CSE 599 I
Accelerated Computing - Programming GPUs
OpenCL / OpenACC
Lecture 20 – Related Programming Models: OpenCL
Lecture 20.1 - OpenCL Data Parallelism Model
Objective

- To Understand the OpenCL programming model
  - basic concepts and data types
  - Kernel structure
  - Application programming interface
  - Simple examples
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 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

```
int id = get_global_id(0);
result[id] = a[id] + b[id];
...%_1%
```

<table>
<thead>
<tr>
<th>work group 0</th>
<th>work group 1</th>
<th>work group 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7</td>
<td>8 9 10 11 12 13 14 15</td>
<td>56 57 58 59 60 61 62 63</td>
</tr>
</tbody>
</table>
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 counterpart 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>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>
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
Module 20 – Related Programming Models: OpenCL
Lecture 20.2 - OpenCL Device Architecture
Objective

- To Understand the OpenCL device architecture
  - Foundation to terminology used in the host code
  - Also needed to understand the memory model for kernels
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

Compute Device

Compute unit 1

Private memory 1

PE 1

Local memory 1

Private memory M

PE M

Compute unit N

Private memory 1

PE 1

Local memory N

Private memory M

PE M

Global/Constant Memory Data Cache

Global Memory

Constant Memory

Compute Device Memory
## 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>global memory</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>constant memory</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>local memory</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>private memory</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>
OpenCL Context

- Contains one or more devices
- OpenCL device memory objects are associated with a context, not a specific device
Module 20 – Related Programming Models: OpenCL
Lecture 20.3 - OpenCL Host Code
Objective

- To learn to write OpenCL host code
  - Create OpenCL context
  - Create work queues for task parallelism
  - Device memory Allocation
  - Kernel compilation
  - Kernel launch
  - Host-device data copy
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) {
  \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 kernel source code as a big string

Gives raw source code string(s) to OpenCL

Set compiler flags, compile source, and retrieve a handle to the “vadd” kernel
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

VECTOR_SIZE = 1024;
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 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
    - 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
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:

```c
  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`. 
Device Memory Allocation and Data Transfer for vadd

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

```c
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

```c
cl_event event=NULL;
clerr= clEnqueueNDRangeKernel(clcmdq, clkern, 2, NULL,
    Gsz, Bsz, 0, NULL, &event);
clerr= clWaitForEvents(1, &event);
cEnqueueReadBuffer(clcmdq, d_C, CL_TRUE, 0,
    N*sizeof(float), h_C, 0, NULL, NULL);
clReleaseMemObject(d_A);
cReleaseMemObject(d_B);
cReleaseMemObject(d_C);
```
Lecture 21.1 - Related Programming Models: OpenACC

Introduction to OpenACC
Objective

- To understand the OpenACC programming model
  - basic concepts and pragma types
  - simple examples
OpenACC

- The OpenACC Application Programming Interface provides a set of
  - compiler directives (pragmas)
  - library routines and
  - environment variables
  that can be used to write data parallel Fortran, C and C++ programs
  that run on accelerator devices including GPUs and CPUs
OpenACC Pragmas

– In C and C++, the #pragma directive is the method to provide to the compiler information that is not specified in the standard language.
  – These pragmas extend the base language
void VecAdd(float *__restrict__ output, const float * input1, const float * input2, int inputLength) {
    #pragma acc parallel loop copyin(input1[0:inputLength], input2[0:inputLength]),
    copyout(output[0:inputLength])
    for(i = 0; i < inputLength; ++i) {
        output[i] = input1[i] + input2[i];
    }
}
Simple Matrix-Matrix Multiplication in OpenACC

1. void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw) 
2. {
3.   #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Mw*Nw]) copyout(P[0:Mh*Nw])
4.   for (int i=0; i<Mh; i++) {
5.     #pragma acc loop
6.       for (int j=0; j<Nw; j++) {
7.         float sum = 0;
8.         for (int k=0; k<Mw; k++) {
9.             float a = M[i*Mw+k];
10.            float b = N[k*Nw+j];
11.               sum += a*b;
12.         }
13.         P[i*Nw+j] = sum;
14.     }
15.   }
16. }


