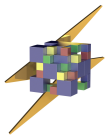
Lecture 16

OpenCL

이재진
서울대학교 데이터사이언스대학원
서울대학교 공과대학 컴퓨터공학부
http://aces.snu.ac.kr/~jlee



THUNDER Research Group
Seoul National University
서울대학교 천둥 연구실

OpenCL

- Open Computing Language

- A framework (parallel programming model) for heterogeneous parallel computing
  - A language, API, libraries, and a runtime system
- License free
- The specification of OpenCL 1.0 was released in late 2008

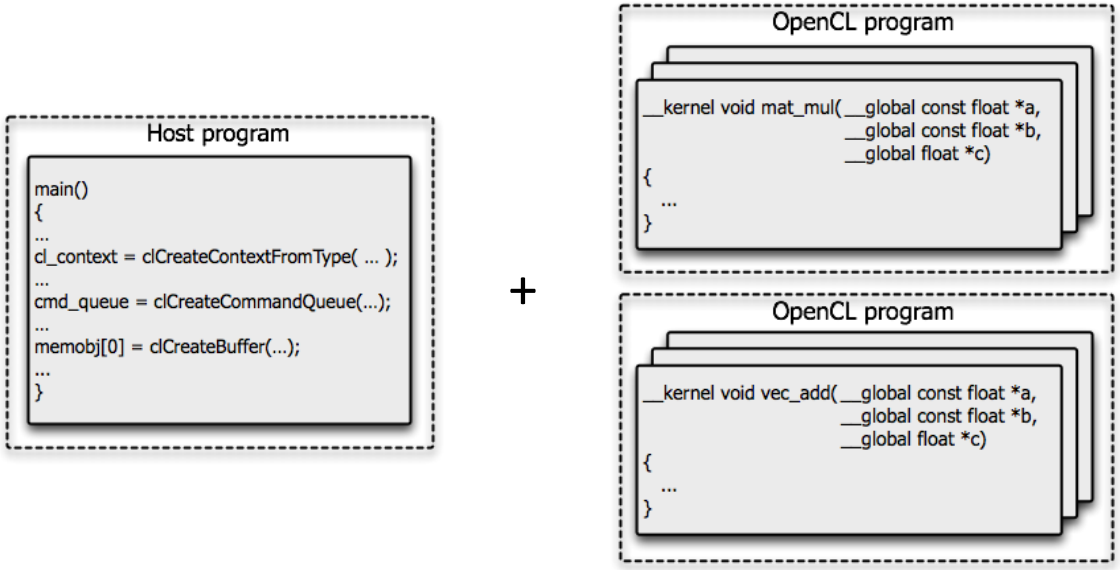
OpenCL (cont'd)

- From mobile devices to supercomputers

- Portable code across different architectures
  - CPUs, GPUs, Cell BE processors, Xeon Phi, etc.
  - Not yet portable performance
- Based on ANSI/ISO C99 standard
- Supported by many vendors, such as Apple, AMD, ARM, IBM, Intel, NVIDIA, Samsung, TI, Qualcomm, etc.

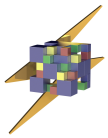
OpenCL Application

- The combination of programs running on a host processor and OpenCL compute devices
  - Compute devices: CPUs, GPUs, Xeon Phi, etc.
- A host program + OpenCL programs
  - OpenCL program: a set of kernels



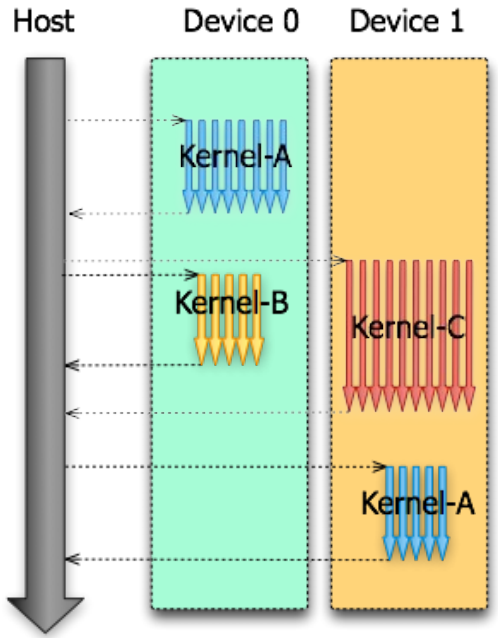
OpenCL C Language

- For kernels
- Based on ISO C99
- Restrictions
  - No standard C99 headers, function pointers, recursion, variable length arrays, and bit fields
- Extensions
  - Vector types
  - Image types
  - Synchronization
  - Address space qualifiers
  - Built-in functions



OpenCL Application

- Host program
  - Executes on the host and manages kernel execution
- Kernels
  - Basic unit of executable code (a function) on compute devices
  - When executed, many instances are created
    - Exploits data parallelism
- The host program and kernels all run in parallel

