What is OpenCL?

- **Cross-platform** parallel computing **API** and C-like language for **heterogeneous computing** devices
- **Code is portable** across various target devices:
  - Correctness is guaranteed
  - Performance of a given kernel is not guaranteed across differing target devices
- **OpenCL implementations already exist** for AMD, ATI, and NVIDIA GPUs, x86 CPUs
- **In principle, OpenCL could also target DSPs, Cell, and perhaps also FPGAs**
More on Multi-Platform Targeting

- Targets a broader range of CPU-like and GPU-like devices than CUDA
  - Targets devices produced by multiple vendors
  - Many features of OpenCL are optional and may not be supported on all devices

- OpenCL codes must be prepared to deal with much greater hardware diversity

- A single OpenCL kernel will likely not achieve peak performance on all device types
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 using barriers or memory fences.
- Work items in different work groups can only sync with each other by launching a new kernel.
## Mapping Data Parallelism Models: OpenCL to CUDA

<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>NDRRange (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 NDRange Configuration

<table>
<thead>
<tr>
<th>Work Group</th>
<th>Global Size(0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Group ID</td>
</tr>
<tr>
<td>0,0</td>
<td>0,1</td>
</tr>
<tr>
<td></td>
<td>…</td>
</tr>
<tr>
<td>1,0</td>
<td>1,1</td>
</tr>
<tr>
<td></td>
<td>…</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

- **Work Group**
- **Local Size(0)**
- **Global Size(1)**
- **Local Size(1)**

**Copyright © 2013 by Yong Cao, Referencing UIUC ECE408/498AL Course Notes**
## Mapping OpenCL indices to CUDA

<table>
<thead>
<tr>
<th>OpenCL API call</th>
<th>Explanation</th>
<th>CUDA equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_global_id(0);</td>
<td>Global index of the work item in the x-dimension</td>
<td>blockIdx.x × blockDim.x + threadIdx.x</td>
</tr>
<tr>
<td>get_local_id(0)</td>
<td>Local index of the work item within the work group in the x-dimension</td>
<td>threadIdx.x</td>
</tr>
<tr>
<td>get_global_size(0);</td>
<td>Size of NDRange in the x-dimension</td>
<td>gridDim.x × blockDim.x</td>
</tr>
<tr>
<td>get_local_size(0);</td>
<td>Size of each work group in the x-dimension</td>
<td>blockDim.x</td>
</tr>
</tbody>
</table>
A Simple Example Matrix Multiplication

A simple matrix multiplication example that illustrates the basic features of memory and thread management in OpenCL programs

- Private register usage
- Work item ID usage
- Memory data transfer API between host and device
- Assume square matrix for simplicity
Square Matrix-Matrix Multiplication

- $P = M \times N$ of size $WIDTH \times WIDTH$
  - Each **work item** calculates one element of $P$
  - $M$ and $N$ are loaded $WIDTH$ times from global memory
OpenCL Context

- Contains one or more devices
- OpenCL memory objects 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 / command queue(s)
- Memory transfers are associated with a command queue (thus a specific device)
OpenCL Device Command Execution

Application → Command → Cmd Queue

Cmd Queue → OpenCL Device

OpenCL Context
OpenCL Context Setup Code (simple)

```
cl_int clerr = CL_SUCCESS;
// create context including all available OpenCL devices
cl_context clctx = clCreateContextFromType(
    0, CL_DEVICE_TYPE_ALL, NULL, NULL, &clerr);

size_t parmsz;
// query number of devices in context
clerr = clGetContextInfo(
    clctx, CL_CONTEXT_DEVICES, 0, NULL, &parmsz);
// now that size is known, allocate list for device info
cl_device_id* cldevs = (cl_device_id *) malloc(parmsz);
// query device info
clerr = clGetContextInfo(
    clctx, CL_CONTEXT_DEVICES, parmsz, cldevs, NULL);
// create command queue for first OpenCL device
cl_command_queue clcmdq = clCreateCommandQueue(
    clctx, cldevs[0], 0, &clerr);
```
Data Allocation

