Week 8 workshop

1. Create a CUDA program to find the first 1 million prime numbers that uses 10 blocks and 100 threads. For this question, you will need to use the formula found in the larger squaring program on Canvas to get the unique id of the thread. This is because you are using more than 1 block. Each thread should find the same amount of prime numbers and store them into an array on the GPU. Once all threads have been completed, the GPU array memory is copied back into the CPU memory, then print them out onto the terminal.

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week8$ cat prime_finder.cu
#include <stdio.h>
#include <cuda.h>

#define NUM_BLOCKS 10
#define THREADS_PER_BLOCK 100
#define TOTAL_THREADS (NUM_BLOCKS * THREADS_PER_BLOCK)

#define PRIMES_PER_THREAD 1000
#define TOTAL_PRIMES (TOTAL_THREADS * PRIMES_PER_THREAD)


__device__ int is_prime(int n) {
    if (n < 2) return 0;
    if (n == 2) return 1;
    if (n % 2 == 0) return 0;
    for (int i = 3; i * i <= n; i += 2) {
        if (n % i == 0)
            return 0;
    }
    return 1;
}

__global__ void find_primes(int *d_primes) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    int count = 0;
    int number = tid * 10000 + 2;

    int base_index = tid * PRIMES_PER_THREAD;

    while (count < PRIMES_PER_THREAD) {
        if (is_prime(number)) {
            d_primes[base_index + count] = number;
            count++;
        }
        number++;
    }
}
```

```
int main() {
    int *h_primes;
    int *d_primes;

    size_t size = TOTAL_PRIMES * sizeof(int);


    h_primes = (int *)malloc(size);
    if (h_primes == NULL) {
        printf("Failed to allocate host memory\n");
        return 1;
    }


    cudaError_t err = cudaMalloc((void **)&d_primes, size);
    if (err != cudaSuccess) {
        printf("cudaMalloc failed: %s\n", cudaGetErrorString(err));
        free(h_primes);
        return 1;
    }

    printf("Finding %d primes using %d blocks and %d threads...\n",
           TOTAL_PRIMES, NUM_BLOCKS, THREADS_PER_BLOCK);


    dim3 blocks(NUM_BLOCKS);
    dim3 threads(THREADS_PER_BLOCK);
    find_primes<<<blocks, threads>>>(d_primes);


    err = cudaGetLastError();
    if (err != cudaSuccess) {
        printf("Kernel launch failed: %s\n", cudaGetErrorString(err));
        cudaFree(d_primes);
        free(h_primes);
        return 1;
    }


    err = cudaDeviceSynchronize();
    if (err != cudaSuccess) {
        printf("cudaDeviceSynchronize failed: %s\n", cudaGetErrorString(err));
        cudaFree(d_primes);
        free(h_primes);
        return 1;
    }


    err = cudaMemcpy(h_primes, d_primes, size, cudaMemcpyDeviceToHost);
    if (err != cudaSuccess) {
        printf("cudaMemcpy failed: %s\n", cudaGetErrorString(err));
        cudaFree(d_primes);
        free(h_primes);
        return 1;
    }

    printf("Successfully found primes!\n\n");


    printf("First 50 primes:\n");
    for (int i = 0; i < 50 && i < TOTAL_PRIMES; i++) {
        printf("%d ", h_primes[i]);
        if ((i + 1) % 10 == 0) printf("\n");
    }


    printf("\n\nLast 50 primes:\n");
    for (int i = TOTAL_PRIMES - 50; i < TOTAL_PRIMES; i++) {
        printf("%d ", h_primes[i]);
        if ((i - (TOTAL_PRIMES - 50) + 1) % 10 == 0) printf("\n");
    }

    printf("\n\nTotal primes found: %d\n", TOTAL_PRIMES);


    cudaFree(d_primes);
    free(h_primes);

