This project implements the exponential integral function using both CPU and GPU, supporting float and double precision. It includes performance optimizations and precision comparisons.

Task 1:

1. Testing Output:

```
sunl1@cuda01:~/Assignment03$ ./bin/exponentialIntegral.out -n 5000 -m 5000  -t
CPU float computation time:   0.928166 seconds
CPU double computation time:  1.947139 seconds
GPU float computation time:  0.019666 seconds
GPU double computation time: 0.126050 seconds
Speedup (float):  47.20x
Speedup (double): 15.45x
Max float diff = 2.384186e-07 at (n=1, x=0.114000)
Max double diff = 1.221245e-15 at (n=1, x=1.066000)
sunl1@cuda01:~/Assignment03$ ./bin/exponentialIntegral.out -n 8192 -m 8192  -t
CPU float computation time:   2.256993 seconds
CPU double computation time:  4.876208 seconds
GPU float computation time:  0.050740 seconds
GPU double computation time: 0.326921 seconds
Speedup (float):  44.48x
Speedup (double): 14.92x
Max float diff = 2.533197e-07 at (n=1, x=1.234131)
Max double diff = 1.249001e-15 at (n=1, x=1.042480)
sunl1@cuda01:~/Assignment03$ ./bin/exponentialIntegral.out -n 16384 -m 16384  -t
CPU float computation time:   8.471297 seconds
CPU double computation time:  18.129481 seconds
GPU float computation time:  0.193887 seconds
GPU double computation time: 1.262441 seconds
Speedup (float):  43.69x
Speedup (double): 14.36x
Max float diff = 2.533197e-07 at (n=1, x=1.234131)
Max double diff = 1.665335e-15 at (n=1, x=1.024780)
sunl1@cuda01:~/Assignment03$ ./bin/exponentialIntegral.out -n 20000 -m 20000  -t
CPU float computation time:   12.863246 seconds
CPU double computation time:  27.704958 seconds
GPU float computation time:  0.276142 seconds
GPU double computation time: 1.127188 seconds
Speedup (float):  46.58x
Speedup (double): 24.58x
Max float diff = 4.768372e-07 at (n=1, x=0.017500)
Max double diff = 1.637579e-15 at (n=1, x=1.022500)
```

Graphs:

