OpenCL - Runtime System

Soumyajit Dey, Assistant Professor,
CSE, IIT Kharagpur

March 17, 2020

Course Organization

Topic	Week	Hours
Review of basic COA w.r.t. performance	1	2
Intro to GPU architectures	2	3
Intro to CUDA programming	3	2
Multi-dimensional data and synchronization	4	2
Warp Scheduling and Divergence	5	2
Memory Access Coalescing	6	2
Optimizing Reduction Kernels	7	3
Kernel Fusion, Thread and Block Coarsening	8	3
OpenCL - runtime system	9	3
OpenCL - heterogeneous computing	10	2
Efficient Neural Network Training/Inferencing	11-12	6

Recap

- GPU architecture
- GPU programming using CUDA
- GPU optimization techniques

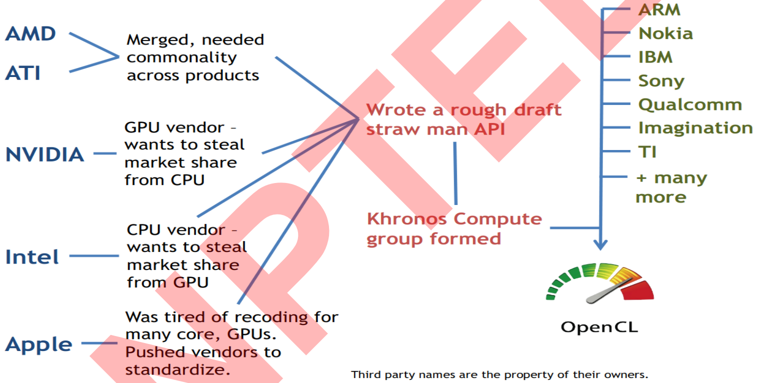
GPGPU Timeline

- 1999-2000 computer scientists from various fields started using GPUs to accelerate a range of scientific applications.GPU programming required the use of graphics APIs such as OpenGL and Cg (C for Graphics).
- 2002 James Fung (University of Toronto) developed OpenVIDIA.
- In November 2006 Nvidia launched CUDA, an API that allows to code algorithms for execution on Geforce GPUs using C programming language.
- Khronos Group defined OpenCL in 2008 supported on AMD, Nvidia and ARM platforms.

OpenCL - Open Computing Language

- Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing across CPUs, GPUs, and other processors like DSP
- A framework for parallel programming includes a language, API, libraries and a runtime system.
- Language support for C, C++, Python, Java

Origin of OpenCL



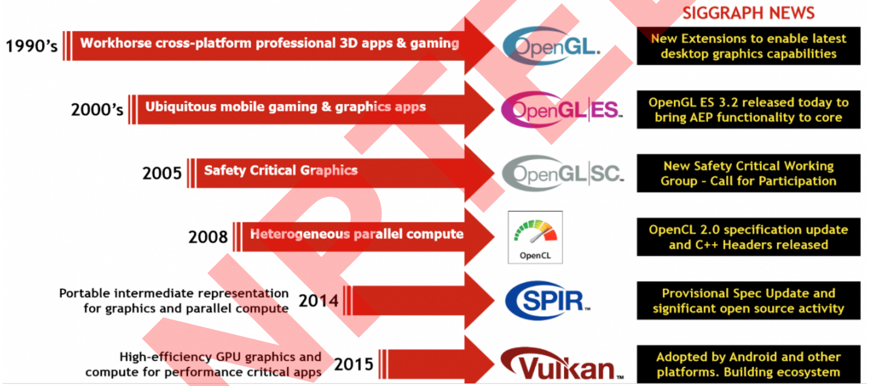
AMD
ATI
Merged, needed commonality across products
NVIDIA
GPU vendor - wants to steal market share from CPU
Intel
CPU vendor - wants to steal market share from GPU
Apple
Was tired of recoding for many core, GPUs. Pushed vendors to standardize.
Wrote a rough draft straw man API
Khronos Compute group formed
ARM
Nokia
IBM
Sony
Qualcomm
Imagination
TI
+ many more
OpenCL
Third party names are the property of their owners.

From "OpenCL: A Hands-on Introduction" by Tim Mattson and Alice Koniges

OpenCL Working Group

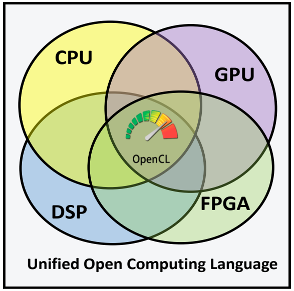
- Diverse industry
  - Processor vendors, system OEM, middleware vendors, application developers
- OpenCL became an important standard upon release by virtue of the market coverage of the companies behind it.

Khronos Group Timeline


SIGGRAPH NEWS
1990's
Workhorse cross-platform professional 3D apps & gaming
OpenGL.
New Extensions to enable latest desktop graphics capabilities
2000's
Ubiquitous mobile gaming & graphics apps
OpenGL ES
OpenGL ES 3.2 released today to bring AEP functionality to core
2005
Safety Critical Graphics
OpenGL SC
New Safety Critical Working Group - Call for Participation
2008
Heterogeneous parallel compute
OpenCL
OpenCL 2.0 specification update and C++ Headers released
Portable intermediate representation for graphics and parallel compute
2014
SPIR
Provisional Spec Update and significant open source activity
High-efficiency GPU graphics and compute for performance critical apps
2015
Vulkan
Adopted by Android and other platforms. Building ecosystem

OpenCL: Portable Heterogeneous Computing

- Open-source, cross-platform, cross-vendor standard that target a wide range of systems like supercomputers, embedded systems, mobile devices
- Enables programming of diverse compute devices like CPU, GPU, DSP, FPGA
- Same code can be executed on all these devices
- Can dynamically interrogate system load and balance across available heterogeneous processors



