Challenge #14

1. Write a CUDA kernel that computes the Fibonacci sequence for N numbers. E.g., N = 220

%%writefile fibonacci.cu

#include <stdio.h>

\_\_global\_\_ void fibonacci\_kernel(unsigned long long \*fib, int N) {

  // Only one thread computes the whole sequence

  if (threadIdx.x == 0 && blockIdx.x == 0) {

    fib[0] = 0;

    if (N > 1) fib[1] = 1;

    for (int i = 2; i < N; ++i) {

      fib[i] = fib[i - 1] + fib[i - 2];

    }

  }

}

int main() {

  const int N = 93; // Upper limit for unsigned long long

  unsigned long long \*fib, \*d\_fib;

  // Allocate memory on host and device

  fib = (unsigned long long\*)malloc(N \* sizeof(unsigned long long));

  cudaMalloc(&d\_fib, N \* sizeof(unsigned long long));

  // Launch 1 thread to do the entire sequence computation

  fibonacci\_kernel<<<1, 1>>>(d\_fib, N);

  // Copy back result

  cudaMemcpy(fib, d\_fib, N \* sizeof(unsigned long long), cudaMemcpyDeviceToHost);

  // Print result

  for (int i = 0; i < N; i++)

    printf("F[%d] = %llu\n", i, fib[i]);

  // Free memory

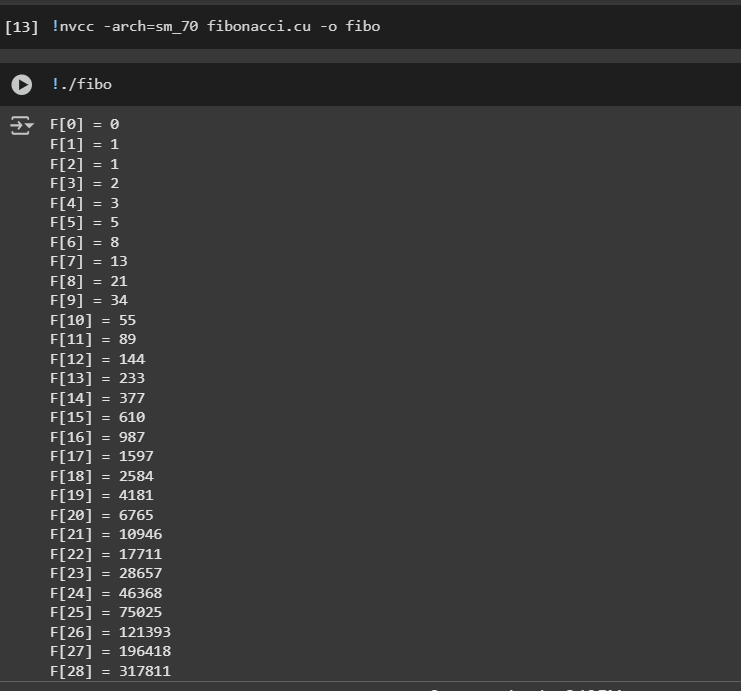
  cudaFree(d\_fib);

  free(fib);

  return 0;

}

Below is the output of the above code.



1. Compare a simple sequential implementation with your CUDA implementation. What do you  
   observe?

| **Feature** | **CPU Sequential Version** | **CUDA Version (Single Thread Kernel)** |
| --- | --- | --- |
| **Execution Model** | Entirely sequential loop on CPU | A single thread in a GPU kernel computes the sequence |
| **Memory Access** | Local in RAM | GPU global memory + transfer back to CPU |
| **Performance for Small N (≤ 93)** | Extremely fast, negligible time | Slower due to GPU call and memory transfer overhead |
| **Suitability for Large N (> 93)** | Needs BigInt (manual handling) | Same as CPU, but even more complex with GPU |
| **Ease of Implementation** | Very simple | Requires setup, allocation, memory management |

For N ≤ 93, the CPU version is superior:

* Faster (no kernel launch/memcpy overhead)
* Simpler (no CUDA setup required)
* No benefit from GPU parallelism (since it's still sequential inside CUDA)

CUDA is designed for **parallel** tasks. Fibonacci is **inherently serial** unless we reformulate the problem using **matrix exponentiation** or **closed-form formulas** like Binet’s formula — then we can parallelize computations like exponentiation or batched Fibonacci.

**Challenges Faced:**

1. **Incorrect Parallelization**: Initially attempted to compute each Fibonacci number using separate threads, which led to incorrect results due to race conditions and overwriting shared memory.
2. **Sequential Nature**: Realized Fibonacci sequence generation is inherently sequential (each value depends on the previous two), making it unsuitable for naive GPU parallelism.
3. **Overflow Limit**: Found that unsigned long long overflows after F93F\_{93}F93​, making it impossible to compute higher values without arbitrary-precision arithmetic.
4. **GPU Underutilization**: Using only one thread on the GPU to maintain correctness reduced the benefit of GPU acceleration.

**What I Learned:**

* CUDA is not always the best tool for problems with sequential dependencies.
* Understanding data dependencies is crucial when deciding how (or whether) to parallelize a problem.
* GPU computation benefits emerge when the workload is highly parallel — not when it is serial.

**Conclusion:**

The Fibonacci sequence is a poor candidate for GPU acceleration in its standard form. It taught me that not all problems benefit from CUDA — especially ones with tight value dependencies. For truly parallel acceleration, problems need to be restructured or reformulated, such as using matrix exponentiation or batched workloads.