OpenCL – Open Computing Language
OpenCL

- A standard based upon C for portable parallel applications
- Task parallel and data parallel applications
- Focuses on multi platform support (multiple CPUs, GPUs, …)
- Development initiated by Apple.
- Developed by Khronos group who also managed OpenGL
- OpenCL 1.0 2008. Released with Mac OS 10.6 (Snow Leopard)
- Similarities with CUDA
- Implementation available for NVIDIA GPUs
OpenCL Programming Model

- Uses data parallel programming model, similar to CUDA
- Host program launches kernel routines as in CUDA, but allows for just-in-time compilation during host execution.
- OpenCL “work items” corresponds to CUDA threads
- OpenCL “work groups” corresponds to CUDA thread blocks
- Work items in same work group can be synchronized with a barrier as in CUDA.
Structure of OpenCL main program

- Get information about platform and devices available on system
- Select devices to use
- Create an OpenCL command queue
- Create memory buffers on device
- Transfer data from host to device memory buffers
- Create kernel program object
- Build (compile) kernel in-line (or load precompiled binary)
- Create OpenCL kernel object
- Set kernel arguments
- Execute kernel
- Read kernel memory and copy to host memory.
Includes

● #include <CL/cl.h>
  //OpenCL header for C
Platform

● "The host plus a collection of devices managed by the OpenCL framework that allow an application to share resources and execute kernels on devices in the platform."

● Platforms represented by a `cl_platform` object, initialized with `clGetPlatformID()`
//Platform

cl_platform_id   platform;

clGetPlatformIDs (1, &platform, NULL);

Number of platform entries
List of OpenCL platforms found. (Platform IDs)
In our case just one platform, identified by &platform

Returns number of OpenCL platforms available. If NULL, ignored.
"The environment within which the kernels execute and the domain in which synchronization and memory management is defined.

The context includes a set of devices, the memory accessible to those devices, the corresponding memory properties and one or more command-queues used to schedule execution of a kernel(s) or operations on memory objects."
```c
//Context

cl_context_properties props[3];
props[0] = (cl_context_properties) CL_CONTEXT_PLATFORM;
props[1] = (cl_context_properties) platform;
props[2] = (cl_context_properties) 0;
cl_context GPUContext =
    clCreateContextFromType(props, CL_DEVICE_TYPE_GPU, NULL, NULL, NULL, NULL);

//Context info

size_t ParmDataBytes;
clGetContextInfo(GPUContext, CL_CONTEXT_DEVICES, 0, NULL, &ParmDataBytes);
cl_device_id* GPUDevices = (cl_device_id*)malloc(ParmDataBytes);
clGetContextInfo(GPUContext, CL_CONTEXT_DEVICES, ParmDataBytes, GPUDevices, NULL);
```
Allocating Memory on Device

Use `clCreateBuffer`:

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

- **OpenCL context**, from `clCreateContextFromType()`
- **Bit field to specify type of allocation/usage** (e.g., `CL_MEM_READ_WRITE`)
- **No of bytes in buffer memory object**
- **Returns memory object**
- **Ptr to buffer data** (May be previously allocated.)
- **Returns error code if an error**
Sample code for allocating memory

// source data on host, two vectors
int *A, *B;
A = new int[N];
B = new int[N];
for(int i = 0; i < N; i++) {
    A[i] = rand()%1000;
    B[i] = rand()%1000;
}

…
// Allocate GPU memory for source vectors
cl_mem GPUVector1 = clCreateBuffer(GPUContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(int)*N, A, NULL);
cl_mem GPUVector2 = clCreateBuffer(GPUContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(int)*N, B, NULL);

// Allocate GPU memory for output vector
cl_mem GPUOutputVector =
    clCreateBuffer(GPUContext, CL_MEM_WRITE_ONLY, sizeof(int)*N, NULL, NULL);
// Build the program (OpenCL JIT compilation)

clBuildProgram(OpenCLProgram, 0, NULL, NULL, NULL, NULL, NULL);
Creating Kernel Objects

