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


// Include files from C standard library.
#include <stdlib.h>
#include <stdio.h>
#include <string.h>  // For memcpy().
#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.


// Include files to use Thrust (a C++ template library for CUDA).
// Thrust v1.7.0 is automatically installed with CUDA Toolkit 6.5.
// Read more about Thrust at the GitHub Thrust project page
// (http://thrust.github.com/).

#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <thrust/scan.h>



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

#define NUM_ELEMS       (5*1FFFFFF)  // Number of elements in input array.

#define BLOCK_SIZE      256
#define NUM_BLOCKS      ( ( (NUM_ELEMS) + (BLOCK_SIZE) - 1 ) / (BLOCK_SIZE) )

#define ELEM_MIN        1       // Minimum value in input array (must not be negative).
#define ELEM_MAX        100000  // Maximum value in input array (must not be negative).



//============================================================================
// CUDA kernel used by GPU_Unique().
//
// Given an input sorted integer array, the kernel marks in the output array
// which elements of the input array should be kept/removed, so that if these
// elements were to be kept/removed, there would be no duplicate elements in
// the sorted array. We want to remove as few elements as possible from the
// input array.
//
// The output of the kernel is an array of 1's and 0's to indicate whether
// the corresponding elements in the input array should be kept or removed --
// a 1 means keep, and 0 means remove. The output array has the same number
// of elements as the input array.
```

```
//
// For example, given the following input array
//
//        inSortedArray[] = [ 1 1 3 3 3 5 5 7 8 8 ]
//
// the output would be
//
//    outSelectionArray[] = [ 1 0 1 0 0 1 0 1 1 0 ]
//
// so that if we keep only those elements in the input array that have a 1
// in the corresponding location in the output array, we will have the
// result [ 1 3 5 7 8 ].
//
// NOTE: You should use shared memory to minimize the number of uncoalesced
// global memory accesses. Shared memory conflicts must be minimized too.
//============================================================================
__global__ void Kernel_MarkUnique(int *inSortedArray, int *outSelectionArray,
    int numElems)
{
    //*********************************************
    //*********** WRITE YOUR CODE HERE **************
    //*********************************************
    __shared__ float sharedBlk[BLOCK_SIZE];
    int tid = blockIdx.x * blockDim.x + threadIdx.x; //index of element in array
    int tx = threadIdx.x;
    if (tid < numElems) {
        sharedBlk[tx] = inSortedArray[tid]; //load element into shared block for processing
        __syncthreads();
        if (tid == 0) {// if first index of entire input array, always return 1 because it is al
            outSelectionArray[tid] = 1;
        }
        else { //checking is needed
            if (tx == 0) {// if its the first element of the block, access to global input varia
                if (sharedBlk[tx] > inSortedArray[tid - 1]) { //If there is an increment in valu
                    outSelectionArray[tid] = 1;
                }
                else { //It is a duplicate
                    outSelectionArray[tid] = 0;
                }
            }
            else { // it is not in the first element, checking can be done with sharedBlk
                if (sharedBlk[tx] > sharedBlk[tx - 1]) { //If there is an increment in values i.
                    outSelectionArray[tid] = 1;
                }
                else { //It is a duplicate
                    outSelectionArray[tid] = 0;
                }
            }
        }

    }

}



//============================================================================
// CUDA kernel used by GPU_Unique().
//
// The kernel copies a selected set of elements from the input array to
// specified locations in the output array.
//
// For an input element inArray[i], if selectionArray[i] is 1, then
// the input element is copied to the output array outArray[].
// The location in the output array it is copied to is
```

```
// scatterAddressArray[i] + addressOffset.
//
// You can assume that no two elements in the input array inArray[]
// will be selected and copied to the same location in the output
// array outArray[].
//
// NOTE: You do not need to use shared memory, but try to keep the
// number of uncoalesced global memory accesses to the minimal.
//============================================================================
__global__ void Kernel_Scatter(int *inArray, int *selectionArray,
    int *scatterAddressArray, int addressOffset,
    int *outArray, int numElems)
{
    //************************************************
    //*********** WRITE YOUR CODE HERE ***************
    //************************************************
    int tid = blockIdx.x * blockDim.x + threadIdx.x; //index of element in array
    if (tid < numElems) {
        if (selectionArray[tid] == 1) {
            outArray[scatterAddressArray[tid] + addressOffset] = inArray[tid];
        }
    }
}



