CME433 - Lab 6
Graphics Processing Unit

I require the use of Linux for this lab (although you can write OpenCL on windows as well…).
Hint: Most problems and questions in this lab are solved by reading this lab manual! Don't skip steps or information… anything else you need can be read about in the link below or by following comments in the example code.

Objectives

The objective of this lab is to study the architecture of a Graphics Processing Unit (GPU) and the OpenCL programming language. By the end of this lab, students should be able to:

- Understand the architecture of the Intel GPU.
- Understand the interaction between the CPU and GPU in a computer.
- Write a basic OpenCL program which takes advantage of parallel programming.

Background

First, this reference is your friend:
https://www.fixstars.com/en/opencl/book/OpenCLProgrammingBook/contents/

OpenCL™ is the first open, royalty-free standard for cross-platform, parallel programming of modern processors found in personal computers, servers, and hand-held or embedded devices. OpenCL (which stands for "Open Computing Language") greatly improves speed and responsiveness for a wide range of applications in numerous market categories from gaming and entertainment to scientific and medical software. To put it simply, OpenCL (Open Computing Language) is "a framework suited for parallel programming of heterogeneous systems". The framework includes the OpenCL C language as well as the compiler and the runtime environment required to run the code written in OpenCL C.

OpenCL C language, as the name suggests, uses almost the exact same syntax as the C language. It is extended to support SIMD vector operations and multiple memory hierarchies, and some features not required for computations, such as printf(), were taken out. The control program using the OpenCL runtime API can be written in C or C++, and does not require a special compiler.

OpenCL explicitly separates the program to be ran on the host side (CPU) and the device side (GPU).

The program to be ran on the device (GPU) is called a kernel. When creating an OpenCL application, this kernel is written in OpenCL C, and compiled using the OpenCL compiler for that device.

The host is programmed using C/C++ using the OpenCL runtime API, which is linked to an OpenCL runtime library implemented for the host-device combination. This code is compiled using compilers such as GCC.

In OpenCL, a kernel can be compiled either ‘online’ or ‘offline’. We will be using ‘online’ for this lab.

- Offline: Kernel binary is read in by the host code
- Online: Kernel source file is read in by the host code

In "offline-compilation", the kernel is pre-built using an OpenCL compiler, and the generated binary is what gets loaded using the OpenCL API.

In "online-compilation", the kernel is built from source during runtime using the OpenCL runtime library. This method is commonly known as JIT (Just in time) compilation. The advantage of this method is that the host-side binary can be distributed in a form that's not device-dependent, and that adaptive compiling of the kernel is possible. It also makes the testing of the kernel easier during development, since it gets rid of the need to compile the kernel each time. However, this is not suited for embedded systems that require real-time processing. Also, since the kernel code is in readable form, this method may not be suited for commercial applications.

OpenCL provides API for the following programming models.

- Data parallel programming model
- Task parallel programming model

In the data parallel programming model, the same kernel is passed through the command queue to be executed simultaneously across the compute units or processing elements.

For the task parallel programming model, different kernels are passed through the command queue to be executed on different compute units or processing elements.

For data parallel programming models, each kernel requires a unique index so the processing is performed on different data sets. OpenCL provides this functionality via the concept of index space.

OpenCL gives an ID to a group of kernels to be run on each compute unit, and another ID for each kernel within each compute unit, which is run on each processing element. The ID for the compute unit is called the Workgroup ID, and the ID for the processing element is called the Work Item ID. When the user specifies the total number of work items (global item), and the number of work items to be ran on each compute unit (local item), the IDs are given for each item by the OpenCL runtime API. The index space is defined and accessed by this ID set. The number of workgroups can be computed by dividing the number of global items by the number of local items.

The ID can have up to 3 dimensions to correspond to the data to process. The retrieval of this ID is necessary when programming the kernel, so that each kernel processes different sets of data.

As a preliminary to this lab, you will need to firstly determine if your 'machine' can use OpenCL. You will also need to be able to compile and run C and OpenCL programs from Linux.

Compiling Programs:

Use this command to compile programs:

g++ -o output source.c -lOpenCL

The breakdown of this command is as follows:
g++: the compiler
-o ouput: make the executable output named 'output'
source.c: the source files to compile
-lOpenCL: include this library (needed for .cl programs)

Want to use a GUI instead of terminal? Go for it! Most GUIs allow you to specify compiler options. GUI too slow? Don't like punching that in every time? Make a makefile (pun intended)!

