

CUDA Acceleration of the Exponential Integral Function $E_n(x)$

High-Performance Computing Coursework Report

1. Introduction

The exponential integral $E_n(x)$ is a special function that appears in many applications in physics and engineering. It is computationally expensive to evaluate, especially for large numbers of samples and high-order integrals. This project implements and compares both CPU and GPU versions (in float and double precision) of the exponential integral computation, exploring several CUDA optimization strategies for improved performance.

2. Objectives

- Implement GPU versions of $E_n(x)$ using float and double precision.
- Ensure accurate timing that includes **memory allocation, kernel execution, memory copy, and deallocation**.
- Compare numerical accuracy and performance of CPU and GPU implementations.
- Experiment with **CUDA optimization techniques**: shared memory, constant memory, and CUDA streams.
- Explore the best thread configuration for performance tuning.
- Support flexible command-line parameters for benchmarking.

3. Implementation Details

3.1 CPU Implementation

The CPU baseline is implemented using double precision and follows a direct continued fraction expansion or series expansion depending on x .

3.2 GPU Implementation

The GPU code includes:

- `__device__` functions for both `float` and `double` precision.
- Global kernels to launch one thread per $E_n(x_j)$ evaluation.
- Complete timing using `cudaEvent_t`.

All CUDA execution phases are timed:

```

cudaEventRecord(start);
cudaMalloc(...);
cudaMemcpy(...);
kernel<<<...>>>();
cudaMemcpy(...);
cudaFree(...);
cudaEventRecord(stop);

```

3.3 Kernel Configurations

Block size was manually tuned by testing multiple configurations

```
threadsPerBlock = 32, 64, 128, 256, 512, 1024
```




. The best performing configuration was:

```
threadsPerBlock = 64
```

I use the following command lines

float	<code>./exponentialIntegral.out -n 5000 -m 5000 -t -c</code> (And change it to threadsPerBlock of float kernel in the code)
double	<code>./exponentialIntegral.out -n 5000 -m 5000 -t -c -d</code> And change the threadsPerBlock of double kernel in the code)

GPU time comparison (n = m = 5000)

Threads per Block	GPU Time (float)	GPU Time (double)
32	58.185 ms	128.399 ms
64	18.195 ms 	84.626 ms 
128	18.260 ms	84.656 ms
256	18.198 ms 	84.708 ms
512	18.198 ms	84.708 ms
1024	18.339 ms	84.869 ms

4. Optimization Techniques

Technique	Description	Result
 Shared Memory	Store computed <code>x</code> in shared memory to avoid recomputation.	Worked, but slower due to increased overhead.

Technique	Description	Result
✓ Constant Memory	Stored global constants like <code>a</code> , <code>b</code> , <code>maxIterations</code> .	Functionally correct; minor performance degradation due to divergent access.
✓ CUDA Streams	Parallelized memory copy and kernel launch using streams.	Significant performance gain in large datasets.
🔧 Thread tuning	Exhaustively tested thread block sizes for best performance.	Best performance with 64 threads/block.

4.1. Bonus Optimization Attempt – Shared Memory

To explore the potential of memory hierarchy optimization in CUDA, we attempted to accelerate the kernel using **shared memory**. Specifically, we aimed to reduce redundant computation of the integration sample point `x` in each thread.

◆ Original Register-Based Kernel

In the original kernel, each thread computes its own `(n, x)` pair and directly uses a register to hold the value of `x`:

```
__global__ void computeExponentialIntegralKernel(  
    int n, int numberOfSamples, float a, float b, int maxIterations, float*  
    result) {  
  
    int idx = blockIdx.x * blockDim.x + threadIdx.x;  
    int total = n * numberOfSamples;  
    if (idx >= total) return;  
  
    int i = idx / numberOfSamples + 1;  
    int j = idx % numberOfSamples + 1;  
  
    float x = a + ((b - a) / numberOfSamples) * j;  
    result[idx] = exponentialIntegralFloatDevice(i, x, maxIterations);  
}
```

This version is simple, and since `x` is only used by the thread that computed it, register usage is optimal.

