Task 1 - CUDA Implementation of the Exponential Integral

The progress can be seen from：https://github.com/StarCloudes/cuda/commits/master/assignment03

Overview

The goal of this task was to port the existing CPU implementation of the exponential integral function $E_n(x)$ to CUDA, in both single and double precision. We implemented and tested GPU versions of the algorithm using custom CUDA kernels.

1、Original Code Test

```
kuangg@cuda01:~/homework/assignment03$ ./exponentialIntegral.out -n 5 -m 10 -v -t
n=5
numberOfSamples=10
a=0
b=10
timing=1
verbose=1
calculating the exponentials on the cpu took: 0.000053 seconds
CPU==> exponentialIntegralDouble (1,1)=0.219384 ,exponentialIntegralFloat  (1,1)=0.219384
CPU==> exponentialIntegralDouble (1,2)=0.0489005 ,exponentialIntegralFloat  (1,2)=0.0489005
CPU==> exponentialIntegralDouble (1,3)=0.0130484 ,exponentialIntegralFloat  (1,3)=0.0130484
CPU==> exponentialIntegralDouble (1,4)=0.00377935 ,exponentialIntegralFloat  (1,4)=0.00377935
```

2、Implementation Details

- We developed two GPU wrapper functions:
  - exponentialIntegralFloatGPUWrapper for single precision
  - exponentialIntegralDoubleGPUWrapper for double precision
- Each wrapper handles:
  - Device memory allocation and deallocation
  - Launching a CUDA kernel with grid-stride loops
  - Memory transfers between host and device
  - Timing using cudaEvent_t with millisecond precision
- We also implemented stream-based versions:
  - exponentialIntegralFloatGPUStreamWrapper
  - exponentialIntegralDoubleGPUStreamWrapper
  - These use asynchronous cudaMemcpyAsync, cudaMallocAsync, and kernel launches in a cudaStream_t.

3、Benchmarking Test

Run the script to run Benchmarking Test

```
cd src
make
./run_benchmark.sh
```

3.1 Results

Size	BlockSize	CPU_time	GPU_time	Speedup
5000x5000	64	2.728345	0.102564	26.6
5000x5000	128	2.745564	0.104004	26.4
5000x5000	256	2.725105	0.103805	26.25
5000x5000	512	2.735155	0.102859	26.59
8192x8192	64	6.991616	0.259345	26.96
8192x8192	128	7.021402	0.269926	26.01
8192x8192	256	6.994776	0.260889	26.81
8192x8192	512	7.016039	0.269749	26.01
16384x16384	64	26.281944	0.967365	27.17
16384x16384	128	26.28158	0.965503	27.22
16384x16384	256	26.26783	0.964185	27.24
16384x16384	512	26.343049	0.97571	27
20000x20000	64	38.138014	1.415732	26.94
20000x20000	128	38.249268	1.407306	27.18
20000x20000	256	38.139398	1.420187	26.86
20000x20000	512	38.268465	1.411075	27.12
8192x20000	64	16.965474	0.615016	27.59
8192x20000	128	16.986019	0.611524	27.78
8192x20000	256	16.965505	0.61157	27.74
8192x20000	512	17.083393	0.628707	27.17
16384x8192	64	13.266245	0.493576	26.88
16384x8192	128	13.18637	0.507545	25.98
16384x8192	256	13.191251	0.483294	27.29
16384x8192	512	13.189766	0.505569	26.09

3.2 Summary of Best Block Sizes (per matrix size):

Matrix Size	Best BlockSize	Speedup	GPU Time (s)	Reason
5000×5000	64	26.60×	0.102564	Smallest GPU time and highest speedup
8192×8192	64	26.96×	0.259345	Fastest execution (very close to 256)
16384×16384	256	27.24×	0.964185	Fastest overall result
20000×20000	128	27.18×	1.407306	Slightly faster than others
8192×20000	128	27.78×	0.611524	Highest overall speedup
16384×8192	256	27.29×	0.483294	Best GPU time and speedup for this size

4、Conclusion:

- **BlockSize 256 is the most consistent and generally best-performing configuration**, especially for large matrices like 16384×16384 or 16384×8192.
- The **highest speedup** was achieved in the case of 8192×20000 and 16384×8192, using **blockSize 128 and 256**, both exceeding **27.7×**.
- **BlockSize 64 only outperformed others in smaller sizes** (like 5000×5000), but was slightly slower for larger inputs.

Task 2 - LLM implementation

Here I used **Claude** to generate the coda code.

