

# Module 20 Lab

## OpenCL Vector Addition

*GPU Teaching Kit – Accelerated Computing*

### OBJECTIVE

The purpose of this lab is to introduce the student to the OpenCL API by implementing vector addition. The student will implement vector addition by writing the GPU kernel code as well as the associated host code.

### PREREQUISITES

Before starting this lab, make sure that:

- You have completed all of the module video lectures.
- You have completed the CUDA Vector Addition lab.

### INSTRUCTIONS

This lab uses a separate build system. Consult the provided Makefile.

- Edit the Makefile variable LIBWB to point to the location of libwb.so. If the Modules were built in /path/to/build, that location should be /path/to/build.
- Edit the Makefile target all to look like `all: template solution` to compile the solution.

Edit the code in the code tab to perform the following:

- Set up an OpenCL context and command queue
- Invoke the OpenCL API to build the kernel
- Allocate device memory
- Copy host memory to device
- Initialize work-group and global sizes
- Enqueue the kernel
- Copy results from device to host
- Free device memory
- Write the OpenCL kernel

Instructions about where to place each part of the code is demarcated by the `//@@` comment lines.

## LOCAL SETUP INSTRUCTIONS

The most recent version of source code for this lab along with the build-scripts can be found on the [Bitbucket repository](#). A description on how to use the [CMake](#) tool in along with how to build the labs for local development found in the [README](#) document in the root of the repository.

The executable generated as a result of compiling the lab can be run using the following command:

```
./OpenCLVectorAdd_Template -e <expected.raw> -i <input1.raw>,<input2.raw> \
-o <output.raw> -t vector
```

where <expected.raw> is the expected output, <input0.raw>,<input1.raw> is the input dataset, and <output.raw> is an optional path to store the results. The datasets can be generated using the dataset generator built as part of the compilation process.

The local CMake does not build this lab. An example Makefile is

```
NVCC=nvcc
INCLUDE= -I../libwb
LIBWB= -L../build -lwb
LIBS= -lOpenCL $(LIBWB)

all: template

template:
    $(NVCC) -std=c++11 template.cpp $(INCLUDE) $(LIBS) -o OpenCLVectorAdd_Template

solution:
    $(NVCC) solution.c $(INCLUDE)

clean:
    rm -f OpenCLVectorAdd_Template
```

## QUESTIONS

- (1) If you have completed some CUDA labs, try to list which CUDA API calls correspond to which sets of OpenCL API calls.

ANSWER: **CUDA's kernel«<>» corresponds to OpenCL's clGetPlatformIDs(), clGetDeviceIDs(), clCreateContext(), clCreateCommandQueue(), clCreateProgramWithSource(), clBuildProgram(), clCreateKernel(), clSetKernelArg(), clEnqueueNDRangeKernel(). CUDA's cudaMalloc() corresponds to OpenCL's clCreateBuffer(). CUDA's cudaMemcpy() corresponds to OpenCL's clEnqueueWriteBuffer() / clEnqueueReadBuffer(). CUDA's cudaFree() corresponds to OpenCL's clReleaseMemObject()**

- (2) Why is the OpenCL kernel stored as a string in the source file instead of directly in the source file like the host code?

ANSWER: The OpenCL model compiles the device code at runtime, so it could come from anywhere. The OpenCL API directly exposes this capability. Lower-level CUDA APIs do something similar.

## CODE TEMPLATE

The following code is suggested as a starting point for students. The code handles the import and export as well as the checking of the solution. Students are expected to insert their code in the sections demarcated with `///@`. Students are expected to keep the other code unchanged. The tutorial page describes the functionality of the `wb*` methods.

```

1  #include <wb.h>
2
3
4  const char *kernelSource = "";
5
6  int main(int argc, char *argv[]) {
7      wbArg_t args;
8      int inputLength;
9      int inputLengthBytes;
10     float *hostInput1;
11     float *hostInput2;
12     float *hostOutput;
13     cl_mem deviceInput1;
14     cl_mem deviceInput2;
15     cl_mem deviceOutput;
16
17     cl_platform_id cpPlatform; // OpenCL platform
18     cl_device_id device_id;    // device ID
19     cl_context context;        // context
20     cl_command_queue queue;    // command queue
21     cl_program program;        // program
22     cl_kernel kernel;          // kernel
23
24     args = wbArg_read(argc, argv);
25
26     wbTime_start(Generic, "Importing data and creating memory on host");
27     hostInput1 =
28         (float *)wbImport(wbArg_getInputFile(args, 0), &inputLength);
29     hostInput2 =
30         (float *)wbImport(wbArg_getInputFile(args, 1), &inputLength);
31     inputLengthBytes = inputLength * sizeof(float);
32     hostOutput = (float *)malloc(inputLengthBytes);
33     wbTime_stop(Generic, "Importing data and creating memory on host");
34
35     wbLog	TRACE, "The input length is ", inputLength);
36     wbLog	TRACE, "The input size is ", inputLengthBytes, " bytes");
37
38     ///@@ Insert code here
39     ///@@ Initialize the workgroup dimensions
40

```

```

41  //@@ Bind to platform
42
43  //@@ Get ID for the device
44
45  //@@ Create a context
46
47  //@@ Create a command queue
48
49  //@@ Create the compute program from the source buffer
50
51  //@@ Build the program executable
52
53  //@@ Create the compute kernel in the program we wish to run
54
55  //@@ Create the input and output arrays in device memory for our
56  //@@ calculation
57
58  //@@ Write our data set into the input array in device memory
59
60  //@@ Set the arguments to our compute kernel
61
62  //@@ Execute the kernel over the entire range of the data set
63
64  //@@ Wait for the command queue to get serviced before reading back results
65
66  //@@ Read the results from the device
67
68  wbSolution(args, hostOutput, inputLength);
69
70  // release OpenCL resources
71  clReleaseMemObject(deviceInput1);
72  clReleaseMemObject(deviceInput2);
73  clReleaseMemObject(deviceOutput);
74  clReleaseProgram(program);
75  clReleaseKernel(kernel);
76  clReleaseCommandQueue(queue);
77  clReleaseContext(context);
78
79  // release host memory
80  free(hostInput1);
81  free(hostInput2);
82  free(hostOutput);
83
84  return 0;
85  }