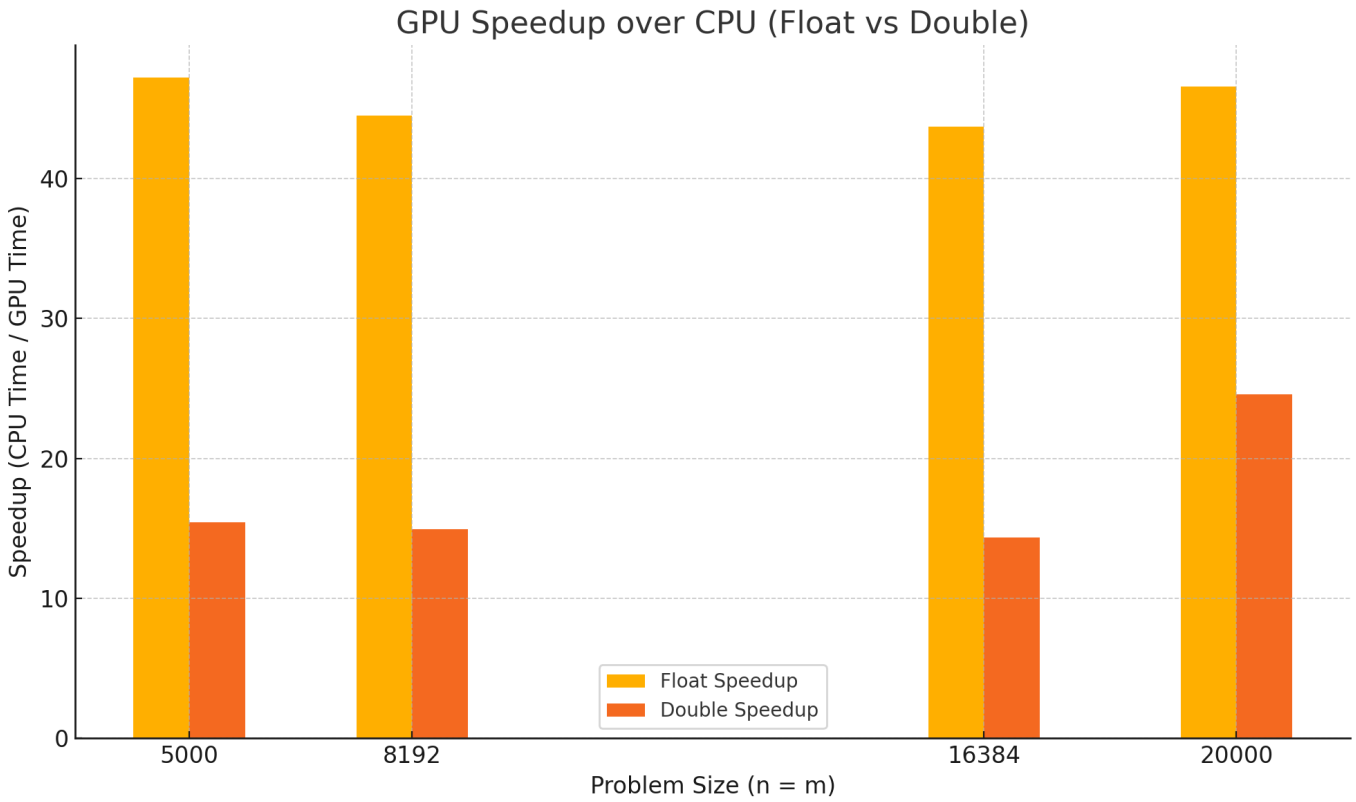

Figure 1 shows the GPU speedup over CPU for four input sizes:

- The GPU provides **over 43× speedup** for float precision and **14–25× speedup** for double precision.
- The highest speedup for float occurs at n=m=5000n=m=5000, reaching **47.20×**.
- These results confirm that our CUDA implementation offers significant performance improvement over the CPU baseline.