Some Observations (1)

1. void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw)
2. {
3.   #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Mw*Nw]) copyout(P[0:Mh*Nw])
4.   for (int i=0; i<Mh; i++) {
5.       #pragma acc loop
6.         for (int j=0; j<Nw; j++) {
7.             float sum = 0;
8.             for (int k=0; k<Mw; k++) {
9.                 float a = M[i*Mw+k];
10.                float b = N[k*Nw+j];
11.                 sum += a*b;
12.            }
13.            P[i*Nw+j] = sum;
14.       }
15.   }
Some Observations (2)

1. void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw)
2. {
3.    #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Mw*Nw]) copyout(P[0:Mh*Nw])
4.    for (int i=0; i<Mh; i++) {
5.        #pragma acc loop
6.        for (int j=0; j<Nw; j++) {
7.            float sum = 0;
8.            for (int k=0; k<Mw; k++) {
9.                float a = M[i*Mw+k];
10.               float b = N[k*Nw+j];
11.               sum += a*b;
12.             }
13.            P[i*Nw+j] = sum;
14.        }
15.    }
16. }

The #pragma at line 3 tells the compiler to generate code for the ‘i’ loop at line 4 through 15 so that the loop iterations are executed at the first level of parallelism on the accelerator.
Some Observations (3)

1. void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw)
2. {
3.   #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Mw*Nw]) copyout(P[0:Mh*Nw])
4.   for (int i=0; i<Mh; i++) {
5.     #pragma acc loop
6.       for (int j=0; j<Nw; j++) {
7.         float sum = 0;
8.         for (int k=0; k<Mw; k++) {
9.           float a = M[i*Mw+k];
10.          float b = N[k*Nw+j];
11.          sum += a*b;
12.        }
13.        P[i*Nw+j] = sum;
14.      }
15.   }
16. }

The copyin() clause and the copyout() clause specify how the compiler should arrange for the matrix data to be transferred between the host and the accelerator.
Some Observations (4)

1. void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw)
2. {
3. #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Mw*Nw]) copyout(P[0:Mh*Nw])
4.  for (int i=0; i<Mh; i++) {
5.    #pragma acc loop
6.      for (int j=0; j<Nw; j++) {
7.        float sum = 0;
8.        for (int k=0; k<Mw; k++) {
9.          float a = M[i*Mw+k];
10.         float b = N[k*Nw+j];
11.         sum += a*b;
12.     }
13.     P[i*Nw+j] = sum;
14.   }
15. }
16. }

The #pragma at line 5 instructs the compiler to map the inner ‘j’ loop to the second level of parallelism on the accelerator.
Motivation

- OpenACC programmers can often start with writing a sequential version and then annotate their sequential program with OpenACC directives.
  - leave most of the details in generating a kernel, memory allocation, and data transfers to the OpenACC compiler.

- OpenACC code can be compiled by non-OpenACC compilers by ignoring the pragmas.
Frequently Encountered Issues

- Some OpenACC pragmas are hints to the OpenACC compiler, which may or may not be able to act accordingly
  - The performance of an OpenACC program depends heavily on the quality of the compiler.
  - It may be hard to figure out why the compiler cannot act according to your hints
  - The uncertainty is much less so for CUDA or OpenCL programs
OpenACC Device Model

Currently OpenACC does not expose synchronization across threads to the programmers.
OpenACC Execution Model
Lecture 21.2 - Related Programming Models: OpenACC

OpenACC Subtleties
Objective

– To understand some important and sometimes subtle details in OpenACC programming
  – parallel loops
  – simple examples to illustrate basic concepts and functionalities
Parallel vs. Loop Constructs

```c
#pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Mw*Nw])
copyout(P[0:Mh*Nw])
for (int i=0; i<Mh; i++) {
    ...
}
```

is equivalent to:

```c
#pragma acc parallel copyin(M[0:Mh*Mw]) copyin(N[0:Mw*Nw])
copyout(P[0:Mh*Nw])
{
    #pragma acc loop
    for (int i=0; i<Mh; i++) {
        ...
    }
}
```