```

## CODE SOLUTION

The following is a possible implementation of the lab. This solution is intended for use only by the teaching staff and should not be distributed to students.

```

1  #include <wb.h>
2
3  #include <CL/opencl.h>
4  #include <math.h>
5  #include <stdio.h>
6  #include <stdlib.h>
7
8  const char *kernelSource =
9      "\n"
10     "__kernel void vecAdd( __global float *in1,          \n"
11     "                        __global float *in2,          \n"
12     "                        __global float *out,           \n"
13     "                        const unsigned int n)           \n"
14     "{                                                     \n"
15     "    //Get our global thread ID                         \n"
16     "    int id = get_global_id(0);                          \n"
17     "                                                     \n"
18     "    //Make sure we do not go out of bounds             \n"
19     "    if (id < n)                                         \n"
20     "        c[id] = a[id] + b[id];                        \n"
21     "                                                     \n"
22     "}"
23     "\n";
24
25 int main(int argc, char *argv[]) {
26     wbArg_t args;
27     int inputLength;
28     int inputLengthBytes;
29     float *hostInput1;
30     float *hostInput2;
31     float *hostOutput;
32     cl_mem deviceInput1;
33     cl_mem deviceInput2;
34     cl_mem deviceOutput;
35
36     cl_platform_id cpPlatform; // OpenCL platform
37     cl_device_id device_id;    // device ID
38     cl_context context;        // context
39     cl_command_queue queue;    // command queue
40     cl_program program;        // program
41     cl_kernel kernel;          // kernel
42
43     args = wbArg_read(argc, argv);
44
45     wbTime_start(Generic, "Importing data and creating memory on host");
46     hostInput1 =
47         (float *)wbImport(wbArg_getInputFile(args, 0), &inputLength);
48     hostInput2 =
49         (float *)wbImport(wbArg_getInputFile(args, 1), &inputLength);
50     inputLengthBytes = inputLength * sizeof(float);
51     hostOutput = (float *)malloc(inputLengthBytes);
52     wbTime_stop(Generic, "Importing data and creating memory on host");
53
54     wbLog	TRACE, "The input length is ", inputLength);

```

```

54     wbLog(TRACE, "The input size is ", inputLengthBytes, " bytes");
55
56     // Initialize the workgroup dimensions
57     size_t globalSize, localSize;
58     cl_int err;
59     /// Insert code here
60     localSize = 64;
61     globalSize = ceil(inputLength / (float)localSize) * localSize;
62
63     // Bind to platform
64     err = clGetPlatformIDs(1, &cpPlatform, NULL);
65
66     // Get ID for the device
67     err =
68         clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
69
70     // Create a context
71     context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
72
73     // Create a command queue
74     queue = clCreateCommandQueue(context, device_id, 0, &err);
75
76     // Create the compute program from the source buffer
77     program = clCreateProgramWithSource(
78         context, 1, (const char *)&kernelSource, NULL, &err);
79
80     // Build the program executable
81     clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
82
83     // Create the compute kernel in the program we wish to run
84     kernel = clCreateKernel(program, "vecAdd", &err);
85
86     // Create the input and output arrays in device memory for our
87     // calculation
88     deviceInput1 = clCreateBuffer(context, CL_MEM_READ_ONLY,
89                                     inputLengthBytes, NULL, NULL);
90     deviceInput2 = clCreateBuffer(context, CL_MEM_READ_ONLY,
91                                     inputLengthBytes, NULL, NULL);
92     deviceOutput = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
93                                     inputLengthBytes, NULL, NULL);
94
95     // Write our data set into the input array in device memory
96     err = clEnqueueWriteBuffer(queue, deviceInput1, CL_TRUE, 0,
97                                 inputLengthBytes, hostInput1, 0, NULL, NULL);
98     err |= clEnqueueWriteBuffer(queue, deviceInput2, CL_TRUE, 0,
99                                 inputLengthBytes, hostInput2, 0, NULL, NULL);
100
101     // Set the arguments to our compute kernel
102     err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &deviceInput1);
103     err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &deviceInput2);
104     err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &deviceOutput);
105     err |= clSetKernelArg(kernel, 3, sizeof(int), &inputLength);
106

```

```

107 // Execute the kernel over the entire range of the data set
108 err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalSize,
109                               &localSize, 0, NULL, NULL);
110
111 // Wait for the command queue to get serviced before reading back results
112 clFinish(queue);
113
114 // Read the results from the device
115 clEnqueueReadBuffer(queue, deviceOutput, CL_TRUE, 0, inputLengthBytes,
116                    hostOutput, 0, NULL, NULL);
117
118 wbSolution(args, hostOutput, inputLength);
119
120 // release OpenCL resources
121 clReleaseMemObject(deviceInput1);
122 clReleaseMemObject(deviceInput2);
123 clReleaseMemObject(deviceOutput);
124 clReleaseProgram(program);
125 clReleaseKernel(kernel);
126 clReleaseCommandQueue(queue);
127 clReleaseContext(context);
128
129 // release host memory
130 free(hostInput1);
131 free(hostInput2);
132 free(hostOutput);
133
134 return 0;
135 }

```