```
/* Project 1
 * Finite Difference Solution of a Vibrating
 * 2D Membrane on a GPU
 * Author: Shawn Hinnebusch
 * Date: 10/30/2020
 *

 * To compile locally: nvcc -O3 -o project1.exe  project1.cu -lm

 * To compile on the CRC:
 * crc-interactive.py -g -u 1 -t 1 -p gtx1080
 * nvcc -O3 -arch=sm_61 -o project1.exe  project1.cu -lm

 * To run:
 *
 * ./project1.exe

 * Create PDF:
 * a2ps project1.cu --pro=color --columns=2 -E --pretty-print='c' -o project1.ps
 | ps2pdf project1.ps

 * Compress: tar czvf Hinnebusch_proj1.tar.gz project1/
 */

#include "timer_nv.h"
#include <float.h>
#include <limits.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>

#define BLOCK_SIZE_X 16
#define BLOCK_SIZE_Y 16
#define MACHINE_PRECISION 1e-10
#define LOOP_MAX 2000
#ifndef TIME
#define TIME 2.0
#endif
#ifndef LX
#define LX 0.50
#endif
#define PI M_PI

typedef float REAL;
typedef int   INT;

void surfaceOutput(const INT nx, const INT ny, const REAL *x)
{
    INT   i, j, ic;
    FILE *output;
    output = fopen("SurfaceOutput.dat", "w");

    for (j = 0; j < ny; j++) {
        for (i = 0; i < nx; i++) {
            ic = j * nx + i;
            fprintf(output, "%f", x[ ic ]);
            if (i < (nx - 1)) { fprintf(output, " "); }
        }
        fprintf(output, "\n");
    }
    fclose(output);
}

void compareOutput(const INT nx, const INT ny, const REAL *analyticSol, const RE
AL *cpu, const REAL *gpu, const REAL dx,
                   const INT i)
{
```



```
    INT  j, ic;
    REAL x = i * dx;

    char fileName[ 50 ] = "results_time=";
    char timeChar[ 64 ];
    sprintf(timeChar, "%.2f", TIME);
    strcat(fileName, timeChar);

    strcat(fileName, "_x=");

    char xValue[ 64 ];
    sprintf(xValue, "%.2f", x);
    strcat(fileName, xValue);
    strcat(fileName, ".dat");

    FILE *output;
    output = fopen(fileName, "w");

    fprintf(output, " y   \tAnalytic\tCPU   \tGPU\n");

    for (j = 0; j < ny; j++) {
        ic     = j * nx + i;
        REAL y = j * dx; // dx = dy
        fprintf(output, "%f\t%f\t%f\t%f\n", y, analyticSol[ ic ], cpu[ ic ], gpu[ ic
 ]);
    }
    fclose(output);
}

void initialize(REAL *matrix, const INT nx, const INT ny, const REAL dx, const R
EAL dy)
{
    INT i, j, ic;

    for (j = 1; j < (ny - 1); j++) {
        for (i = 1; i < (nx - 1); i++) {
            ic             = j * nx + i;
            REAL x         = i * dx;
            REAL y         = j * dy;
            matrix[ ic ] = 0.1 * (4.0 * x - x * x) * (2.0 * y - y * y);
        }
    }
}

void phiFirstIteration(const REAL *phiCurrent, REAL *phiPrev, const INT nx, cons
t INT ny, const REAL h, const REAL dt)
{
    INT  i, j, ic, IP1, IM1, jP1, jM1;
    REAL waveConst = 5.0 * dt * dt / (2.0 * h * h);

    for (j = 1; j < (ny - 1); j++) {
        for (i = 1; i < (nx - 1); i++) {
            ic             = j * nx + i;
            IP1            = j * nx + (i + 1);
            IM1            = j * nx + (i - 1);
            jP1            = (j + 1) * nx + i;
            jM1            = (j - 1) * nx + i;
            phiPrev[ ic ] = phiCurrent[ ic ]
                            + waveConst
                              * (phiCurrent[ IP1 ] + phiCurrent[ IM1 ] + phiCurr
ent[ jM1 ] + phiCurrent[ jP1 ]
                                 - 4.0 * phiCurrent[ ic ]);
        }
    }
}

void phiNext(REAL *phiNew, const REAL *phiCurrent, const REAL *phiPrev, const IN
T nx, const INT ny, const REAL h,
             const REAL dt)
```

```
{
    INT  i, j, ic, IP1, IM1, jP1, jM1;
    REAL waveConst = 5.0 * dt * dt / (h * h);

    for (j = 1; j < (ny - 1); j++) {
        for (i = 1; i < (nx - 1); i++) {
            ic             = j * nx + i;