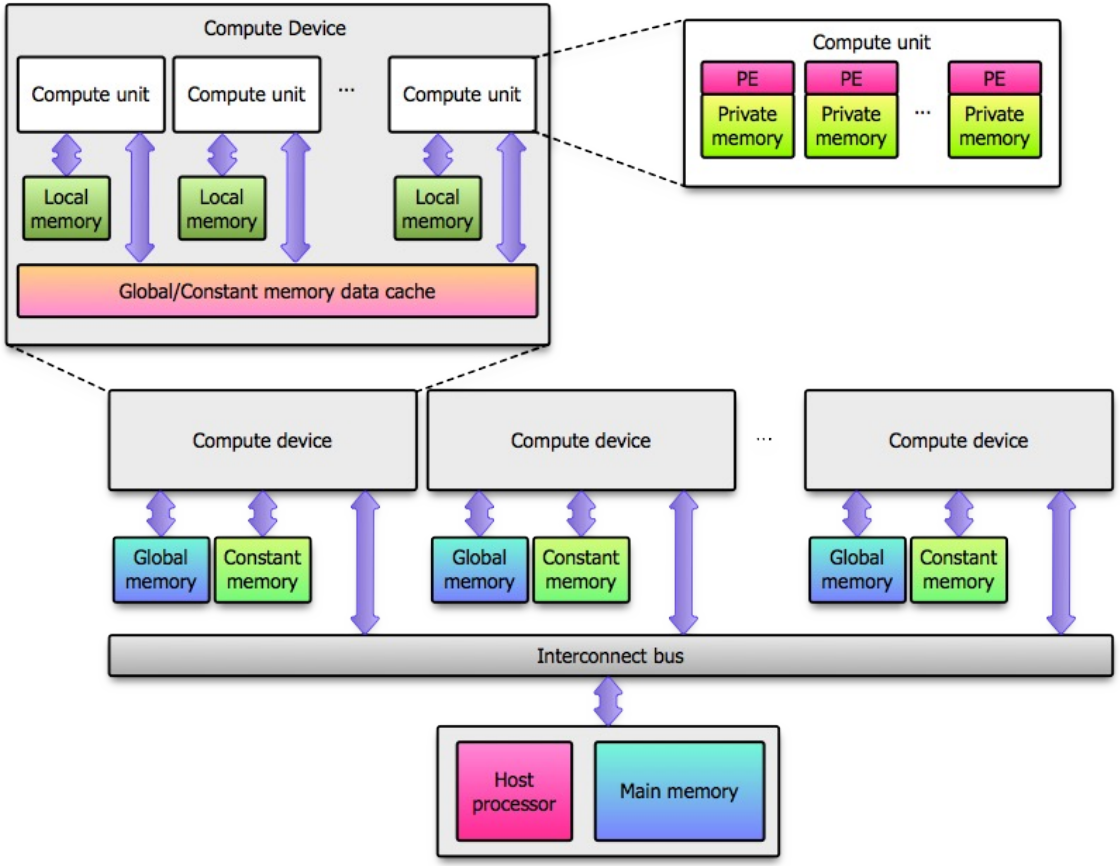


OpenCL Framework

- A software system that contains the set of components to support OpenCL application development and execution
  - To use a host and one or more compute devices as a single system

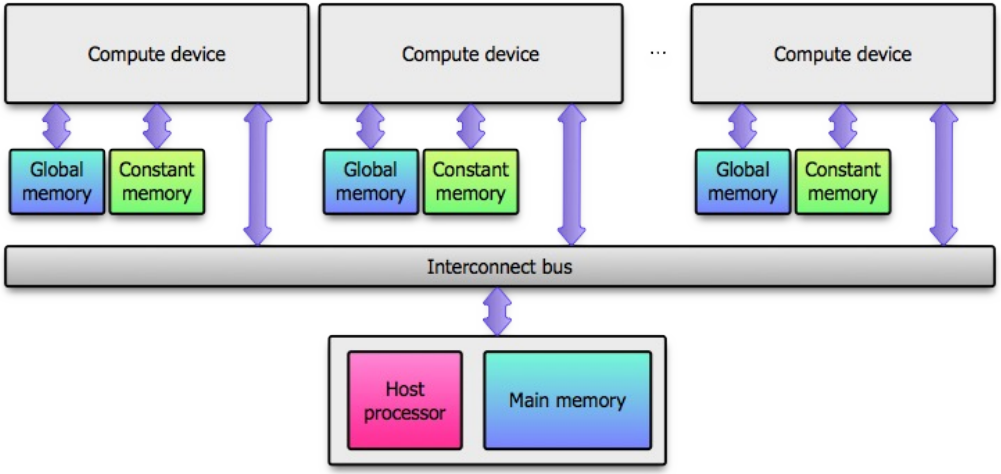
- OpenCL platform layer
- OpenCL runtime
- OpenCL compiler (kernel compilation)

OpenCL Platform Model



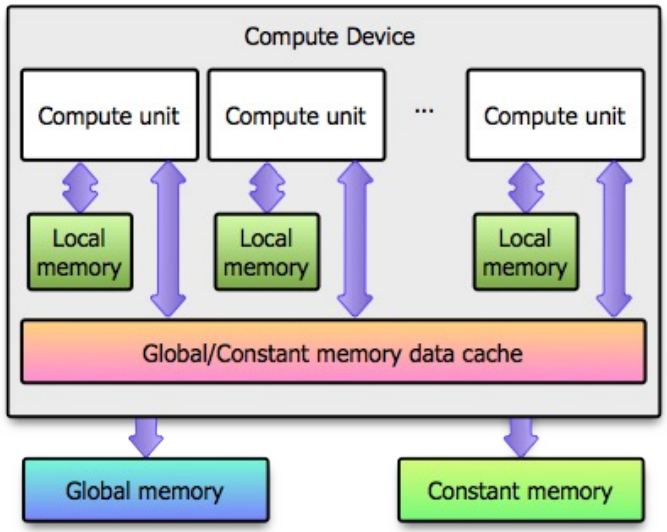
OpenCL Platform Model (cont'd)

- One host + one or more compute devices
- Compute devices may not access main memory
- Constant memory is read-only for compute devices
- The host can access global memory and constant memory (read/write)



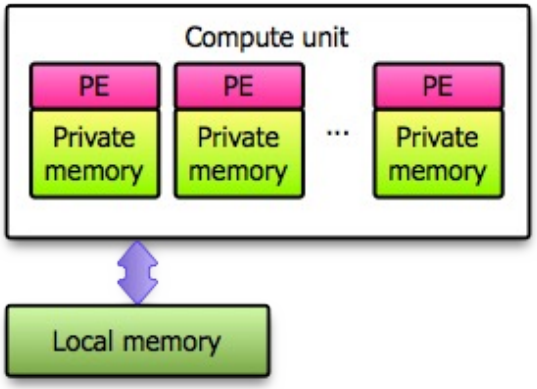
Compute Devices

- A collection of one or more Compute Units (CUs)
- Local memory is local to a CU
- The host may not access local memory