(a parallel region that consists of a single loop)
More on Parallel Construct

```c
#pragma acc parallel copyout(a) num_gangs(1024) num_workers(32)
{
    a = 23;
}
```

1024*32 workers will be created. a=23 will be executed redundantly by all 1024 gang leads

- A parallel construct is executed on an accelerator
- One can specify the number of gangs and number of workers in each gang
  - Equivalent to CUDA blocks and threads
What Does Each “Gang Loop” Do?

```c
#pragma acc parallel num_gangs(1024)
{
    for (int i=0; i<2048; i++) {
        ...
    }
}

#pragma acc parallel num_gangs(1024)
{
    #pragma acc loop gang
    for (int i=0; i<2048; i++) {
        ...
    }
}
```
Worker Loop

#pragma acc parallel num_gangs(1024) num_workers(32)
{
    #pragma acc loop gang
    for (int i=0; i<2048; i++) {
        #pragma acc loop worker
        for (int j=0; j<512; j++) {
            foo(i,j);
        }
    }
}

1024*32=32K workers will be created, each executing 1M/32K = 32 instance of foo()
A More Substantial Example

- Statements 1, 3, 5, 6 are redundantly executed by 32 gangs

```c
#pragma acc parallel num_gangs(32)
{
    Statement 1;
    #pragma acc loop gang
    for (int i=0; i<n; i++) {
        Statement 2;
    }
    Statement 3;
    #pragma acc loop gang
    for (int i=0; i<m; i++) {
        Statement 4;
    }
    Statement 5;
    if (condition) Statement 6;
}
```
A More Substantial Example

- The iterations of the n and m for-loop iterations are distributed to 32 gangs
- Each gang could further distribute the iterations to its workers
  - The number of workers in each gang will be determined by the compiler/runtime

```c
#pragma acc parallel num_gangs(32)
{
    Statement 1;
    #pragma acc loop gang
    for (int i=0; i<n; i++) {
        Statement 2;
    }
    Statement 3;
    #pragma acc loop gang
    for (int i=0; i<m; i++) {
        Statement 4;
    }
    Statement 5;
    if (condition) Statement 6;
}
```
Avoiding Redundant Execution

- Statements 1, 3, 5, 6 will be executed only once
- Iterations of the n and m loops will be distributed to 32 workers

```c
#pragma acc parallel
num_gangs(1) num_workers(32)
{
    Statement 1;
    #pragma acc loop worker
    for (int i=0; i<n; i++) {
        Statement 2;
    }
    Statement 3;
    #pragma acc loop worker
    for (int i=0; i<m; i++) {
        Statement 4;
    }
    Statement 5;
    if (condition)  Statement 6;
}
```
Kernel Regions

- Kernel constructs are descriptive of programmer intentions
  - The compiler has a lot of flexibility in its use of the information
- This is in contrast with Parallel, which is prescriptive of the action for the compile follow

```c
#pragma acc kernels
{
  #pragma acc loop gang(1024)
  for (int i=0; i<2048; i++) {
    a[i] = b[i];
  }

  #pragma acc loop gang(512)
  for (int j=0; j<2048; j++) {
    c[j] = a[j]*2;
  }
  for (int k=0; k<2048; k++) {
    d[k] = c[k];
  }
}
```
Kernel Regions

- Code in a kernel region can be broken into multiple CUDA/OpenCL kernels
- The i, j, k loops can each become a kernel
  - The k-loop may even remain as host code
- Each kernel can have a different gang/worker configuration

```c
#pragma acc kernels
{
  #pragma acc loop gang(1024)
  for (int i=0; i<2048; i++) {
    a[i] = b[i];
  }

  #pragma acc loop gang(512)
  for (int j=0; j<2048; j++) {
    c[j] = a[j]*2;
  }

  for (int k=0; k<2048; k++) {
    d[k] = c[k];
  }
}
```
The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License.