OpenCL Language Specification

- Subset of ISO C99 with language extensions
- Well defined numerical accuracy (IEEE 754 rounding with specified max error)
- Rich set of built-in functions: cross, dot, sin, cos, pow, log . . .
- Can be compiled JIT/online or offline
- Execute compute kernels across multiple devices

CUDA and OpenCL Correspondence

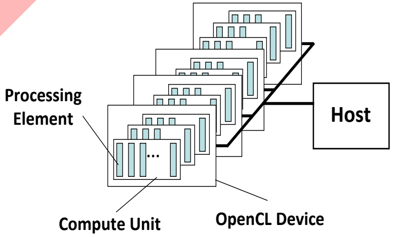
CUDA	**OpenCL**
GPU	device
multiprocessor	compute unit
scalar core	processing element
thread	work-item
thread-block	workgroup
grid	NDRange
global memory	global memory
shared memory	local memory
local memory	private memory

CUDA and OpenCL Correspondence

CUDA	OpenCL
__global__ function	__kernel function
__device__ function	no qualification needed
__constant__ variable	__constant variable
__device__ variable	__global variable
__shared__ variable	__local variable

OpenCL Platform Model

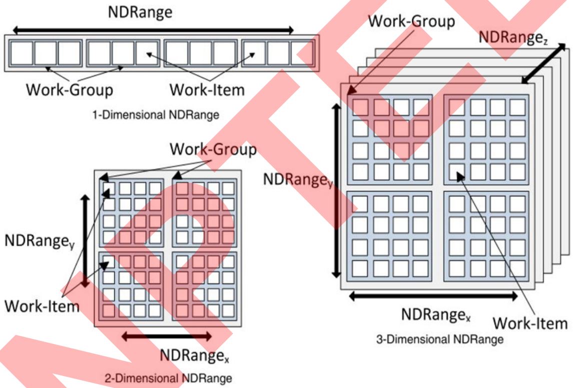
- A host is connected to one or more OpenCL devices
- OpenCL device is collection of one or more compute units
- A compute unit is composed of one or more processing elements
- Processing elements execute code as SIMD or SPMD



OpenCL Execution Model

- The two main execution units in OpenCL are the kernels and the host program.
- When a kernel is submitted for execution by the host, an index space is defined
- An instance of the kernel called a work-item executes for each point in this index space.
- Work-items are organized into work-groups
- An NDRange is an N-dimensional index space, where N is one, two or three

OpenCL Execution Model



(Figure from reference[1])

OpenCL Execution Model: Context

A context is an abstract container that exists on the host
The context includes the following resources:

- **Devices:** The collection of OpenCL devices to be used by the host
- **Kernels:** The OpenCL functions that run on OpenCL devices
- **Program Objects:** The program source and executables that implement the kernels
- **Memory Objects:** A set of memory objects visible to the host and the OpenCL devices.

OpenCL Execution Model: Context

- The context is created and manipulated by the host using functions from the OpenCL API
- It coordinates the mechanisms for host–device interaction
- It manages the memory objects that are available to the devices
- It keeps track of the programs and kernels that are created for each device
- Limiting the context to a given platform helps to fully utilize a system comprising resources from a mixture of vendors (ICD:"Installable Client Driver")

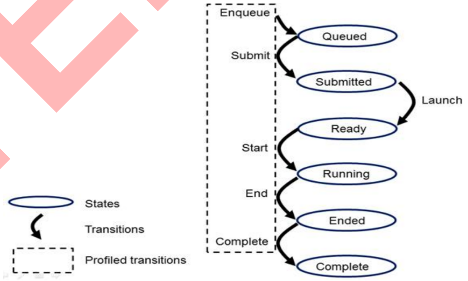
OpenCL Execution Model: Command Queue

- Host creates a data structure called command-queue to co-ordinate execution of the kernels on the devices.
- Host places commands into the command-queue which are then scheduled onto the devices within the context.
    - Kernel execution commands
    - Memory commands
    - Synchronization commands
- These execute asynchronously between the host and the device
    - In-order Execution (default)
    - Out-of-order Execution

OpenCL Execution Model: Command State

Regardless of whether the command-queue resides on the host or a device, each command passes through six states.

- Queued
- Submitted
- Ready
- Running
- Ended
- Complete



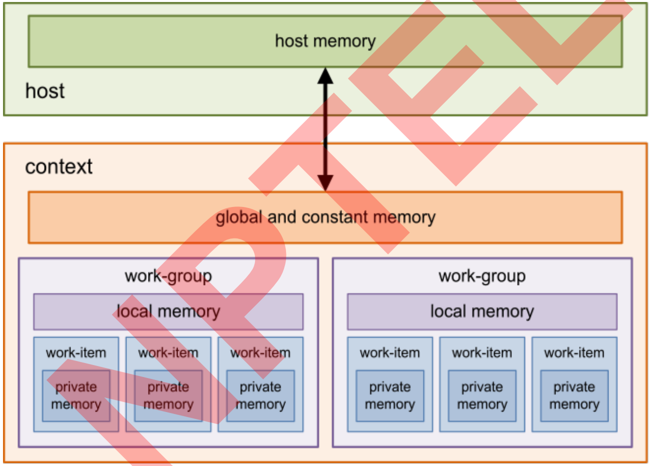
(Figure from reference[1])

OpenCL Execution Model: Synchronization

Synchronization refers to mechanisms that constrain the order of execution between two or more units of execution.

- **Work-group synchronization:** Constraints on the order of execution for work-items in a single work-group
- **Command synchronization:** Constraints on the order of commands launched for execution

OpenCL Memory Model



Memory Objects

The contents of global memory are memory objects. Use the OpenCL type cl_mem.