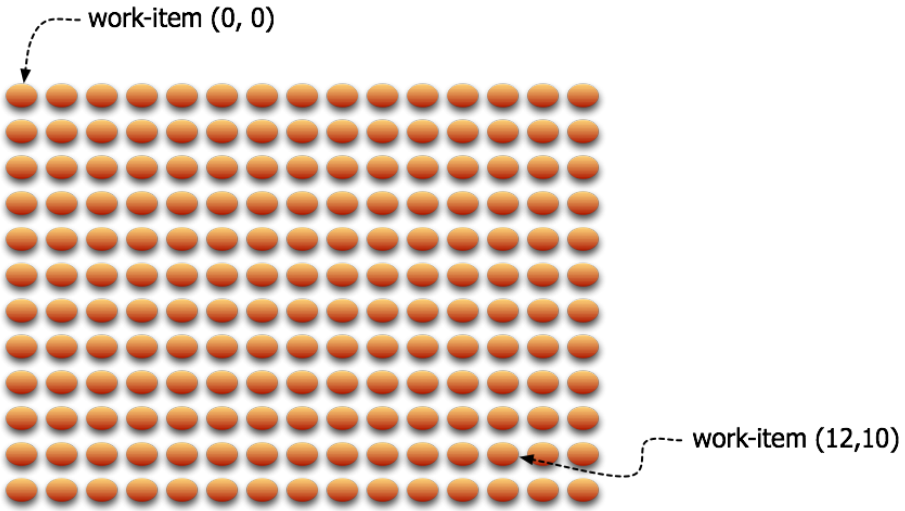
Compute Units

- An SM in a GPU
- A collection of one or more Processing Elements (PEs)
  - PEs are ALUs in an SM
- PEs execute code as SIMD units or SPMD units
- Each PE has its own private memory (e.g., context in an SM)
- Local memory (e.g., shared context in an SM) is shared between PEs



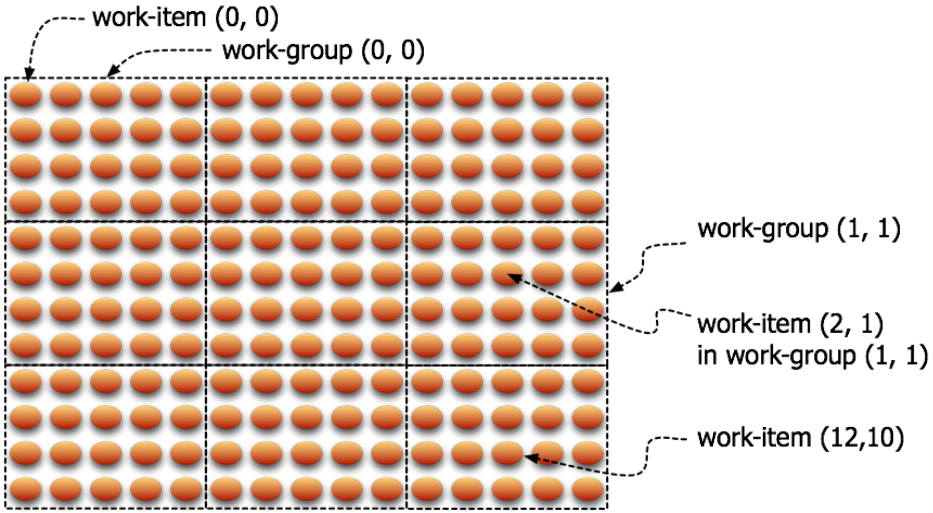
Kernel Index Space

- N-dimensional index space (N = 1, 2, or 3)
- Defines the total number of work-items (a kernel instance) that execute in parallel
  - SPMD
- A work-item executes for each point in the space



Work-groups

- Work-items are organized into work-groups
- A work-group executes on a compute unit
  - For GPUs, a set of work-groups are context switched on an SM
- Choose the dimension and size that are best for your kernel



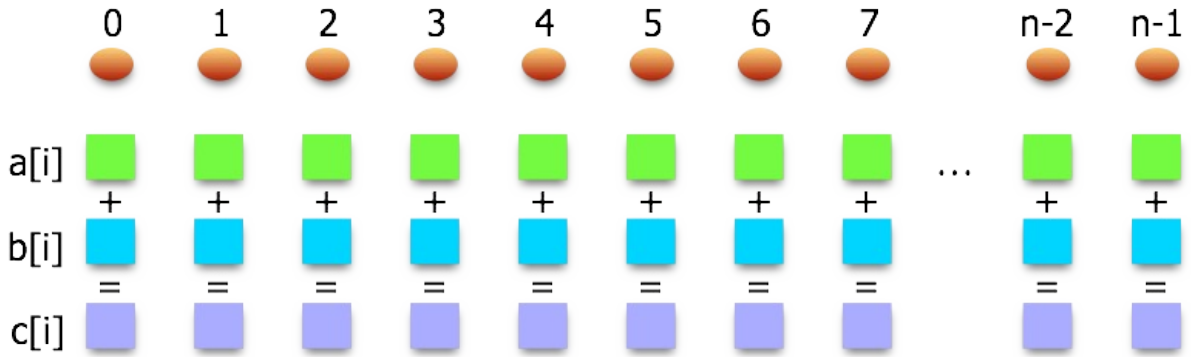
Kernel Example

```
void vec_add(int n,
             const float *a,
             const float *b,
             float *c)
{
    int i;
    for (i = 0; i < n; i++)
        c[i] = a[i] + b[i];
}
```

```
__kernel void vec_add( __global const float *a,
                       __global const float *b,
                       __global float *c)
{
    int id = get_global_id(0);

    c[id] = a[id] + b[id];
}
```

Data Parallel Programming Model

- A set of instructions from the kernel are applied concurrently to each point in the kernel index space
  - SPMD
- The index space defines how the data maps onto the work-items in the index space