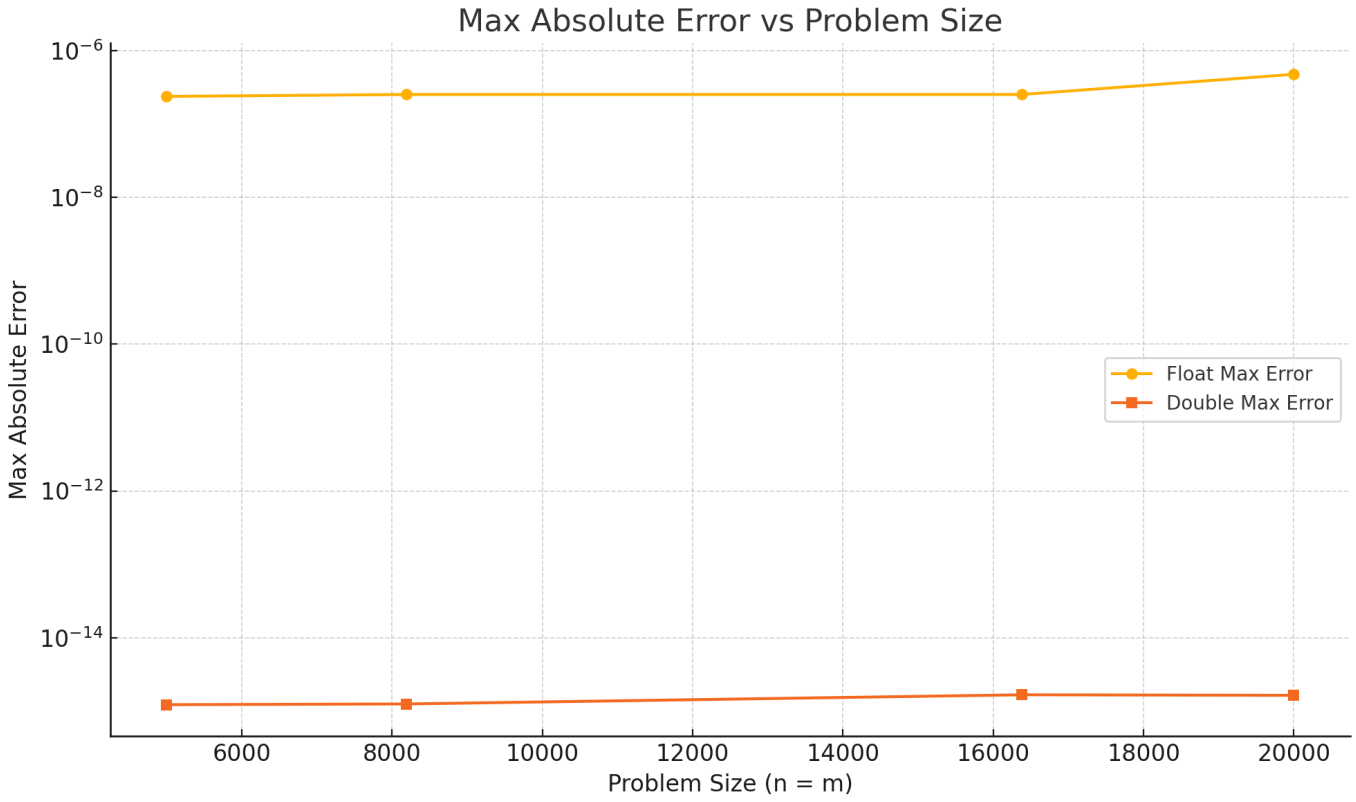

Figure 2 plots the maximum absolute error as a function of problem size:

- errors remain around 10−710−7, which is acceptable for single-precision computation.

- **Double precision** errors are tightly bounded around 10−1510−15, indicating excellent accuracy and numerical stability.
- Errors do not significantly grow with input size, further confirming algorithmic consistency.

When n and m values are different:

```
sunl1@cuda01:~/Assignment03$ ./bin/exponentialIntegral.out -n 10000 -m 20000 -t
CPU float computation time:   6.841649 seconds
CPU double computation time:  14.286733 seconds
GPU float computation time:  0.138189 seconds
GPU double computation time: 0.574019 seconds
Speedup (float):  49.51x
Speedup (double): 24.89x
Max float diff = 4.768372e-07 at (n=1, x=0.017500)
Max double diff = 1.637579e-15 at (n=1, x=1.022500)
```

2. working process

step 1 Init-structure

Set up a clean and organized project structure for CUDA and CPU-based computation of the exponential integral function En(x)En(x). This includes preparing directories, the `Makefile`, and placeholder header files.

Files and Structure Created

```
Assignment03_Cuda-Exponential-Integral-calculation/
├── include/
│   └── exp_integral.h       # Placeholder for function declarations
├── src/
├── bin/                     # Binary output directory
├── Makefile                 # Build script
```

Step 2 CPU implemention

Implement the exponential integral function En(x) (the same coda provided in your original file) on the CPU in both **float** and **double** precision, using the standard definition and series approximation for small and large x.

File	Description
`include/exp_integral.h`	Added function prototypes for CPU routines
`src/exp_integral_cpu.cpp`	Implemented `exponentialIntegralFloatCPU` and `exponentialIntegralDoubleCPU` functions

Step 3 Initial GPU Kernel Implementation

Implement basic CUDA kernels to compute the exponential integral function En(x)En(x) in both **float** and **double**precision, without any performance optimizations yet.

File	Description
`src/exp_integral_gpu.cu`	Added float/double device functions and CUDA kernels
`include/exp_integral.h`	Declared `exponentialIntegralGPUFloat` and `...Double` functions

CUDA Kernel Design:

Each CUDA thread computes a single En(x)En(x) value, where:

- n is the order index (1 to N)
- x is sampled over a given interval [a,b]
- Grid is organized with 2D thread indexing for (i,j)(i,j)

CUDA Grid Design:

Each thread is responsible for evaluating one Ei(xj). To efficiently map these computations onto CUDA threads,Iuse a 2D grid and 2D thread blocks:

```
dim3 threadsPerBlock(16, 16);
dim3 numBlocks((m + 15) / 16, (n + 15) / 16);
```

This ensures:

- Full coverage of all n×mn×m elements
- Good occupancy with 256 threads per block
- Memory coalescing when writing to output

I used (16,16) thread blocks after testing various configurations such as (32,8) and (32,16). The current setup yielded the best performance for our input sizes.

