```c
/*
 * ME 2054 Parallel Scientific Computing
 * Project 1 - Finite Difference Solution of a Vibrating 2D Membrane on a GPU
 * Due: November 6,2018
 *
 * Author: Dustin (Ting-Hsuan) Ma
 *
 * Compile: nvcc -O2 GPUmembrane.cu -o GPUrun.exe
 * Clang: ../clang-format -i GPUmembrane.cu
 */

#include "timer.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/resource.h>

#define LX 4.0f
#define LY 2.0f
#define NX 41
#define NY 21
#define DX LX / (REAL)(NX - 1)
#define DY LY / (REAL)(NY - 1)
#define H DX

#define C sqrt(5.f)
#define DT 0.4f * (H / C)

#define ENDTIME 1.0f
#define INFINITE 50

// Making calculation part easier
#define IC i + j *NX
#define IP1 (i + 1) + j *NX
#define IM1 (i - 1) + j *NX
#define JP1 i + (j + 1) * NX
#define JM1 i + (j - 1) * NX

#define BLOCKSIZE 16

typedef float REAL;
typedef int   INT;

__global__ void GPU_calculatingWave(REAL *now, REAL *old, REAL *out)
{
        // Row index and Colum index
        INT i = blockIdx.x * blockDim.x + threadIdx.x;
        INT j = blockIdx.y * blockDim.y + threadIdx.y;

        // Linear indexing
        INT ic = IC, ip1 = IP1, im1 = IM1, jp1 = JP1, jm1 = JM1;

        if (i > 0 && i < NX - 1 && j > 0 && j < NY - 1) {
                out[ic] = 2.f * now[ic] - old[ic]
                        + ((C * C * DT * DT) / (H * H))
                          * (now[ip1] + now[im1] + now[jp1] + now[jm1] - 4.f * now[ic]);
        }
}

void CPU_calculatingWave(REAL *now, REAL *old, REAL *out)
{
        for (INT j = 1; j < NY - 1; j++) {
                for (INT i = 1; i < NX - 1; i++) {
                        INT ic = IC, ip1 = IP1, im1 = IM1, jp1 = JP1, jm1 = JM1;
                        out[ic] = 2.f * now[ic] - old[ic]
                                + ((C * C * DT * DT) / (H * H))
                                  * (now[ip1] + now[im1] + now[jp1] + now[jm1] - 4.f * now[ic]);
                }
        }
}

void initializeMatrices(REAL *in)
{
        INT  i, j, idx;
        REAL x, y = 0.0f;

        for (j = 0; j < NY; j++) {
                x = 0.0f;
                for (i = 0; i < NX; i++) {
                        idx = IC;
                        // Eq 2.
                        in[idx] = 0.1f * (4.f * x - (x * x)) * (2.f * y - (y * y));
                        x += DX;
                }
                y += DY;
        }
```



```c
}

void applyingBoundary(REAL *in)
{
        INT i, j, idx;
        for (j = 0; j < NY; j++) {
                for (i = 0; i < NX; i++) {
                        idx = IC;
                        // Eq 4 - 7
                        if (i == 0 || i == NX) in[idx] = 0.0f;
                        if (j == 0 || j == NY) in[idx] = 0.0f;
                }
        }
}

void initializeSolution(REAL *in, REAL *out)
{
        for (INT j = 1; j < NY - 1; j++) {
                for (INT i = 1; i < NX - 1; i++) {
                        INT ic = IC, ip1 = IP1, im1 = IM1, jp1 = JP1, jm1 = JM1;

                        out[ic] = in[ic]
                                + (0.5 * (C * C * DT * DT) / (H * H))
                                  * (in[ip1] + in[im1] + in[jp1] + in[jm1] - 4.f * in[ic]);
                }
        }
}

void analyticalSolution(REAL *out)
{
        INT  idx, i, j;
        REAL x, m, n, y = 0;

        for (j = 0; j < NY; j++) {
                x = 0.f;
                for (i = 0; i < NX; i++) {
                        idx = IC;
                        for (m = 1.f; m <= INFINITE; m += 2.f) {
                                for (n = 1.f; n <= INFINITE; n += 2.f) {
                                        out[idx] += 0.426050f / (m * m * m * n * n * n)
                                                * cos(ENDTIME * sqrt(5.f) * M_PI / 4.f * sqrt(m
* m + 4.f * n * n))
