Exercise 1 (EX1)

In this first set of exercises and homework you will achieve the following:

- Get your system set up with Visual Studio 2019 Community edition and the CUDA Toolkit 11.x to allow you to build GPU projects using both CUDA and OpenCL.
- Get some experience using the CUDA and OpenCL host-side APIs to query your system for its hardware properties and capabilities, which will be discussed in detail during the class and will be used in later exercises and homeworks.
    - this will also serve as a review/reminder of basic C programming syntax and usage of the Visual Studio IDE.
- To "get your feet wet", you'll write your first GPU kernels using the simpler CUDA runtime API to implement the host application to execute them on your GPU. The runtime API is nice because it makes it extremely simple to get a basic CUDA application up-and-running.
    - It turns out the runtime API is sometimes needed to use other CUDA libraries, such as cuFFT, cuBLAS, and others. We'll discuss this more later on...
- Use basic GPU API error handling.
- Start to become familiar with using the CUDA C++ Programming Guide and the OpenCL API and C Language specifications as references to implement host and kernel code. This will be a vital skill throughout the course.
    - NOTE: I'm intentionally not spelling everything out for you entirely for these programming exercises, but I am providing all the necessary reference links, to "inspire" you to practice using the references and get comfortable figuring some of this stuff out yourself. An important skill.

1. Get VS19 Community and CUDA TK 11 installed

a. see lecture #1 slides for the download links

2. Configuring a VS19 project for building CUDA and OpenCL

This information will be useful throughout the course, as we'll be creating many CUDA and OpenCL VS19 projects, which you'll need to understand how to set up to build successfully!

After creating a new solution and an empty C++ project using the wizard,
In VS Project Properties window

Set Configuration: All Configurations
Platform: x64 (assuming everyone is on 64-bit OS these days.)

VC++ Directories

Include Directories: <CUDA Toolkit install path>\include;<CUDA Toolkit install path>\common\inc;
Library Directories: <CUDA Toolkit install path>\lib;

C/C++
General
Additional Include Directories: C:\Users\colin\Source\Repos\shared (whereever you put your shared folders)

Linker/Input
Additional Dependencies: x64\opencl.lib; x64\cudart_static.lib, (etc…)
In your VS Projects source files (.cpp)
files in the Shared folder location subfolders {src, inc, lib} need to be referenced in #includes as

    #include <inc/oclUtils.h> (for example, etc.)

Exercise 1.A. Platform/Device Query

In exercise 1.A. you'll use both the CUDA Driver API and the OpenCL API to implement host-side applications to query your system for CUDA/OpenCL platform and hardware properties and capabilities. Later in the class we will study the details of what this information means and how it can be used to design and optimize your device kernel codes.
NOTE: for this exercise no GPU kernels will be defined or executed.

=== CUDA driver API

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#driver-api
https://docs.nvidia.com/cuda/cuda-driver-api/index.html

// CUDA driver API routines (prefixed with "cu")
#include <cuda.h>

using CUDA driver API:
link to cuda.lib, implementation in cuda.dll/.so

- VS Linker/Additional Dependencies: cuda.lib

function entry points prefixed with "cu"
initialize with cuInit()

relevant API functions
cuInit, cuDeviceGetCount, cuDeviceComputeCapability, cuDriverGetVersion, cuDeviceTotalMem, cuDeviceGetAttribute, cuDeviceGetName

Tasks

1. enumerate the available CUDA devices and get their names
2. For each CUDA device, query the following attributes and print them to console
    a. CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILTY_MAJOR
    b. CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILTY_MINOR
    c. Cuda driver version
    d. CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT
    e. CU_DEVICE_ATTRIBUTE_CLOCK_RATE
    f. CU_DEVICE_ATTRIBUTE_MEMORY_CLOCK_RATE
    g. CU_DEVICE_ATTRIBUTE_GLOBAL_MEMORY_BUS_WIDTH
    h. CU_DEVICE_ATTRIBUTE_L2_CACHE_SIZE
    i. CU_DEVICE_ATTRIBUTE_TOTAL_CONSTANT_MEMORY
    j. CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK
    k. CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_BLOCK
    l. CU_DEVICE_ATTRIBUTE_WARP_SIZE
    m. CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_MULTIPROCESSOR
    n. CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK
    o. CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_X
    p. CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Y
    q. CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Z
    r. CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_X
    s. CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Y
    t. CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Z

=== OpenCL API (host)

https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf

# OpenCL API routines (prefixed with"cl")
#include <CL/opencl.h> // will be a subdirectory of CUDA/include folder

static link to: opencl.lib, implemented in opencl.dll (and vendor driver DLLs)

- VS Linker/Additional Dependencies: x64/opencl.lib

relevant API functions:
clGetPlatformIDs
clGetPlatformInfo
clGetDeviceIDs
clGetDeviceInfo

- you'll see there is a huge list of Device Info parameters, with various return types

Tasks

1. enumerate the available OpenCL devices and get their names

2. For the GPU device type, retrieve the following info
    a. CL_DEVICE_MAX_COMPUTE_UNITS
    b. CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS
    c. CL_DEVICE_MAX_WORK_ITEM_SIZES
    d. CL_DEVICE_MAX_WORK_GROUP_SIZE
    e. CL_DEVICE_MAX_CLOCK_FREQUENCY
    f. CL_DEVICE_MAX_MEM_ALLOC_SIZE
    g. CL_DEVICE_SINGLE_FP_CONFIG
    h. CL_DEVICE_DOUBLE_FP_CONFIG
    i. CL_DEVICE_NAME
    j. CL_DEVICE_EXTENSIONS
3. Print out results of above queries to command console

Exercise 1.B. HelloGPUWorld kernel

In exercise 1.B. you will use the CUDA runtime API to implement a simple CUDA kernel and use the printf statement to output information from your GPU when the kernel executes.

In your <filename>.cu file
#include <stdio.h> // to pull in printf

=== CUDA runtime API

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-c-runtime
https://docs.nvidia.com/cuda/cuda-runtime-api/index.html

// CUDA runtime routines (prefixed with "cuda")
#include <cuda_runtime.h>

static link to: cudart_static.lib, implemented in cudart.dll/.so

- Linker/Additional Dependencies: cudart_static.lib

Using CUDA runtime API:
no explicit init (auto-inits when first runtime fn called)
Kernels operate out of device memory, so the runtime provides functions to allocate, deallocate, and copy device memory, as well as transfer data between host memory and device memory.

- malloc/free memory on host
- cudaMalloc/cudaFree memory on device
- cudaMemcpy - transfer between host & device memory

Tasks

1. Define a CUDA kernel named: helloGPUworld

2. using printf in the kernel body, and the built-in grid, block, and thread index and dimension variables, print out the kernel instance indices info to the console.
3. in the host application, experiment with various values of the execution configuration (exec cfg), such as
    a. <<<1,16>>>
    b. <<<4,4>>>
    c. <<<16,1>>>
    d. Run each case several times (like 10 times).
4. Some questions to consider:
    a. What do you observe about the results?
    b. Does the behavior of blocks and threads seem to differ?
    c. What does this imply about computations you try to perform in the kernels?

Exercise 1.C. Simplest Vector Addition

In exercise 1.C you will use the CUDA runtime API to execute the simplest version of the vector addition GPU kernel. The same basic project setup will be used as for Ex 1.B.

Include CUDA error handling (refer to the Lecture #1 slides for CUDA Error Handling examples)

Questions:

- how large can you make the vectors to be added before errors occur?
- What are the errors?
- Can you relate the errors to any device properties you queried in EX 1.A.?