//============================================================================
// Used by GPU_Unique().
//
// Use Thrust's sort algorithm to sort the input integer array on the GPU,
// in non-decreasing order. The sort is performed in-place,
//
// NOTE:
// * The input/output array is already allocated in the device memory.
//============================================================================
static void GPU_SortIntegerArray(int *d_inoutArray, int numElems)
{
    thrust::device_ptr<int> dev_ptr(d_inoutArray);
    thrust::sort(dev_ptr, dev_ptr + numElems);
}



//============================================================================
// Used by GPU_Unique().
//
// Use Thrust's scan algorithm to compute the "inclusive" all-prefix sums on the GPU.
// Also produces the sum of all elements in the input array in the output
// parameter *h_outInArraySum.
//
// NOTE: The input and output arrays are already allocated in the device memory.
//============================================================================
static void GPU_AllPrefixSums(int *d_inArray, int *d_outArray, int numElems,
    int *h_outInArraySum)
{
    thrust::device_ptr<int> in_dev_ptr(d_inArray);
    thrust::device_ptr<int> out_dev_ptr(d_outArray);
    thrust::inclusive_scan(in_dev_ptr, in_dev_ptr + numElems, out_dev_ptr);

    // Get the sum of all the elements in the input array. This can be obtained
    // from the last element in the all-prefix-sums array.
    checkCudaErrors(cudaMemcpy(h_outInArraySum, d_outArray + numElems - 1,
        sizeof(int), cudaMemcpyDeviceToHost));

    // Using Thrust, the above memory copy can be written as:
```

```
    // *h_outInArraySum = out_dev_ptr[ numElems - 1 ];
}



//============================================================================
// GPU version.
//
// Given an input integer array, the function produces an output array
// which is a sorted version of the input array, but with duplicate
// elements removed. The output array is sorted in non-decreasing order.
// The function also produces the number of unique elements in the
// output array in the parameter (*numUniqueElems).
//
// For example, if the input array is [ 5 3 7 5 8 3 1 3 1 8 ], the
// output array would be [ 1 3 5 7 8 ].
//
// When this function is called, sufficient memory storage must have
// already been allocated for the output array. The safest is to allocate
// as much memory as for the input array.
//
// Here, a scan-and-scatter approach is used to do the stream compaction
// on the GPU. The following example demonstrates the steps.
//
// (0) Input array:
//              inputArray[] = [ 5 3 7 5 8 3 1 3 1 8 ]
//
// (1) Sort inputArray[]:
//             sortedArray[] = [ 1 1 3 3 3 5 5 7 8 8 ]
//
// (2) Mark the unique elements in sortedArray[]:
//          selectionArray[] = [ 1 0 1 0 0 1 0 1 1 0 ]
//
// (3) Scan selectionArray[] ("inclusive" all-prefix sums):
//     scatterAddressArray[] = [ 1 1 2 2 2 3 3 4 5 5 ]
//
// (4) Scatter sortedArray[] into outputArray[] using scatterAddressArray[] - 1:
//             outputArray[] = [ 1 3 5 7 8 ]
//
// Note that the number of unique elements in the output array is the
// value of the last element in scatterAddressArray[].
//
// IMPORTANT: Step (1) to (4) must be computed on the GPU.
//
//============================================================================
static void GPU_Unique(const int inputArray[], int numInputElems,
    int outputArray[], int *numUniqueElems)
{
    if (numInputElems < 1)
    {
        (*numUniqueElems) = 0;
        return;
    }

    //------------------------------------------------------------------------
    // Allocate device memory and copy input array from host memory to
    // device memory.
    //------------------------------------------------------------------------