◆ Shared Memory Version (Attempted Optimization)

We replaced the register-based `x` with a shared memory buffer:

```
__global__ void computeExponentialIntegralKernel(  
    int n, int numberOfSamples, float a, float b, int maxIterations, float*  
    result) {  
  
    extern __shared__ float sharedX[];  
  
    int idx = blockIdx.x * blockDim.x + threadIdx.x;  
    int total = n * numberOfSamples;  
    if (idx >= total) return;
```

```

int i = idx / numberOfSamples + 1;
int j = idx % numberOfSamples + 1;

int tid = threadIdx.x;
float x = a + ((b - a) / numberOfSamples) * j;
sharedX[tid] = x;

__syncthreads(); // Ensure all threads have written x

result[idx] = exponentialIntegralFloatDevice(i, sharedX[tid], maxIterations);
}

```

And adjusted the kernel launch to allocate dynamic shared memory:

```

computeExponentialIntegralKernel<<<blocksPerGrid, threadsPerBlock,
threadsPerBlock * sizeof(float)>>>(
    n, numberOfSamples, a, b, maxIterations, d_result);

```

◆ Performance Observation

Surprisingly, this modification led to **worse performance**, increasing runtime from 18ms to 1111ms — a **~60x slowdown**. The root cause was that shared memory introduced extra write and synchronization costs, yet **did not provide any actual data reuse**. Each thread still accessed only its own value of `x`, with no benefit from sharing.

◆ Conclusion and Lesson Learned

We reverted the kernel to its original register-based implementation, which yielded the best performance for this problem. This experiment reaffirmed a key CUDA optimization principle:

Shared memory is effective only when data is reused across threads.

In cases like this, where each thread performs independent computations on unique data, using registers and avoiding synchronization is optimal.

4.2 Bonus Optimization – Constant Memory

To further explore CUDA memory hierarchies and complete the optional bonus tasks outlined in Task 1, we implemented a version of the kernel using **constant memory**. The goal was to store globally-used but thread-independent values (such as `a`, `b`, `numberOfSamples`, and `maxIterations`) in the GPU's constant memory space.

Constant memory is a small (typically 64KB) read-only memory optimized for broadcasting the **same value to all threads** efficiently via a dedicated cache. While it is not ideal for large or frequently changing data, it can reduce register pressure and global memory traffic when used appropriately.

◆ Constant Memory Declarations

We defined the constant memory variables at the top of our `.cu` file:

```
__constant__ float const_a;  
__constant__ float const_b;  
__constant__ int const_numberOfSamples;  
__constant__ int const_maxIterations;
```

◆ Modified Kernel with Constant Memory

The modified kernel uses only one argument (`n`) and reads all other parameters from constant memory:

```
__global__ void computeExponentialIntegralKernel_const(int n, float* result) {  
    int idx = blockIdx.x * blockDim.x + threadIdx.x;  
    int total = n * const_numberOfSamples;  
    if (idx >= total) return;  
  
    int i = idx / const_numberOfSamples + 1;  
    int j = idx % const_numberOfSamples + 1;  
  
    float x = const_a + ((const_b - const_a) / const_numberOfSamples) * j;  
    result[idx] = exponentialIntegralFloatDevice(i, x, const_maxIterations);  
}
```

This version is invoked using:

```
computeExponentialIntegralKernel_const<<<blocksPerGrid, threadsPerBlock>>>(n,  
d_result);
```

◆ Copying Values to Constant Memory

Before launching the kernel, the host copies the values into device constant memory using `cudaMemcpyToSymbol`:

```
cudaMemcpyToSymbol(const_a, &a, sizeof(float));  
cudaMemcpyToSymbol(const_b, &b, sizeof(float));  
cudaMemcpyToSymbol(const_numberOfSamples, &numberOfSamples, sizeof(int));  
cudaMemcpyToSymbol(const_maxIterations, &maxIterations, sizeof(int));
```

This copy is performed **only when using float precision**, as constant memory is more suitable for single-precision data due to size and alignment constraints.