In order to determine if your 'machine' supports OpenCL, compile and run the program clDeviceQuery.cpp (or deviceQuery.c). You will get an output that looks similar to appendix A.

For your convenience, those files are stored in the directory: /*opt*/opencl/
You will need to copy them to your NFS as you don’t have write permissions to that location.

The first thing to look for is the GPU (with this being a GPU lab, we need a supported GPU). There is a wealth of information about the GPU, some of which will be necessary when doing this lab (and will be explained in the lab, and referenced in the example vectorAdd.c given).

Particularly, these items are of interest for this lab:
CL_DEVICE_VERSION
CL_DEVICE_MAX_CLOCK_FREQUENCY
CL_DEVICE_MAX_COMPUTE_UNITS
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS
CL_DEVICE_MAX_WORK_ITEM_SIZES
CL_DEVICE_MAX_WORK_GROUP_SIZES

Procedure

Read the background section. The whole thing. Don't be lazy. Don't skim it, read it again.

Download, compile, and run clDeviceQuery.cpp (or deviceQuery.c)

Download, compile, and run the example given (vectorAdd.c and vectorAdd.cl).

Read through the code and understand it. I'd recommend giving this a read-through:
https://www.fixstars.com/en/opencl/book/OpenCLProgrammingBook/basic-program-flow/

You can read through another example here:
https://www.fixstars.com/en/opencl/book/OpenCLProgrammingBook/first-opencl-program/

You are given 3 template files: matrixMul.c, matrixMul_host.c, matrixMul_kernel.cl

Step 1: complete matrixMul.c
This will be your CPU comparison – just write an ordinary 'C' program which computes a matrix multiplication. You will need to use a timer to compute how long it takes to perform the operation.

Reminder of Matrix multiplication:

$$\mathbf{AB} = \begin{pmatrix} a & b & c \\ p & q & r \\ u & v & w \end{pmatrix} \begin{pmatrix} \alpha & \beta & \gamma \\ \lambda & \mu & \nu \\ \rho & \sigma & \tau \end{pmatrix} = \begin{pmatrix} a\alpha + b\lambda + c\rho & a\beta + b\mu + c\sigma & a\gamma + b\nu + c\tau \\ p\alpha + q\lambda + r\rho & p\beta + q\mu + r\sigma & p\gamma + q\nu + r\tau \\ u\alpha + v\lambda + w\rho & u\beta + v\mu + w\sigma & u\gamma + v\nu + w\tau \end{pmatrix},$$

Start with a small matrix (4x4) so you can output the results and verify that your programs are working.
Use Matlab (or similar program) to verify that the result is correct.
**Take a screen shot of this result, the Matlab/other result, and include them in your report.
Make a small table which has several matrix sizes and their computation times (up to 512x512 minimum)
**Include this table in your report.

Step 2: Complete matrixMul_host.c and matrixMul_kernel.cl
Use the vectorAdd.c and vectorAdd.cl programs as reference, complete these programs and compare the results to your CPU calculations.

Hint: A common mistake in writing the CL program is to write a fully sequential program – you are only writing the program which ONE work item would execute. Think parallel! Give the Appendix sections a good read-through before doing this part of the lab.

Start with a small matrix (4x4) so you can output the results and verify that your programs are working.
**Take a screen shot of this matrix and include it in your report.
Using the table that you generated with your CPU, run your GPU program for comparison.
**Include this table in your report.

**Summarize your findings and the results of the learning objectives of this lab in your report.

Appendix A: Example OpenCL device query output:

clDeviceQuery Starting...