    return 0;
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week8$
```

output

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week8$ ls
index  index.cu  prime_finder.cu
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week8$ nvcc -o prime_finder prime_finder.cu
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week8$ ./prime_finder
Finding 1000000 primes using 10 blocks and 100 threads...
Successfully found primes!

First 50 primes:
2 3 5 7 11 13 17 19 23 29
31 37 41 43 47 53 59 61 67 71
73 79 83 89 97 101 103 107 109 113
127 131 137 139 149 151 157 163 167 173
179 181 191 193 197 199 211 223 227 229


Last 50 primes:
10005431 10005451 10005473 10005487 10005497 10005509 10005511 10005539 10005559 10005581
10005601 10005607 10005623 10005629 10005631 10005641 10005661 10005679 10005683 10005703
10005727 10005731 10005773 10005791 10005823 10005829 10005833 10005857 10005859 10005871
10005881 10005883 10005887 10005929 10005937 10005953 10005967 10005973 10005977 10005997
10006001 10006039 10006043 10006063 10006067 10006097 10006109 10006111 10006121 10006151


Total primes found: 1000000
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week8$
```

2. Create a program that calculates an estimate of Pi using the Leibniz algorithm in CUDA. You should first create a standard CPU program (preferably multithreaded) and then convert this into CUDA. Do a comparison of your CPU program to your CUDA version and analyse the time difference. Change the number of blocks and threads and see whether this has any effect on the time. We will discuss this at the next Lecture. Below is the formula:

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week8$ cat pi_cpu.cu
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/time.h>
#include <math.h>
#include <cuda.h>

#define NUM_CPU_THREADS 8
#define NUM_TERMS 1000000000L
#define M_PI 3.14159265358979323846

typedef struct {
    long start;
    long end;
    double result;
} ThreadData;

double get_time() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec + tv.tv_usec * 1e-6;
}

void* cpu_leibniz_thread(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    double sum = 0.0;
    for (long k = data->start; k < data->end; k++) {
        double term = (k % 2 == 0 ? 1.0 : -1.0) / (2.0 * k + 1.0);
        sum += term;
    }
    data->result = sum;
    return NULL;
}

double cpu_leibniz_single() {
    double sum = 0.0;
    for (long k = 0; k < NUM_TERMS; k++) {
        double term = (k % 2 == 0 ? 1.0 : -1.0) / (2.0 * k + 1.0);
        sum += term;
    }
    return 4.0 * sum;
}

double cpu_leibniz_multi() {
    pthread_t threads[NUM_CPU_THREADS];
    ThreadData thread_data[NUM_CPU_THREADS];
    long terms_per_thread = NUM_TERMS / NUM_CPU_THREADS;

    for (int i = 0; i < NUM_CPU_THREADS; i++) {
        thread_data[i].start = i * terms_per_thread;
        thread_data[i].end = (i + 1) * terms_per_thread;
        if (i == NUM_CPU_THREADS - 1) thread_data[i].end = NUM_TERMS;
        pthread_create(&threads[i], NULL, cpu_leibniz_thread, &thread_data[i]);
    }

    double sum = 0.0;
    for (int i = 0; i < NUM_CPU_THREADS; i++) {
        pthread_join(threads[i], NULL);
        sum += thread_data[i].result;
    }

    return 4.0 * sum;
}

__global__ void cuda_leibniz(double* partial_sums, long num_terms) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int total_threads = gridDim.x * blockDim.x;
    double sum = 0.0;

    for (long k = tid; k < num_terms; k += total_threads) {
        double term = (k % 2 == 0 ? 1.0 : -1.0) / (2.0 * k + 1.0);
        sum += term;
    }

    partial_sums[tid] = sum;
}

double gpu_leibniz(int num_blocks, int num_threads) {
    int total_threads = num_blocks * num_threads;
    double* h_partial_sums = (double*)malloc(total_threads * sizeof(double));
    double* d_partial_sums;

    cudaMalloc(&d_partial_sums, total_threads * sizeof(double));
    cuda_leibniz<<<num_blocks, num_threads>>>(d_partial_sums, NUM_TERMS);
    cudaDeviceSynchronize();
    cudaMemcpy(h_partial_sums, d_partial_sums, total_threads * sizeof(double), cudaMemcpyDeviceToHost);

    double sum = 0.0;
    for (int i = 0; i < total_threads; i++) {
        sum += h_partial_sums[i];
    }

    cudaFree(d_partial_sums);
```

```
}

double gpu_leibniz(int num_blocks, int num_threads) {
    int total_threads = num_blocks * num_threads;
    double* h_partial_sums = (double*)malloc(total_threads * sizeof(double));
    double* d_partial_sums;

