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 <intput1.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(), clSetKernekArg(), clEnqueuNDRangeKernel(). 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 is the sections demarcated with `//@@`. Students expected the other code unchanged. The tutorial page describes the functionality of the `wb*` methods.

```
#include <wb.h>


const char *kernelSource = "";

int main(int argc, char *argv[]) {
  wbArg_t args;
  int inputLength;
  int inputLengthBytes;
  float *hostInput1;
  float *hostInput2;
  float *hostOutput;
  cl_mem deviceInput1;
  cl_mem deviceInput2;
  cl_mem deviceOutput;

  cl_platform_id cpPlatform; // OpenCL platform
  cl_device_id device_id;    // device ID
  cl_context context;        // context
  cl_command_queue queue;    // command queue
  cl_program program;        // program
  cl_kernel kernel;          // kernel

  args = wbArg_read(argc, argv);

  wbTime_start(Generic, "Importing data and creating memory on host");
  hostInput1 =
      (float *)wbImport(wbArg_getInputFile(args, 0), &inputLength);
  hostInput2 =
      (float *)wbImport(wbArg_getInputFile(args, 1), &inputLength);
  inputLengthBytes = inputLength * sizeof(float);
  hostOutput       = (float *)malloc(inputLengthBytes);
  wbTime_stop(Generic, "Importing data and creating memory on host");

  wbLog(TRACE, "The input length is ", inputLength);
  wbLog(TRACE, "The input size is ", inputLengthBytes, " bytes");

  //@@ Insert code here
  //@@ Initialize the workgroup dimensions

```

```
  //@@ Bind to platform

  //@@ Get ID for the device

  //@@ Create a context

  //@@ Create a command queue

  //@@ Create the compute program from the source buffer

  //@@ Build the program executable

  //@@ Create the compute kernel in the program we wish to run

  //@@ Create the input and output arrays in device memory for our
  //@@ calculation

  //@@ Write our data set into the input array in device memory

  //@@ Set the arguments to our compute kernel

  //@@ Execute the kernel over the entire range of the data set

  //@@ Wait for the command queue to get serviced before reading back results

  //@@ Read the results from the device

  wbSolution(args, hostOutput, inputLength);

  // release OpenCL resources
  clReleaseMemObject(deviceInput1);
  clReleaseMemObject(deviceInput2);
  clReleaseMemObject(deviceOutput);
  clReleaseProgram(program);
  clReleaseKernel(kernel);
  clReleaseCommandQueue(queue);
  clReleaseContext(context);

  // release host memory
  free(hostInput1);
  free(hostInput2);
  free(hostOutput);

  return 0;
}
```

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.

```
#include <wb.h>

#include <CL/opencl.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

const char *kernelSource =
    "\n"
    "__kernel void vecAdd(  __global float *in1,                      \n"
    "                       __global float *in2,                      \n"
    "                       __global float *out,                      \n"
    "                       const unsigned int n)                     \n"
    "{                                                                \n"
    "    //Get our global thread ID                                   \n"
    "    int id = get_global_id(0);                                   \n"
    "                                                                 \n"
    "    //Make sure we do not go out of bounds                       \n"
    "    if (id < n)                                                  \n"
    "        c[id] = a[id] + b[id];                                   \n"
    "}                                                                \n"
    "\n";

int main(int argc, char *argv[]) {
  wbArg_t args;
  int inputLength;
  int inputLengthBytes;
  float *hostInput1;
  float *hostInput2;
  float *hostOutput;
  cl_mem deviceInput1;
  cl_mem deviceInput2;
  cl_mem deviceOutput;

  cl_platform_id cpPlatform; // OpenCL platform
  cl_device_id device_id;    // device ID
  cl_context context;        // context
  cl_command_queue queue;    // command queue
  cl_program program;        // program
  cl_kernel kernel;          // kernel

  args = wbArg_read(argc, argv);

  wbTime_start(Generic, "Importing data and creating memory on host");
  hostInput1 =
      (float *)wbImport(wbArg_getInputFile(args, 0), &inputLength);
  hostInput2 =
      (float *)wbImport(wbArg_getInputFile(args, 1), &inputLength);
  inputLengthBytes = inputLength * sizeof(float);
  hostOutput       = (float *)malloc(inputLengthBytes);
  wbTime_stop(Generic, "Importing data and creating memory on host");

  wbLog(TRACE, "The input length is ", inputLength);
```

```
wbLog(TRACE, "The input size is ", inputLengthBytes, " bytes");

// Initialize the workgroup dimensions
size_t globalSize, localSize;
cl_int err;
//@@ Insert code here
localSize  = 64;
globalSize = ceil(inputLength / (float)localSize) * localSize;

// Bind to platform
err = clGetPlatformIDs(1, &cpPlatform, NULL);

// Get ID for the device
err =
    clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);

// Create a context
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);

// Create a command queue
queue = clCreateCommandQueue(context, device_id, 0, &err);

// Create the compute program from the source buffer
program = clCreateProgramWithSource(
    context, 1, (const char **)&kernelSource, NULL, &err);

// Build the program executable
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// Create the compute kernel in the program we wish to run
kernel = clCreateKernel(program, "vecAdd", &err);

// Create the input and output arrays in device memory for our
// calculation
deviceInput1 = clCreateBuffer(context, CL_MEM_READ_ONLY,
                              inputLengthBytes, NULL, NULL);
deviceInput2 = clCreateBuffer(context, CL_MEM_READ_ONLY,
                              inputLengthBytes, NULL, NULL);
deviceOutput = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                              inputLengthBytes, NULL, NULL);

// Write our data set into the input array in device memory
err = clEnqueueWriteBuffer(queue, deviceInput1, CL_TRUE, 0,
                           inputLengthBytes, hostInput1, 0, NULL, NULL);
err |= clEnqueueWriteBuffer(queue, deviceInput2, CL_TRUE, 0,
                            inputLengthBytes, hostInput2, 0, NULL, NULL);

// Set the arguments to our compute kernel
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &deviceInput1);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &deviceInput2);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &deviceOutput);
err |= clSetKernelArg(kernel, 3, sizeof(int), &inputLength);
```

```
  // Execute the kernel over the entire range of the data set
  err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalSize,
                               &localSize, 0, NULL, NULL);

  // Wait for the command queue to get serviced before reading back results
  clFinish(queue);

  // Read the results from the device
  clEnqueueReadBuffer(queue, deviceOutput, CL_TRUE, 0, inputLengthBytes,
                      hostOutput, 0, NULL, NULL);

  wbSolution(args, hostOutput, inputLength);

  // release OpenCL resources
  clReleaseMemObject(deviceInput1);
  clReleaseMemObject(deviceInput2);
  clReleaseMemObject(deviceOutput);
  clReleaseProgram(program);
  clReleaseKernel(kernel);
  clReleaseCommandQueue(queue);
  clReleaseContext(context);

  // release host memory
  free(hostInput1);
  free(hostInput2);
  free(hostOutput);

  return 0;
}
```