2 devices found supporting OpenCL on: Intel(R) OpenCL

Device Intel(R) HD Graphics

CL_DEVICE_NAME: Intel(R) HD Graphics
CL_DEVICE_VENDOR: Intel(R) Corporation
CL_DRIVER_VERSION: r3.1.58620
CL_DEVICE_TYPE: CL_DEVICE_TYPE_GPU
CL_DEVICE_MAX_COMPUTE_UNITS: 20
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS: 3
CL_DEVICE_MAX_WORK_ITEM_SIZES: 256 / 256 / 256
CL_DEVICE_MAX_WORK_GROUP_SIZE: 256
CL_DEVICE_MAX_CLOCK_FREQUENCY: 1150 MHz
CL_DEVICE_ADDRESS_BITS: 64
CL_DEVICE_MAX_MEM_ALLOC_SIZE: 815 MByte
CL_DEVICE_GLOBAL_MEM_SIZE: 1630 MByte
CL_DEVICE_ERROR_CORRECTION_SUPPORT: no
CL_DEVICE_LOCAL_MEM_TYPE: local
CL_DEVICE_LOCAL_MEM_SIZE: 64 KByte
CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE: 834764 KByte
CL_DEVICE_QUEUE_PROPERTIES: CL_QUEUE_PROFILING_ENABLE
CL_DEVICE_IMAGE_SUPPORT: 1
CL_DEVICE_MAX_READ_IMAGE_ARGS: 128
CL_DEVICE_MAX_WRITE_IMAGE_ARGS: 128

CL_DEVICE_IMAGE <dim> 2D_MAX_WIDTH 16384
2D_MAX_HEIGHT 16384
3D_MAX_WIDTH 2048
3D_MAX_HEIGHT 2048
3D_MAX_DEPTH 2048
CL_DEVICE_PREFERRED_VECTOR_WIDTH_<t> CHAR 16, SHORT 8, INT 4, FLOAT 1, DOUBLE 1

Device Intel(R) Core(TM) i5-4570T CPU @ 2.90GHz

CL_DEVICE_NAME: Intel(R) Core(TM) i5-4570T CPU @ 2.90GHz
CL_DEVICE_VENDOR: Intel(R) Corporation
CL_DRIVER_VERSION: 1.2.0.330

CL_DEVICE_TYPE: CL_DEVICE_TYPE_CPU
CL_DEVICE_MAX_COMPUTE_UNITS: 4
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS: 3
CL_DEVICE_MAX_WORK_ITEM_SIZES: 8192 / 8192 / 8192
CL_DEVICE_MAX_WORK_GROUP_SIZE: 8192
CL_DEVICE_MAX_CLOCK_FREQUENCY: 2900 MHz
CL_DEVICE_ADDRESS_BITS: 64
CL_DEVICE_MAX_MEM_ALLOC_SIZE: 1928 MByte
CL_DEVICE_GLOBAL_MEM_SIZE: 7713 MByte
CL_DEVICE_ERROR_CORRECTION_SUPPORT: no
CL_DEVICE_LOCAL_MEM_TYPE: global
CL_DEVICE_LOCAL_MEM_SIZE: 32 KByte
CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE: 128 KByte
CL_DEVICE_QUEUE_PROPERTIES:CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE
CL_DEVICE_QUEUE_PROPERTIES: CL_QUEUE_PROFILING_ENABLE
CL_DEVICE_IMAGE_SUPPORT: 1
CL_DEVICE_MAX_READ_IMAGE_ARGS: 480
CL_DEVICE_MAX_WRITE_IMAGE_ARGS: 480

CL_DEVICE_IMAGE <dim> 2D_MAX_WIDTH 16384
2D_MAX_HEIGHT 16384
3D_MAX_WIDTH 2048
3D_MAX_HEIGHT 2048
3D_MAX_DEPTH 2048
CL_DEVICE_PREFERRED_VECTOR_WIDTH_<t> CHAR 1, SHORT 1, INT 1, FLOAT 1, DOUBLE 1

clDeviceQuery, Platform Name = Intel(R) OpenCL, Platform Version = OpenCL 1.2 , NumDevs = 2, Device = Intel(R) HD Graphics, Device = Intel(R) Core(TM) i5-4570T CPU @ 2.90GHz