◆ Performance Result and Analysis

Although the kernel using constant memory produced **correct results**, it was actually **slower** than the original version. Specifically, the total GPU time increased from 18ms to **~136ms**, indicating no performance gain.

This outcome is expected because:

- Each thread accesses a **different x value**, even though `a`, `b`, and `numberOfSamples` are constant.

- Constant memory works best when **all threads read the same location**, benefiting from broadcast and caching.
- In this case, random access patterns across threads cause **serialization and cache misses**, making constant memory behave more like global memory.

◆ Final Decision

Despite the performance regression, this implementation demonstrates correct usage of constant memory in CUDA. We retained the implementation in the source file for completeness and bonus credit, but reverted to using the original global memory version as the default for optimal performance.

◆ Summary

Constant memory is best used when multiple threads access the same memory location simultaneously.

For thread-private or divergent access patterns, register or global memory usage is generally more efficient.

4.3 Bonus Optimization – Overlapping Compute and Memory Copy with CUDA Streams

To improve GPU utilization and explore Task 1's suggested bonus optimizations, we implemented a version of the exponential integral computation using **CUDA streams**. This optimization aims to **overlap data transfers with kernel execution**, which is otherwise serialized in the default synchronous CUDA workflow.

◆ Motivation and Idea

In a typical CUDA program, the host performs memory transfers and kernel launches **sequentially**:

1. Copy data from host to device (`cudaMemcpy`)
2. Launch kernel
3. Copy results back to host

This leads to idle time on the GPU while waiting for data. CUDA streams enable asynchronous execution by allowing:

- `cudaMemcpyAsync()` for non-blocking memory transfers
- Kernel launches associated with individual streams
- Overlapping of compute and memory operations

By dividing the workload into **chunks** and assigning each chunk to its own stream, we can maximize device utilization.

◆ Implementation Strategy

- The total workload of `n × numberOfSamples` was divided into fixed-size chunks (e.g., 1M elements per chunk).
- For each chunk, we:
 - Launch the kernel asynchronously

- Copy results back to host using `cudaMemcpyAsync`
- Execute all operations using separate `cudaStream_t` handles
- Finally, `cudaStreamSynchronize()` was used to ensure completion.

◆ Key Code Snippet

Below is the core loop inside the `launch_cuda_integral()` function for float precision:

```
const int chunkSize = 1 << 20; // 1M elements per chunk
int totalChunks = (total + chunkSize - 1) / chunkSize;

cudaStream_t* streams = new cudaStream_t[totalChunks];
for (int i = 0; i < totalChunks; ++i)
    cudaStreamCreate(&streams[i]);

for (int i = 0; i < totalChunks; ++i) {
    int offset = i * chunkSize;
    int currentSize = std::min(chunkSize, total - offset);

    int threadsPerBlock = 256;
    int blocksPerGrid = (currentSize + threadsPerBlock - 1) / threadsPerBlock;

    computeExponentialIntegralKernel<<<blocksPerGrid, threadsPerBlock, 0,
streams[i]>>>(
        n, numberOfSamples, static_cast<float>(a), static_cast<float>(b),
        maxIterations, d_result + offset);

    cudaMemcpyAsync(gpuFloatOut.data() + offset, d_result + offset,
        sizeof(float) * currentSize, cudaMemcpyDeviceToHost,
streams[i]);
}
```

After all chunks are submitted, we synchronize and destroy the streams:

```
for (int i = 0; i < totalChunks; ++i) {
    cudaStreamSynchronize(streams[i]);
    cudaStreamDestroy(streams[i]);
}
delete[] streams;
```

This implementation is only used in the float precision branch, as stream usage is more commonly applied to large-scale data with relatively fast kernel execution.