- `clCreateBuffer();`
- Requires five parameters
  - OpenCL context
  - Allocation and usage flags
  - Size in bytes
  - Host memory pointer
  - Returned error code
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
Device-to-Host Data Transfer

- clEnqueueReadBuffer();
- memory data transfer to host
- Requires nine parameters
  - OpenCL command queue pointer
  - Destination OpenCL memory buffer
  - Blocking flag
  - Offset in bytes
  - Size in bytes of written data
  - Destination host memory pointer
  - List of events to be completed before execution of this command
  - Event object tied to this command
OpenCL Memory Systems

- **__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
- Programmer manages device memory explicitly

### OpenCL Memory Types vs. CUDA Equivalents

<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>
void MatrixMulOnDevice(float* M, float* N, float* P, int Width) {
    int size = Width * Width * sizeof(float);
    cl_mem Md, Nd, Pd;
    Md = clCreateBuffer(cl_ctxt, CL_MEM_READ_WRITE, mem_size_M, NULL, NULL);
    Nd = clCreateBuffer(cl_ctxt, CL_MEM_READ_WRITE, mem_size_N, NULL, &ciErrNum);
    Pd = clCreateBuffer(cl_ctxt, CL_MEM_READ_WRITE, mem_size_P, NULL, NULL);
    clEnqueueWriteBuffer(clcmdque, Md, CL_FALSE, 0, mem_size_M, 
                        (const void *)M, 0, 0, NULL);
    clEnqueueWriteBuffer(clcmdque, Nd, CL_FALSE, 0, mem_size_N, 
                        (const void *)N, 0, 0, NULL);
    clEnqueueWriteBuffer(clcmdque, Pd, CL_FALSE, 0, mem_size_P, 
                        (const void *)P, 0, 0, NULL);
Output Matrix Data Transfer (Host-side Code)

2. // Kernel invocation code – to be shown later

3. // Read P from the device
   clEnqueueReadBuffer(cmdque, Pd, CL_FALSE, 0, mem_size_P, (void*)P), 0, 0, &ReadDone);

   // Free device matrices
   clReleaseMemObject(Md);
   clReleaseMemObject(Nd);
   clReleaseMemObject(Pd);
}
Matrix Multiplication Using Multiple Work Groups

- Break up $P_d$ into tiles
- Each work group calculates one tile
  - Each work item calculates one element
  - Set work group size to tile size
A Very Small Example

WIDTH = 4; TILE_WIDTH = 2
Each work group has 2*2 = 4 work items

WIDTH/TILE_WIDTH = 2
Use 2*2 = 4 work groups
__kernel void MatrixMulKernel(__global float* Md, __global float* Nd, __global float* Pd, int Width) {
    // Calculate the row index of the Pd element and M
    int Row = get_global_id(1);
    // Calculate the column index of Pd and N
    int Col = get_global_id(0);

    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k) {
        Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];
    }
    Pd[Row*Width+Col] = Pvalue;
}
// Setup the execution configuration
size_t cl_DimBlock[2], cl_DimGrid[2];
cl_DimBlock[0] = TILE_WIDTH;
cl_DimBlock[1] = TILE_WIDTH;
cl_DimGrid[0] = Width;
cl_DimGrid[1] = Width;
clSetKernelArg(clkern, 0, sizeof (cl_mem), (void*)(&deviceP));
clSetKernelArg(clkern, 1, sizeof (cl_mem), (void*)(&deviceM));
clSetKernelArg(clkern, 2, sizeof (cl_mem), (void*)(&deviceN));
clSetKernelArg(clkern, 3, sizeof (int), (void *)(&Width));

// Launch the device kernel
clEnqueueNDRangeKernel(clcmandque, clkern, 2, NULL,
    cl_DimGrid, cl_DimBlock, 0, NULL,
    &DeviceDone);
A Real Application Example -- Electrostatic Potential Maps
Electrostatic Potential Maps