// Create a handle to the compiled OpenCL function

cl_kernel OpenCLVectorAdd = clCreateKernel(OpenCLProgram, "vectorAdd", NULL)
Function to copy from buffer object to host

```c
cl_int clEnqueueReadBuffer (  
    cl_command_queue command_queue, 
    cl_mem buffer, 
    cl_bool blocking_read, 
    size_t offset, 
    size_t cb, 
    void *ptr, 
    cl_uint num_events_in_wait_list, 
    const cl_event *event_wait_list, 
    cl_event *event)
```
Copy from host memory to buffer object

```c
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)
```
// Cleanup

free(GPUDevices);

clReleaseKernel(OpenCLVectorAdd);

clReleaseProgram(OpenCLProgram);

clReleaseCommandQueue(GPUCommandQueue);

clReleaseContext(GPUContext);

clReleaseMemObject(GPUVector1);

clReleaseMemObject(GPUVector2);

clReleaseMemObject(GPUOutputVector);
## OpenCL to CUDA Model Mapping

<table>
<thead>
<tr>
<th>OpenCL Parallelism Concept</th>
<th>CUDA Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel</td>
<td>kernel</td>
</tr>
<tr>
<td>host program</td>
<td>host program</td>
</tr>
<tr>
<td>NDRange (index space)</td>
<td>grid</td>
</tr>
<tr>
<td>work item</td>
<td>thread</td>
</tr>
<tr>
<td>work group</td>
<td>block</td>
</tr>
</tbody>
</table>
Overview of OpenCL Execution Model

The execution model consists of a grid of work items and work groups. Each work group has a unique group ID, which is used to index into the grid. The grid is defined by local and global sizes, where the local size is the number of work items within a work group, and the global size is the maximum number of work groups.

- **Work Group**: A group of work items with a unique group ID.
- **Work Item**: A unit of work within a work group.
- **Local Size**: Size of the work group, indicates how many work items are contained within a single work group.
- **Global Size**: Size of the entire grid, indicates the maximum number of work groups that can exist.

The grid is structured as follows:

```
<table>
<thead>
<tr>
<th>Local Size(0)</th>
<th>Global Size(0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Size(1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Group ID</td>
</tr>
<tr>
<td></td>
<td>0,0</td>
</tr>
<tr>
<td></td>
<td>0,1</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>1,0</td>
</tr>
<tr>
<td></td>
<td>1,1</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
```

The diagram illustrates the hierarchical structure of work items and work groups, showing how the local and global sizes define the layout of the execution model.
## Mapping of OpenCL to CUDA

<table>
<thead>
<tr>
<th>OpenCL API Call</th>
<th>Explanation</th>
<th>CUDA Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_global_id(0);</code></td>
<td>global index of the work item in the x dimension</td>
<td><code>blockIdx.x×blockDim.x+threadIdx.x</code></td>
</tr>
<tr>
<td><code>get_local_id(0)</code></td>
<td>local index of the work item within the work group in the x dimension</td>
<td><code>blockIdx.x</code></td>
</tr>
<tr>
<td><code>get_global_size(0);</code></td>
<td>size of NDRange in the x dimension</td>
<td><code>gridDim.x ×blockDim.x</code></td>
</tr>
<tr>
<td><code>get_local_size(0);</code></td>
<td>Size of each work group in the x dimension</td>
<td><code>blockDim.x</code></td>
</tr>
</tbody>
</table>
Conceptual OpenCL Device Architecture
## Mapping OpenCL Memory Types to CUDA

<table>
<thead>
<tr>
<th>OpenCL Memory Types</th>
<th>CUDA Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>global memory</td>
<td>global memory</td>
</tr>
<tr>
<td>constant memory</td>
<td>constant memory</td>
</tr>
<tr>
<td>local memory</td>
<td>shared memory</td>
</tr>
<tr>
<td>private memory</td>
<td>Local memory</td>
</tr>
</tbody>
</table>
OpenCL Context for Device Management