    // Allocate device memory.
    int *d_sortedArray, *d_selectionArray, *d_scatterAddressArray, *d_outputArray;

    checkCudaErrors(cudaMalloc((void**)&d_sortedArray, numInputElems * sizeof(int)));
    checkCudaErrors(cudaMalloc((void**)&d_selectionArray, numInputElems * sizeof(int)));
    checkCudaErrors(cudaMalloc((void**)&d_scatterAddressArray, numInputElems * sizeof(int)));
```

```
    checkCudaErrors(cudaMalloc((void**)&d_outputArray, numInputElems * sizeof(int)));

    // Will contain the number of unique elements in the output array.
    int numSelectedElems = 0;

    // Copy host input array to device memory.
    checkCudaErrors(cudaMemcpy(d_sortedArray, inputArray, numInputElems * sizeof(int),
        cudaMemcpyHostToDevice));


    //---------------------------------------------------------------------------
    // Do Step (1) to (4).
    //---------------------------------------------------------------------------

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



    // (1) Sort inputArray[]:
    //             sortedArray[] = [ 1 1 3 3 3 5 5 7 8 8 ]
    GPU_SortIntegerArray(d_sortedArray, numInputElems);

    // (2) Mark the unique elements in sortedArray[]:
    //          selectionArray[] = [ 1 0 1 0 0 1 0 1 1 0 ]
    Kernel_MarkUnique << <NUM_BLOCKS, BLOCK_SIZE >> >(d_sortedArray, d_selectionArray, numInputE

    // (3) Scan selectionArray[] ("inclusive" all-prefix sums):
    //     scatterAddressArray[] = [ 1 1 2 2 2 3 3 4 5 5 ]
    GPU_AllPrefixSums(d_selectionArray, d_scatterAddressArray, numInputElems, &numSelectedElems)

    // (4) Scatter sortedArray[] into outputArray[] using scatterAddressArray[] - 1:
    //             outputArray[] = [ 1 3 5 7 8 ]
    Kernel_Scatter << <NUM_BLOCKS, BLOCK_SIZE >> >(d_sortedArray, d_selectionArray,
        d_scatterAddressArray, -1,
        d_outputArray, numInputElems);

        //---------------------------------------------------------------------------
        // Copy the final result from the device memory to the host memory.
        //---------------------------------------------------------------------------

        checkCudaErrors(cudaMemcpy(outputArray, d_outputArray, numSelectedElems * sizeof(int),
        cudaMemcpyDeviceToHost));

    (*numUniqueElems) = numSelectedElems;


    //---------------------------------------------------------------------------
    // Clean up.
    //---------------------------------------------------------------------------

    // Free device memory.
    checkCudaErrors(cudaFree(d_sortedArray));
    checkCudaErrors(cudaFree(d_selectionArray));
    checkCudaErrors(cudaFree(d_scatterAddressArray));
    checkCudaErrors(cudaFree(d_outputArray));
}




//============================================================================
// Quicksort to sort the input integer array in-place in ascending order.
// To sort the entire input array, call Quicksort(array, 0, numElems-1).
```

```
//============================================================================
#define SWAP(x, y, t)  ((t)=(x),(x)=(y),(y)=(t))

static void Quicksort(int a[], int first, int last)
{
    int tmp; // Temporary variable for SWAP.

    if (first < last)
    {
        int pivot = a[first];
        int i = first - 1;
        int j = last + 1;

        while (true)
        {
            do { j--; } while (a[j] > pivot);
            do { i++; } while (a[i] < pivot);

            if (i < j)
                SWAP(a[i], a[j], tmp);
            else
                break;
        }

