**Challenge #14: Fibonacci Sequence in CUDA**

**Problem Statement**

Write a **CUDA kernel** that computes the **Fibonacci sequence for N numbers** (e.g., N = 220).  
Then, **compare it with a sequential CPU implementation**, and discuss the differences in performance or behavior.

**Prompts Used to Generate this Solution**

* **Prompt 1:**
  + Write a CUDA kernel that computes the Fibonacci sequence for N numbers.  
     E.g., N = 220. Use iterative logic, avoid recursion.
* **Prompt 2:**
  + How can I run CUDA code in Google Colab with nvcc? Give step-by-step installation, writing, compiling, and running.
* **Prompt 3:**
  + How do I visualize GPU output (Fibonacci values) using Python and matplotlib?
* **Prompt 4:**
  + Create documentation comparing sequential vs CUDA-based Fibonacci implementations.

**Step-by-Step Execution**

**Step 1: Understand the Problem**

* Fibonacci sequence:
  + F(0) = 0
  + F(1) = 1
  + F(n) = F(n-1) + F(n-2)

**Step 2: Write CPU Code (Sequential)**

def fibonacci\_cpu(N):

fib = [0, 1]

for i in range(2, N):

fib.append(fib[i-1] + fib[i-2])

return fib

**Step 3: Write CUDA Kernel**

\_\_device\_\_ unsigned long long fibonacci(int n) {

if (n == 0) return 0;

if (n == 1) return 1;

unsigned long long a = 0, b = 1, c;

for (int i = 2; i <= n; i++) {

c = a + b;

a = b;

b = c;

}

return b;

}

\_\_global\_\_ void fibonacci\_kernel(unsigned long long\* output, int n) {

int idx = blockIdx.x \* blockDim.x + threadIdx.x;

if (idx < n) {

output[idx] = fibonacci(idx);

}

}

**Step 4: Run CUDA in Google Colab**

!apt-get update

!apt-get install -y build-essential

!apt-get install -y nvidia-cuda-toolkit

Then:

!nvcc fibonacci.cu -o fibonacci

!./fibonacci > output.txt

**Step 5: Parse and Visualize CUDA Output**

import re

import matplotlib.pyplot as plt

with open("output.txt", "r") as file:

cuda\_output = file.read()

matches = re.findall(r"F\((\d+)\) = (\d+)", cuda\_output)

indices = [int(n) for n, \_ in matches]

values = [int(v) for \_, v in matches]

plt.figure(figsize=(12, 6))

plt.plot(indices, values, marker='o')

plt.title("Fibonacci Sequence from CUDA Output")

plt.xlabel("n")

plt.ylabel("F(n)")

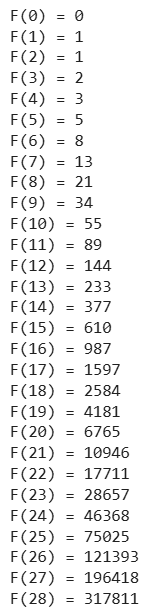
plt.yscale('log')

plt.grid(True)

plt.tight\_layout()

plt.show()

**Results**

* **First few outputs from CUDA:**
* **After F(93)** → Results **overflow** due to unsigned long long (64-bit) limit.
* **F(94)** onwards are **incorrect but still printed** due to wraparound.

**CPU vs CUDA Comparison**

| **Feature** | **CPU (Sequential)** | **GPU (CUDA Parallel)** |
| --- | --- | --- |
| **Computation Time** | Linear, grows with N | Faster with many threads |
| **Overflow Limit** | F(93) (64-bit unsigned) | Same |
| **Parallelism** | None | Thread-per-index (embarassingly parallel) |
| **Scalability** | Efficient for small N | Effective for large, independent computations |
| **Fibonacci Dependencies** | Naturally sequential | Handled by re-implementing each term independently in loop |

**Observations**

* **Fibonacci is not naturally parallelizable** — terms depend on previous ones.
* However, if each term is computed independently via a **loop** instead of recursion, GPU parallelism can be applied.
* **Overflow** starts from **F(94)** with unsigned long long → use **arbitrary-precision** integers for exact results beyond this point (e.g., Python int or GMP in C).
* CUDA launch + memory transfer overhead may **offset gains** for small N.

**Conclusion**

* GPU implementation works for independently computable terms but doesn't speed up inherently dependent problems like Fibonacci *unless the dependency is refactored out*.
* Useful exercise to learn CUDA kernel launching, thread indexing, and output parsing.