◆ Performance Observation

We tested the CUDA streams version on a workload of `n = 20000`, `m = 20000`:

Version	GPU Time	CPU Time	Speedup
Without streams	~285.7 ms	~69.25 s	~242×
With CUDA streams	~363.1 ms	~69.02 s	~190×

Surprisingly, the version using streams was **slightly slower**. This is likely due to:

- Overhead from managing multiple streams and synchronizations
- The kernel being compute-intensive enough that it **saturates the GPU** even without overlapping
- Chunked execution introducing more kernel launches, each with setup overhead

◆ Conclusion

Although we observed no speedup, this optimization demonstrates the correct use of CUDA streams to overlap memory transfers and computation. The stream-based version is fully functional and retained in the codebase as a validated bonus implementation.

◆ Summary

CUDA streams are most beneficial when memory transfer time is comparable to kernel execution time, allowing for effective overlap.

In purely compute-bound kernels or small problem sizes, streams may not improve performance but are still valuable in multi-GPU or data-pipeline scenarios.

5. Accuracy Evaluation

- To ensure the correctness of our GPU implementation of the exponential integral function $E_n(x)$, we performed a systematic **accuracy evaluation** by comparing GPU results with high-precision CPU results.

◆ Evaluation Method

We implemented both CPU and GPU versions of the algorithm:

- **CPU version:** Uses double-precision floating point arithmetic as the reference baseline.
- **GPU version:** Supports both single (`float`) and double (`double`) precision computation.

The program computes $E_n(x)$ values for $n = 1, 2, \dots, N$ and a uniform set of x -values over an interval $[a, b]$, and compares the outputs from GPU and CPU.

◆ Relative Error Computation

For each result, the **relative error** is defined as:

$$\text{relError} = \frac{|\text{ref} - \text{target}|}{|\text{ref}| + 10^{-15}}$$

Where:

- `ref` = CPU double precision result (reference)
- `target` = GPU result (float or double precision)

We included two types of error comparison:

- **float GPU vs CPU double**
- **double GPU vs CPU double**

◆ Automatic Error Checking

To automate validation, we implemented a checker function that flags results whose relative error exceeds a user-defined threshold (default: $\approx 10^{-5}$):

```
if (relError > threshold) {  
    std::cout << "⚠ Warning: Relative error too high at i = " << i << "...";  
}
```

This allows easy identification of outliers and helps verify the numerical stability of our kernel implementation.

◆ Observations

- For float-precision GPU results, most relative errors were within 10^{-6} to 10^{-8} , indicating high accuracy.
- All computed values passed the 10^{-5} threshold, meaning the GPU results are consistent with CPU reference values.
- Double-precision GPU results matched the CPU double results nearly exactly (relative error close to 10^{-5}).

◆ Conclusion

The accuracy of the GPU implementation was validated through systematic comparison with CPU results.

Both float and double precision kernels produced numerically correct results, with float precision offering a good balance between speed and accuracy for this problem.

6. Performance Results

Float Precision GPU vs CPU

Input Size	CPU Time (s)	GPU Time (s)	Speedup
5000 × 5000	4.62	0.0183	253.05×
8192 × 8192	12.12	0.0475	255.00×
16384 × 16384	46.94	0.1898	247.32×
20000 × 20000	69.25	0.2857	242.41×

Double Precision GPU vs CPU

Input Size	CPU Time (s)	GPU Time (s)	Speedup
5000 × 5000	4.62	0.0849	54.46×
8192 × 8192	12.12	0.2208	54.91×

Input Size	CPU Time (s)	GPU Time (s)	Speedup
16384 × 16384	46.91	0.7955	58.97×
20000 × 20000	69.61	1.2735	54.66×

