GPU Teaching Kit

Accelerated Computing

Lecture 20 – Related Programming Models: OpenCL

Lecture 20.1 - OpenCL Data Parallelism Model

Objective

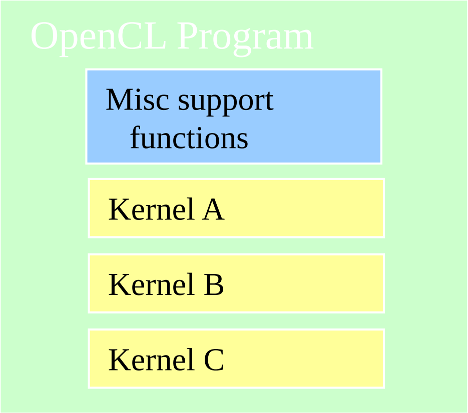
- To Understand the OpenCL programming model
  - basic concepts and data types
  - Kernel structure
  - Application programming interface
  - Simple examples

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

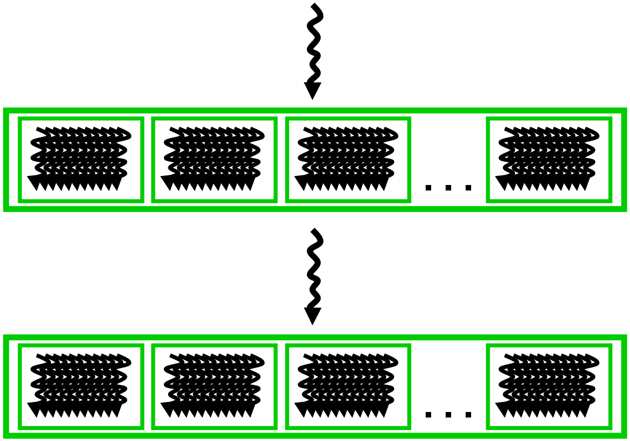
OpenCL Programs

– 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.

OpenCL Parallelism Concept	CUDA Equivalent
host	host
device	device
kernel	kernel
host program	host program
NDRange (index space)	grid
work item	thread
work group	block

