```
//============================================================
// STUDENT NAME: Adrian Chan Ee Ray
// MATRIC NO.   : A0122061B
// NUS EMAIL    : Acer@u.nus.edu
// COMMENTS TO GRADER:
// <comments to grader, if any>
//
// ============================================================
//
// FILE: convolution.cu


// Include files from C standard library.
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

// Includes CUDA.
#include <cuda_runtime.h>

// Includes helper functions from CUDA Samples SDK.
#include <helper_cuda.h>
#include <helper_functions.h> // helper functions for SDK examples



/////////////////////////////////////////////////////////////////////////////
// CONSTANTS & GLOBAL VARIABLES
/////////////////////////////////////////////////////////////////////////////

// FILTER_WIDTH must be odd, and BLOCK_SIZE >= FILTER_WIDTH.
#define FILTER_WIDTH    249

// Number of CUDA threads per thread block. BLOCK_SIZE >= FILTER_WIDTH.
#define BLOCK_SIZE      256

// Number of elements in the data.
// Note that DATA_SIZE is always a multiple of BLOCK_SIZE.
#define DATA_SIZE       (2048 * BLOCK_SIZE)

// Number of CUDA thread blocks.
#define NUM_BLOCKS      ( ( (DATA_SIZE) + (BLOCK_SIZE) - 1 ) / (BLOCK_SIZE) )




//===========================================================================
// CUDA Kernel 1.
// Does not use shared memory.
// Does not care about memory coalesces.
//
// Compute the convolution of the data and the filter.
// Filter width (filterWidth) must be odd, and the filter's
// origin is the center element, that is the element
// filter[ filterWidth/2 ].
//
// For each output element output[i] that does not have enough input
// data elements in its neighborhood (that is when i < (filterWidth/2) or
// when i >= (dataSize - filterWidth/2)), the output element output[i]
// will have value 0.0.
//
//===========================================================================
__global__ void GPU_Convolve1( const float *data, int dataSize,
                               const float *filter, int filterWidth,
```

```
                                float *output )
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    int filterRadius = filterWidth / 2;

    //***********************************************
    //*********** WRITE YOUR CODE HERE **************
    //***********************************************
    // set the defualt output to be zero
    output[tid] = 0.0;
    float result = 0.0;
    // if there is enough input in the neighborhood
    if (tid < dataSize - filterRadius && tid >= filterRadius)
    {
        int beginning = tid - filterRadius; // the start of input
        for (int j = 0; j < filterWidth; j++) {
            result += filter[j] * data[beginning + j];
            }
        output[tid] = result;
        }
    }

}



//=============================================================================
// CUDA Kernel 2.
// Use shared memory.
// Care about memory coalesces.
// Care about shared memory conflicts.
//
// Compute the convolution of the data and the filter.
// Filter width (filterWidth) must be odd, and the filter's
// origin is the center element, that is the element
// filter[ filterWidth/2 ].
//
// For each output element output[i] that does not have enough input
// data elements in its neighborhood (that is when i < (filterWidth/2) or
// when i >= (dataSize - filterWidth/2)), the output element output[i]
// will have value 0.0.
//
// Assume that filterWidth <= BLOCK_SIZE.
// Assume that dataSize is a multiple of BLOCK_SIZE.
//
//=============================================================================
__global__ void GPU_Convolve2( const float *data, int dataSize,
                                const float *filter, int filterWidth,
                                float *output )
{
    __shared__ float filterS[ BLOCK_SIZE ];
    __shared__ float dataS[ 3 * BLOCK_SIZE ];

    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int tx = threadIdx.x;

    int filterRadius = filterWidth / 2;

    //***********************************************
    //*********** WRITE YOUR CODE HERE **************
    //***********************************************

    // write the filter array into shared memory
    filterS[tx] = 0.0;
```

```
    if (tx< filterWidth) { //Read in more than the number of filter elements to shared memory
        filterS[tx] = filter[tx];
    }

    // write the input data array into shared memory

    if (blockIdx.x > 0) // if this is not the first block
    {   // read the input from the previous block, and write to shared memory
        dataS[tx] = data[tid - BLOCK_SIZE];
    }
    if (blockIdx.x < NUM_BLOCKS - 1) // if this is not the last block
    {   // read the input from the next block, and write to shared memory
        dataS[tx + 2 * BLOCK_SIZE] = data[tid + BLOCK_SIZE];
    }
    // read the input from the current block, and write to shared memory
    dataS[tx + BLOCK_SIZE] = data[tid];

    // sync to make sure all input data are loaded before computation
    __syncthreads();

    // set the defualt output to be zero
    output[tid] = 0.0;
    float result = 0.0;
    // if there is enough input in the neighborhood
    if (tid >= filterRadius && tid < dataSize - filterRadius)
    {
        int beginning = tx + BLOCK_SIZE - filterRadius; // the start of input
        for (int j = 0; j < filterWidth; j++){
        //For best efficiency, you should avoid writing to output[tid] in the loop. Store the su
            result += filterS[j] * dataS[beginning + j];
        }
        output[tid] = result;
    }
}




//=============================================================================
// CPU version.
