Introduction to OpenCL
Background

- OpenCL was initiated by Apple and maintained by the Khronos Group (also home of OpenGL) as an industry standard API
  - For cross-platform parallel programming in CPUs, GPUs, DSPs, FPGAs,…
- OpenCL draws heavily on CUDA
  - Easy to learn for CUDA programmers
- OpenCL host code is much more complex and tedious due to desire to maximize portability and to minimize burden on vendors
OpenCL Programs

- An OpenCL “program” is a C program that contains one or more “kernels” and any supporting routines that run on a target device.
- An OpenCL kernel is the basic unit of parallel code that can be executed on a target device.

```
OpenCL Program

<table>
<thead>
<tr>
<th>Misc support functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel A</td>
</tr>
<tr>
<td>Kernel B</td>
</tr>
<tr>
<td>Kernel C</td>
</tr>
</tbody>
</table>
```
OpenCL Execution Model

- Integrated host+device app C program
  - Serial or modestly parallel parts in host C code
  - Highly parallel parts in device SPMD kernel C code
Mapping between OpenCL and CUDA data parallelism model concepts.

<table>
<thead>
<tr>
<th>OpenCL Parallelism Concept</th>
<th>CUDA Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>host</td>
</tr>
<tr>
<td>device</td>
<td>device</td>
</tr>
<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>
OpenCL Kernels

- Code that executes on target devices
- Kernel body is instantiated once for each work item
  - An OpenCL work item is equivalent to a CUDA thread
- Each OpenCL work item gets a unique index

```c
__kernel void vadd(__global const float *a,
                    __global const float *b,
                    __global float *result)
{
    int id = get_global_id(0);
    result[id] = a[id] + b[id];
}
```
Array of Work Items

- An OpenCL kernel is executed by an array of work items
  - All work items run the same code (SPMD)
  - Each work item can call get_global_id() to get its index for computing memory addresses and make control decisions

```c
int id = get_global_id(0);
result[id] = a[id] + b[id];
```
Work Groups: Scalable Cooperation

- Divide monolithic work item array into work groups
  - Work items within a work group cooperate via shared memory and barrier synchronization
  - Work items in different work groups cannot cooperate
- OpenCL counter part of CUDA Thread Blocks
## OpenCL Dimensions and Indices

<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>threadIdx.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>
Multidimensional Work Indexing
OpenCL Data Parallel Model Summary

- Parallel work is submitted to devices by launching kernels
- Kernels run over global dimension index ranges (NDRange), broken up into “work groups”, and “work items”
- Work items executing within the same work group can synchronize with each other with barriers or memory fences
- Work items in different work groups can’t sync with each other, except by terminating the kernel
OpenCL Hardware Abstraction

- OpenCL exposes CPUs, GPUs, and other Accelerators as “devices”
- Each device contains one or more “compute units”, i.e. cores, Streaming Multiprocessors, etc...
- Each compute unit contains one or more SIMD “processing elements”, (i.e. SP in CUDA)
OpenCL Device Architecture

![Diagram of OpenCL Device Architecture](image-url)
## OpenCL Device Memory Types

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Host access</th>
<th>Device access</th>
<th>CUDA Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>global memory</strong></td>
<td>Dynamic allocation; Read/write access</td>
<td>No allocation; Read/write access by all work items in all work groups, large and slow but may be cached in some devices.</td>
<td>global memory</td>
</tr>
<tr>
<td><strong>constant memory</strong></td>
<td>Dynamic allocation; read/write access</td>
<td>Static allocation; read-only access by all work items.</td>
<td>constant memory</td>
</tr>
<tr>
<td><strong>local memory</strong></td>
<td>Dynamic allocation; no access</td>
<td>Static allocation; shared read-write access by all work items in a work group.</td>
<td>shared memory</td>
</tr>
<tr>
<td><strong>private memory</strong></td>
<td>No allocation; no access</td>
<td>Static allocation; Read/write access by a single work item.</td>
<td>registers and local memory</td>
</tr>
</tbody>
</table>
An Example of Physical Reality Behind OpenCL Abstraction
OpenCL Context

- Contains one or more devices
- OpenCL device memory objects are associated with a context, not a specific device
OpenCL Context

- Contains one or more devices
- OpenCL memory objects are associated with a context, not a specific device
- clCreateBuffer() is the main data object allocation function
  - error if an allocation is too large for any device in the context
- Each device needs its own work queue(s)
- Memory copy transfers are associated with a command queue (thus a specific device)
OpenCL Context Setup
Code (simple)

```c
cl_int clerr = CL_SUCCESS;
cl_context clctx = clCreateContextFromType(0, CL_DEVICE_TYPE_ALL, NULL, NULL, &clerr);

size_t parmsz;
clerr = clGetContextInfo(clctx, CL_CONTEXT_DEVICES, 0, NULL, &parmsz);

cl_device_id* cldevs = (cl_device_id *) malloc(parmsz);
clerr = clGetContextInfo(clctx, CL_CONTEXT_DEVICES, parmsz, cldevs, NULL);

cl_command_queue clcmdq = clCreateCommandQueue(clctx, cldevs[0], 0, &clerr);
```
OpenCL Kernel Compilation: vadd

const char* vaddsrc =
    "__kernel void vadd(__global float *d_A, __global float
    *d_B, __global float *d_C, int N) {
    ...
    ...
    
    cl_program clpgm;
    clpgm = clCreateProgramWithSource(clctx, 1, &vaddsrc,
    NULL, &clerr);

    char clcompileflags[4096];
    sprintf(clcompileflags, "-cl-mad-enable");
    clerr = clBuildProgram(clpgm, 0, NULL, clcompileflags,
    NULL, NULL);
    cl_kernel clkern = clCreateKernel(clpgm, "vadd", &clerr);
OpenCL Device Memory Allocation

- `clCreateBuffer()`;
  - Allocates object in the device Global Memory
  - Returns a pointer to the object
  - Requires five parameters
    - OpenCL context pointer
    - Flags for access type by device (read/write, etc.)
    - Size of allocated object
    - Host memory pointer, if used in copy-from-host mode
    - Error code

- `clReleaseMemObject()`
  - Frees object
    - Pointer to freed object
OpenCL Device Memory Allocation (cont.)