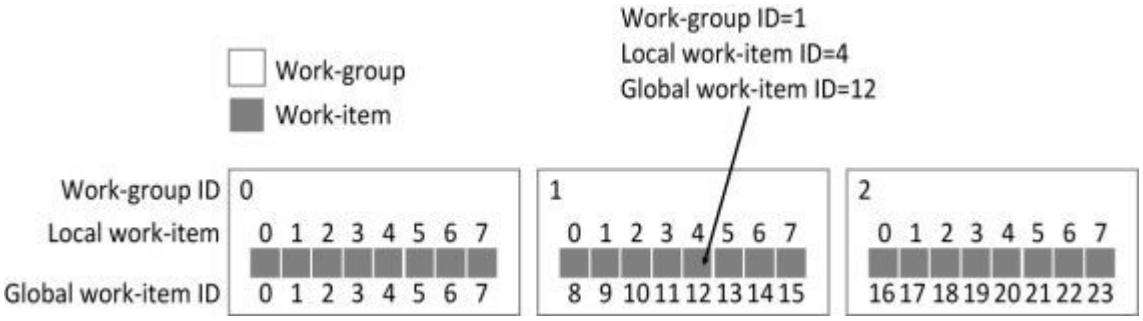
System Info:

Local Time/Date = 13:40:00, 12/02/2016
CPU Name: Intel(R) Core(TM) i5-4570T CPU @ 2.90GHz
# of CPU processors: 4
Linux version 3.10.0-514.el7.x86_64 (mockbuild@x86-039.build.eng.bos.redhat.com) (gcc version 4.8.5 20150623 (Red Hat 4.8.5-11) (GCC)) #1 SMP Wed Oct 19 11:24:13 EDT 2016

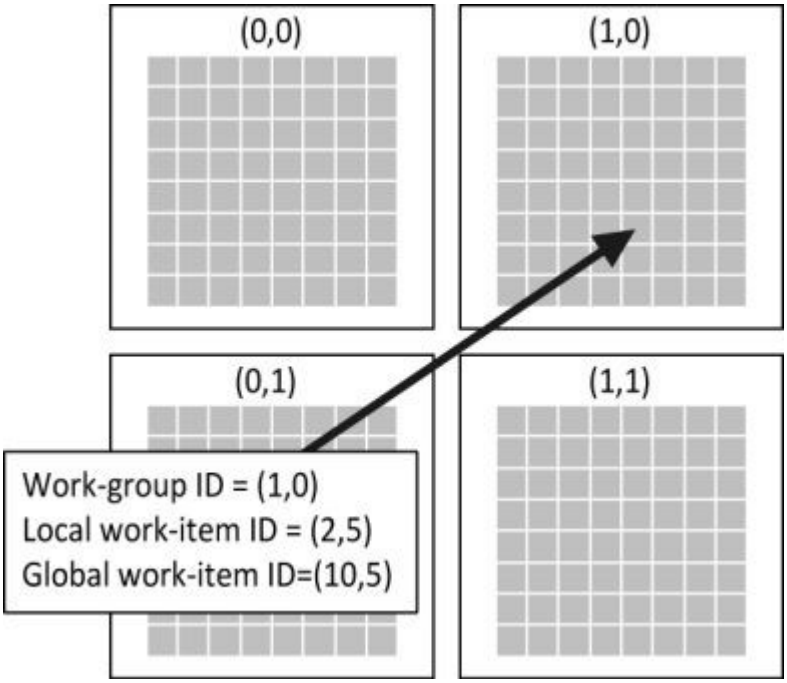
TEST PASSED

Appendix B: Local and Global Work Groups

https://www.fixstars.com/en/opencl/book/OpenCLProgrammingBook/calling-the-kernel/



get_group_id(0) 1
get_group_id(1) 0
get_global_id(0) 10
get_global_id(1) 5
get_local_id(0) 2
get_local_id(1) 5



Appendix C: Restrictions

Below lists the restricted portion from the C language in OpenCL C.

- The pointer passed as an argument to a kernel function must be of type __global, __constant, or __local.
- Pointer to a pointer cannot be passed as an argument to a kernel function.
- Bit-fields are not supported.
- Variable length arrays and structures with flexible (or unsized) arrays are not supported.
- Variadic macros and functions are not supported.
- C99 standard headers cannot be included
- The extern, static, auto and register storage-class specifiers are not supported.
- Predefined identifiers are not supported.
- Recursion is not supported.
- The function using the __kernel qualifier can only have return type void in the source code.
- Writes to a pointer of type char, uchar, char2, uchar2, short, ushort, and half are not supported.
- Support for double precision floating-point is currently an optional extension. It may or may not be implemented.

