AMD Accelerated Parallel Processing TECHNOLOGY

SAMPLE

Template

1 Overview

1.1 Location

`$<AMDAPPSDKSamplesInstallPath>\samples\opencl\cl\1.x`

1.2 How to Run

See the *Getting Started* guide for how to build samples. You first must compile the sample.

Use the command line to change to the directory where the executable is located. The pre-compiled sample executable is at
`$<AMDAPPSDKSamplesInstallPath>\samples\opencl\bin\x86\` for 32-bit builds, and
`$<AMDAPPSDKSamplesInstallPath>\samples\opencl\bin\x86_64\` for 64-bit builds.

Type `Template`. This initializes input from 0 to 255. Multiplies each input element by a scalar multiplier (the default is 2), and stores the result in the output using OpenCL kernels.

2 Introduction

This is a stand-alone OpenCL sample, independent of any utility libraries in the SDK. It is an easy sample for a new user to start coding and learn OpenCL.

The Template kernel is an OpenCL kernel that multiplies each element of the input array with a scalar, then stores it in the output array.

A new user can write a new sample by replacing the Template sample.

3 Implementation Details

This section shows how to write a simple OpenCL program. An OpenCL program comprises of:

1. Allocating and initializing any input and output memory.
2. Initializing OpenCL.
    a. Open a device (CPU/GPU) specific context.
    b. Query for a list of devices in that particular context.
    c. Open a context on the device of your choice.
    d. Create a command-queue on the context, which is associated to a device.

       OpenCL objects such as memory, program, and kernel objects are created using a context. Operations on these objects are performed using a command-queue. The command-queue can be used to queue a set of operations (referred to as commands) in order. See reference [1].

e. Create memory buffers (the memory buffers that an OpenCL program/kernel can access).

f. Create and build an OpenCL program.

   An OpenCL program consists of a set of kernels. Programs also can contain auxiliary functions called by the `__kernel` functions and constant data. There are different ways of creating an OpenCL program; one way is to create from a source, the other is to create from the binary that is obtained by compiling the OpenCL program.

g. Create a Kernel by providing the kernel function name.

   A kernel is a program that runs on an OpenCL device/compute device.

3. Provide arguments to the kernels, and run the kernels.
   a. All the arguments that must be passed to the kernel are set here.
   b. Run the kernel.
4. Clean up and release all the structures created for OpenCL.
   a. Release the created OpenCL memory buffers.
   b. Release the context created.
5. Deallocate any memory allocated specifically for this program unrelated to OpenCL.

See reference [1] for a detailed explanation of the terminology.

3.1 Initializing OpenCL

```
context = clCreateContextFromType(0, CL_DEVICE_TYPE_CPU, NULL, NULL, &status);
```

This creates an OpenCL context for the CPU devices on the system; on successful creation, it returns a valid context. The status is set to `CL_SUCCESS` on successful creation of *context*. For error codes, see reference [1].

```
status = clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &deviceListSize);
```

By passing only `CL_CONTEXT_DEVICES` as the second argument and `deviceListSize` as the fifth, we get the number of devices in this context to allocate memory for the list of devices.

```
status = clGetContextInfo(context, CL_CONTEXT_DEVICES, deviceListSize, devices, NULL);
```

This passes the *deviceListSize* and *devices* in order to get all the information regarding each of devices on the device list.

```
commandQueue = clCreateCommandQueue( context, devices[0], 0, &status);
```

This creates a command-queue on a specific device; here, *devices[0]*.

```
inputBuffer = clCreateBuffer( context, CL_MEM_READ_WRITE | CL_MEM_USE_HOST_PTR,
sizeof(cl_uint) * width, input, &status);
```

A memory buffer (accessible to OpenCL kernels and programs) is created by providing the necessary arguments. `CL_MEM_READ_WRITE` indicates that the memory buffer object can be used to read from it and write to it. `CL_MEM_USE_HOST_PTR` indicates that the buffer uses only a host pointer to create the buffer object. This pointer is the *input* of size *sizeof(cl_uint)*width*. This means any read and write operation in the OpenCL kernel is similar to reading and writing to the *input* array. On successful creation of *inputBuffer*, the status stores `CL_SUCCESS`.

```
const char * filename  = "Template_Kernels.cl";
const char * source    = convertToString(filename).c_str();
size_t sourceSize[]     = { strlen(source) };
```

The `filename` stores the path for the openCL kernel. The kernel is converted into a string and stored in `source`; `sourceSize` stores the length of the kernels.

```
program = clCreateProgramWithSource(
              context,
                    1,
              &source,
           sourceSize,
              &status);
```

This creates a program object for a context so that it can be run on each device that belongs to *devices* array. The `1` denotes the number of devices on which the program object is to be created.

```
status = clBuildProgram(program, 1, devices, NULL, NULL, NULL);
```

This compiles and links all the binary files supplied to the program in the previous step, and builds the `program` object for all the devices in the `devices` list, which is given as an argument. The `1` denotes the number of devices in the `devices` list.

```
kernel = clCreateKernel(program, "templateKernel", &status);
```

This provides a handle to a particular kernel by passing the entry point (the name by which the kernel function is defined).

3.2 Run OpenCL Programs

```
globalThreads[0] = width;
localThreads[0]  = 1;
```

These variables define the number of times a kernel executes. Here, it executes in a 1-dimensional index space, with the number of threads equal to *width.*

```
status = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&outputBuffer);
status = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&inputBuffer);
status = clSetKernelArg(kernel, 2, sizeof(cl_uint), (void *)&multiplier);
```

This sets the first, second, and third arguments to the kernel.

```
status = clEnqueueNDRangeKernel(
                                commandQueue,
                                kernel,
                                1,
                                NULL,
                                globalThreads,
                                localThreads,
```

```
                                0,
                                NULL,
                                &events[0]);
```

This queues the instructions for a particular devices through the *commandQueue* of that device. It instructs the device to run the kernel in 1-dimensional index space (third argument) with the dimensions stored in *globalThreads* array. The `events[0]` returns an event object that identifies the particular kernel execution instance.

```
status = clWaitForEvents(1, &events[0]);
```

This is a waiting loop that breaks on successful execution of the kernel.

3.2.1 Release and Clean Up OpenCL

```
status = clReleaseMemObject(inputBuffer);
```

This releases the structures created to maintain the memory buffer object used. In this example, it releases all the memory allocation for maintaining `inputBuffer`.

```
status = clReleaseContext(context);
```

This releases the context and all the devices that it has created.

3.2.2 OpenCL Kernel

```
__kernel void templateKernel(__global unsigned int * output,
                             __global  unsigned int * input,
                             __const  unsigned int multiplier)
```

The `__kernel` denotes that the function is a kernel function. It has three arguments: *output* array (the output buffer is passed as an argument), *input* array (the input Buffer created is passed as an argument), and multiplier (which is a constant).

```
uint tid = get_global_id(0);
```

This provides the thread id in the global execution space. For this kernel, the execution space is the same size as that of the array (*output, input*). Thus, each instance of the kernel that is executed has associated with it an element in the array.

```
output[tid] = input[tid] * multiplier;
```

Depending on the threadid(*tid*), we take an element in the input array, multiply it with a scalar, and store the output at the corresponding location in the output array.

4 References

1. http://www.khronos.org/opencl/

Contact

Advanced Micro Devices, Inc.
One AMD Place
P.O. Box 3453
Sunnyvale, CA, 94088-3453
Phone: +1.408.749.4000

For AMD Accelerated Parallel Processing:
URL: **developer.amd.com/appsdk**
Developing: **developer.amd.com/**
Support: **developer.amd.com/appsdksupport**