    cudaMalloc(&d_partial_sums, total_threads * sizeof(double));
    cuda_leibniz<<<num_blocks, num_threads>>>(d_partial_sums, NUM_TERMS);
    cudaDeviceSynchronize();
    cudaMemcpy(h_partial_sums, d_partial_sums, total_threads * sizeof(double), cudaMemcpyDeviceToHost);

    double sum = 0.0;
    for (int i = 0; i < total_threads; i++) {
        sum += h_partial_sums[i];
    }

    cudaFree(d_partial_sums);
    free(h_partial_sums);

    return 4.0 * sum;
}

int main() {
    double pi_estimate, start_time, end_time;

    printf("Calculating Pi using Leibniz formula with %ld terms\n", NUM_TERMS);
    printf("Actual Pi value: %.10f\n", M_PI);
    printf("================================================================\n\n");

    // CPU Single-threaded
    start_time = get_time();
    pi_estimate = cpu_leibniz_single();
    end_time = get_time();
    printf("CPU Single-threaded:\n");
    printf("  Pi estimate: %.10f\n", pi_estimate);
    printf("  Error: %.10f\n", fabs(M_PI - pi_estimate));
    printf("  Time: %.6f seconds\n\n", end_time - start_time);

    // CPU Multi-threaded
    start_time = get_time();
    pi_estimate = cpu_leibniz_multi();
    end_time = get_time();
    printf("CPU Multi-threaded (%d threads):\n", NUM_CPU_THREADS);
    printf("  Pi estimate: %.10f\n", pi_estimate);
    printf("  Error: %.10f\n", fabs(M_PI - pi_estimate));
    printf("  Time: %.6f seconds\n\n", end_time - start_time);

    printf("================================================================\n");
    printf("CUDA Performance Analysis:\n");
    printf("================================================================\n\n");

    int configs[][2] = {
        {10, 100},
        {100, 100},
        {256, 256},
        {512, 512},
        {1000, 256},
        {1024, 1024}
    };

    int num_configs = sizeof(configs) / sizeof(configs[0]);

    for (int i = 0; i < num_configs; i++) {
        int blocks = configs[i][0];
        int threads = configs[i][1];

        start_time = get_time();
        pi_estimate = gpu_leibniz(blocks, threads);
        end_time = get_time();

        printf("CUDA Configuration: %d blocks x %d threads = %d total threads\n",
               blocks, threads, blocks * threads);
        printf("  Pi estimate: %.10f\n", pi_estimate);
        printf("  Error: %.10f\n", fabs(M_PI - pi_estimate));
        printf("  Time: %.6f seconds\n\n", end_time - start_time);
    }

    printf("================================================================\n");

    return 0;
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week8$
```

Output

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/week8$ ./pi_cpu
Calculating Pi using Leibniz formula with 1000000000 terms
Actual Pi value: 3.1415926536
================================================================

CPU Single-threaded:
  Pi estimate: 3.1415926526
  Error: 0.0000000010
  Time: 2.288179 seconds

CPU Multi-threaded (8 threads):
  Pi estimate: 3.1415926526
  Error: 0.0000000010
  Time: 0.362342 seconds

================================================================
CUDA Performance Analysis:
================================================================

CUDA Configuration: 10 blocks x 100 threads = 1000 total threads
  Pi estimate: 3.1415926526
  Error: 0.0000000010
  Time: 0.958926 seconds

CUDA Configuration: 100 blocks x 100 threads = 10000 total threads
  Pi estimate: 3.1415926526
  Error: 0.0000000010
  Time: 0.153560 seconds

CUDA Configuration: 256 blocks x 256 threads = 65536 total threads
  Pi estimate: 3.1415926526
  Error: 0.0000000010
  Time: 0.105227 seconds

CUDA Configuration: 512 blocks x 512 threads = 262144 total threads
  Pi estimate: 3.1415926526
  Error: 0.0000000010
  Time: 0.108159 seconds

CUDA Configuration: 1000 blocks x 256 threads = 256000 total threads
  Pi estimate: 3.1415926526
  Error: 0.0000000010
  Time: 0.102635 seconds

CUDA Configuration: 1024 blocks x 1024 threads = 1048576 total threads
  Pi estimate: 3.1415926526
  Error: 0.0000000010
  Time: 0.111645 seconds
```