- **Buffer:** Stored as a block of contiguous memory and used as a general purpose object to hold built in types (such as int, float), vector types, or user-defined data used in an OpenCL program. Can be manipulated through pointers much as one would with any block of memory in C.
- **Image:** Holds one, two or three dimensional images. Formats are based on the standard image formats used in graphics applications. Must be managed by functions defined in the OpenCL API.

OpenCL Memory Model

	Global	Constant	Local	Private
Host	Dynamic Allocation	Dynamic Allocation	Dynamic Allocation	No Allocation
	Read/Write access to buffers and images	Read/Write access	No access	No access
Kernel	Static Allocation for program scope variables	Static Allocation	Static Allocation.	Static Allocation.
	Read/Write access	Read-only access	Read/Write access	Read/Write access

OpenCL Programming Model

- **Data Parallel Model:** Defines a computation in terms of a sequence of instructions applied to multiple elements of a memory object
- **Task Parallel Model:** Defines a model in which a single instance of a kernel is executed independent of any index space.
- Hybrids of these two models

OpenCL Data Parallel Programming Model

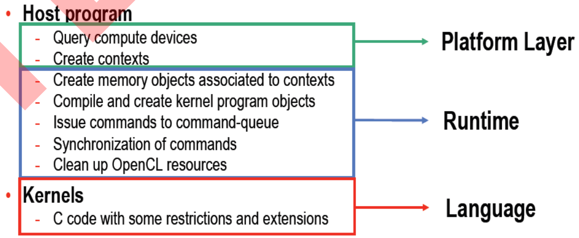
- OpenCL provides a hierarchical data parallel programming model
- In **explicit model** programmer defines the total number of work-items to execute in parallel and how the work-items are divided among work-groups.
- In **implicit model** programmer specifies only the total number of work-items to execute in parallel and the division into work-groups is managed by the OpenCL implementation.

OpenCL Task Parallel Programming Model

- Logically equivalent to executing a kernel on a compute unit with a work-group containing a single work-item
- Users express parallelism by -
  - Using vector data types implemented by the device
  - Enqueuing multiple tasks

OpenCL Program Structure

- Platform Layer API
  - Abstraction layer for diverse computational resources
  - Query, select and initialize compute devices
  - Create compute contexts and work-queues
- Runtime API
  - Launch compute kernel
  - Set kernel execution configuration
  - Manage scheduling, compute, and memory resources
  - Synchronization



Host program
- Query compute devices
- Create contexts
- Create memory objects associated to contexts
- Compile and create kernel program objects
- Issue commands to command-queue
- Synchronization of commands
- Clean up OpenCL resources
Kernels
- C code with some restrictions and extensions
Platform Layer
Runtime
Language

OpenCL and CUDA kernel Code

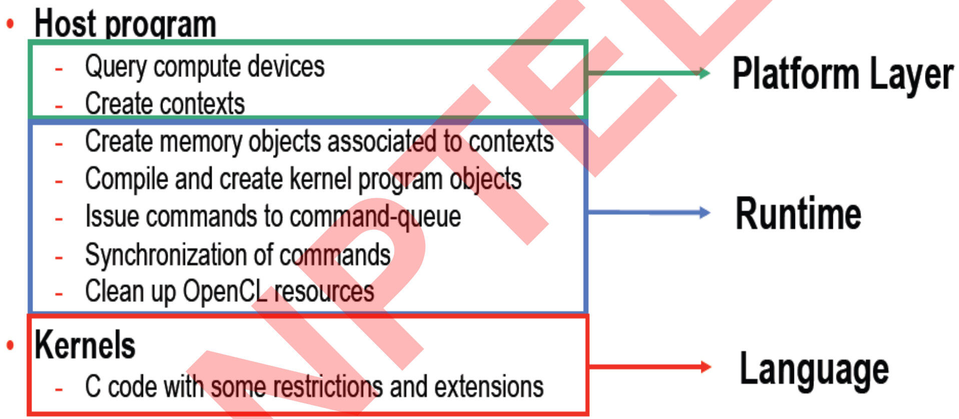
OpenCL

```
__kernel void vadd(__global float* a,
    __global float* b, __global float*
    c, const unsigned int n)
{
  int i = get_global_id(0);
  if(i < n)
    c[i] = a[i] + b[i];
}
```

CUDA

```
__global__ void vectorAdd(float* a,
    float* b, float* c, unsigned int n)
{
  int i= threadIdx.x + blockDim.x *
      blockIdx.x;
  if(i < n)
    c[i] = a[i] + b[i];
}
```

OpenCL Host Code Overview



- Host program
  - Query compute devices
  - Create contexts
  - Create memory objects associated to contexts
  - Compile and create kernel program objects
  - Issue commands to command-queue
  - Synchronization of commands
  - Clean up OpenCL resources
- Kernels
  - C code with some restrictions and extensions

(Figure from reference[2])

OpenCL Host Code

Initialisation and declaration

```
#include <stdio.h>
#include <stdlib.h>
#include <CL/cl.h>
#define LENGTH (1024) // length of vectors a, b, and c

int main(int argc, char** argv)
{
  size_t dataSize = sizeof(float) * LENGTH;
  float* h_a = (float *)malloc(dataSize);       // a vector
  float* h_b = (float *)malloc(dataSize);       // b vector
  float* h_c = (float *)malloc(dataSize);       // c vector (result)
  cl_int  err; // error code returned from OpenCL call

  // Fill vectors a and b with random float values
  int i = 0;
  for(int i = 0; i < LENGTH; i++){
    h_a[i] = rand() / (float)RAND_MAX;
    h_b[i] = rand() / (float)RAND_MAX;
  }
```

OpenCL Host Code Cont.

