MAP55616-03 - CUDA Exponential Integral Calculation

Yu Liu (liuy41@tcd.ie)

https://github.com/afkbjb/cuda3$_e xponential_i ntegral$

Task 1

First, I measured the average CPU execution times (using the `-g` option) for float and double precision at each problem size separately (averaged over multiple runs, recorded as `CPU_float_avg` and `CPU_double_avg`).

$n = m$	CPU float avg (ms)	CPU double avg (ms)
5000	778.280	1798.650
8192	1989.171	4555.087
16384	7414.355	17009.648
20000	10681.267	24788.801

```
//sample results
liuy41@cuda01:~/cuda3$ ./exponentialIntegral.out -n 5000 -m 5000 -g -t
./exponentialIntegral.out -n 8192 -m 8192 -g -t
./exponentialIntegral.out -n 16384 -m 16384 -g -t
./exponentialIntegral.out -n 20000 -m 20000 -g -t
==== Timing ====
CPU (float) : 782.324 ms
CPU (double): 1806.418 ms

==== Timing ====
CPU (float) : 2022.229 ms
CPU (double): 4561.238 ms

==== Timing ====
CPU (float) : 7545.378 ms
CPU (double): 17075.648 ms

==== Timing ====
CPU (float) : 10806.218 ms
CPU (double): 24830.597 ms
=================
```

CUDA Implementation

- Separate kernels `exponentialIntegralFloat` and `exponentialIntegralDouble` are implemented to compute float and double precision in parallel.
- The main program allocates memory, copies inputs, calls kernels, copies outputs, and checks errors for both precisions separately, ensuring $|CPU - GPU| \leq 1e - 5$.

Constant Memory

Constants such as Euler's constant, convergence threshold, and maximum iterations are stored in the `__constant__` memory area, optimizing shared read access by threads.

Performance Testing

I wrote a script `task1.sh` to test gpu performance at different sizes ($n = m = 5000, 8192, 16384, 20000$) and various block sizes (16 to 1024). The test results are saved to the file `gpu_only.csv`.

CUDA Streams

I created two cuda streams (`streamF` and `streamD`) to allow float and double data transfers and kernel launches to run concurrently. This improved bandwidth usage and kernel efficiency. The results with streams are saved in `gpu_streams.csv`. Compared to the version using only constant memory, the performance improved a bit but not much.

Block Size

Added `-B` parameter to support runtime setting of `threadsPerBlock` for testing different block sizes' performance.

Based on four rounds of testing (results in `runtime.csv`), considering execution time and stability of both float and double, the final chosen block sizes are:

$n = m$	Selected Block Size
5000	128
8192	128
16384	256
20000	256

Speedup

Combining GPU test results at the best block sizes with the CPU average times:

$n = m$	Best Blk	GPU float	GPU double	CPU float	CPU double	Float Speedup	Double Speedup
5000	128	17.733	69.433	778.280	1798.650	43.9	25.9
8192	128	46.575	181.750	1989.171	4555.087	42.7	25.1
16384	256	185.755	707.506	7414.355	17009.648	39.9	24.0
20000	256	273.976	1037.674	10681.267	24788.801	39.0	23.9

Task 2

Using Claude Sonnet 4 with the prompt:

"convert the following code into a CUDA version, use reasonable optimization, explain it".

The code was saved as `llm.cu` and was successfully tested. It uses a fixed block size of 256 threads per block. I didn't test other sizes to find a better setup.

Analysis

- **Constant memory:** The Euler constant and `maxIterations` are put into GPU constant memory so all threads can access them faster. It's similar to my implementation.
- **One kernel for float & double with simple timing:** It combined the float and double calculations into one kernel to save kernel launch time. But this means can't tune or measure their performance separately. Also, the timing method is basic, without splitting float and double.

Performance Comparison

LLM version works correctly, but my manually optimized version from Task 1 runs faster overall—especially for smaller sizes. It may because float and double are calculated together in one kernel, so the performance can't be separately optimized for each type.

In contrast, task1 uses two streams to run float and double kernels concurrently, so the total GPU time is limited by the slower kernel instead of the simple addition.

$n = m$	Task1 GPU (ms)	LLM (ms)
5000	69.433	303.494
8192	181.750	385.041
16384	707.506	1032.267
20000	1037.674	1295.653

```
//Sample results
./llm -n 5000 -m 5000 -c -t
./llm -n 8192 -m 8192 -c -t
./llm -n 16384 -m 16384 -c -t
./llm -n 20000 -m 20000 -c -t

GPU computation took: 0.303494 seconds
GPU computation took: 0.385041 seconds
GPU computation took: 1.032267 seconds
GPU computation took: 1.295653 seconds

./llm -n 5000 -m 5000 -g -t
./llm -n 8192 -m 8192 -g -t
./llm -n 16384 -m 16384 -g -t
./llm -n 20000 -m 20000 -g -t

CPU computation took: 2.743872 seconds
CPU computation took: 7.039535 seconds
CPU computation took: 26.538541 seconds
CPU computation took: 38.500617 seconds
```