            IP1            = j * nx + (i + 1);
            IM1            = j * nx + (i - 1);
            jP1            = (j + 1) * nx + i;
            jM1            = (j - 1) * nx + i;
            phiNew[ ic ] = 2.0 * phiCurrent[ ic ] - phiPrev[ ic ]
                           + waveConst
                           * (phiCurrent[ IP1 ] + phiCurrent[ IM1 ] + phiCurre
nt[ jM1 ] + phiCurrent[ jP1 ]
                              - 4.0 * phiCurrent[ ic ]);
        }
    }
}

__global__ void phiNextGPU(REAL *phiNew, const REAL *phiCurrent, const REAL *phi
Prev, const INT nx, const INT ny,
                           const REAL h, const REAL dt)
{
    INT i, j, ic, IP1, IM1, jP1, jM1;
    i = blockIdx.x * blockDim.x + threadIdx.x;
    j = blockIdx.y * blockDim.y + threadIdx.y;

    REAL waveConst = 5.0 * dt * dt / (h * h);

    if (i != 0 && i < (nx - 1) && j != 0 && j < (ny - 1)) {
        ic  = j * nx + i;
        IP1 = j * nx + (i + 1);
        IM1 = j * nx + (i - 1);
        jP1 = (j + 1) * nx + i;
        jM1 = (j - 1) * nx + i;
        phiNew[ ic ]
        = 2.0 * phiCurrent[ ic ] - phiPrev[ ic ]
          + waveConst
            * (phiCurrent[ IP1 ] + phiCurrent[ IM1 ] + phiCurrent[ jM1 ] + phiCu
rrent[ jP1 ] - 4.0 * phiCurrent[ ic ]);
    }
}

REAL phiInnerLoop(const REAL x, const REAL y, const REAL t)
{
    REAL result   = 0;
    REAL residual = 0;

    REAL prevIter = 1e3;
    for (INT m = 1; m < LOOP_MAX; m += 2) {
        for (INT n = 1; n < LOOP_MAX; n += 2) {
            residual = (1.0 / (m * m * m * n * n * n) * cos((t * sqrt(5.0) * PI
* 0.25) * sqrt(m * m + 4.0 * n * n))
                        * sin(m * PI * x * 0.25) * sin(n * PI * y * 0.5));
            result   = result + residual;

            if ((fabs(residual) / fabs(result)) < MACHINE_PRECISION) { break; }
        }
        if ((fabs(result - prevIter) / fabs(result)) < MACHINE_PRECISION) { brea
k; }
        prevIter = result;
    }

    result = 0.426050 * result;
    return result;
}

void analyticalSolFunc(const INT nx, const INT ny, const REAL dx, const REAL dy,
```



```
 const REAL t, REAL *matrix)
{
    INT ic;

    for (int j = 1; j < (ny - 1); j++) {
        for (int i = 1; i < (nx - 1); i++) {
            ic           = j * nx + i;
            REAL x       = i * dx;
            REAL y       = j * dy;
            matrix[ ic ] = phiInnerLoop(x, y, t);
        }
    }
}

void findMax(const REAL *phi, const INT nx, const INT ny, const INT iter, REAL *
maxValue, INT *maxValueTime)
{
    INT i, j, ic;

    for (j = 1; j < (ny - 1); j++) {
        for (i = 1; i < (nx - 1); i++) {
            ic = j * nx + i;
            if (fabs(phi[ ic ]) > *maxValue) {
                *maxValue     = fabs(phi[ ic ]);
                *maxValueTime = iter;
            }
        }
    }
}

INT main( )
{
    // mesh size must be 513x257, 1026x513, and 2049x1025
    const int  nx     = 41;
    const int  ny     = 21;
    const REAL length = 4.0;
    const REAL width  = 2.0;
    const int  size   = ny * nx;
    const REAL dx     = length / (nx - 1);
    const REAL dy     = width / (ny - 1);
    const REAL h      = dx;
    REAL *     temp;

    // multiple by 0.1 to make sure dt is small enough
    const REAL dt          = 1.0 * h / sqrt(5.0) * 0.1;
    const int  numOfLoops  = ( int ) ceil(TIME / dt);
    INT        part3ConstX = ( int ) ceil(LX * length / dx);

    // ########################## Alloc Memory ##########################

    // Alloc memory for arrays
    REAL *phiCurrent_GPU, *phiPrev_GPU, *phiNew_GPU;
    cudaMallocManaged(&phiCurrent_GPU, size * sizeof(*phiCurrent_GPU));
    cudaMallocManaged(&phiPrev_GPU, size * sizeof(*phiPrev_GPU));