- Electrostatic potentials evaluated on 3-D lattice:
  \[ V_i = \sum_j \frac{q_j}{4\pi\epsilon_0|\mathbf{r}_j - \mathbf{r}_i|} \]

- Applications include:
  - Ion placement for structure building
  - Time-averaged potentials for simulation
  - Visualization and analysis

Isoleucine tRNA synthetase
At each lattice point, sum potential contributions for all atoms in the simulated structure:

\[
potential[j] += \frac{\text{charge}[i]}{r_{ij}}
\]

- \(potential[j]\): potential at lattice point being evaluated
- \(atom[i]\):
- \(r_{ij}\): distance from lattice point \(j\) to atom \([i]\)
DCS Data Parallel Decomposition

Unrolling increases computational tile size

Work groups: 64-256 threads

Work items compute up to 8 potentials, skipping by memory coalescing width

Padding waste

(unrolled, coalesced)
NDRange of work groups

Copyright © 2013 by Yong Cao, Referencing UIUC ECE408/498AL Course Notes
Direct Coulomb Summation in OpenCL

NDRange containing all work items, decomposed into work groups

Lattice padding

Work groups: 64-256 threads

Work items compute up to 8 potentials, skipping by memory coalescing width

Copyright © 2013 by Yong Cao, Referencing UIUC ECE408/498AL Course Notes
Direct Coulomb Summation Kernel Setup

**OpenCL:**

```c
__kernel void clenergy(...) {
    unsigned int xindex = (get_global_id(0) -
                           get_local_id(0)) * UNROLLX +
                           get_local_id(0);
    unsigned int yindex = get_global_id(1);
    unsigned int outaddr = get_global_size(0) *
                           UNROLLX * yindex + xindex;
```

**CUDA:**

```c
__global__ void cuenergy (...) {
    unsigned int xindex = blockIdx.x *
                          blockDim.x * UNROLLX +
                          threadIdx.x;
    unsigned int yindex = blockIdx.y *
                          blockDim.y + threadIdx.y;
    unsigned int outaddr = gridDim.x *
                           blockDim.x * UNROLLX *
                           yindex + xindex;
```
for (atomid=0; atomid<numatoms; atomid++) {
    float dy = coory - atominfo[atomid].y;
    float dyz2 = (dy * dy) + atominfo[atomid].z;
    float dx1 = coorx – atominfo[atomid].x;
    float dx2 = dx1 + gridspacing_coalesce;
    float dx3 = dx2 + gridspacing_coalesce;
    float dx4 = dx3 + gridspacing_coalesce;
    float charge = atominfo[atomid].w;
    energyvalx1 += charge * rsqrtf(dx1*dx1 + dyz2);
    energyvalx2 += charge * rsqrtf(dx2*dx2 + dyz2);
    energyvalx3 += charge * rsqrtf(dx3*dx3 + dyz2);
    energyvalx4 += charge * rsqrtf(dx4*dx4 + dyz2);
}
DCS Inner Loop (OpenCL on NVIDIA GPU)

...for (atomid=0; atomid<numatoms; atomid++) {
    float dy = coory - atominfo[atomid].y;
    float dyz2 = (dy * dy) + atominfo[atomid].z;
    float dx1 = coorx - atominfo[atomid].x;
    float dx2 = dx1 + gridspacing_coalesce;
    float dx3 = dx2 + gridspacing_coalesce;
    float dx4 = dx3 + gridspacing_coalesce;
    float charge = atominfo[atomid].w;
    energyvalx1 += charge * native_rsqrt(dx1*dx1 + dyz2);
    energyvalx2 += charge * native_rsqrt(dx2*dx2 + dyz2);
    energyvalx3 += charge * native_rsqrt(dx3*dx3 + dyz2);
    energyvalx4 += charge * native_rsqrt(dx4*dx4 + dyz2);
}
DCS Inner Loop (OpenCL on AMD CPU)

```c
float4 gridspacing_u4 = { 0.f, 1.f, 2.f, 3.f };  
gridspacing_u4 *= gridspacing_coalesce;  
float4 energyvalx=0.0f;

...  
for (atomid=0; atomid<numatoms; atomid++) {  
  float dy = coory - atominfo[atomid].y;  
  float dyz2 = (dy * dy) + atominfo[atomid].z;  
  float4 dx = gridspacing_u4 + (coorx - atominfo[atomid].x);  
  float charge = atominfo[atomid].w;  
  energyvalx1 += charge * native_rsqrt(dx1*dx1 + dyz2);
}
```
Why Two Different OpenCL Kernels???