//
// Compute the convolution of the data and the filter.
// Filter width (filterWidth) must be odd, and the filter's
// origin is the center element, that is the element
// filter[ filterWidth/2 ].
//
// For each output element output[i] that does not have enough input
// data elements in its neighborhood (that is when i < (filterWidth/2) or
// when i >= (dataSize - filterWidth/2)), the output element output[i]
// will have value 0.0.
//
//=============================================================================
static void CPU_Convolve( const float *data, int dataSize,
                          const float *filter, int filterWidth,
                          float *output )
{
    int filterRadius = filterWidth / 2;

    for ( int i = 0; i < dataSize; i++ ) output[i] = 0.0;

    for ( int i = filterRadius; i < (dataSize - filterRadius); i++ )
        for ( int k = 0; k < filterWidth; k++ )
            output[i] += filter[k] * data[ i - filterRadius + k ];
}
```

```
//============================================================================
// Returns a random value in the range [min, max] from a uniform distribution.
//============================================================================
inline static double UniformRandom( double min, double max )
{
    return ( ((double)rand()) / RAND_MAX ) * (max - min) + min;
}



//============================================================================
// Generates a set of random floating-point numbers in the range [min,max]
// and put them in the the array A.
//============================================================================
static void GenerateRandomArray( float *A, int numElems, float min, float max )
{
    for ( int i = 0; i < numElems; i++ )
        A[i] = (float) UniformRandom( min, max );
}



//============================================================================
// Return true iff all corresponding elements in the float arrays A and B
// are approximately equal (i.e. the absolute difference is within the
// given epsilon).
//============================================================================
static bool FloatArrayEqual( const float *A, const float *B, int numElems, float epsilon )
{
    for ( int i = 0; i < numElems; i++ )
        if ( fabs( A[i] - B[i] ) > epsilon ) return false;

    return true;
}



void WaitForEnterKeyBeforeExit( void )
{
    fflush( stdin );
    getchar();
}



//============================================================================
// The main function
//============================================================================
int main( int argc, char** argv )
{
    atexit( WaitForEnterKeyBeforeExit );

    // Set seed for rand().
    srand( 123 );

    // Use command-line specified CUDA device, otherwise use device with highest Gflops/s.
    int devID = findCudaDevice( argc, (const char **)argv );

    // Create a timer.
    StopWatchInterface *timer = 0;
    sdkCreateTimer( &timer );
```

```
//-----------------------------------------------------------------------------
// Allocate memory and generate test data.
//-----------------------------------------------------------------------------

    // Allocate host memory for filter, input data and result arrays.
    float *h_filter = (float *) malloc( FILTER_WIDTH * sizeof(float) );
    float *h_data = (float *) malloc( DATA_SIZE * sizeof(float) );
    float *h_output = (float *) malloc( DATA_SIZE * sizeof(float) );

    // Allocate host memory for receiving results from the GPU.
    float *d2h_output1 = (float *) malloc( DATA_SIZE * sizeof(float) );
    float *d2h_output2 = (float *) malloc( DATA_SIZE * sizeof(float) );

    // Allocate device memory.
    float *d_filter, *d_data, *d_output;
    checkCudaErrors( cudaMalloc( (void**) &d_filter, FILTER_WIDTH * sizeof(float) ) );
    checkCudaErrors( cudaMalloc( (void**) &d_data, DATA_SIZE * sizeof(float) ) );
    checkCudaErrors( cudaMalloc( (void**) &d_output, DATA_SIZE * sizeof(float) ) );

    // Fill the host filter and data arrays with random floating-point numbers.
    GenerateRandomArray( h_filter, FILTER_WIDTH, 0.0, 1.0 );
    GenerateRandomArray( h_data, DATA_SIZE, 1.0, 5.0 );


//-----------------------------------------------------------------------------
// Print some program parameter values.
//-----------------------------------------------------------------------------
    printf( "Filter width = %d\n", FILTER_WIDTH );
    printf( "Data size = %d\n", DATA_SIZE );
    printf( "Thread block size = %d\n", BLOCK_SIZE );
    printf( "Number of thread blocks = %d\n", NUM_BLOCKS );
    printf( "\n\n" );


//-----------------------------------------------------------------------------
// Perform computation on CPU.
//-----------------------------------------------------------------------------
    printf( "CPU COMPUTATION:\n" );

    // Reset and start timer.
    sdkResetTimer( &timer );
    sdkStartTimer( &timer );

    // Compute on CPU.