1、Overall Code Comparison

Feature / Module	My Implementation	LLM-Generated Implementation
Float/Double **device** funcs	✅ Implemented using **device** + **constant** memory	✅ Implemented, passes maxIterations as a parameter
Kernel logic	✅ Separate kernels for float/double (no shared memory)	✅ Includes standard and shared-memory-optimized kernels
Shared memory optimization	❌ Not used	✅ Uses **shared** constants to reduce redundant calculations
Texture memory optimization	❌ Not implemented	✅ Provided experimental kernel using tex1Dfetch()
Stream version	✅ Implemented with async malloc, memcpy, and kernel launch	✅ Uses cudaStream_t for float and double streams
CUDA timing	✅ Accurate with cudaEventRecord + ElapsedTime	❌ Uses gettimeofday(), less precise for CUDA timing
Unified wrapper function	❌ Separate wrapper for float/double	✅ computeExponentialIntegralsCuda() handles both
Error handling	⚠️ Minimal checks or default returns	✅ Robust checkCudaError() wrapper used throughout

2、Optimization Techniques Used by the LLM

Technique	Description
1. Shared Memory	Frequently used constants like a, b, and division = (b - a)/m are loaded into **shared** memory to avoid redundant computation or global loads.
2. Texture Memory	Experimental version using tex1Dfetch() to read a and b from CUDA texture memory. Texture memory is read-only and cached.
3. CUDA Streams	Float and double kernels run in separate CUDA streams to overlap kernel execution with memory transfers (cudaMemcpyAsync, cudaMallocAsync).
4. Unified Timing	Although less accurate (uses gettimeofday()), the total execution time for float and double paths is measured and reported separately.
5. Modular Design	Exposes computeExponentialIntegralsCuda(...) and computeExponentialIntegralsCudaAdvanced(...) with flags to toggle shared/stream/texture usage.
6. Grid/Block Configuration Logging	Logs the number of CUDA blocks and threads per block during launch to help with performance tuning.

3、Reuslts Comparison

Compared to the LLM-generated version, my CUDA implementation achieved higher performance across all test sizes. This is primarily due to more precise event-based timing, better memory layout, and kernel specialization without excessive abstraction.

While the LLM version provides a cleaner modular structure with support for shared and texture memory, those optimizations did not contribute measurable performance gains in this task. Both versions produce numerically accurate results.

```
kuangg@cuda01:~/homework/assignment03$ ./exponentialIntegral.out -n 10000 -m 10000 -g -t

  [GPU float] malloc time     : 0.000096 seconds
  [GPU float] kernel time     : 0.005155 seconds
  [GPU float] memcpy time     : 0.065075 seconds
  [GPU float] total cuda time : 0.070326 seconds
  [GPU double] malloc time     : 0.001047 seconds
  [GPU double] kernel time     : 0.188515 seconds
  [GPU double] memcpy time     : 0.130297 seconds
  [GPU double] total cuda time : 0.319859 seconds
  [CPU] Execution time: 10.130379 seconds
  [GPU] Execution time: 0.390186 seconds
  [Speedup] CPU / GPU total = 25.96x
  [Summary] Float mismatches : 0
  [Summary] Double mismatches: 0
kuangg@cuda01:~/homework/assignment03$
kuangg@cuda01:~/homework/assignment03$ ./AI/bin/exponentialIntegral -t  -n 10000 -m 10000
GPU: Computing 100000000 exponential integrals...
Launching float kernel with 390625 blocks of 256 threads
Launching double kernel with 390625 blocks of 256 threads
GPU computation completed successfully!
  Total time: 1.21609 seconds
  Float time: 0.178894 seconds
  Double time: 0.516112 seconds
Calculating the exponentials on the CPU took: 10.148440 seconds
Calculating the exponentials on the GPU took: 1.216091 seconds total
  - Float precision: 0.178894 seconds
  - Double precision: 0.516112 seconds
GPU speedup: 8.35x

=== Comparing CPU vs GPU Results ===
All results match within tolerance (1e-05)
```

```
kuangg@cuda01:~/homework/assignment03/AI$ ./bin/exponentialIntegral -t  -n 5000 -m 5000
GPU: Computing 25000000 exponential integrals...
Launching float kernel with 97657 blocks of 256 threads
Launching double kernel with 97657 blocks of 256 threads
GPU computation completed successfully!
  Total time: 0.506802 seconds
  Float time: 0.0457809 seconds
  Double time: 0.135983 seconds
Calculating the exponentials on the CPU took: 2.739665 seconds
Calculating the exponentials on the GPU took: 0.506802 seconds total
  - Float precision: 0.045781 seconds
  - Double precision: 0.135983 seconds
GPU speedup: 5.41x

=== Comparing CPU vs GPU Results ===
All results match within tolerance (1e-05)
kuangg@cuda01:~/homework/assignment03/AI$  cd ..
kuangg@cuda01:~/homework/assignment03$ ./exponentialIntegral.out -n 5000 -m 5000 -g -t
[GPU float] malloc time     : 0.000097 seconds
[GPU float] kernel time     : 0.001439 seconds
[GPU float] memcpy time     : 0.016517 seconds
[GPU float] total cuda time : 0.018053 seconds
[GPU double] malloc time     : 0.000110 seconds
[GPU double] kernel time     : 0.052171 seconds
[GPU double] memcpy time     : 0.032867 seconds
[GPU double] total cuda time : 0.085147 seconds
[CPU] Execution time: 2.741589 seconds
[GPU] Execution time: 0.103200 seconds
[Speedup] CPU / GPU total = 26.57x
[Summary] Float mismatches : 0
[Summary] Double mismatches: 0
```