CUDA-Based Fibonacci Sequence

Goal:

The objective was to:

- Implement the Fibonacci sequence computation in CUDA.
- Compare CUDA's parallel implementation with a sequential CPU implementation.
- Analyze performance, scalability, and gain deeper understanding of GPU parallelism for a simple dependency-based algorithm.

My Approach:

1. Understanding the Problem

The Fibonacci sequence is a recursive problem where each term depends on the previous two terms.
While the formula looks simple, the dependency between terms creates a serial dependency chain that is not naturally parallel.

2. Using LLM to brainstorm parallel approaches:

LLM Prompt I gave to GPT:
"How can I write a CUDA kernel for computing Fibonacci sequence for N numbers? Is it parallelizable?"

GPT's Insight:

- Fibonacci sequence is not naturally parallel for standard computation.
- Two options:
  - Use recursion and memoization on CPU side.
  - Use wavefront parallelism or prefix sum-like techniques to parallelize Fibonacci in CUDA.
  - Or simply compute independent Fibonacci numbers using the closed-form Binet’s formula (approximate but parallel).

3. Implemented two versions:
   a. Sequential CPU version (Baseline):
      i. Simple loop-based code.
      ii. O(N) time complexity.
   b. CUDA GPU version:
      i. Each CUDA thread computes one Fibonacci number independently.
      ii. Used a simple iterative loop inside each thread.
      iii. Even though threads are independent, each thread computes its own Fibonacci number serially.
4. Code Setup
   a. Used Google Colab for CUDA access (Tesla K80 GPUs).
   b. CUDA kernel written in Fibonacci.ipynb.

c. Both CPU and GPU versions tested for same inputs (N up to 220).
d. Verified correctness of results between CPU & GPU versions.

I generated simple output graphs (see README.md and notebook) to visualize execution time differences as problem size grows.

Insights:

- Algorithms like Fibonacci highlight when hardware acceleration makes sense.
- Data dependency limits the gains of parallelism.
- CUDA shines best for tasks where computations across data elements are independent.