    CPU_Convolve( h_data, DATA_SIZE, h_filter, FILTER_WIDTH, h_output );

    // Stop timer.
    sdkStopTimer( &timer );
    printf( "Processing time = %.3f ms\n", sdkGetTimerValue( &timer ) );

    // Print some results.
    printf( "First element  = %.8f\n", h_output[0] );
    printf( "Middle element = %.8f\n", h_output[ DATA_SIZE / 2 ] );
    printf( "Last element   = %.8f\n", h_output[ DATA_SIZE - 1 ] );
    printf( "\n\n" );


//-----------------------------------------------------------------------------
// Perform computation on GPU using Kernel 1 (not using shared memory).
//-----------------------------------------------------------------------------
    printf( "GPU COMPUTATION 1 (not using Shared Memory):\n" );

    // Reset and start timer.
    sdkResetTimer( &timer );
    sdkStartTimer( &timer );
```

```
    // Copy host memory to device.
    checkCudaErrors( cudaMemcpy( d_filter, h_filter, FILTER_WIDTH * sizeof(float), cudaMemcpyHos
    checkCudaErrors( cudaMemcpy( d_data, h_data, DATA_SIZE * sizeof(float), cudaMemcpyHostToDevi

    // Clear the output array in device memory.
    checkCudaErrors( cudaMemset( d_output, 0, DATA_SIZE * sizeof(float) ) );

    // Execute the kernel.
    GPU_Convolve1 <<<NUM_BLOCKS, BLOCK_SIZE>>> ( d_data, DATA_SIZE, d_filter, FILTER_WIDTH, d_ou

    // Check if kernel execution generated any error.
    getLastCudaError( "Kernel execution failed" );

    // Copy result from device memory to host.
    checkCudaErrors( cudaMemcpy( d2h_output1, d_output, DATA_SIZE * sizeof(float), cudaMemcpyDev

    // Stop timer.
    sdkStopTimer( &timer );
    printf( "Processing time = %.3f ms\n", sdkGetTimerValue( &timer ) );

    // Print some results.
    printf( "First element  = %.8f\n", d2h_output1[0] );
    printf( "Middle element = %.8f\n", d2h_output1[ DATA_SIZE / 2 ] );
    printf( "Last element   = %.8f\n", d2h_output1[ DATA_SIZE - 1 ] );

    // Check result with reference result computed by CPU.
    bool equal1 = FloatArrayEqual( h_output, d2h_output1, DATA_SIZE, 0.001f );
    printf( "Verify GPU result... %s\n", (equal1)? "PASS" : "FAIL" );
    printf( "\n\n" );


//------------------------------------------------------------------------------
// Perform computation on GPU using Kernel 2 (using shared memory).
//------------------------------------------------------------------------------
    printf( "GPU COMPUTATION 2 (using Shared Memory):\n" );

    // Reset and start timer.
    sdkResetTimer( &timer );
    sdkStartTimer( &timer );

    // Copy host memory to device.
    checkCudaErrors( cudaMemcpy( d_filter, h_filter, FILTER_WIDTH * sizeof(float), cudaMemcpyHos
    checkCudaErrors( cudaMemcpy( d_data, h_data, DATA_SIZE * sizeof(float), cudaMemcpyHostToDevi

    // Clear the output array in device memory.
    checkCudaErrors( cudaMemset( d_output, 0, DATA_SIZE * sizeof(float) ) );

    // Execute the kernel.
    GPU_Convolve2 <<<NUM_BLOCKS, BLOCK_SIZE>>> ( d_data, DATA_SIZE, d_filter, FILTER_WIDTH, d_ou

    // Check if kernel execution generated any error.
    getLastCudaError( "Kernel execution failed" );

    // Copy result from device memory to host.
    checkCudaErrors( cudaMemcpy( d2h_output2, d_output, DATA_SIZE * sizeof(float), cudaMemcpyDev

    // Stop timer.
    sdkStopTimer( &timer );
    printf( "Processing time = %.3f ms\n", sdkGetTimerValue( &timer ) );

    // Print some results.
    printf( "First element  = %.8f\n", d2h_output2[0] );
    printf( "Middle element = %.8f\n", d2h_output2[ DATA_SIZE / 2 ] );
    printf( "Last element   = %.8f\n", d2h_output2[ DATA_SIZE - 1 ] );
```

```
    // Check result with reference result computed by CPU.
    bool equal2 = FloatArrayEqual( h_output, d2h_output2, DATA_SIZE, 0.001f );
    printf( "Verify GPU result... %s\n", (equal2)? "PASS" : "FAIL" );
    printf( "\n\n" );


//-----------------------------------------------------------------------------
// Clean up.
//-----------------------------------------------------------------------------
    // Destroy the timer.
    sdkDeleteTimer( &timer );

    // Free up memory.
    free( h_filter );
    free( h_data );
    free( h_output );
    free( d2h_output1 );
    free( d2h_output2 );
    checkCudaErrors( cudaFree( d_filter ) );
    checkCudaErrors( cudaFree( d_data ) );
    checkCudaErrors( cudaFree( d_output ) );

    cudaDeviceReset();
}
```