Set up platform:

```
// Set up platform
cl_uint numPlatforms;

/*cl_int clGetPlatformIDs(cl_uint numOfEntries,cl_platform_id *platforms,
    cl_uint *numOfPlatforms)*/

err = clGetPlatformIDs(0, NULL, &numPlatforms); // Find number of platforms
if (numPlatforms == 0)
{
  printf("Found 0 platforms!\n");
  return -1;
}
cl_platform_id Platform[numPlatforms];
err = clGetPlatformIDs(numPlatforms, Platform, NULL);// Get all platforms
```

OpenCL Host Code Cont.

Find all available devices:

```
/*cl_int clGetDeviceIDs(cl_platform_id platform,cl_device_type deviceType,
    cl_uint numOfEntries, cl_device_id *devices,cl_uint *numOfDevices)*/

 cl_device_id all_devices[numPlatforms][2][100]= { 0 }; //For each platform,
     there can be two device types and each type can have multiple devices
 for (i = 0; i < numPlatforms; i++)
 {
   cl_uint numDevices;
   err=clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_GPU, 0, NULL, &numDevices);
   err = clGetDeviceIDs(Platform[i], CL_DEVICE_TYPE_GPU,  numDevices,
       all_devices[numPlatforms][0], NULL);
   err=clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_CPU, 0, NULL, &numDevices);
   err = clGetDeviceIDs(Platform[i], CL_DEVICE_TYPE_CPU,  numDevices,
       all_devices[numPlatforms][1], NULL);
 }
 // For this example we have choosen the 1st GPU device in 1st platform
 cl_device_id device_id=all_devices[0][0][0];
```

Context

- The context is created and manipulated by the host using functions from the OpenCL API
- It coordinates the mechanisms for host–device interaction
- It manages the memory objects that are available to the devices
- It keeps track of the programs and kernels that are created for each device
- Limiting the context to a given platform helps to fully utilize a system comprising resources from a mixture of vendors (ICD:"Installable Client Driver")

Command Queues

- Creates a data structure called command-queue to co-ordinate execution of the kernels on the devices.
- The command-queue can be used to queue a set of operations (referred to as commands) in order.
- Before OpenCL 2.0, commands could be enqueued only from host. OpenCL 2.0 allows both host-side and device-side command-queue that allows a child kernel to be enqueued directly from another kernel executing on a device
- To create a host or device command-queue on a specific device, API used is clCreateCommandQueueWithProperties. (clCreateCommandQueue)
- We can set the property parameter of the Command-Queue in the clCreateCommandQueue command

Command Queues Properties

The Command-Queue Properties that can be set are-

- CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE : Determines whether the commands queued in the command-queue are executed in-order or out-of-order. .
- CL_QUEUE_PROFILING_ENABLE : Enable or disable profiling of commands in the command-queue.
- CL_QUEUE_ON_DEVICE - Indicates that this is a device queue. Only out-of-order device queues are supported.
- CL_QUEUE_ON_DEVICE_DEFAULT - Indicates that this is the default device queue and used with CL_QUEUE_ON_DEVICE. There can only be one default device queue per context.
- If not specified, an in-order host command queue is created for the specified device

OpenCL Host Code Cont.

Create context and command queue

```
/*Ideally for each platform(from different vendors), there will be one context
     and each device has one command-queue. Single device can have one or
    multiple command-queues but a command-queue can be associated with only
    one device(To be discussed later). For given example we have created only
    one context and command-queue for the chosen GPU device */

//cl_context clCreateContext(const cl_context_properties *properties,cl_uint
    num_devices,const cl_device_id *devices,void CL_CALLBACK *pfn_notify,void
    *user_data,cl_int *errcode_ret)
  cl_context context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);

//cl_command_queue clCreateCommandQueueWithProperties(cl_context context,
    cl_device_id device,cl_command_queue_properties properties,cl_int *
    errcode_ret)
  cl_command_queue commands = clCreateCommandQueueWithProperties(context,
      device_id, 0, &err);
```

Program Object

- An OpenCL program consists of a set of kernels that are identified as functions declared with the kernel qualifier in the program source.
- The program executable can be generated online or offline by the OpenCL compiler for the appropriate target device

Program Object

A program object encapsulates the following information:

- An associated context
- A program source or binary
- Successfully built program executable, library or compiled binary
- List of devices
- Build option used
- Number of kernel objects currently attached

Program Object

The OpenCL APIs used to create a program object for a context-

- **clCreateProgramWithSource:** Load source code into that object
- **clCreateProgramWithIL:** Load code in an intermediate language into that object
- **clCreateProgramWithBinary:** Load binary bits into that object
- **clCreateProgramWithBuiltInKernels:** Loads the information related to the built-in kernels into that object

Program Object

Other OpenCL APIs related to Program Object-

- **clBuildProgram:** Build (compile and link) a program executable for all the devices or a specific device(s) in the OpenCL context associated with the program
- **clCompileProgram:** Compile a program's source
- **clLinkProgram:** Link a set of compiled program objects and libraries and create a library or executable
- **clGetProgramInfo:** Return information about a program object
- **clGetProgramBuildInfo:** Return build information for each device in the program object

Kernel Object

- A kernel is a function declared in a program in OpenCL.
- A kernel is identified by the __kernel qualifier applied to any function in a program.
- A kernel object encapsulates the specific __kernel function declared in a program and the argument values to be used when executing this __kernel function.

Kernel Object

OpenCL APIs related to Kernel Object-

- **clCreateKernel:** Create a kernel object
- **clCreateKernelsInProgram:** Create kernel objects for all kernel functions in a program
- **clSetKernelArg:** Set the argument value for a specific argument of a kernel
- **clGetKernelInfo:** Return information about a kernel object
- **clGetKernelArgInfo:** Return information about the arguments of a kernel
- **clEnqueueNDRangeKernel:** Enqueue a command to execute a kernel on a device

OpenCL Host Code Cont.

