CUDA Exponential Integral Report

colemaa3@tcd.ie

May 2025

1 CUDA Grid and Block Configuration for Large Problem Sizes

Efficient CUDA kernel execution requires carefully chosen grid and block layout to fully utilize GPU resources. For a 2D computation space such as the exponential integral evaluation over $n \times m$ sample points, we use a 2D grid of 2D blocks. The kernel indexes each sample at (i, j) in the range $[0, n) \times [0, m)$.

1.1 Recommended Block Size

A good CUDA block size balances between:

- Sufficient parallelism to keep GPU multiprocessors busy and hide idling with computation.
- Resource constraints (e.g., registers and shared memory, some are better for tasks than others depending on inputs (wrapped vectors etc.))
- Warp efficiency (multiples of 32 threads)

It is widely-known that the choice that works well for a broad range of problems with GPUs is:

```
dim3 block(32, 32);
```

This creates a 2D block with $32 \times 32 = 1024$ threads per block, which is the maximum allowed on most modern NVIDIA GPUs.

1.2 Grid Size Calculation

Given a block size of `(32, 32)`, the grid dimensions should cover the entire $n \times m$ domain:

$$\texttt{dim3 grid}\left(\left\lceil \frac{m}{32} \right\rceil, \left\lceil \frac{n}{32} \right\rceil\right)$$

Problem Size ($n \times m$)	Block Size	Grid Size
5000×5000	$(32, 32)$	$(157, 157)$
8192×8192	$(32, 32)$	$(256, 256)$
16384×16384	$(32, 32)$	$(512, 512)$
20000×20000	$(32, 32)$	$(625, 625)$

Table 1: Recommended grid and block dimensions for large problem sizes

1.3 Example Grid Sizes

1.4 Implementation Note

In the CUDA kernel launch in my source files, configuring this would appear as:

```
dim3 block(32, 32);
dim3 grid((m + 31) / 32, (n + 31) / 32);
computeKernel<<<grid, block>>>(...);
```

Using these dimensions ensures that:

- Every point in the $n \times m$ domain is covered
- No excess threads operate outofbounds (guarded with ifconditions inside the kernel)
- Parallelism maximized and memory coalescing ēncouraged!

2 Expected GPU Speedup vs CPU

The exponential integral computation is highly parallel and computationally intensive, making it wellsuited for GPU acceleration. Given the independence of each $E_n(x)$ evaluation, the problem maps efficiently to CUDA threads. The expected speedup of the GPU relative to the CPU increases with problem size due to improved utilization of GPU resources. We would expect the cost of parallelism and thread management overhead to be far outweighed by the benefits through speedup for much larger problem sizes, as I describe in Table 2.

2.1 Estimated Speedup by Problem Size

Expected GPU speedup over CPU for increasing problem sizes

2.2 Performance Factors for speedup

My results are not numerically correct for the GPU branch of computation, thus I discuss theoretical speedup factors based on problem size in the following section:

Problem Size ($n \times m$)	**Expected Speedup**	**Notes**
100×100	1x – 2x	GPU underutilized; overhead from memory copies and kernel launch dominate
5000×5000	10x – 30x	Parallelism better utilized; memory bandwidth more effective
8192×8192	20x – 50x	High thread occupancy and coalesced memory access
16384×16384	30x – 60x	Full GPU utilization; CPU likely bottlenecked by sequential execution
20000×20000	30x – 70x+	I would expect continued strong scaling assuming kernel efficiency and memory capacity

Table 2: Estimated GPU speedup over CPU for different problem sizes

- **Favouring GPU:** High compute intensity, independent data points, effective 2D grid/block usage.
- **Limiting GPU speedup:** Host-device memory transfer costs, slow convergence in iterations, limited double-precision throughput on consumer GPUs.
- **CPU Comparison:** Estimates assume single-threaded CPU execution. Multi-threaded CPU implementations would reduce the observed speedup.

3 Code Refactoring w/ LLM and potential Performance Improvements

I used the AI tool Kimi.ai to refactor my code for the exponential integral solving with CUDA. I still have systematic disparities between CPU and GPU results, GPU results are routinely 0, but the CPU appears to be evaluating the integral correctly at each stage. The new files are: `llm_main.cpp`, `llm_cpu_implement.cpp`, and `llm_cuda_implement.cu/.h`, which replace `main_imp.cu`, `cpu_implement.cpp`, and `cuda_implement.cu/.h`, respectively.

3.1 Modularization and Code Organization

- The functionality has been separated into three main components:
  - **CPU Logic**: Encapsulated in `llm_cpu_implement.cpp`, with floating- and double-precision versions of the exponential integral function and command-line parsing routines.
  - **GPU Logic**: Refactored into `llm_cuda_implement.cu/.h`, with clearly separated device functions, kernel launches, and memory management.
  - **Main Program Flow**: Delegated to `llm_main.cpp`, which coordinates CPU/GPU execution, result comparison, and timing.

- Header file `llm_cuda_implement.h` provides clean declarations for CUDA entry points and result output, enhancing readability and reducing coupling.

3.2 GPU Kernel Improvements

- Device functions for exponential integrals now include better numerical stability by guarding against division-by-zero and very small denominators via `EPSILON`.
- The GPU kernels (`computeKernelFloat`, `computeKernelDouble`) use 2D thread blocks for better parallelization over input ranges (n, m).
- Computed results are written directly into linear device arrays and mapped back to 2D vectors, improving host-device data transfer efficiency.

3.3 Performance-Oriented Enhancements

- Timing instrumentation is now included using `cudaEvent` for GPU and `gettimeofday` for CPU, enabling precise profiling.
- Memory allocations and deallocations are explicitly tracked and minimized to reduce GPU memory overhead.
- Error-prone command-line parsing logic was centralized and simplified, reducing the chance of misconfiguration.
- Result comparison with configurable thresholds (`1e-5`) ensures numerical accuracy while enabling performance trade-offs.

3.4 Expected Performance Impact

- **Execution Time Reduction:** Due to optimized GPU memory access, kernel parallelization, and CUDA streamlining.
- **Scalability:** The revised implementation scales better with larger sample sizes and higher orders due to improved memory and computation layout.
- **Maintainability:** Modular design makes future enhancements (e.g., mixed precision, batched kernels, asynchronous execution) easier to integrate.

3.5 Conclusion

In theory, the LLM redesigned file structure and logic significantly improve the separation of concerns and enable more efficient execution on both CPU and GPU. These changes are expected to result in improved performance and reduced overhead (threads, parallelism).