Vector Addition Example

```
main(){

float srcA[N], srcB[N], srcC[N];

// initialize srcA and srcB
...
vec_add(m, srcA, srcB, srcC);
...
}
```

```
void vec_add(int n,
             const float *A,
             const float *B,
             float *C)
{
    int i;
    for (i=0; i<n; i++)
        c[i] = a[i] + b[i];
}
```

```
__kernel void vec_add( __global const float *A,
                       __global const float *B,
                       __global float *C)
{
    int id = get_global_id(0);

    C[id] = A[id] + B[id];
}
```

Host Program

```
//////////////////
// Host program //
//////////////////

#include <CL/cl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE  1024

// Kernel source code
const char* kernel_src =
                  "__kernel void vec_add(__global const float* A, "
                  "__global const float* B, "
                  "__global float* C) {"
                  "  int id = get_global_id(0);"
                  "  C[id] = A[id] + B[id];"
                  "}";
```

Initializing Vectors

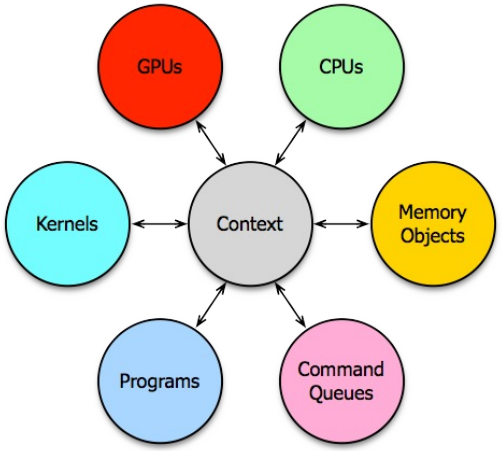
```
int main(int argc, char** argv) {

  float*  hostA;
  float*  hostB;
  float*  hostC;

  size_t  sizeA, sizeB, sizeC;

  sizeA = SIZE * sizeof(float);
  sizeB = SIZE * sizeof(float);
  sizeC = SIZE * sizeof(float);

  hostA = (float*) malloc(sizeA);
  hostB = (float*) malloc(sizeB);
  hostC = (float*) malloc(sizeC);

  for (int i = 0; i < SIZE; i++) {
    hostA[i] = (float) i;
    hostB[i] = (float) i * 2;
  }
```

Obtaining OpenCL Platforms and Devices

```
  cl_platform_id    platform;

// Obtain a list of available OpenCL platforms
  clGetPlatformIDs(1, &platform, NULL);

  cl_device_id      device;

// Obtain the list of available devices on the OpenCL platform
  clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
```

Creating an OpenCL Context

- OpenCL context
  - The environment in which the kernels execute
  - The domain in which synchronization and memory management is defined




```
  cl_context        context;

// Create an OpenCL context on a GPU device

  context = clCreateContext(0, 1, &device, NULL, NULL, NULL);
```

Command-queues and Commands

- Command-queues
  - Contain commands that will be executed on a specific compute device
    - Attached to the specific compute device by the host program
    - Commands are issued in-order or out-of-order

- Commands
  - OpenCL operations that are submitted to command-queues for execution
    - Kernel execution commands
    - Memory commands
    - Synchronization commands

Memory Objects

- Buffer objects
  - One dimensional data (array)
    - Scalar type, vector type, user defined type
  - Accessed by a pointer in the kernel

- Image objects
  - Two- or three-dimensional data
    - Textures, frame buffers, images
  - Cannot be accessed by a pointer
    - Use built-in functions

Creating Command-queues

```
  cl_command_queue  command_queue;

// Create a command queue and attach it to the compute device
//  (in-order queue)
  command_queue = clCreateCommandQueue(context, device, 0, NULL);
```

Allocating Memory Objects

- Allocate OpenCL memory objects to store the data accessed by the kernel
- Two types of memory objects
  - Buffer objects and image objects

```
  cl_mem              bufferA;
  cl_mem              bufferB;
  cl_mem              bufferC;

// Allocate buffer memory objects
  bufferA = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeA,
                           NULL, NULL);
  bufferB = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeB,
                           NULL, NULL);
  bufferC = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeC,
                           NULL, NULL);
```

Compiling and Building the OpenCL Program

- On-line or off-line (binary image)
- Compile and link the kernel code

```
  cl_program        program;

  size_t kernel_src_len = strlen(kernel_src);

// Create an OpenCL program object for the context
//   and load the kernel source into the program object
  program = clCreateProgramWithSource(context, 1,
                                      (const char**) &kernel_src,
                                      &kernel_src_len, NULL);

// Build (compile and link) the program executable
//   from the source or binary for the device
  clBuildProgram(program, 1, &device, NULL, NULL, NULL);
```

Building OpenCL Programs

- OpenCL version 1.0 and 1.1
  - Using clBuildProgram()
  - Preprocessing, compilation and linking all together
- OpenCL version 1.2
  - Using clCompileProgram() and clLinkProgram()
  - Separation of compilation and linking
  - Modular software development

Creating Kernel Objects

```
  cl_kernel         kernel;

// Create a kernel object from the program
  kernel = clCreateKernel(program, "vec_add", NULL);
```

Setting the Arguments of the Kernel

- Arguments to the kernel are referred by indices that go from 0 for the leftmost argument to n-1, where n is the total number of arguments declared by a kernel

```
__kernel void vec_add( __global const float *A,
                       __global const float *B,
                       __global float *C)
{
    int id = get_global_id(0);

    C[id] = A[id] * B[id];
}
```

```
// Set the arguments of the kernel
  clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*) &bufferA);
  clSetKernelArg(kernel, 1, sizeof(cl_mem), (void*) &bufferB);
  clSetKernelArg(kernel, 2, sizeof(cl_mem), (void*) &bufferC);
```

Preparing the Input Data

```
// Copy the input vectors to the corresponding buffers
  clEnqueueWriteBuffer(command_queue, bufferA, CL_FALSE,
                       0, sizeA, hostA, 0, NULL, NULL);
  clEnqueueWriteBuffer(command_queue, bufferB, CL_FALSE,
                       0, sizeB, hostB, 0, NULL, NULL);
```

