Lecture 3
Kernel programming

Outline

- OpenCL Review
- Data Transfer
- Kernel Programming
- Memory Model
- Synchronization
- Data Type

OpenCL Programming Flow

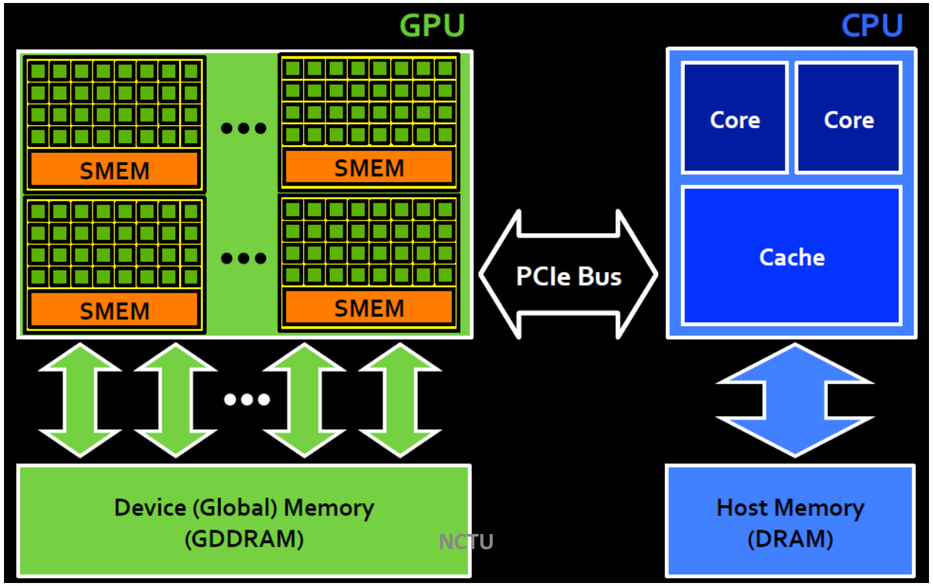
Standard Execute Flow

Step 1: Copy data to GPU memory

Step 2: Launch the kernels on GPU

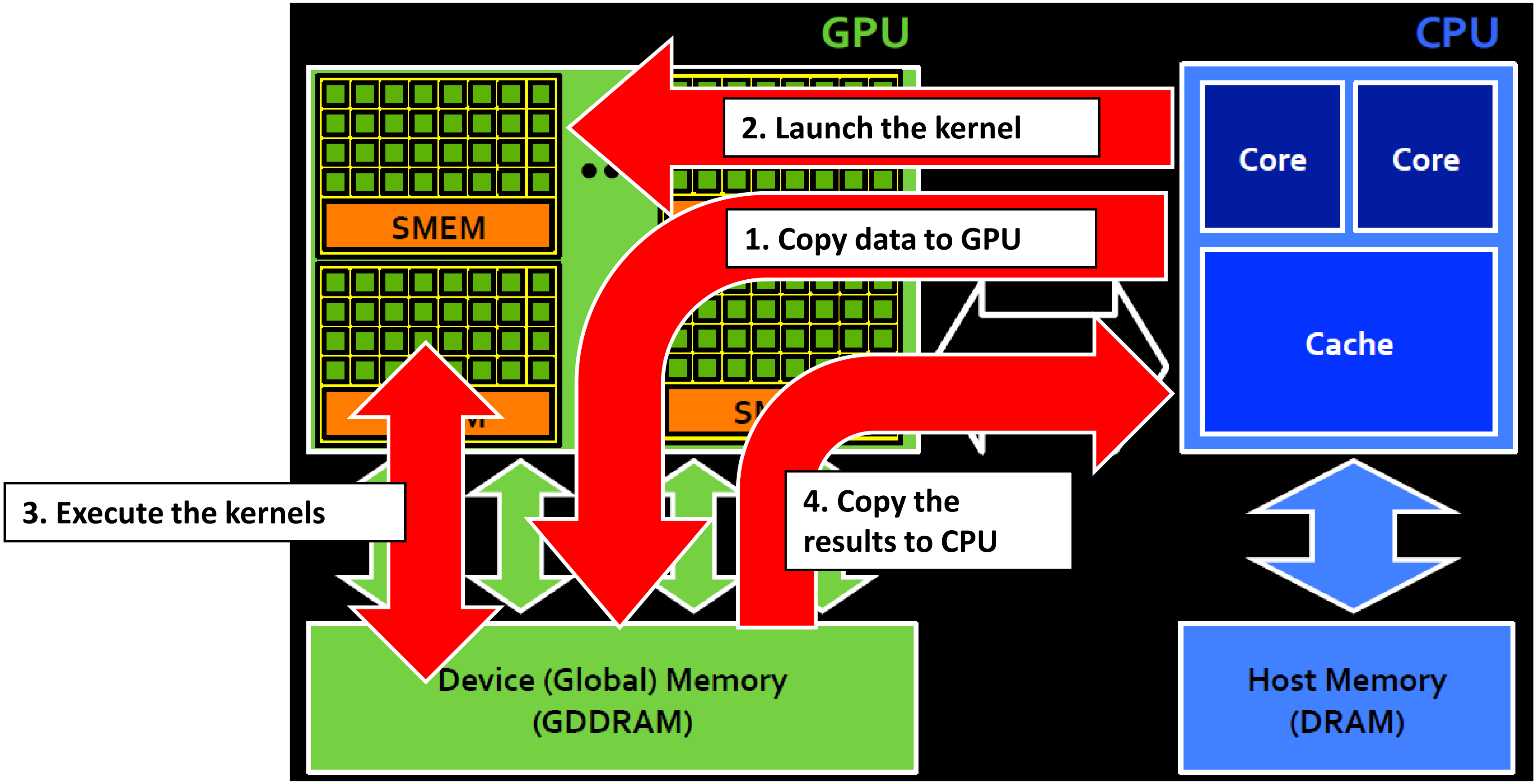
Step 3: Execute kernels on GPU

Step 4: Copy data to CPU memory



OpenCL Programming Flow

Standard Execute Flow



Outline

- OpenCL Review
- Data Transfer
- Kernel Programming
- Memory Model
- Synchronization
- Data Type

OpenCL Framework

Program Flow

1. Read Hello_world.cl file
2. Create OpenCL structure
3. Send the data to GPU
4. Send the task to GPU
   clEnqueueNDRangeKernel();
5. Get the result from GPU
6. Done!

- Create Platforms
- Create Devices
- Create Contexts
- Create Programs
- Create Kernels
- Create Buffers
- Create CommandQueue

Send the data to GPU

Setting Kernel Arguments

- **You have to use the function "*clSetKernelArg()*" to sent the data to the target device in OpenCL.**
  - *clSetKernelArg(cl_kernel kernel,*
    cl_uint index,
    size_t size,
    *const void *value)*

- **kernel: A valid kernel object.**
- **index: The argument index.**
- **size: Specifies the size of the argument value.**
- ***value: point to the data(4 types) that will be transferred to the kernel.**

OpenCL Framework

How to Send The Data to Device

1. Create memory buffer
2. Write the data into device's buffer
3. Use *clSetKernelArg* to the kernel.

- Kernel code

```
__kernel void
dp_mul( __global const float *a,
        __global const float *b,
        __global float *c)
{
   int id = get_global_id(0);
   c[id] = a[id] * b[id];
} // execute over n "work items"
```

- Host code

```
cl_mem bufA;
cl_mem bufB;
cl_mem bufC;
// Create a buffer object that will contain the data  from the host array A
bufA = clCreateBuffer(context, CL_MEM_READ_ONLY, datasize, NULL, &status);
// Create a buffer object that will contain the data  from the host array B
bufB = clCreateBuffer(context, CL_MEM_READ_ONLY, datasize, NULL, &status);
// Create a buffer object that will hold the output data
bufC = clCreateBuffer(context, CL_MEM_WRITE_ONLY, datasize, NULL, &status);
```