Appendix D: Address Space Qualifiers

OpenCL supports 4 memory types: global, constant, local, and private. In OpenCL C, these address space qualifiers are used to specify which memory on the device to use. The memory type corresponding to each address space qualifiers are shown in Table 5.1.

Qualifier	Corresponding memory
__global (or global)	Global memory
__local (or local)	Local memory
__private (or private)	Private memory
__constant (or constant)	Constant memory

The "__" prefix is not required before the qualifiers, but we will continue to use the prefix in this text for consistency. If the qualifier is not specified, the variable gets allocated to "__private", which is the default qualifier.

The private address space is the default for variables. Variables in the private address space are only accessible within the (individual) work-item.

The local address space name is used to describe variables that need to be allocated in local memory and are shared by all work-items of a work-group. Local memory that belongs to another work-group may not be accessed. These variables cannot be initialized.

The global address space is… global.

Appendix E: Advanced OpenCL Goodies

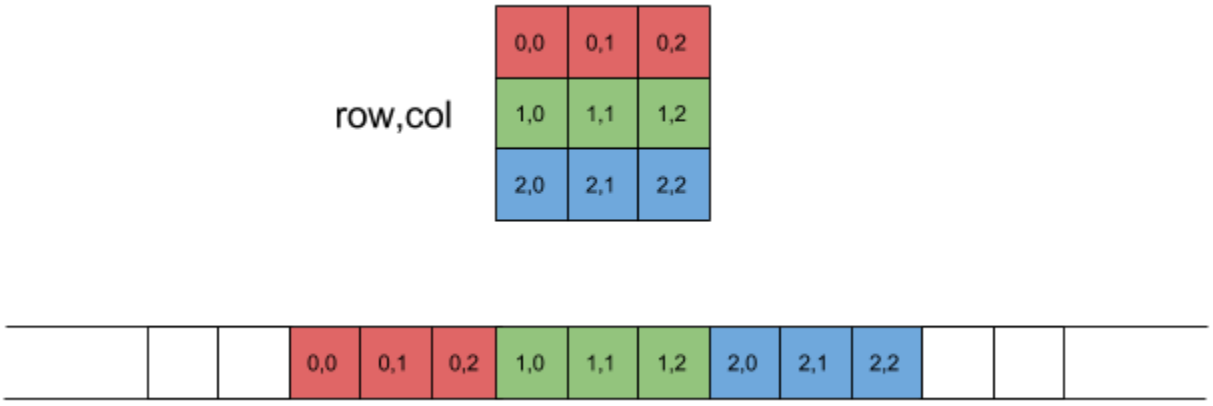
OpenCL contains several built-in functions for scalar and vector operations. These can be used **without** having to include a header file or linking to a library. These include functions related to:
- Work-item Functions (get_work_dim, get_global_id, get_local_id, get_group_id, etc.)
- Math Functions (sin, cos, tan, log, powr, remainder, etc.)
- Integer Functions (abs, clamp, clz, min, max, upsample, popcount)
- Common Functions (clamp, min, max, degrees, radians, mix, step, sign)
- Geometric Functions (cross, dot, distance, length, normalize)
- Relational Functions
- Vector Functions
- Barrier Functions
- Memory Functions
- Image Functions

OpenCL variable types

Data types	Bit width	Remarks
bool	Undefined	
char	8	
unsigned char, uchar	8	
short	16	
unsigned short, ushort	16	
int	32	
unsigned int, uint	32	
long	64	
unsigned long, ulong	64	
float	32	
half	16	
size_t	*	stddef.h not required
intptr_t	*	stddef.h not required
uintptr_t	*	stddef.h not required

Appendix F: Computing Large Matrix Hints

1. The matrix operations can be performed in standard two dimensional format, but can also be performed in one dimension.. How? Think of the matrix as a large array where the location in the array is equal to the column_number * row_number.



2. In order to achieve efficiency out of the GPU, you **MUST** use more than one dimension in your workspace. *The dimensions used in the calculation workspace does not have to match the dimensions of the input arrays.* For example, if you choose to do hint #1 as a single dimension array, the data can still be mapped to two dimensions in the workspace. This is essentially the reverse operation of hint #1. Failure to use more than one dimension limits the workspace to the maximum amount of work IDs equal to the max dimensional size. This is most cases will be too small, and it ties up blocks of the GPU which could be used for other operations (architecture dependent).