Win21 EX2/HW1

DUE:

Overview

Work with the OpenCL (OCL) API and the CUDA Driver API (more powerful and low-level than the CUDA runtime API) to implement host application frameworks which are reusable to launch various different kernels, and support measurement of kernel execution times.
Work with the OpenCL C and CUDA C++ kernel language extensions, and implement several new computational kernels which include features such as 2D computation domains (index spaces), out-of-bounds handling, grid-stride loops, and native vector data types.
Perform verification of results between CPU and GPU versions including proper handling of numerical precision issues which may result in non-identical results even from “correct” calculations.

Goals/Outcomes

1. Implement reusable “boilerplate” host application framework projects in both OpenCL and CUDA Driver APIs, which will be used throughout the remainder of course.
2. Learn how to allocate memory on host and device and transfer data-memory in both directions (HtoD, DtoH).
3. Learn how to hook up kernel arguments to be passed from host to the device kernel instances.
4. Get some experience using OpenCL C and CUDA C++ kernel language syntax.
5. Implement new computational GPU kernels in OpenCL and CUDA
    a. matrix addition
    b. dot product
    c. BLAS Level 2 D/SGEMV (Double/Single GEneral Matrix-Vector product)
6. Perform result verification between CPU baseline and GPU compute kernel versions, accounting for possibility of numerical precision issues.

Procedures

1. Implement host application framework projects which can be reused to launch several different GPU kernels. Implement 2 host app frameworks, one using OpenCL API and one using the CUDA Driver (DRV) API. Follow the guidelines in **Lecture 3 - Part 1** slides on the basic steps for host applications.

a. Include the code comments from Lecture 1 Slide 45 (Basic Host Application Template) in your host application and use them to fill in the implementation code for each sub-section
b. For OpenCL: use the provided utility header file `read_source.h`, which contains the function `read_source( )`.
    i. This routine will read a separate source code file containing a kernel function and return it as a char * array which can be passed directly to the OpenCL API routines to compile the kernel.
    ii. File is provided in <class web>/Files/src
c. For CUDA: you'll need to compile your cuda kernel.cu file using nvcc to generate a PTX
    i. `nvcc -o %SRCPATH%\%SRCFILE%.ptx %SRCPATH%\%SRCFILE%.cu -ptx`
    ii. then you pass the kernel.ptx to cuModuleLoad( )
    iii. see Lec 3 - Part 1 slide #34.
d. use the 'advanced' kernel arg method (*extra* parameter) in the CUDA DRV API host app to pass the kernel arguments when launching kernels with `cuLaunchKernel()`.
    i. see Lec 3 - Part 1 slide #34 for example using the "simple" kernel arg method.
e. Also refer to
    i. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#kernel-execution
    ii. https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__EXEC.html#group__CUDA__EXEC_1gb8f3dc3031b40da29d5f9a7139e52e15
f. Ensure both the host app frameworks build and run error-free.
    i. Dev Hint: you can use a 'dummy' kernel with no arguments to test initially.

2. **Implement kernel functions** in both CUDA and OpenCL. Build and Launch these kernels from within your two host applications and follow the specific instructions for each kernel below:

a. matrix add

i. Pass data from host to device as a *row-major* linearized 2D matrix array.
ii. Include correct boundary handling & grid-stride looping
iii. iterate kernel threads row-wise & col-wise
- HW Q: in your kernel functions, include comments which clearly explain if row-wise or col-wise thread access pattern is being used, and how you know this.
- HW Q: profile using high-resolution performance counters around the kernel dispatch call to obtain estimates of kernel execution time.
- https://docs.microsoft.com/en-us/windows/win32/api/profileapi/
    - Be sure to stop the timing after the post-launch Host-GPU synchronization call point.
        - HW Q: Why?

        - HW Q: why is this not a fully satisfactory measurement of the kernel execution time?
    - HW Q: is one faster? why?

b. dot product

i. in your kernel, make use of the built-in OCL/CUDA vector datatypes
    - implement kernel variants using float2 and float4
    - using as large data input vectors as possible on your HW, measure the execution times of the float/float2/float4 variants.
        - HW Q: Make a table to record the execution times for each kernel in OCL and CUDA. Is there a noticeable difference?

c. BLAS Level 2 DGEMV (or SGEMV if your system doesnt have D)

i. https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms#Level_2

ii. All scalar, vector, and matrix arguments should be floating-point
    - Use either single (S, 32-bit) or double (D, 64-bit) depending on what your HW supports.
    - D should be supported on “most” NVIDIA
    - on Intel use Intel OpenCL SDK and query
        - CL_DEVICE_DOUBLE_FP_CONFIG
        - requires >= OpenCL 1.2

3. Include verification routines in the host application to VERIFY results of same computations on CPU vs GPU
    a. thus a serial CPU routine will need to be implemented for each calc also.
    b. Use a tolerance threshold for result agreement due to numerical precision diffs, as discussed in Lecture 2.
    c. HW Q: At what tolerance value do verifications start to fail?
    d. HW Q: which of the 3 kernels exhibits the most severe result disagreements between CPU and GPU versions? Why?

SUBMIT

1. Submit answers to above highlighted HW Q questions.
2. Submit your OpenCL and CUDA host application source code
    a. .cpp and .h files only - not all project & solution files and other by-products!
3. Submit your kernel functions (both OpenCL and CUDA) for all kernel types and variants implemented.