Create and build program object

```
/* cl_program clCreateProgramWithSource(cl_context context,cl_uint count,const
      char **strings,const size_t *lengths,cl_int *errcode_ret) */
  cl_program program = clCreateProgramWithSource(context, 1, (const char **) &
        KernelSource, NULL, &err);

/*  cl_int clBuildProgram(cl_program program,cl_uint num_devices,const
    cl_device_id *device_list,const char *options,void (*pfn_notify)(
    cl_program, void *user_data),void *user_data) */
  err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

/*  cl_kernel clCreateKernel(cl_program  program,const char *kernel_name,
    cl_int *errcode_ret) */
  cl_kernel ko_vadd = clCreateKernel(program, "vadd", &err);
```

Buffer Object

- Stored as a block of contiguous memory and used as a general purpose object to hold built in types (such as int, float), vector types, or user-defined data used in an OpenCL program.
- Can be manipulated through pointers much as one would with any block of memory in C.
- A buffer object stores a one-dimensional collection of elements.
- Elements of a buffer object can be a scalar data type (such as an int, float), vector data type, or a user-defined structure.

Buffer Object

APIs used for buffer objects-

- **clCreateBuffer:** Creates a buffer object
- **clEnqueueReadBuffer:** Enqueue commands to read from a buffer object to host memory
- **clEnqueueWriteBuffer:** Enqueue commands to write to a buffer object from host memory
- **clEnqueueCopyBuffer:** Enqueue a command to copy a buffer object identified by source to destination buffer
- **clEnqueueFillBuffer:** Enqueue a command to fill a buffer object with a pattern of a given pattern size
- **clEnqueueMapBuffer:** Enqueue a command to map a region of the buffer object into the host address space and returns a pointer to this mapped region

OpenCL Host Code Cont.

Create memory objects

```
/* cl_mem clCreateBuffer(cl_context context,cl_mem_flags flags,size_t size,
    void *host_ptr,cl_int *errcode_ret) */

  // Create the input (a, b) arrays in device memory
  cl_mem d_a  = clCreateBuffer(context,  CL_MEM_READ_ONLY |
      CL_MEM_COPY_HOST_PTR,  dataSize, h_a, &err);
  cl_mem d_b  = clCreateBuffer(context,  CL_MEM_READ_ONLY |
      CL_MEM_COPY_HOST_PTR,  dataSize, h_b, &err);
  // Create the output (c) array in device memory
  cl_mem d_c  = clCreateBuffer(context,  CL_MEM_READ_WRITE, dataSize, NULL, &
      err);
```

OpenCL Host Code Cont.

Set kernel arguments

```
/*  cl_int clSetKernelArg(cl_kernel kernel,cl_uint arg_index,size_t arg_size,
    const void *arg_value) */

  // Set the arguments to our compute kernel
  err  = clSetKernelArg(ko_vadd, 0, sizeof(cl_mem), &d_a);
  err = clSetKernelArg(ko_vadd, 1, sizeof(cl_mem), &d_b);
  err = clSetKernelArg(ko_vadd, 2, sizeof(cl_mem), &d_c);
  err = clSetKernelArg(ko_vadd, 3, sizeof(unsigned int), &count);
```

OpenCL Host Code Cont.

Calculate global and local work size

```
/* cl_int clGetDeviceInfo(cl_device_id  device,cl_device_info  param_name,
    size_t  param_value_size,void  *param_value,size_t  *param_value_size_ret
    ) */

  const int count = LENGTH;
  cl_uint max_work_itm_dims;
  err = clGetDeviceInfo( device_id, CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS, sizeof
      (cl_uint), &max_work_itm_dims, NULL);
  size_t *max_loc_size = (size_t*)malloc(max_work_itm_dims*sizeof(size_t));
  err = clGetDeviceInfo( device_id, CL_DEVICE_MAX_WORK_ITEM_SIZES,
      max_work_itm_dims*sizeof(size_t), max_loc_size, NULL);
  size_t  global_work_size = count;
  size_t local_work_size=1;
  for (i=0;i<max_work_itm_dims;i++)
    local_work_size*=max_loc_size[i];
```

OpenCL Host Code Cont.

Writing input data to device from host

```
// cl_int clEnqueueWriteBuffer(cl_command_queue command_queue,cl_mem buffer,
    cl_bool blocking_write,size_t offset,size_t cb,const void *ptr,cl_uint
    num_events_in_wait_list,const cl_event *event_wait_list,cl_event *event)

  //Write the data from host to the compute device
  err = clEnqueueReadBuffer( commands, d_a, CL_TRUE, 0, sizeof(float) * count,
       h_a, 0, NULL, NULL );
  err = clEnqueueReadBuffer( commands, d_b, CL_TRUE, 0, sizeof(float) * count,
       h_b, 0, NULL, NULL );
```

OpenCL Host Code Cont.

Executing the kernel on device

```
//cl_int clEnqueueNDRangeKernel(cl_command_queue command_queue,cl_kernel
    kernel,cl_uint work_dim,const size_t *global_work_offset,const size_t *
    global_work_size,const size_t *local_work_size,cl_uint
    num_events_in_wait_list,const cl_event *event_wait_list,cl_event *event)

  // Execute the kernel over the entire range of our 1d input data set
  err = clEnqueueNDRangeKernel(commands, ko_vadd, 1, NULL, &global_work_size
      , &local_work_size, 0, NULL, NULL);

  //global_work_size (local_work_size) point to an array of work_dim unsigned
      values that describe the number of global (local) work-items in work_dim
       dimensions that will execute the kernel function.
```

Work-pool

- The work-groups associated with kernel-instance are placed into a logical pool of “ready to execute” work-groups called work-pool.
- OpenCL does not constrain the order how work-groups are scheduled for execution in the actual device from the work-pool
- Once in the work-pool, independent execution of work-groups can happen in any order and could be interleaved
- For each device there can be only one work-pool used by all command-queues associated with that device

OpenCL Host Code Cont.