        Quicksort(a, first, j);
        Quicksort(a, j + 1, last);
    }
}

#undef SWAP



//============================================================================
// CPU version.
//
// Given an input integer array, the function produces an output array
// which is a sorted version of the input array, but with duplicate
// elements removed. The output array is sorted in non-decreasing order.
// The function also produces the number of unique elements in the
// output array in the parameter (*numUniqueElems).
//
// When this function is called, sufficient memory storage must have
// already been allocated for the output array. The safest is to allocate
// as much memory as for the input array.
//============================================================================
static void CPU_Unique(const int inputArray[], int numInputElems,
    int outputArray[], int *numUniqueElems)
{
    if (numInputElems < 1)
    {
        (*numUniqueElems) = 0;
        return;
    }

    int *sortedArray = (int *)malloc(numInputElems * sizeof(int));
    memcpy(sortedArray, inputArray, numInputElems * sizeof(int));

    Quicksort(sortedArray, 0, numInputElems - 1);

    outputArray[0] = sortedArray[0];
    int uniqueCount = 1;

    for (int i = 1; i < numInputElems; i++)
        if (sortedArray[i] != sortedArray[i - 1])
```

```
            outputArray[uniqueCount++] = sortedArray[i];

    (*numUniqueElems) = uniqueCount;
}



//============================================================================
// Generates a set of random integers, each has value from elemMin to
// elemMax, and put them in the array intArray[].
//============================================================================
static void GenerateRandomIntegers(int intArray[], int numElems, int elemMin, int elemMax)
{
    for (int i = 0; i < numElems; i++)
    {
        int rand32 = rand() * (RAND_MAX + 1) + rand();
        intArray[i] = rand32 % (elemMax - elemMin + 1) + elemMin;
    }
}



//============================================================================
// Return true iff all corresponding elements in the int
// arrays A and B are equal.
//============================================================================
static bool IntArrayEqual(const int A[], const int B[], int numElems)
{
    for (int i = 0; i < numElems; i++)
        if (A[i] != B[i]) 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(927);

    // 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 host memory and generate test data.
```

```
    //---------------------------------------------------------------------

    // Allocate host memory for input integer array.
    int *inputArray = (int *)malloc(NUM_ELEMS * sizeof(int));

    // Allocate host memory for result arrays.
    int *cpu_uniqueArray = (int *)malloc(NUM_ELEMS * sizeof(int));
    int *gpu_uniqueArray = (int *)malloc(NUM_ELEMS * sizeof(int));

    // Number of unique elements in input array computed by different methods.
    int cpu_numUniqueElems = 0;
    int gpu_numUniqueElems = 0;

    // Fill the input array with random integers.
    GenerateRandomIntegers(inputArray, NUM_ELEMS, ELEM_MIN, ELEM_MAX);


    //---------------------------------------------------------------------
    // Print some program parameter values.
    //---------------------------------------------------------------------
    printf("NUM_ELEMS  = %d\n", NUM_ELEMS);
    printf("BLOCK_SIZE = %d\n", BLOCK_SIZE);
    printf("ELEM_MIN   = %d\n", ELEM_MIN);
    printf("ELEM_MAX   = %d\n", ELEM_MAX);
    printf("\n\n");


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

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

    // Compute on CPU.
    CPU_Unique(inputArray, NUM_ELEMS, cpu_uniqueArray, &cpu_numUniqueElems);

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

    // Print some results.
    printf("Number of unique elements = %d\n", cpu_numUniqueElems);
    printf("\n\n");


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

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

    // Compute on GPU.
    GPU_Unique(inputArray, NUM_ELEMS, gpu_uniqueArray, &gpu_numUniqueElems);

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

    // Print some results.
```

```
    printf("Number of unique elements = %d\n", gpu_numUniqueElems);
    printf("\n");

    // Check result with reference result computed by CPU.
    bool equal = (gpu_numUniqueElems == cpu_numUniqueElems) &&
        IntArrayEqual(cpu_uniqueArray, gpu_uniqueArray, cpu_numUniqueElems);
    printf("Verify GPU result... %s\n", (equal) ? "PASS" : "FAIL");
    printf("\n\n");


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

    // Free up memory.
    free(inputArray);
    free(cpu_uniqueArray);
    free(gpu_uniqueArray);

    cudaDeviceReset();
}
```