Application → Kernel → Cmd Queue

Cmd Queue → OpenCL Device

OpenCL Device
OpenCL Specification

- **Four Parts:**
  1. **Platform model:** Specifies that there is one processor coordinating execution (the host) and one or more processors capable of executing OpenCL C code (the devices). It defines an abstract hardware model that is used by programmers when writing OpenCL C functions (called kernels) that execute on the devices.
  2. **Execution model:** Defines how the OpenCL environment is configured on the host and how kernels are executed on the device. This includes setting up an OpenCL context on the host, providing mechanisms for host–device interaction, and defining a concurrency model used for kernel execution on devices.
  3. **Memory model:** Defines the abstract memory hierarchy that kernels use, regardless of the actual underlying memory architecture. The memory model closely resembles current GPU memory hierarchies, although this has not limited adoptability by other accelerators.
  4. **Programming model:** Defines how the concurrency model is mapped to physical hardware.
// Perform an element-wise addition of A and B and store in C
// N work-items will be created to execute this kernel.

__kernel 
void vecadd(__global int *A,
           __global int *B, __global int *C)
{
    // Get the work-item’s unique ID
    int idx = get_global_id(0);

    // Add the corresponding locations of
    // 'A' and 'B’, and store the result in 'C'.
}
Vector Addition – prelim

// This program implements a vector addition using OpenCL
// System includes
#include <stdio.h>
#include <stdlib.h>

// OpenCL includes
#include <CL/cl.h>

// OpenCL kernel to perform an element-wise add of two arrays
const char* programSource =
  "__kernel
  "void vecadd(__global int *A, __global int *B, __global int *C) {
    int idx = get_global_id(0);
  }";
int main() {

    // This code executes on the OpenCL host
    // Host data
    int *A = NULL; // Input array
    int *B = NULL; // Input array
    int *C = NULL; // Output array

    // Elements in each array
    const int elements = 2048;

    // Compute the size of the data
    size_t datasize = sizeof(int)*elements;

    // Allocate space for input/output data
    A = (int*)malloc(datasize);
    B = (int*)malloc(datasize);
    C = (int*)malloc(datasize);

    // Initialize the input data
    for(int i = 0; i < elements; i++) {
        A[i] = i;
        B[i] = i;
    }

    // Use this to check the output of each API call
    cl_int status;
}
OpenCL Platform Model

FIGURE 2.2
The platform model defines an abstract architecture for devices.
CLInfo

- CLInfo - Uses clGetPlatformInfo() and clGetDeviceInfo()
Vector Addition – Step 1

//____________________________________________________________________________
// STEP 1: Discover and initialize the platforms
//____________________________________________________________________________
cl_uint numPlatforms = 0;
cl_platform_id *platforms = NULL;

// Use clGetPlatformIDs() to retrieve the number of platforms
status = clGetPlatformIDs(0, NULL, &numPlatforms);

// Allocate enough space for each platform
platforms =
    (cl_platform_id*)malloc(numPlatforms*sizeof(cl_platform_id));

// Fill in platforms with clGetPlatformIDs()
status = clGetPlatformIDs(numPlatforms, platforms, NULL);
Vector Addition – Step 2

```c
// STEP 2: Discover and initialize the devices

cl_uint numDevices = 0;
cl_device_id *devices = NULL;

// Use clGetDeviceIDs() to retrieve the number of devices present
status = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, 0, NULL, &numDevices);

// Allocate enough space for each device
devices = (cl_device_id*)malloc(numDevices*sizeof(cl_device_id));

// Fill in devices with clGetDeviceIDs()
status = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, numDevices, devices, NULL);
```
// STEP 3: Create a context

cl_context context = NULL;

// Create a context using clCreateContext() and
// associate it with the devices

context = clCreateContext(NULL, numDevices,
                          devices, NULL, NULL, &status);
Vector Addition – Step 4

```c
//STEP 4: Create a command queue
cl_command_queue cmdQueue;

// Create a command queue using clCreateCommandQueue(),
// and associate it with the device you want to execute on
cmdQueue = clCreateCommandQueue(context, devices[0], 0, &status);
```
OpenCL Memory Model