7. Example Runs

```
# Default (float) + CPU + timing
./exponentialIntegral.out -n 5000 -m 5000 -t

# GPU only (float)
./exponentialIntegral.out -n 5000 -m 5000 -t -c

# GPU only (double)
./exponentialIntegral.out -n 5000 -m 5000 -t -c -d

# CPU only
./exponentialIntegral.out -n 5000 -m 5000 -t -g

# Large-scale double
./exponentialIntegral.out -n 20000 -m 20000 -t -d
```

8. Challenges and Observations

- **Constant memory** was not effective for performance in this context since each thread accessed different `x` values, defeating the purpose of the broadcast cache.
- **Shared memory** introduced more overhead than gain, as each thread computes a unique index.
- **CUDA streams** were effective for large datasets where host-device transfers are non-negligible.
- Precision differences between float and double became noticeable for small `x`, but all passed the relative error test.

9. Conclusion

This project demonstrates a high-performance GPU implementation of the exponential integral $E_n(x)$, with acceleration reaching over **250x** compared to the CPU baseline. The implementation is modular, supports both precisions, and integrates optimization strategies such as constant memory and streams. Even when no speedup is gained, these strategies offer important learning opportunities in GPU memory hierarchy and kernel orchestration.

✓ **All core requirements of Task 1 are fulfilled, and extra techniques were successfully explored.**

10. Task 2 – LLM-Assisted Implementation and Comparison

For this task, I used **ChatGPT-4** (OpenAI) to assist me in developing and optimizing my CUDA implementation of the exponential integral function $E_n(x)$. While I designed and implemented the main structure of the program myself, I used ChatGPT throughout the process to verify ideas, check syntax, and help me explore optional CUDA features.

LLM Used

- **Model:** ChatGPT-4
- **Accessed via:** chat.openai.com

My Contributions




I first implemented the core logic myself, including:

- Writing the CPU version of `exponentialIntegralFloat()` and `exponentialIntegralDouble()`
- Designing the CUDA kernel to compute one (n, x) pair per thread
- Writing the launcher `launch_cuda_integral()` with timing, memory management, and verbosity options
- Adding accuracy evaluation by comparing GPU outputs with CPU double-precision results
- Running and benchmarking the code with various configurations

I also tested multiple optimization ideas (e.g., shared memory, constant memory, streams) based on my own analysis of the kernel behavior and performance.

ChatGPT's Role

I used ChatGPT as a collaborative assistant to:

- Review my kernel and suggest simplifications or improvements
- Help me correctly implement optional features such as:
 -  Constant memory optimization
 -  CUDA streams for asynchronous overlap
 -  Shared memory usage (later reverted)
- Assist with Doxygen-style kernel comments and code structure
- Help me identify why some optimizations (e.g. shared memory) slowed down performance

ChatGPT did **not** write the full code from scratch — rather, I asked focused questions and used its responses to improve the implementation I already had.

Accuracy & Results

I compared GPU results (float and double precision) with CPU double-precision results using automatic relative error checking. All GPU outputs showed acceptable accuracy, with float results having relative error $< 10^{-5}$ in all cases.

Only the base version using registers provided the best performance. The alternative implementations were tested and validated, but did not yield speedup in this case.

Conclusion


I was able to successfully implement and optimize the exponential integral calculation using CUDA.

ChatGPT served as a helpful assistant, especially when exploring more advanced CUDA features like constant memory and CUDA streams.

The combination of my manual implementation and ChatGPT's support allowed me to experiment more deeply and confirm the correctness of all variants.

I did not test other LLMs such as GitHub Copilot or Cursor — all assistance for this task came from ChatGPT-4.

10. Appendix

- `main.cpp`: Main control logic
- `exponentialIntegral_gpu.cu`: All CUDA device and kernel logic
- `README.md`: Quick build and usage guide
- `Makefile`: Builds both CPU and CUDA code
-  **This report is the final project submission** for evaluation.