                                                * sin(m * M_PI * x / 4.f) * sin(n * M_PI * y / 2
.f);
                                }
                        }
                        x += DX;
                }
                y += DY;
        }
}

void outputMatrix(REAL *in)
{
        INT i, j, idx;
        for (j = 0; j < NY; j++) {
                for (i = 0; i < NX; i++) {
                        idx = i + j * NX;
                        printf("%7.3f ", in[idx]);
                }
                printf("\n");
        }
        printf("\n");
}

INT main()
{
        printf("End time = %f\n", ENDTIME);

        // Running CFL check
        if (sqrt(C) * DT / H < 1.0f) {
                printf("CFL condition is met\n");
        } else {
                printf("CFL condition is not met, try again\n");
                return EXIT_SUCCESS;
        }

        // Calculating Analitical Solution
        REAL *Exact_phi = (REAL *) calloc(NX * NY, sizeof(*Exact_phi));
        analyticalSolution(Exact_phi);
        printf("========================Exact Solution=========================\n");
        outputMatrix(Exact_phi);

        // Allocating memory for CPU
        REAL *phi     = (REAL *) calloc(NX * NY, sizeof(*phi));
        REAL *phi_old = (REAL *) calloc(NX * NY, sizeof(*phi_old));
```

```
	REAL *phi_new = (REAL *) calloc(NX * NY, sizeof(*phi_new));

	// Initializing Mesh, boundaries, and solution
	initializeMatrices(phi);
	applyingBoundary(phi);
	initializeSolution(phi, phi_old);

	// Starting time
	double start, finish;

	GET_TIME(start);

	// Solving time function until endtime is reached
	REAL time = 0.0f, *tmp_CPU;
	while (time < ENDTIME) {
		CPU_calculatingWave(phi, phi_old, phi_new);

		tmp_CPU = phi;
		phi     = phi_new;
		phi_new = phi_old;
		phi_old = tmp_CPU;

		time += DT;
	}
	GET_TIME(finish);

	// Outputing CPU solution
	printf("========================Final Solution CPU=========================\n");
	outputMatrix(phi);
	printf("elapsed wall time (Host) = %3.5f s\n", (finish - start));
	printf("\n");

	// Allocating memory for GPU
	REAL *phi_d, *phi_old_d, *phi_new_d;
	cudaMallocManaged(&phi_d, NX * NY * sizeof(*phi_d));
	cudaMallocManaged(&phi_new_d, NX * NY * sizeof(*phi_new_d));
	cudaMallocManaged(&phi_old_d, NX * NY * sizeof(*phi_old_d));

	// Restarting Problem for GPU
	initializeMatrices(phi_d);
	applyingBoundary(phi_d);
	initializeSolution(phi_d, phi_old_d);

	// Setting up device enviromnet
	dim3 dimBlock(BLOCKSIZE, BLOCKSIZE);
	dim3 dimGrid(NX / dimBlock.x + 1, NY / dimBlock.y + 1);

	// Time enviorment to test cuda speed up
	REAL        elapsedTime;        // records in [ms]
	cudaEvent_t timeStart, timeStop; // cudaEvent_t initializes variable used in event time
	cudaEventCreate(&timeStart);
	cudaEventCreate(&timeStop);
	cudaEventRecord(timeStart, 0);

	// Solving time function until endtime is reached
	REAL Itertime = 0.0f, *tmp_GPU;

	while (Itertime < ENDTIME) {
		GPU_calculatingWave<<<dimGrid, dimBlock>>>(phi_d, phi_old_d, phi_new_d);
		cudaDeviceSynchronize();

		tmp_GPU   = phi_d;
		phi_d     = phi_new_d;
		phi_new_d = phi_old_d;
		phi_old_d = tmp_GPU;

		Itertime += DT;
	}

	cudaEventRecord(timeStop, 0);
	cudaEventSynchronize(timeStop);
	cudaEventElapsedTime(&elapsedTime, timeStart, timeStop);

	// Outputting Matrix
	printf("========================Final Solution GPU=========================\n");
	outputMatrix(phi_d);
	printf("elapsed wall time (Device) = %3.5f s\n", elapsedTime / 1000);

	// Deallocating memory
	cudaEventDestroy(timeStart);
	cudaEventDestroy(timeStop);

	free(phi);
	free(phi_old);
	free(phi_new);
	free(Exact_phi);
```



```
	cudaFree(phi_d);
	cudaFree(phi_new_d);
	cudaFree(phi_old_d);

	phi     = NULL;
	phi_old = NULL;
	phi_new = NULL;

	Exact_phi = NULL;

	return EXIT_SUCCESS;
}
```