FIGURE 2.3
The abstract memory model defined by OpenCL.
Vector Addition – Step 5

//_____________________________________________________________________________________
// STEP 5: Create device buffers
//_____________________________________________________________________________________

cl_mem bufferA; // Input array on the device
cl_mem bufferB; // Input array on the device
cl_mem bufferC; // Output array on the device

// Use clCreateBuffer() to create a buffer object (d_A)
// that will contain the data from the host array A
bufferA = clCreateBuffer( context, CL_MEM_READ_ONLY, datasize, NULL, &status);

// Use clCreateBuffer() to create a buffer object (d_B)
// that will contain the data from the host array B
bufferB = clCreateBuffer( context, CL_MEM_READ_ONLY, datasize, NULL, &status);

// Use clCreateBuffer() to create a buffer object (d_C)
// with enough space to hold the output data
bufferC = clCreateBuffer(context, CL_MEM_WRITE_ONLY, datasize, NULL, &status);
Vector Addition – Step 6

//STEP 6: Write host data to device buffers

// Use clEnqueueWriteBuffer() to write input array A to
// the device buffer bufferA
status = clEnqueueWriteBuffer(cmdQueue, bufferA, CL_FALSE, 0, datasize, A, 0, NULL, NULL);

// Use clEnqueueWriteBuffer() to write input array B to
// the device buffer bufferB
status = clEnqueueWriteBuffer(cmdQueue, bufferB, CL_FALSE, 0, datasize, B, 0, NULL, NULL);
Vector Addition – Step 7

// STEP 7: Create and compile the program

cl_program program = clCreateProgramWithSource(context,
   1, (const char**)&programSource, NULL, &status);

// Build (compile) the program for the devices with
// clBuildProgram()

status = clBuildProgram(program, numDevices,
devices, NULL, NULL, NULL);
// STEP 8: Create the kernel

cl_kernel kernel = NULL;

// Use clCreateKernel() to create a kernel from the vector addition function (named "vecadd")
kernel = clCreateKernel(program, "vecadd", &status);
Vector Addition – Step 9

//———————————————————————————————————

// STEP 9: Set the kernel arguments

//———————————————————————————————————

// Associate the input and output buffers with the
// kernel using clSetKernelArg()

status = clSetKernelArg(kernel, 0, sizeof(cl_mem), &bufferA);
status |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &bufferB);
status |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &bufferC);
//STEP 10: Configure the work-item structure

// Define an index space (global work size) of work
// items for execution.
// A workgroup size (local work size) is not required,
// but can be used.

size_t globalWorkSize[1];

// There are 'elements' work-items
globalWorkSize[0] = elements;
// STEP 11: Enqueue the kernel for execution

// Execute the kernel by using
// clEnqueueNDRangeKernel().
// ’globalWorkSize’ is the 1D dimension of the
// work-items

status = clEnqueueNDRangeKernel(cmdQueue, kernel, 1,
        NULL, globalWorkSize, NULL, 0, NULL, NULL);
Vector Addition – Step 12

//..............................................................................................................
// STEP 12: Read the output buffer back to the host
//..............................................................................................................
// Use clEnqueueReadBuffer() to read the OpenCL output buffer (bufferC) to the host output array (C)

clEnqueueReadBuffer(cmdQueue, bufferC, CL_TRUE, 0, datasize, C, 0, NULL, NULL);
// Verify the output
bool result = true;
for(int i = 0; i < elements; i++) {
    if(C[i] != i+i) {
        result = false;
        break;
    }
}
if(result) {
    printf("Output is correct\n");
} else {
    printf("Output is incorrect\n");
}
Vector Addition – Step 13

//______________________________________________________________________________
// STEP 13: Release OpenCL resources
//______________________________________________________________________________
// Free OpenCL resources
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(cmdQueue);
clReleaseMemObject(bufferA);
clReleaseMemObject(bufferB);
clReleaseMemObject(bufferC);
clReleaseContext(context);

// Free host resources
free(A); free(B); free(C); free(platforms); free(devices);