// This program implements a vector addition using OpenCL

#include <stdio.h>

#include <stdlib.h>

#include <CL/cl.h>

// OpenCL kernel to perform an element-wise add of two arrays

const char\* programSource =

“\_\_kernel \n”

“void vecadd(\_\_global int \*A, \n”

“ \_\_global int \*B, \n”

“ \_\_global int \*C) \n”

“{ \n”

“ \n”

“ // Get the work-item’s unique ID \n”

“ int idx = get\_global\_id(0); \n”

“ \n”

“ // Add the corresponding locations of \n”

“ // ’A’ and ’B’, and store the result in ’C’. \n”

“ C[idx] = A[idx] + B[idx]; \n”

“} \n”

;

int main() {

// This code executes on the OpenCL host

// Host data

int \*A = NULL; // Input array

int \*B = NULL; // Input array

int \*C = NULL; // Output array

// Elements in each array

const int elements = 2048;

// Compute the size of the data

size\_t datasize = sizeof(int)\*elements;

// Allocate space for input/output data

A = (int\*)malloc(datasize);

B = (int\*)malloc(datasize);

C = (int\*)malloc(datasize);

// Initialize the input data

for(int i = 0; i < elements; i++) {

A[i] = i;

B[i] = i;

}

// Use this to check the output of each API call

cl\_int status;

//———————————————————————————————————————————————————

// STEP 1: Discover and initialize the platforms

//———————————————————————————————————————————————————

cl\_uint numPlatforms = 0;

cl\_platform\_id \*platforms = NULL;

// Use clGetPlatformIDs() to retrieve the number of

// platforms

status = clGetPlatformIDs(0, NULL, &numPlatforms);

// Allocate enough space for each platform

platforms =

(cl\_platform\_id\*)malloc(

numPlatforms\*sizeof(cl\_platform\_id));

// Fill in platforms with clGetPlatformIDs()

status = clGetPlatformIDs(numPlatforms, platforms,

NULL);

//———————————————————————————————————————————————————

// STEP 2: Discover and initialize the devices

//———————————————————————————————————————————————————

cl\_uint numDevices = 0;

cl\_device\_id \*devices = NULL;

// Use clGetDeviceIDs() to retrieve the number of

// devices present

status = clGetDeviceIDs(

platforms[0],

CL\_DEVICE\_TYPE\_ALL,

0,

NULL,

&numDevices);

// Allocate enough space for each device

devices =

(cl\_device\_id\*)malloc(

numDevices\*sizeof(cl\_device\_id));

// Fill in devices with clGetDeviceIDs()

status = clGetDeviceIDs(

platforms[0],

CL\_DEVICE\_TYPE\_ALL,

numDevices,

devices,

NULL);

//———————————————————————————————————————————————————

// STEP 3: Create a context

//———————————————————————————————————————————————————

cl\_context context = NULL;

// Create a context using clCreateContext() and

// associate it with the devices

context = clCreateContext(

NULL,

numDevices,

devices,

NULL,

NULL,

&status);

//———————————————————————————————————————————————————

// STEP 4: Create a command queue

//———————————————————————————————————————————————————

cl\_command\_queue cmdQueue;

// Create a command queue using clCreateCommandQueue(),

// and associate it with the device you want to execute

// on

cmdQueue = clCreateCommandQueue(

context,

devices[0],

0,

&status);

//———————————————————————————————————————————————————

// STEP 5: Create device buffers

//———————————————————————————————————————————————————

cl\_mem bufferA; // Input array on the device

cl\_mem bufferB; // Input array on the device

cl\_mem bufferC; // Output array on the device

// Use clCreateBuffer() to create a buffer object (d\_A)

// that will contain the data from the host array A

bufferA = clCreateBuffer(

context,

CL\_MEM\_READ\_ONLY,

datasize,

NULL,

&status);

// Use clCreateBuffer() to create a buffer object (d\_B)

// that will contain the data from the host array B

bufferB = clCreateBuffer(

context,

CL\_MEM\_READ\_ONLY,

datasize,

NULL,

&status);

