```
// 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 void vecadd(__global int *A, __global int *B, __global
int *C)

{
   // Get the work-item's unique ID
   int idx = get_global_id(0);

   // Add the corresponding locations of
   // 'A' and 'B', and store the result in 'C'.
   C[idx] = A[idx] + B[idx];
}
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
```

```
  // 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
  // ——————————————————————————————————————
  // Use clEnqueueWriteBuffer() to write input array A to
//  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);
```