Launching the Kernel

- Set the kernel index space
- Enqueue the kernel command
- Kernel execution is asynchronous to the host

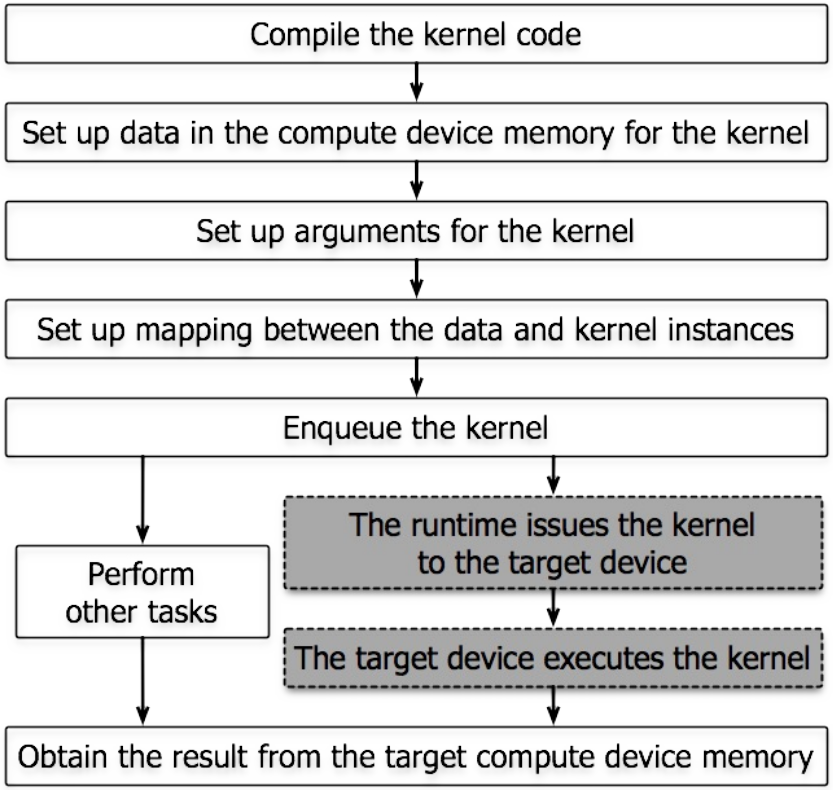
```
// The kernel index space is one dimensional
// Specify the number of total work-items in the index space
  size_t global[1] = { SIZE };

// Specify the number of total work-items in a work-group
  size_t local[1] = { 16 };

// Execute the kernel
  clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
                         global, local, 0, NULL, NULL);
```

Obtaining the Result from the Device

- **`clFinish(command_queue,...)`** does not return until all previously queued commands in command_queue have been processed and completed

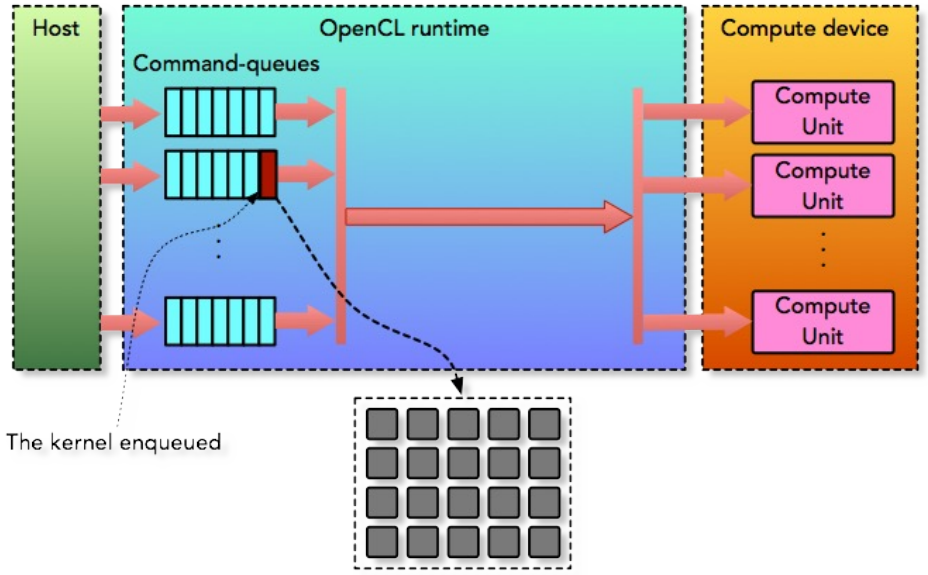
```
// Wait until the kernel command completes
//(no need to wait because the command_queue is an in-order queue)
// clFinish(command_queue)

// Copy the result from bufferC to hostC
  clEnqueueReadBuffer(command_queue, bufferC, CL_TRUE, 0, sizeC,
                      hostC, 0, NULL, NULL);

// Print the result
  for (int i = 0; i < SIZE; i++) {
    printf("C[%d] = %f\n", i, hostC[i]);
  }

  return 0;
}
```

