a positive integer): `./task1_cub n`
 - Example expected output:

```
3141
0.012
```
- c) On an *Euler compute node*:
- Run `task1_thrust` for value $n = 2^{10}, 2^{11}, \dots, 2^{30}$ and generate a pattern of time vs. `n` in log – log scale.
 - Run `task1_cub` for value $n = 2^{10}, 2^{11}, \dots, 2^{30}$ and generate a pattern of time vs. `n` in log – log scale.
 - Overlay the above two patterns on top of the plot you generated for HW05 `task1` in a file called `task1.pdf`. If you dropped HW05, then using information derived from [this post](#) or other Piazza posts is fine.
- d) Comment on the performance of the three implementations you came up with above.

2. (35 pts)

- (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). You can refer to Lecture 18 and 19 for `thrust::sort` examples. There are other valid options as well, for instance, `thrust::unique`.
 - `thrust::reduce_by_key` could be helpful.
- (b) Write a test program `task2.cu` which does the following:
- Create and fill with random `int` numbers in the range `[0, 500]` 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.
 - Allocate two other `thrust::device_vectors`, `values` and `counts`, then call your `count` function to fill these two arrays with the results of this counting operation.
 - 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 task2.cu count.cu -Xcompiler -O3 -Xcompiler -Wall -Xptxas -O3 -std c++17 -o task2`
 - Run by submitting a `sbatch` script (where `n` is a positive integer): `./task2 n`
 - Example expected output:
`370`
`23`
`0.13`
- (c) On an Euler *compute node*, run `task2` for value `n = 25, 26, ..., 224` and generate a plot of time vs. `n` in log – log scale in a file called `task2.pdf`.

3. (25 pts) Write a C++ program in a file called `task3.cpp` which does the following:

- Launches four OpenMP threads.
- Prints out the number of threads launched, with the format `Number of threads: x` (followed by a newline), where `x` is the total number of threads. This should be printed only once.
- Lets each thread introduce itself, with the print format `I am thread No. i` (followed by a newline), where `i` is the thread number. Each thread should do that only once.
- Computes and prints out the factorial of integers from 1 to 8, `a!=b` (followed by a newline), where `a` is one of the 8 integers, and `b` is the result of `a!`. This should be done in parallel with all 4 threads.

How to go about it, and what the expected output looks like:

- Compile: `g++ task3.cpp -Wall -O3 -std=c++17 -o task3 -fopenmp`
- Run by submitting a `sbatch` script: `./task3`
- Example expected output (as you can see, the order matters not):

```
Number of threads: 4
I am thread No. 0
I am thread No. 3
I am thread No. 1
I am thread No. 2
3!=6
5!=120
4!=24
6!=720
7!=5040
8!=40320
1!=1
2!=2
```

Important note: this problem is meant to get you started with OpenMP, it is not CUDA anymore, so the following changes need to be made to your `slurm` script:

- `#SBATCH --gres=gpu:1` should be removed since GPU is not required in this assignment.
- `#SBATCH --nodes=1 --cpus-per-task=4` (or `-N 1 -c 4` for short) should be added, which requests one node with 4 cores. In this course, `--cpus-per-task` should generally be no more than 20.