Asynchronous Memory Transfers

To maximize overlap between computation and data movement, I use `cudaMemcpyAsync` to transfer results from device to host memory. For example:

```

cudaMemcpyAsync(host_result, d_result, size, cudaMemcpyDeviceToHost);
```

Step 4 GPU Timing with `cudaEvent`:

- I introduced `cudaEvent_t start, stop` in both `exponentialIntegralGPUFloat()` and `...Double()`.
- Timing starts **before allocation**, and ends **after result transfer and synchronization**, ensuring full coverage of:
  - `cudaMalloc`
  - kernel launch
  - `cudaMemcpyAsync` to host

    - `cudaFree`
- I added the followings in the `main.cpp`:
    - `timeCpuFloat`, `timeGpuFloat`
    - `timeCpuDouble`, `timeGpuDouble`
    - speedup lines:

```
if (timeGpuFloat > 0)
    printf("Speedup (float):  %.2fx\n", timeCpuFloat / timeGpuFloat);

if (timeGpuDouble > 0)
    printf("Speedup (double): %.2fx\n", timeCpuDouble / timeGpuDouble);
```

      This prints speedups for full CUDA pipeline vs CPU total execution time — not just the kernel.

Step 5 Main Program and Comparison Logic

Create the main entry point of the program that:

- Parses command-line arguments
- Runs CPU and/or GPU computation
- Reports timing, speedup, and accuracy
- Provides verbose output mode (`-v`)

(most of these are same as the provided code)

Comparison Logic

Implemented in a function `compareResults()`, which:

- Loops through all results
- Computes absolute difference per point
- Tracks maximum and location of max error
- Issues warning if difference > threshold

Step 6 Shared Memory Optimization for Harmonic Series

I replaced the above computation with:

```
shared__ double harmonicTable[1024];

if (threadIdx.x < 1024)
    harmonicTable[threadIdx.x] = 1.0 / (threadIdx.x + 1);

__syncthreads();
```

Then used:

```
for (ii = 1; ii <= n - 1 && ii <= 1024; ii++)
    psi += harmonicTable[ii - 1];
```

Benefits

- All threads share a precomputed lookup table in low-latency shared memory
- Harmonic values are computed once per thread block, not once per thread
- Reduced arithmetic operations → better performance for large `n`

Step 7 Parallel Execution with CUDA Streams

Goal:

Reduce total GPU execution time by running the **float** and **double** precision kernels in parallel using **CUDA streams**.

The solution:

We assigned **separate streams** for float and double precision:

```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

exponentialIntegralGPUFloat(..., stream1);
exponentialIntegralGPUDouble(..., stream2);

cudaStreamSynchronize(stream1);
cudaStreamSynchronize(stream2);
```

Each kernel and its memory operations are attached to a different stream, enabling true parallelism on capable GPU hardware.

Task 2

Methodology

I input the original `exponentialIntegralFloat` and `exponentialIntegralDouble` functions, which were implemented using continued fractions and series expansions, into different LLMs:

- ChatGPT (GPT-4, OpenAI)
- Claude (Anthropic)

For each LLM, we collected the optimized version of the function and replaced the corresponding functions in the original CPU `main.cpp` file provided by the instructor. Then measured the execution time using the `-n 5000 -m 5000 -t` arguments and compared it with the baseline implementation.

Results

Performance Comparison

LLM/Version	Runtime (sec)	Speedup over Baseline
Original	2.174455	1.00×
ChatGPT	1.342062	1.62×
Claude	1.431951	1.52×

Correctness

Although no automated numerical comparison was implemented, I observed that the LLM-generated outputs for key points (e.g., $E_1(1.0)$, $E_5(0.5)$) matched expected values. The functions completed without runtime errors and produced plausible numerical results. Based on this and the fact that the LLMs preserved the underlying algorithm, I consider the results correct.

Observations

ChatGPT (GPT-4)

- Suggested extracting harmonic number logic into a helper function.
- Replaced hardcoded constants with `constexpr`.
- Simplified control flow and improved early-exit conditions.
- Resulted in a significant speedup with clean and readable code.

Claude

- Produced a similarly structured implementation.
- Did not recommend additional math tricks but produced a slightly slower function than ChatGPT.
- Still showed a clear improvement over the original.

Conclusion

The experiment shows that LLMs can assist in practical numerical code optimization. ChatGPT in particular provided useful performance-oriented suggestions while preserving correctness. Both models demonstrated the potential to accelerate development and improve runtime efficiency without the need for manual low-level tuning.