Reading back the result from device to host

```
// cl_int clEnqueueReadBuffer(cl_command_queue command_queue,cl_mem buffer,
    cl_bool blocking_read,size_t offset,size_t size,void *ptr,cl_uint
    num_events_in_wait_list,const cl_event *event_wait_list,cl_event *event)

  // Read back the result from the compute device
  err = clEnqueueReadBuffer( commands, d_c, CL_TRUE, 0, sizeof(float) * count,
       h_c, 0, NULL, NULL );

  // Test the results
  int correct = 0;
  for(int i = 0; i < count; i++)
    if(h[c]==h_a[i] + h_b[i])
      correct++;

  printf("Vector add to find C = A+B:  %d out of %d results were correct.\n",
      correct, count);
```

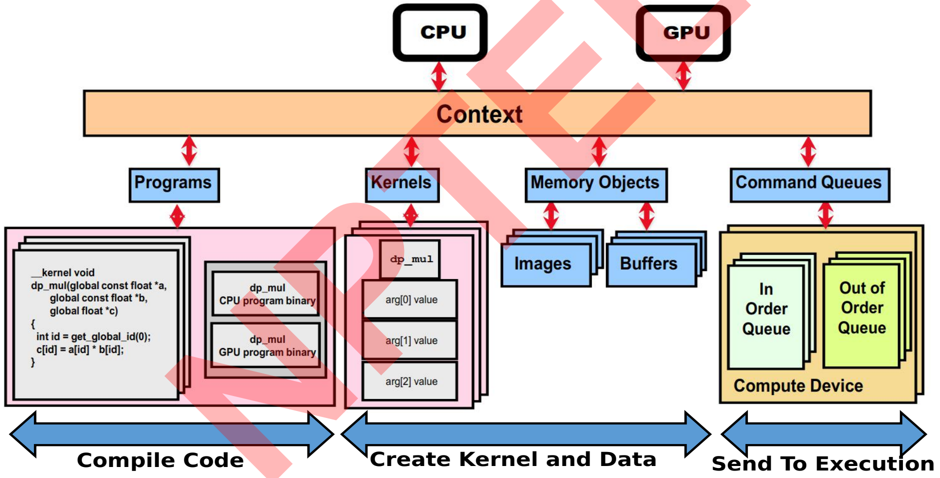
OpenCL Host Code Cont.

Cleanup and exit

```
  clReleaseMemObject(d_a);
  clReleaseMemObject(d_b);
  clReleaseMemObject(d_c);
  clReleaseProgram(program);
  clReleaseKernel(ko_vadd);
  clReleaseCommandQueue(commands);
  clReleaseContext(context);
  free(h_a);
  free(h_b);
  free(h_c);

  return 0;
}
```

OpenCL Runtime System Overview



```
__kernel void
dp_mul(global const float *a,
      global const float *b,
      global float *c)
{
  int id = get_global_id(0);
  c[id] = a[id] * b[id];
}
```

(Figure from reference[2])

OpenCL Synchronization

Synchronization refers to mechanisms that constrain the order of execution between two or more units of execution.

- **Work-group synchronization:** Constraints on the order of execution for work-items in a single work-group
- **Command synchronization:** Constraints on the order of commands launched for execution

Work-group Synchronization

- Synchronization between work-items in a single work-group is done using following command -
  - void barrier(cl_mem_fence_flags flags) (work_group_barrier from OpenCL 2.0 onwards)
- Options for flags are-
  - CLK_LOCAL_MEM_FENCE
  - CLK_GLOBAL_MEM_FENCE
- All the work-items in a work-group must execute the barrier before any are allowed to continue execution beyond the barrier
- Work-group barrier must be encountered by all work-items of a work-group executing the kernel or by none at all

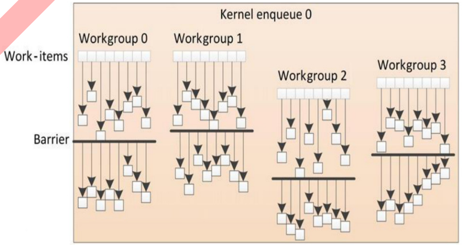
Work-group Synchronization Example

Kernel Code

```
__kernel void simpleKernel(__global float *a,
    __global float *b, __local float *localbuf
    ) {
//cache data to local memory
localbuf[get_local_id(0)] = a[get_global_id(0)
    ];

//wait untill all work_items have read the
    data
work_group_barrier(CLK_LOCAL_MEM_FENCE);

//perform the operation and save in output
    buffer
unsigned int addr = (get_local_id(0)+1) %
    get_local_size(0);
b[get_global_id(0)] = localbuf[get_local_id(0)
    ] + localbuf[addr];
}
```



(Figure from reference[2])

Event Object

- An event object can be used to track the execution status of a command.
- The OpenCL API calls that enqueue commands to command-queue(s), create new event object that is returned in the event argument.
- In case of an error enqueuing the command in the command-queue the event argument does not return an event object.
- Can query the value of an event from the host. For example to track the progress of a command.

Event Object

Execution status of an enqueued command at any given point in time can be one of the following:

- CL_QUEUED
- CL_SUBMITTED
- CL_RUNNING
- CL_COMPLETE

Event Object

OpenCL APIs related to Event Object-

- **clCreateUserEvent:** Create a user event object
- **clSetUserEventStatus:** Set the execution status of a user event object
- **clWaitForEvents:** Waits on the host thread for commands identified by event objects in event_list to complete
- **clGetEventInfo:** Return information about an event object
- **clSetEventCallback:** Register a user callback function for a specific command execution status. The registered callback function will be called when the execution status of command associated with event changes to an execution status equal to or past the status specified by command_exec_status.