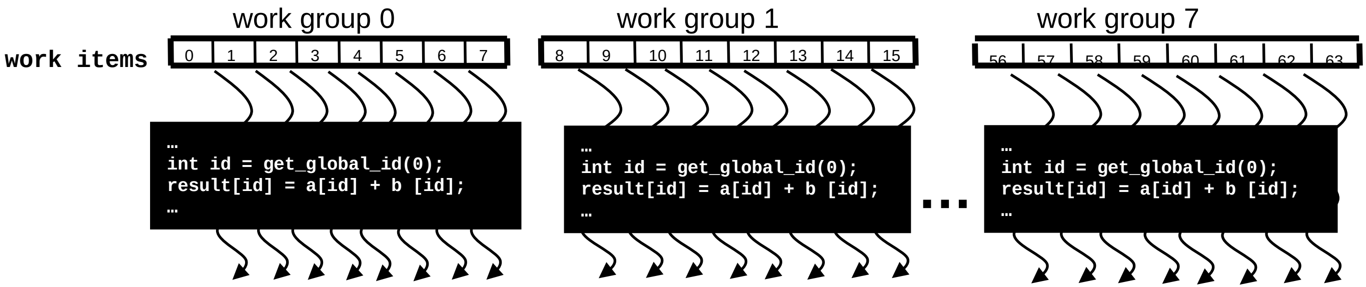
OpenCL Kernels

- 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

```
__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



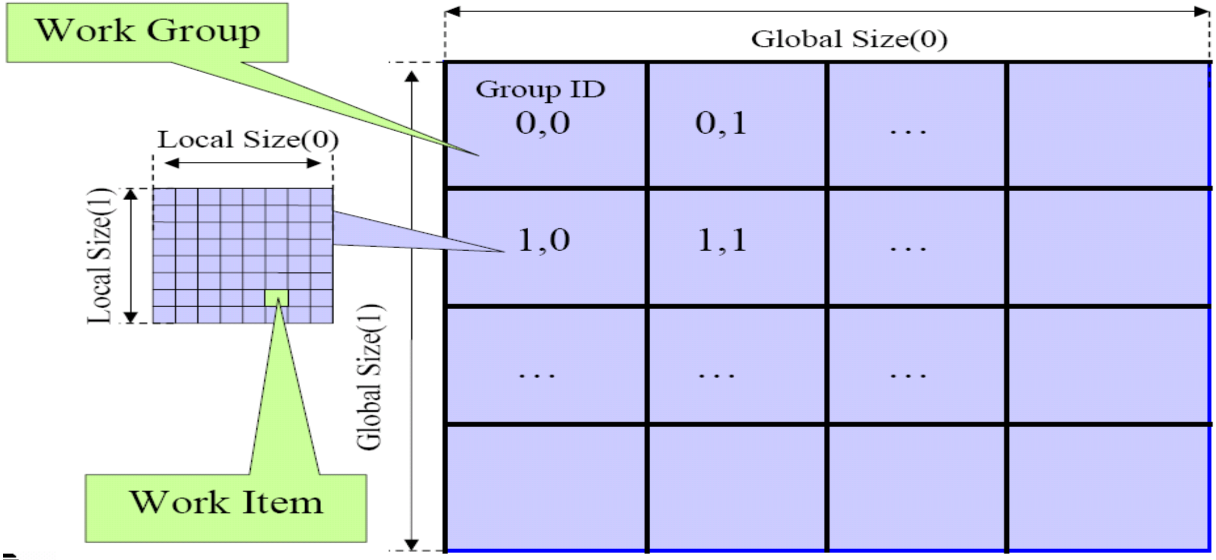
Work Groups: Scalable Cooperation

- 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

OpenCL API Call	Explanation	CUDA Equivalent
get_global_id(0);	global index of the work item in the x dimension	blockIdx.x*blockDim.x +threadIdx.x
get_local_id(0)	local index of the work item within the work group in the x dimension	threadIdx.x
get_global_size(0);	size of NDRange in the x dimension	gridDim.x*blockDim.x
get_local_size(0);	Size of each work group in the x dimension	blockDim.x

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

GPU Teaching Kit

Accelerated Computing

GPU Teaching Kit

Accelerated Computing

Module 20 – Related Programming Models: OpenCL

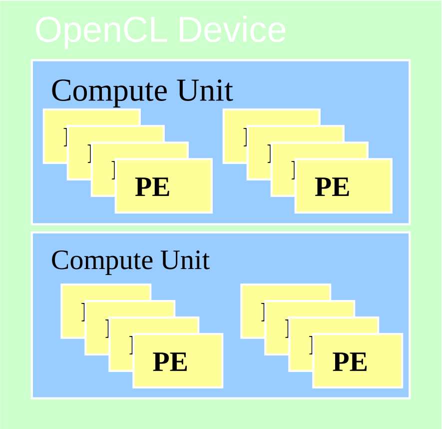
Lecture 20.2 - OpenCL Device Architecture

Objective

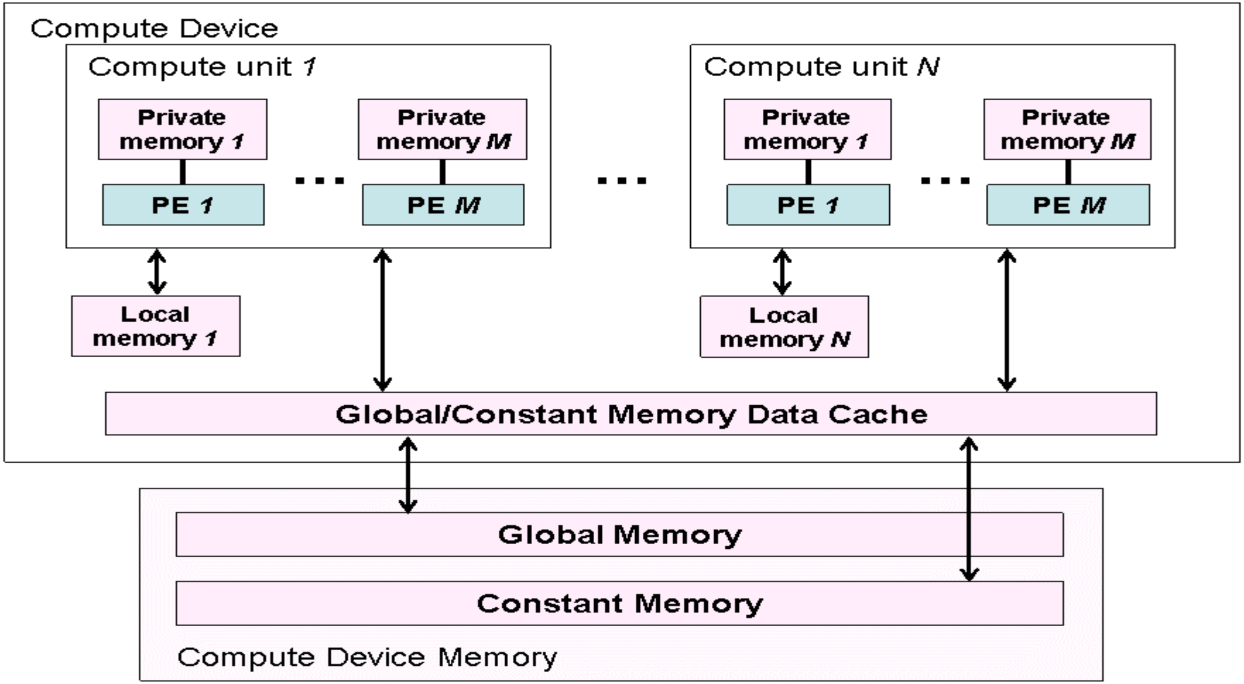
- To Understand the OpenCL device architecture
  - Foundation to terminology used in the host code
  - Also needed to understand the memory model for kernels

OpenCL Hardware Abstraction

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



OpenCL Device Architecture

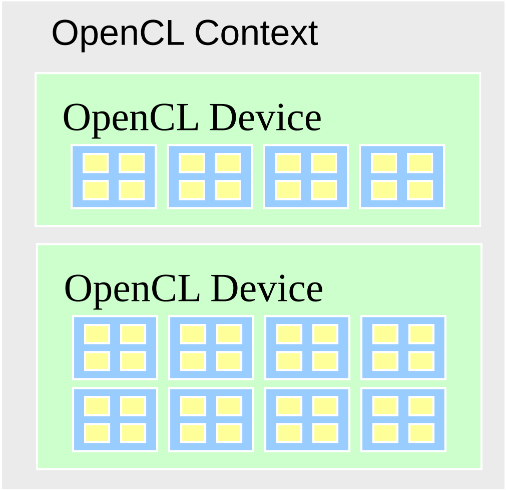


OpenCL Device Memory Types

Memory Type	Host access	Device access	CUDA Equivalent
global memory	Dynamic allocation; Read/write access	No allocation; Read/write access by all work items in all work groups, large and slow but may be cached in some devices.	**global memory**
constant memory	Dynamic allocation; read/write access	Static allocation; read-only access by all work items.	**constant memory**
local memory	Dynamic allocation; no access	Static allocation; shared read-write access by all work items in a work group.	**shared memory**
private memory	**No allocation; no access**	**Static allocation; Read/write access by a single work item.**	**registers and local memory**

OpenCL Context

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



NVIDIA

GPU Teaching Kit

Accelerated Computing

ILLINOIS

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

GPU Teaching Kit

Accelerated Computing

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)

