```
/* Homework Assignment #5
 * Simple CPU program to add two long vectors
 * and to calculate a 1D stencil
 * Author: Shawn Hinnebusch
 * Date: 10/19/2020
 *
 * Part 2 code is VECTORADD 1
 * Part 3-5 code is VECTORADD 0

 * To compile locally: nvcc -O3 -o hw5.exe  hw5.cu -lm

 * To compile on the CRC:
 * crc-interactive.py -g -u 1 -t 1 -p gtx1080
 * nvcc -O3 -gencode arch=compute_61,code=sm_61 -o hw5.exe  hw5.cu -lm

 * To run:
 * ./hw5.exe <vector size>
 * ./hw5.exe 1e8

 * Create PDF:
 * a2ps hw5.cu --pro=color --columns=2 -E --pretty-print=’c’ -o hw5.ps | ps2pdf
hw5.ps

 * Compress: tar czvf Hinnebusch_hw5.tar.gz hw05/
 */

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

#define VECTORADD 0 // else stencil

typedef float REAL;
typedef int   INT;
#define RADIUS 3
#define BLOCK_SIZE 256

// ########################### Stencil 1D Functions ###########################
__global__ void stencil_1d(const REAL *in, REAL *out, const INT n)
{
    __shared__ REAL temp[ BLOCK_SIZE + 2 * RADIUS ];
    INT             gindex = threadIdx.x + blockIdx.x * blockDim.x;
    INT             lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    __syncthreads( );
    temp[ lindex ] = in[ gindex ];
    // Fills temp vector with previous value unless its the first block
    if (threadIdx.x < RADIUS && gindex > RADIUS) {
        temp[ lindex - RADIUS ] = in[ gindex - RADIUS ];
        // Check to not exceed the largest block size before filling in
        // the last 2 ghost cells
        if (gindex + BLOCK_SIZE < n) { temp[ lindex + BLOCK_SIZE ] = in[ gindex
+ BLOCK_SIZE ]; }
    }
    __syncthreads( );

    // Apply the stencil
    REAL result = 0.0;
    if (gindex >= RADIUS && gindex < (n - RADIUS)) {
        for (int offset = -RADIUS; offset <= RADIUS; offset++) {
            result += temp[ lindex + offset ];
        }
    }

    // Store the result
    out[ gindex ] = result;
```

```
}

void stencil_1d_cpu(const REAL *in, REAL *out, const INT n)
{
    for (int i = RADIUS; i < (n - RADIUS); i++) {
        for (int offset = -RADIUS; offset <= RADIUS; offset++) {
            out[ i ] += in[ i + offset ];
        }
    }
}

// ########################### 2 Vector Functions ###########################

__global__ void vector_add_gpu(const INT n, const REAL *a, const REAL *b, REAL *
c)
{
    INT tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < n) c[ tid ] = a[ tid ] + b[ tid ];
}

void vector_add_cpu(const INT n, const REAL *a, const REAL *b, REAL *c)
{
    for (int i = 0; i < n; i++)
        c[ i ] = a[ i ] + b[ i ];
}

// ########################### Main Function ###########################

int main(INT argc, char *argv[])
{
    if (argc < 2) {
        perror("Command-line usage: executableName <vector size>");
        exit(1);
    }

    int n = atof(argv[ 1 ]);

    printf("N: %d\n", n);

    // Initialize and alloc memory for arrays
    REAL *x, *y;
    cudaMallocManaged(&x, n * sizeof(*x));
    cudaMallocManaged(&y, n * sizeof(*y));

#if VECTORADD
    REAL *z;
    cudaMallocManaged(&z, n * sizeof(*z));
    // Init vectors for addition
    for (int i = 0; i < n; i++) {
        x[ i ] = 3.5;
        y[ i ] = 1.5;
    }
#else // stencil
    // Init vectors for stencil
    // note y is for the result should start at zero
    for (int i = 0; i < n; i++) {
        x[ i ] = 1.0;
        y[ i ] = 0.0;
    }
#endif

    // CPU Run
    StartTimer( );

#if VECTORADD
    vector_add_cpu(n, x, y, z);
#else // stencil
    stencil_1d_cpu(x, y, n);
```

```
#endif

    double cpu_elapsedTime = GetTimer( ); // elapsed time is in seconds

    // GPU Run
    cudaEvent_t timeStart, timeStop;
    cudaEventCreate(&timeStart);
    cudaEventCreate(&timeStop);
    float gpu_elapsedTime; // type float, precision is milliseconds (ms) !!!

    int nBlocks = (n + BLOCK_SIZE - 1) / BLOCK_SIZE;

    cudaEventRecord(timeStart, 0); // 2nd argument zero, cuda streams

#if VECTORADD
    vector_add_gpu< < < nBlocks, BLOCK_SIZE > > >(n, x, y, z);
#else // stencil
    stencil_1d< < < nBlocks, BLOCK_SIZE > > >(x, y, n);
#endif

    cudaDeviceSynchronize( );

    cudaEventRecord(timeStop, 0);
    cudaEventSynchronize(timeStop);

    cudaEventElapsedTime(&gpu_elapsedTime, timeStart, timeStop);

    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);

    // Used to print final results of CPU or GPU
    // for (int i = 0; i < n; i++) {
    //     printf("%f\n", y[i]);
    //}

    cudaFree(x);
    cudaFree(y);

// Free additional vector for addition
#if VECTORADD
    cudaFree(z);
#endif

    return EXIT_SUCCESS;
}
```