- Existing OpenCL implementations don’t necessarily auto-vectorize your code for the native hardware’s SIMD vector width
- Although you can run the same code on very different devices and get the correct answer, performance will vary wildly...
- In many cases, getting peak performance on multiple device types or hardware from different vendors currently requires multiple OpenCL kernels
OpenCL Host Code

- Roughly analogous to CUDA driver API:
  - Memory allocations, memory copies, etc
  - Create and manage device context(s) and associated work queue(s), etc...
  - OpenCL uses reference counting on all objects

- OpenCL programs are normally compiled entirely at runtime, which must be managed by host code
OpenCL Kernel Compilation Example

const char* clenergysrc =
“__kernel __attribute__((
(reqd_work_group_size_hint(BLOCKSIZEX, BLOCKSIZEY, 1)))

“void clenergy(int numatoms, float gridspacing, __global float
*energy, __constant float4 *atominfo) { 

cl_program clpgm;

clpgm = clCreateProgramWithSource(clctx, 1, &clenergysrc, NULL, &clerr);
char clcompileflags[4096];
sprintf(clcompileflags, "-DUNROLLX=%d -cl-fast-relaxed-math -cl-single-
precision-constant -cl-denorms-are-zero -cl-mad-enable", UNROLLX);
clerr = clBuildProgram(clpgm, 0, NULL, clcompileflags, NULL, NULL);
cl_kernel clkern = clCreateKernel(clpgm, "clenergy", &clerr);

OpenCL kernel source code as a big string

Gives raw source code string(s) to OpenCL

Set compiler flags, compile source, retrieve handle to the "clenergy" kernel
Host Code: OpenCL Kernel Launch

1. `doutput = clCreateBuffer(clctx, CL_MEM_READ_WRITE, volmemsz, NULL, NULL);`
2. `datominfo = clCreateBuffer(clctx, CL_MEM_READ_ONLY, MAXATOMS * sizeof(cl_float4), NULL, NULL);`

... 
3. `clerr = clSetKernelArg(clkern, 0, sizeof(int), &runatoms);`
4. `clerr = clSetKernelArg(clkern, 1, sizeof(float), &zplane);`
5. `clerr = clSetKernelArg(clkern, 2, sizeof(cl_mem), &doutput);`
6. `clerr = clSetKernelArg(clkern, 3, sizeof(cl_mem), &datominfo);`
7. `cl_event event;`
8. `clerr = clEnqueueNDRangeKernel(clcmdq, clkern, 2, NULL, Gsz, Bsz, 0, NULL, &event);`
9. `clerr = clWaitForEvents(1, &event);`
10. `clerr = clReleaseEvent(event);`

... 
11. `clEnqueueReadBuffer(clcmdq, doutput, CL_TRUE, 0, volmemsz, energy, 0, NULL, NULL);`
12. `clReleaseMemObject(doutput);`
13. `clReleaseMemObject(datominfo);`
To Learn More

- Khronos OpenCL headers, specification, etc: [http://www.khronos.org/registry/cl/](http://www.khronos.org/registry/cl/)
- Khronos OpenCL samples, tutorials, etc: [http://www.khronos.org/developers/resources/opencl/](http://www.khronos.org/developers/resources/opencl/)
- Chapter 11 of our textbook