Command Synchronization

- Command synchronization is defined in terms of distinct synchronization points.
- A synchronization point between a pair of commands (A and B) assures that results of command A happens-before command B is launched
- The synchronization points occur between commands in host command-queues and between commands in device-side command-queues.
- All OpenCL API functions that enqueue commands return an event that identifies the status of command.
- Value of the event associated with the command is set to CL_COMPLETE when done

Command Synchronization

The synchronization points defined in OpenCL include:

- **Completion of a command:** A kernel-instance is complete after all of the work-groups in the kernel and all of its child kernels have completed. This is signaled to the host, parent kernel or other kernels within command queues by setting the value of the event associated with a kernel to CL_COMPLETE.
- **clWaitForEvents:** This function waits on the host thread for commands identified by event objects in event_list to CL_COMPLETE. The events specified in event_list act as synchronization points.

Command Synchronization

The synchronization points defined in OpenCL include:

- **Blocking Commands:** Command execution can be blocking or non-blocking. For blocking commands, the OpenCL API functions that enqueue commands don't return until the command has completed. Some of the blocking commands are clEnqueueReadBuffer, clEnqueueWriteBuffer with blocking_read and blocking_write set to CL_TRUE
- **Command-queue barrier:** Ensures that all previously enqueued commands to a command-queue have finished execution before any following commands enqueued in the command-queue can begin execution. The OpenCL API functions are clEnqueueBarrierWithWaitList, clEnqueueMarkerWithWaitList.

Command Synchronization

The synchronization points defined in OpenCL include:

- **clFlush:** All previously queued OpenCL commands in command_queue are issued to the device associated with command_queue. clFlush only guarantees that all queued commands to command_queue will eventually be submitted to the appropriate device. There is no guarantee that they will be complete after clFlush returns.
- **clFinish:** All previously queued OpenCL commands in command_queue are issued to the associated device, and the function blocks until all previously queued commands have completed. clFinish does not return until all previously queued commands in command_queue have been processed and completed.

Command Synchronization

- **Command-queue barrier:**
  - Ensures that all previously queued commands have finished execution and updating memory objects before subsequently enqueued commands begin execution
  - Can only be used to synchronize between commands in a single command-queue
- **Waiting on an event**
  - All OpenCL API functions that enqueue commands return an event that identifies the status of command
  - Value of the event associated with the command is set to CL_COMPLETE when done
- **clFinish:** Blocks until all previously enqueued commands in the command queue have completed

Out-of-order Execution of Kernels and Memory Object Commands

- The OpenCL functions that are submitted to a command-queue are enqueued in the order the calls are made but can be configured to execute in-order or out-of-order.
- The properties argument in clCreateCommandQueueWithProperties can be used to specify the execution order.
- In out-of-order execution mode there is no guarantee that the enqueued commands will finish execution in the order they were queued.

In-order Execution of Kernels

If the CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE property of a command-queue is not set, the commands enqueued to a command-queue execute in order. For example,

- If an application calls clEnqueueNDRangeKernel to execute kernel A followed by a clEnqueueNDRangeKernel to execute kernel B
- The application can assume that kernel A finishes first and then kernel B is executed.
- If the memory objects output by kernel A are inputs to kernel B then kernel B will see the correct data in memory objects produced by execution of kernel A.

In case of out-of-order execution, there is no guarantee that kernel A will finish before kernel B starts execution.

In-order Execution Example Code

```
// Perform setup of platform, context and create buffers
...
// Create queue leaving parameters as default so queue is in-order queue
clCreateCommandQueue(context,devices[0],0,0);
...
clEnqueueWriteBuffer(queue,bufferA,CL_TRUE,0,10*sizeof(int),a,0,NULL,NULL);
clEnqueueWriteBuffer(queue,bufferB,CL_TRUE,0,10*sizeof(int),b,0,NULL,NULL);
...
clEnqueueNDRangeKernel(queue,kernelA,1,NULL,globalws,localws,0,NULL,NULL);
clEnqueueReadBuffer(queue,bufferOut,CL_TRUE,0,10*sizeof(int),out,0,0,0);
...
clEnqueueNDRangeKernel(queue,kernelB,1,NULL,globalws,localws,0,NULL,NULL);
...
clFinish(queue); // cl_int clFinish(cl_command_queue command_queue)
```

Out-of-order Execution of Kernels

- If the CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE property of a command-queue is set when the command-queue is created, the commands enqueued to a command-queue execute in out-of-order.
- In out-of-order execution mode there is no guarantee that the enqueued commands will finish execution in the order they were queued.
- It is possible that an earlier clEnqueueNDRangeKernel call to execute kernel A identified by event A may execute and/or finish later than a clEnqueueNDRangeKernel call to execute kernel B which was called by the application at a later point in time.

Out-of-order Execution of Kernels

To guarantee a specific order of execution of kernels, following can be done-

- Wait on a event A by specifying in the event_wait_list argument to clEnqueueNDRangeKernel for kernel B.
- Marker (clEnqueueMarkerWithWaitList) or barrier (clEnqueueBarrierWithWaitList) command can be enqueued to the command-queue. This ensures that previously enqueued commands identified by the list of events to wait for (or all previous commands) have finished.
- clSetEventCallback to call registered callback function when the execution status of command associated with an event changes to given status.