- Code example:
  - Allocate a 1024 single precision float array
  - Attach the allocated storage to d_a
  - “d_” is often used to indicate a device data structure

```c
VECTOR_SIZE = 1024;
closing {cl_mem d_a;
int size = VECTOR_SIZE* sizeof(float);
d_a = clCreateBuffer(clctx,
    CL_MEM_READ_ONLY, size, NULL, NULL);
...
clReleaseMemObject(d_a);
```
OpenCL Memories

- **__global** – large, long latency
- **__private** – on-chip device registers
- **__local** – memory accessible from multiple PEs or work items. May be SRAM or DRAM, must query…
- **__constant** – read-only constant cache
- Device memory is managed explicitly by the programmer, as with CUDA
OpenCL Device Command Execution

Application → Command → Cmd Queue

Command → Cmd Queue

OpenCL Device

OpenCL Context
OpenCL Host-to-Device Data Transfer

- `clEnqueueWriteBuffer();`
  - Memory data transfer to device
  - Requires nine parameters
    - OpenCL command queue pointer
    - Destination OpenCL memory buffer
    - Blocking flag
    - Offset in bytes
    - Size (in bytes) of written data
    - Host memory pointer
    - List of events to be completed before execution of this command
    - Event object tied to this command
  - Asynchronous transfer later...
OpenCL Host-to-Device Data Transfer

- `clEnqueueWriteBuffer();`
  - Memory data transfer to device
  - Requires nine parameters
    - OpenCL command queue pointer
    - Destination OpenCL memory buffer
    - Blocking flag
    - Offset in bytes
    - Size (in bytes) of written data
    - Source host memory pointer
    - List of events to be completed before execution of this command
    - Event object tied to this command
OpenCL Device-to-Host Data Transfer

- `clEnqueueReadBuffer();`
  - memory data transfer to host
  - requires nine parameters
    - OpenCL command queue pointer
    - Source OpenCL memory buffer
    - Blocking flag
    - Offset in bytes
    - Size of bytes of read data
    - Destination host memory pointer
    - List of events to be completed before execution of this command
    - Event object tied to this command
    - Asynchronous transfer later
OpenCL Device-to-Host Data Transfer

- `clEnqueueReadBuffer();`
  - Memory data transfer to host
  - requires nine parameters
    - OpenCL command queue pointer
    - Source OpenCL memory buffer
    - Blocking flag
    - Offset in bytes
    - Size of bytes of read data
    - Destination host memory pointer
    - List of events to be completed before execution of this command
    - Event object tied to this command
OpenCL Host-Device Data Transfer (cont.)

- Code example:
  - Transfer a 64 * 64 single precision float array
  - a is in host memory and d_a is in device memory

```c
clEnqueueWriteBuffer(clcmdq, d_a, CL_FALSE, 0, mem_size, (const void *)a, 0, 0, NULL);

clEnqueueReadBuffer(clcmdq, d_result, CL_FALSE, 0, mem_size, (void *) host_result, 0, 0, NULL);
```
OpenCL Host-Device Data Transfer (cont.)

- `clCreateBuffer` and `clEnqueueWriteBuffer` can be combined into a single command using special flags.
- Eg:

```
  d_A = clCreateBuffer(clctxt, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, mem_size, h_A, NULL);
```

- Combination of 2 flags here. `CL_MEM_COPY_HOST_PTR` to be used only if a valid host pointer is specified.
- This creates a memory buffer on the device, and copies data from `h_A` into `d_A`.
- Includes an implicit `clEnqueueWriteBuffer` operation, for all devices/command queues tied to the context `clctxt`. 
float *h_A = ..., *h_B = ...;
   // allocate device (GPU) memory
cl_mem d_A, d_B, d_C;
d_A = clCreateBuffer(clctx, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, N *sizeof(float), h_A, NULL);
d_B = clCreateBuffer(clctx, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, N *sizeof(float), h_B, NULL);
d_C = clCreateBuffer(clctx, CL_MEM_WRITE_ONLY, N *sizeof(float), NULL, NULL);
Device Kernel Configuration
Setting for vadd

clkern=clCreateKernel(clpgm, "vadd", NULL);
...
clerr= clSetKernelArg(clkern, 0, sizeof(cl_mem),(void *)&d_A);
clerr= clSetKernelArg(clkern, 1, sizeof(cl_mem),(void *)&d_B);
clerr= clSetKernelArg(clkern, 2, sizeof(cl_mem),(void *)&d_C);
clerr= clSetKernelArg(clkern, 3, sizeof(int), &N);
Device Kernel Launch and Remaining Code for vadd

cl_event event=NULL;
clerr= clEnqueueNDRangeKernel(clcmdq, clkern, 2,
    NULL,Gsz, Bsz, 0, NULL, &event);
clerr= clWaitForEvents(1, &event);
clEnqueueReadBuffer(clcmdq, d_C, CL_TRUE, 0,
    N*sizeof(float), h_C, 0, NULL, NULL);
clReleaseMemObject(d_A);
clReleaseMemObject(d_B);
clReleaseMemObject(d_C);
}