What the Host does for Kernel Execution



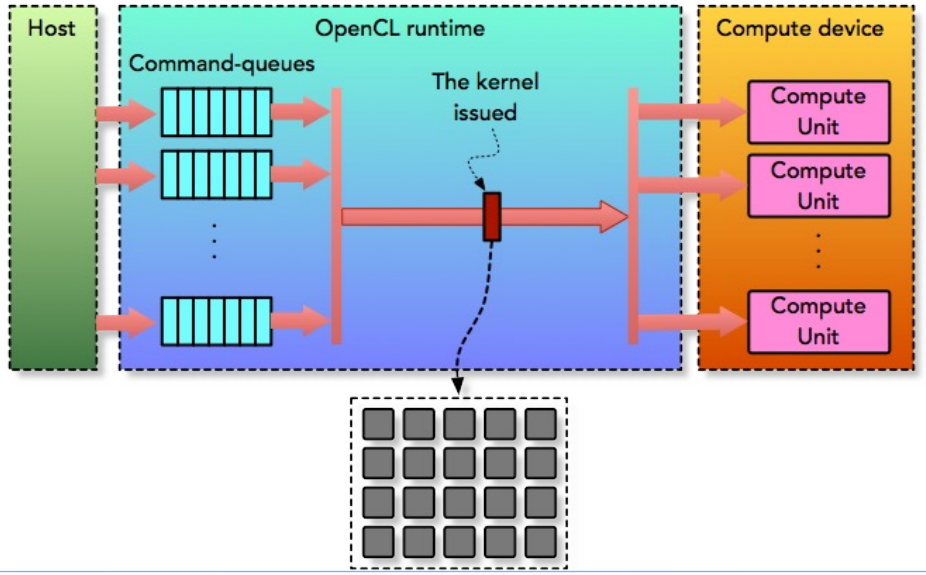
OpenCL Runtime

- The host program inserts the kernel command to a command-queue



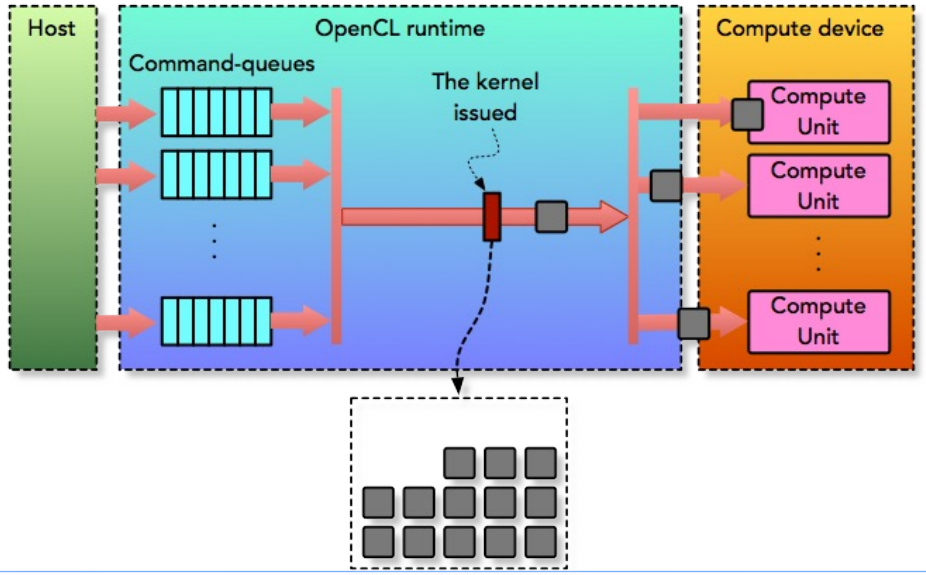
OpenCL Runtime (cont'd)

- The runtime issues the command to the target device
  - In-order or out-of-order (depending on the queue type)
  - After resolving dependences between all enqueued commands



OpenCL Runtime (cont'd)

- The runtime distributes the kernel workload to CUs in the target device
  - The granularity of the distribution: a work-group



Work-group Barriers

- Synchronization only between work-items within a single work-group
  - The barrier must be encountered by all work-items of the work-group executing the kernel or by none at all

- No synchronization mechanism between different work-groups

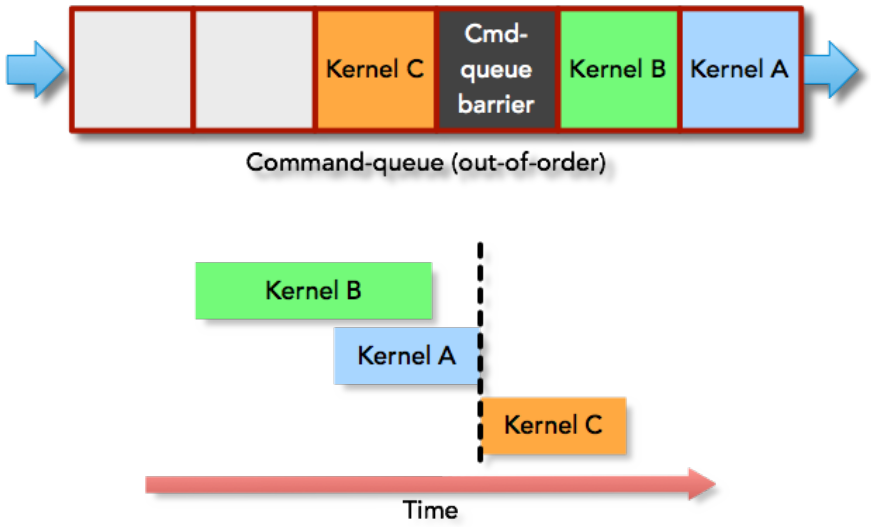
Work-group Barriers (cont'd)

- `void barrier(cl_mem_fence_flags flags)`
  - `flags`
    - **`CLK_LOCAL_MEM_FENCE`** guarantees completion of read/write accesses to the local memory
    - **`CLK_GLOBAL_MEM_FENCE`** guarantees completion of read/write accesses to the global memory

```
__kernel foo( __global float *a, __local float *b )
{
    int id = get_global_id(0);

    if (id == 0) {
        barrier(CLK_GLOBAL_MEM_FENCE);
    } else {
        b[id] = a[id];
    }
    // error
}
```