// Use clCreateBuffer() to create a buffer object (d\_C)

// with enough space to hold the output data

bufferC = clCreateBuffer(

context,

CL\_MEM\_WRITE\_ONLY,

datasize,

NULL,

&status);

//———————————————————————————————————————————————————

// STEP 6: Write host data to device buffers

//———————————————————————————————————————————————————

// Use clEnqueueWriteBuffer() to write input array A to

// the device buffer bufferA

status = clEnqueueWriteBuffer(

cmdQueue,

bufferA,

CL\_FALSE,

0,

datasize,

A,

0,

NULL,

NULL);

// Use clEnqueueWriteBuffer() to write input array B to

// the device buffer bufferB

status = clEnqueueWriteBuffer(

cmdQueue,

bufferB,

CL\_FALSE,

0,

datasize,

B,

0,

NULL,

NULL);

//———————————————————————————————————————————————————

// STEP 7: Create and compile the program

//———————————————————————————————————————————————————

// Create a program using clCreateProgramWithSource()

cl\_program program = clCreateProgramWithSource(

context,

1,

(const char\*\*)&programSource,

NULL,

&status);

// Build (compile) the program for the devices with

// clBuildProgram()

status = clBuildProgram(

program,

numDevices,

devices,

NULL,

NULL,

NULL);

//———————————————————————————————————————————————————

// STEP 8: Create the kernel

//———————————————————————————————————————————————————

cl\_kernel kernel = NULL;

// Use clCreateKernel() to create a kernel from the

// vector addition function (named "vecadd")

kernel = clCreateKernel(program, "vecadd", &status);

//———————————————————————————————————————————————————

// STEP 9: Set the kernel arguments

//———————————————————————————————————————————————————

// Associate the input and output buffers with the

// kernel

// using clSetKernelArg()

status = clSetKernelArg(

kernel,

0,

sizeof(cl\_mem),

&bufferA);

status j= clSetKernelArg(

kernel,

1,

sizeof(cl\_mem),

&bufferB);

status j= clSetKernelArg(

kernel,

2,

sizeof(cl\_mem),

&bufferC);

//———————————————————————————————————————————————————

// STEP 10: Configure the work-item structure

//———————————————————————————————————————————————————

// Define an index space (global work size) of work

// items for

// execution. A workgroup size (local work size) is not

// required,

// but can be used.

size\_t globalWorkSize[1];

// There are ’elements’ work-items

globalWorkSize[0] = elements;

//———————————————————————————————————————————————————

// STEP 11: Enqueue the kernel for execution

//———————————————————————————————————————————————————

// Execute the kernel by using

// clEnqueueNDRangeKernel().

// ’globalWorkSize’ is the 1D dimension of the

// work-items

status = clEnqueueNDRangeKernel(

cmdQueue,

kernel,

1,

NULL,

globalWorkSize,

NULL,

0,

NULL,

NULL);

//———————————————————————————————————————————————————

// STEP 12: Read the output buffer back to the host

//———————————————————————————————————————————————————

// Use clEnqueueReadBuffer() to read the OpenCL output

// buffer (bufferC)

// to the host output array (C)

clEnqueueReadBuffer(

cmdQueue,

bufferC,

CL\_TRUE,

0,

datasize,

C,

0,

NULL,

NULL);

// Verify the output

bool result = true;

for(int i = 0; i < elements; i++) {

if(C[i] != i+i) {

result = false;

break;

}

}

if(result) {

printf("Output is correct\n");

} else {

printf("Output is incorrect\n");

}

//———————————————————————————————————————————————————

// STEP 13: Release OpenCL resources

//———————————————————————————————————————————————————

// Free OpenCL resources

clReleaseKernel(kernel);

clReleaseProgram(program);

clReleaseCommandQueue(cmdQueue);

clReleaseMemObject(bufferA);

clReleaseMemObject(bufferB);

clReleaseMemObject(bufferC);

clReleaseContext(context);

// Free host resources

free(A);

free(B);

free(C);

free(platforms);

free(devices);

}