```
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);
```

Where commands
will be sent.

OpenCL Kernel Compilation: vadd

OpenCL kernel source code as a big string

```
const char* vaddsrc =
   "__kernel void vadd(__global float *d_A, __global float *d_B,
__global float *d_C, int N) { \n"   […etc and so forth…]
```

Gives raw source code string(s) to OpenCL

```
cl_program clpgm;
clpgm = clCreateProgramWithSource(clctx, 1, &vaddsrc, NULL,
&clerr);
```

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

```
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

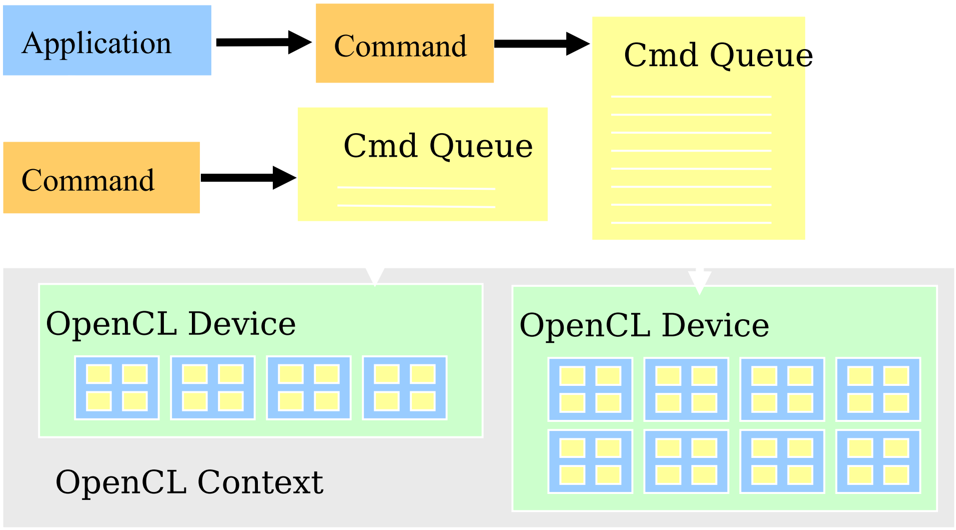
OpenCL Device Memory Allocation (cont.)

- 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

```
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



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

```
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.

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

```
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);
}
```

GPU Teaching Kit

Accelerated Computing