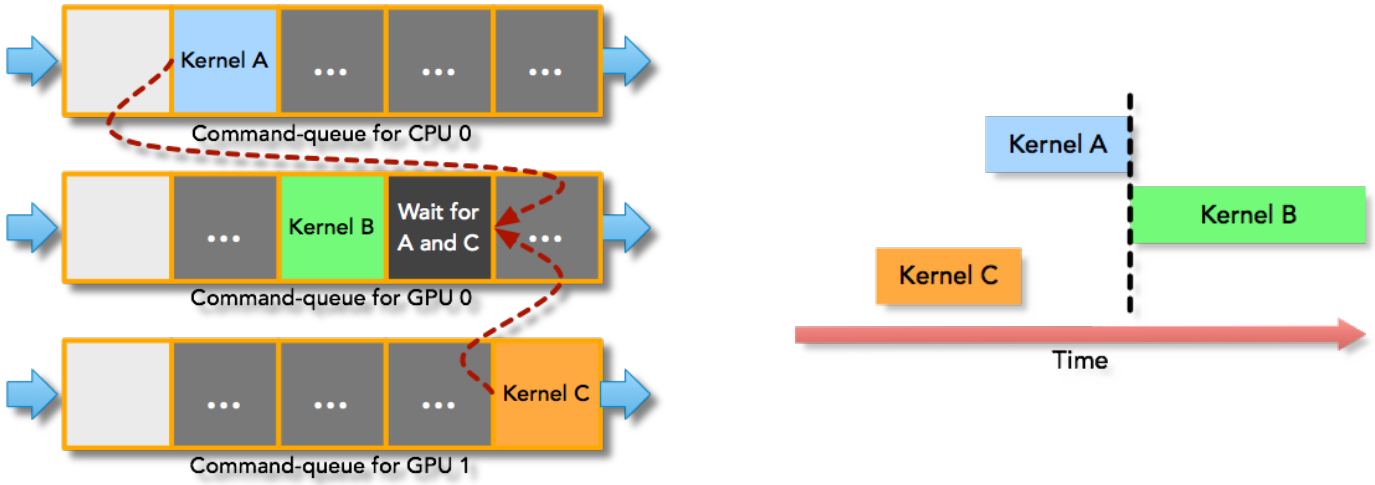
Synchronization Between Commands

- Command-queue barrier
  - Between commands in a single command-queue



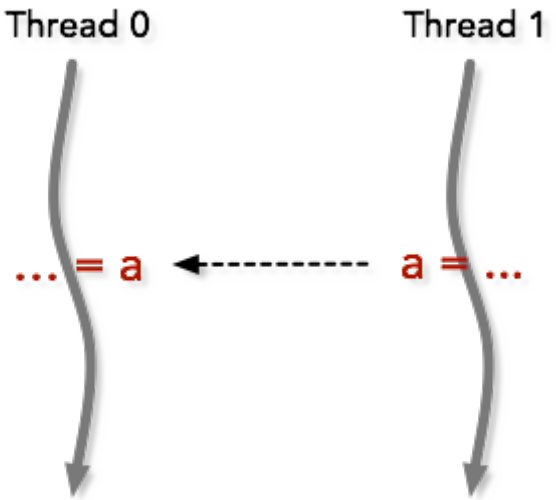
Synchronization Between Commands (cont'd)

- All OpenCL API functions that enqueue commands return an event object
- Event synchronization
  - Between commands in different (possibly the same) command-



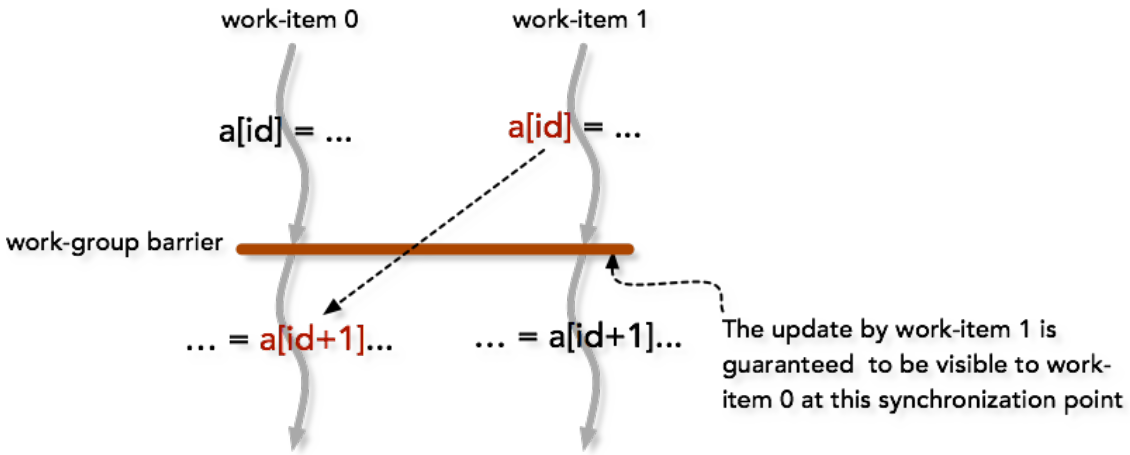
Memory Consistency Model

- When is the updates of memory by a thread visible to another thread?
- Memory access ordering



Memory Consistency Model (cont'd)

- A relaxed memory consistency model
  - Memory is not consistent at all times
- Local/global memory is consistent across work-items in a single work-group at a work-group barrier
  - Not between different work-groups



Memory Consistency Model (cont'd)

- Consistency for memory objects is enforced at a synchronization point
  - clFinish
  - Work-group barriers
  - Command-queue barriers
  - Event synchronization

Vector Type

- Format: **`type`*`n`***
  - **`type`**: char, uchar, short, ushort, int, uint, long, ulong, float, double
  - ***`n`***: 2, 3, 4, 8, 16

OpenCL C Vector Type	Host API Type
char***n***	cl_char***n***
uchar***n***	cl_uchar***n***
short***n***	cl_short***n***
ushort***n***	cl_ushort***n***
int***n***	cl_int***n***
uint***n***	cl_uint***n***
long***n***	cl_long***n***
ulong***n***	cl_ulong***n***
float***n***	cl_float***n***
double***n***	cl_double***n***

Vector Literals

- To initialize variables of vector type
- Format - (type)(initializer)
  - type - vector type
  - initializer - m expressions, where m is the number of elements in the vector type
  - Format examples
    - (float4)(float, float, float, float)
    - (float4)(float2, float, float)
    - (float4)(float, float2, float)
    - (float4)(float, float, float2)
    - (float4)(float2, float2)
    - (float4)(float3, float)
    - (float4)(float, float3)
    - (float4)(float)