Out-of-Order Execution Example Code

```
// Perform setup of platform, context, queue and create buffers
...
cl_event writeEventA;
cl_event writeEventB;
cl_event kernelEvent;
cl_event readEvent;
clEnqueueWriteBuffer(queue,bufferA,CL_TRUE,0,10*sizeof(int),a,0,NULL,&
    writeEventA);
clEnqueueWriteBuffer(queue,bufferB,CL_TRUE,0,10*sizeof(int),b,0,NULL,&
    writeEventB);
// Set kernel arguments
...
size_t localws[1] = {2}; size_t globalws[1] = {10};
// Wait on both writes before executing the kernel
cl_event eventList[2];
eventList[0] = writeEventA;
eventList[1] = writeEventB;
```

Out-of-Order Execution Example Code

```
clEnqueueNDRangeKernel(queue,kernel,1,NULL,globalws,localws,2,eventList,&
    kernelEvent);
// Decrease reference count on events
clReleaseEvent(writeEventA);
clReleaseEvent(writeEventB);
// Read will wait on kernel completion to run
clEnqueueReadBuffer(queue,bufferOut,CL_TRUE,0,10*sizeof(int),out,1,&
    kernelEvent,&readEvent);
clReleaseEvent(kernelEvent);
// Block until the read has completed
// cl_int clWaitForEvents(cl_uint num_events,const cl_event *event_list)
clWaitForEvents(1, &readEvent);
clReleaseEvent(readEvent);
```

Out-of-Order Execution Example Code

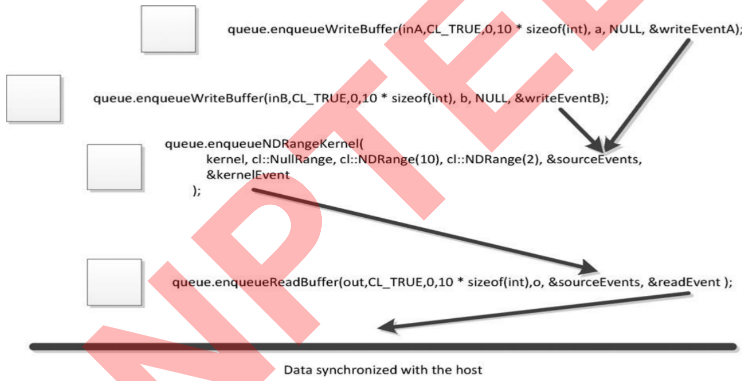
Another approach

```
clEnqueueNDRangeKernel(queue,kernel,1,NULL,globalws,localws,2,eventList,&
    kernelEvent);

// Read will wait on kernel completion to run
clEnqueueReadBuffer(queue,bufferOut,CL_TRUE,0,10*sizeof(int),out,1,&
    kernelEvent,&readEvent);

// Set the callback such that callbackFunction is called when readEvent
    indicates that the event has completed  (CL_COMPLETE)
// cl_int clSetEventCallback(cl_event event,cl_int  command_exec_callback_type
     ,void (CL_CALLBACK  *pfn_event_notify) (cl_event event, cl_int
    event_command_exec_status, void *user_data),void *user_data)
errcode=clSetEventCallback(readEvent,CL_COMPLETE,callbackFunction,(void *)&
    ipargs);
clReleaseEvent(readEvent);
```

Dependencies explicitly defined



```
queue.enqueueWriteBuffer(inA,CL_TRUE,0,10 * sizeof(int), a, NULL, &writeEventA);

queue.enqueueWriteBuffer(inB,CL_TRUE,0,10 * sizeof(int), b, NULL, &writeEventB);

queue.enqueueNDRangeKernel(
        kernel, cl::NullRange, cl::NDRange(10), cl::NDRange(2), &sourceEvents,
        &kernelEvent
    );

queue.enqueueReadBuffer(out,CL_TRUE,0,10 * sizeof(int),o, &sourceEvents, &readEvent );
```

(Figure from reference[2])

Profiling Operations on Memory Objects and Kernels

- Profiling of OpenCL commands can be enabled by using a command-queue created with CL_QUEUE_PROFILING_ENABLE flag set in properties argument to clCreateCommandQueueWithProperties.
- When profiling is enabled, the event objects that are created from enqueuing a command store a timestamp for each of their state transitions.
- The OpenCL APIs used for profiling is **clGetEventProfilingInfo**
- Given by 64-bit value that describes the current device time counter in nanoseconds when the command identified by event is done

Profiling Operations Example

```
...
clCreateCommandQueue(ctx, dev_id, CL_QUEUE_PROFILING_ENABLE, &status)
...
clEnqueueNDRangeKernel(queue,kernel,1,NULL,globalws,localws,2,eventList,&
    execEvent);
...
//Get profile info related to the event execEvent
//cl_int clGetEventProfilingInfo(cl_event event,  cl_profiling_info
    param_name, size_t param_value_size, void *param_value, size_t *
    param_value_size_ret)
//Give the current device time counter in nanoseconds when the command
    identified by event is enqueued,submitted in a command-queue by the
    host, starts and finish execution on the device
```

Profiling Operations Example

```
//current device time counter in nanoseconds when the command identified
    by the event is in the specified state.
status = clGetEventProfilingInfo(execEvent, CL_PROFILING_COMMAND_QUEUED,
    sizeof(cl_ulong), &queued, NULL);
status = clGetEventProfilingInfo(execEvent, CL_PROFILING_COMMAND_SUBMIT,
    sizeof(cl_ulong), &submit, NULL);
status = clGetEventProfilingInfo(execEvent, CL_PROFILING_COMMAND_START,
    sizeof(cl_ulong), &start, NULL);
status = clGetEventProfilingInfo(execEvent, CL_PROFILING_COMMAND_END,
    sizeof(cl_ulong), &end, NULL);
...
printf("clEnqueueNDRangeKernel profiling details:\nQueued: %llu, Submit: %
    llu, Start: %llu,Finish: %llu ",queued,submit,start,end);
...
```

Refernces

- Khronos OpenCL Working Group. *The OpenCL Specification Version-2.1*. 2018 February 13,
- Kaeli DR, Mistry P, Schaa D, Zhang DP. *Heterogeneous computing with OpenCL 2.0*. Morgan Kaufmann; 2015 Jun 18.