    cudaMallocManaged(&phiNew_GPU, size * sizeof(*phiNew_GPU));

    // Memory Allocation for CPU only functions
    REAL *analyticSol    = ( REAL * ) calloc(nx * ny, sizeof(*analyticSol));
    REAL *phinew_CPU     = ( REAL * ) calloc(nx * ny, sizeof(*phinew_CPU));
    REAL *phiCurrent_CPU = ( REAL * ) calloc(nx * ny, sizeof(*phiCurrent_CPU));
    REAL *phiPrev_CPU    = ( REAL * ) calloc(nx * ny, sizeof(*phiPrev_CPU));

    // ########################## Analytical Solution #######################
    // Solve the Analytic Solution
    analyticalSolFunc(nx, ny, dx, dy, TIME, analyticSol);

    // ########################## Create mesh and init ######################

    // GPU Initialize to beginning value and calculate prev iter
```

```
    initialize(phiCurrent_GPU, nx, ny, dx, dy);
    phiFirstIteration(phiCurrent_GPU, phiPrev_GPU, nx, ny, h, dt);

    // CPU Initialize to beginning value and calculate prev iter
    initialize(phiCurrent_CPU, nx, ny, dx, dy);
    phiFirstIteration(phiCurrent_CPU, phiPrev_CPU, nx, ny, h, dt);

    // ########################### GPU Run ###########################
    // Setup CUDA and solve the finite difference
    dim3 threadsPerBlock(BLOCK_SIZE_X, BLOCK_SIZE_Y);
    dim3 numBlocks((nx - 1) / threadsPerBlock.x + 1, (ny - 1) / threadsPerBlock.
y + 1);

    // Start timing
    cudaEvent_t timeStart, timeStop;
    cudaEventCreate(&timeStart);
    cudaEventCreate(&timeStop);
    float gpu_elapsedTime;          // type float, precision is milliseconds!!
    cudaEventRecord(timeStart, 0); // 2nd argument zero, cuda streams

    // Loop to go forward in time
    for (int i = 0; i < numOfLoops; i++) {
        phiNextGPU< < < numBlocks, threadsPerBlock > > >(phiNew_GPU, phiCurrent_
GPU, phiPrev_GPU, nx, ny, h, dt);
        cudaDeviceSynchronize( );
        temp           = phiPrev_GPU;
        phiPrev_GPU    = phiCurrent_GPU;
        phiCurrent_GPU = phiNew_GPU;
        phiNew_GPU     = temp;
    }
    // stop time
    cudaEventRecord(timeStop, 0);
    cudaEventSynchronize(timeStop);
    cudaEventElapsedTime(&gpu_elapsedTime, timeStart, timeStop);

    // ########################### CPU Run ###########################
    // Used to find max
    // REAL maxValue = 0;
    // INT maxValueTime = 0;

    StartTimer( );

    int i;
    for (i = 0; i < numOfLoops; i++) {
        phiNext(phinew_CPU, phiCurrent_CPU, phiPrev_CPU, nx, ny, h, dt);
        temp           = phiPrev_CPU;
        phiPrev_CPU    = phiCurrent_CPU;
        phiCurrent_CPU = phinew_CPU;
        phinew_CPU     = temp;
        // max function for part 4
        // findMax(phiCurrent_CPU,nx, ny,i, & maxValue, & maxValueTime);
    }

    double cpu_elapsedTime = GetTimer( ); // elapsed time is in seconds
    printf("elapsed wall time (CPU) = %5.4f ms\n", cpu_elapsedTime * 1000.);
    printf("elapsed wall time (GPU) = %5.4f ms\n\n", gpu_elapsedTime);
    cudaEventDestroy(timeStart);
    cudaEventDestroy(timeStop);

    // max magnitude for part 4
    // printf("max = %f\n",maxValue);
    // printf("time at max = %f\n",maxValueTime*dt);

    // ########################### Write to output ###########################
    // Write output to file
    // constant value of x
    compareOutput(nx, ny, analyticSol, phiCurrent_CPU, phiCurrent_GPU, dx, part3
ConstX);
    // entire matrix to plot a surface
```



```
    surfaceOutput(nx, ny, phiCurrent_CPU);

    // Free variables
    free(analyticSol);
    free(phinew_CPU);
    free(phiCurrent_CPU);
    free(phiPrev_CPU);
    cudaFree(phiCurrent_GPU);
    cudaFree(phiPrev_GPU);
    cudaFree(phiNew_GPU);

    return EXIT_SUCCESS;
}
```