Vector Literals (cont'd)

- float4 f1 = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
- float4 f2 = (float4)((float2)(1.0f, 2.0f), (float2)(3.0f, 4.0f));
- float4 f3 = (float4)(1.0f, (float2)(2.0f, 3.0f), 4.0f);
- float4 f4 = (float4)((float3)(1.0f, 2.0f, 3.0f), 4.0f);
- float4 f5 = (float4)(1.0f, 2.0f); // ERROR
- uint4 u1 = (uint4)(1);
- uint4 u2 = (uint4)(1, 1, 1, 1);

Vector Components

- Access vector components with letter indices
  - <vector2>.xy
  - <vector3>.xyz
  - <vector4>.xyzw

```
float2 p;
p.x = 1.0f;
p.z = 1.0f;                    // ERROR

float4 a, b, c;
a.xyzw = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
b = a.wzyx;                    // b = (4.0f, 3.0f, 2.0f, 1.0f)
c = a.xxyy;                    // c = (1.0f, 1.0f, 2.0f, 2.0f)

a.xw = (float2)(5.0f, 6.0f);  // a = (5.0f, 2.0f, 3.0f, 6.0f)
b.xx = (float2)(3.0f, 4.0f);              // ERROR
c.xy = (float4)(1.0f, 2.0f, 3.0f, 4.0f);  // ERROR
```

Vector Components (cont'd)

- Access vector components with numeric indices
- All numeric indices must be preceded by the letter s or S

vector*n*	Indices
vector2	0, 1
vector3	0, 1, 2
vector4	0, 1, 2, 3
vector8	0, 1, 2, 3, 4, 5, 6, 7
vector16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F

```
float4 f1, f2;
f1.xyzw = f2.s0123;
f1 = f2.x12w;          // ERROR

float16 f3 = (float16)(0.0f);
f3.sA = 10.0f;
f3.S0f1 = (float3)(0.0f, 15.0f, 1.0f);
```

Vector Components (cont'd)

- Access vector components with suffixes

Suffix	Elements
.lo	the lower half of a given vector
.hi	the upper half of a given vector
.odd	the elements of a given vector at odd positions
.even	the elements of a given vector at even positions

```
float4 vf = (4.0f, 3.0f, 2.0f, 1.0f);

float2 low = vf.lo;     // vf.xy
float2 high = vf.hi;    // vf.zw
float x = low.low;      // low.x
float y = low.hi;       // low.y

float2 odd = vf.odd;    // vf.yw
float2 even = vf.even;  // vf.xz
```

Vector Operation

```
float4 a, b, c;

c = a + b;
```

=

```
float4 a, b, c;

c.x = a.x + b.x;
c.y = a.y + b.y;
c.z = a.z + b.z;
c.w = a.w + b.w;
```

OpenCL C Built-in Functions

- Work-item functions
- Math functions
- Integer functions
- Geometric functions
- Relational functions
- Vector data load/store functions
- Synchronization and memory fence functions
- Atomic functions
- printf
- Image related functions

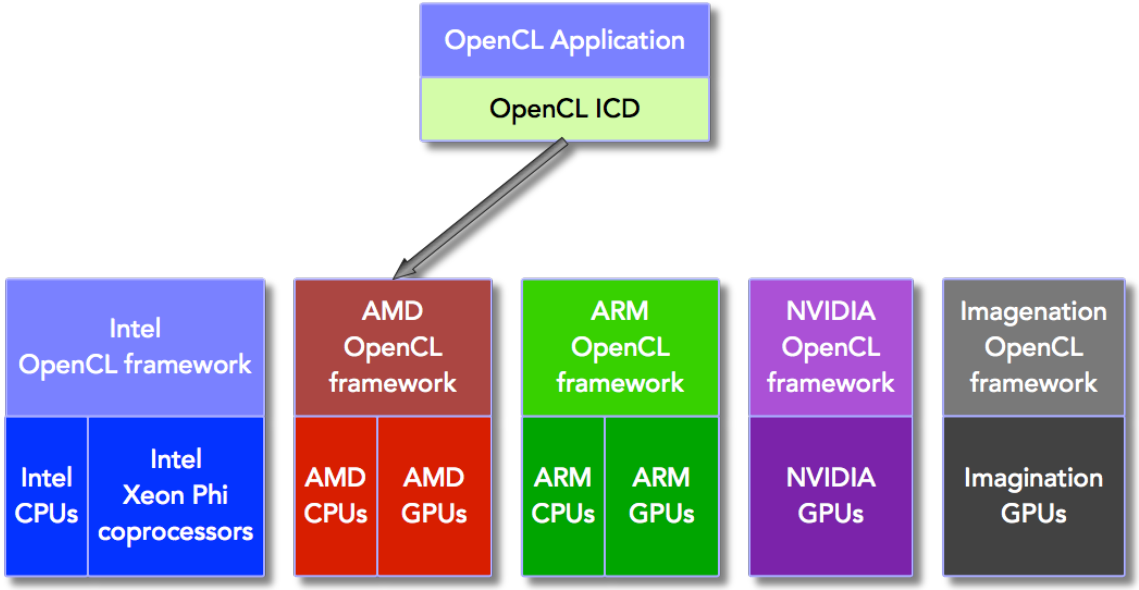
Limitations

- Current OpenCL implementations are targeting parallelism for multiple compute devices under a single operating system instance

- An application for a heterogeneous cluster
  - MPI + OpenCL or MPI + CUDA
  - Complicated, less portable, and hard to maintain

OpenCL ICD

- The OpenCL ICD enables multiple OpenCL implementations to coexist under the same system (an operating system instance)
- The user explicitly specifies which framework to use



Limitations of the OpenCL ICD

- Users need to explicitly specify which framework is used in their applications

- Cannot share objects (buffers, events, etc.) across different frameworks in the same application