OpenCL Framework

How to Send The Data to Device

1. Create memory buffer
2. Write the data into device's buffer
3. Use *clSetKernelArg* to the kernel.

- **Kernel code**

```
__kernel void
dp_mul( __global const float *a,
        __global const float *b,
        __global float *c)
{
   int id = get_global_id(0);
   c[id] = a[id] * b[id];
}  // execute over n "work items"
```

- **Host code**

```
// Write input array A to the device buffer bufferA
status = clEnqueueWriteBuffer(cmdQueue, bufA, CL_FALSE, 0, datasize, A, 0, NULL, NULL);
// Write input array B to the device buffer bufferB
status = clEnqueueWriteBuffer(cmdQueue, bufB, CL_FALSE, 0, datasize, B, 0, NULL, NULL);
// Associate the input and output buffers with the kernel
status = clSetKernelArg(kernel, 0, sizeof(cl_mem), &bufA);
status = clSetKernelArg(kernel, 1, sizeof(cl_mem), &bufB);
status = clSetKernelArg(kernel, 2, sizeof(cl_mem), &bufC);
```

OpenCL Framework

How to Send The Task to Device

```
cl_int clEnqueueNDRangeKernel (cl_command_queue command_queue,
                               cl_kernel kernel,
                               cl_uint work_dim,
                               const size_t *global_work_offset,
                               const size_t *global_work_size,
                               const size_t *local_work_size,
                               cl_uint num_events_in_wait_list,
                               const cl_event *event_wait_list,
                               cl_event *event)
```

NDRange can be 1-, 2-, 3-dimemsions

- **Global_work_size**
  Define the number of global work-items in work_dim dimensions
- **Local_work_size**
  Define the number of work-items in work_dim dimensions make up a work-group

OpenCL Execution Model

Decompose Task Into W*ork-items*

- **Every kernel declaration must start with "__kernel".**
- **Every kernel function must return void.**
- **Four address space qualifiers.**
  - __global
  - __const
  - __local
  - __private

- **OpenCL C code**

```
__kernel void
dp_mul( __global const float *a,
        __global const float *b,
        __global float *c)
{
   int id = get_global_id(0);
   c[id] = a[id] * b[id];
}  // execute over n "work items"
```

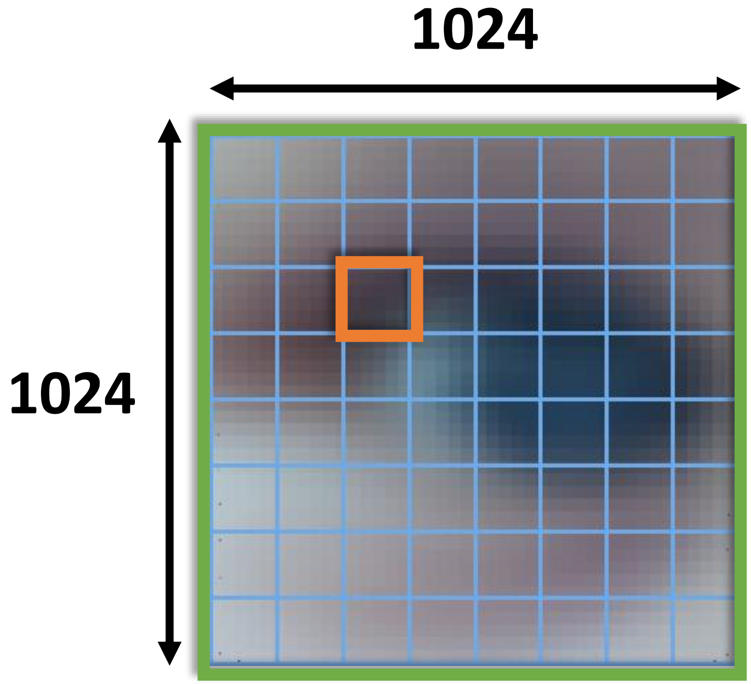
OpenCL Execution Model

An N-dimension Domain of Work-items

- Kernels are executed across a global domain of *work-items*
- Work-items are grouped into local domain of *work-groups*

Global dimensions: 1024 x 1024

Local dimensions: 128 x 128



Outline

- OpenCL Review
- Data Transfer
- Kernel Programming
- Memory Model
- Synchronization
- Data Type

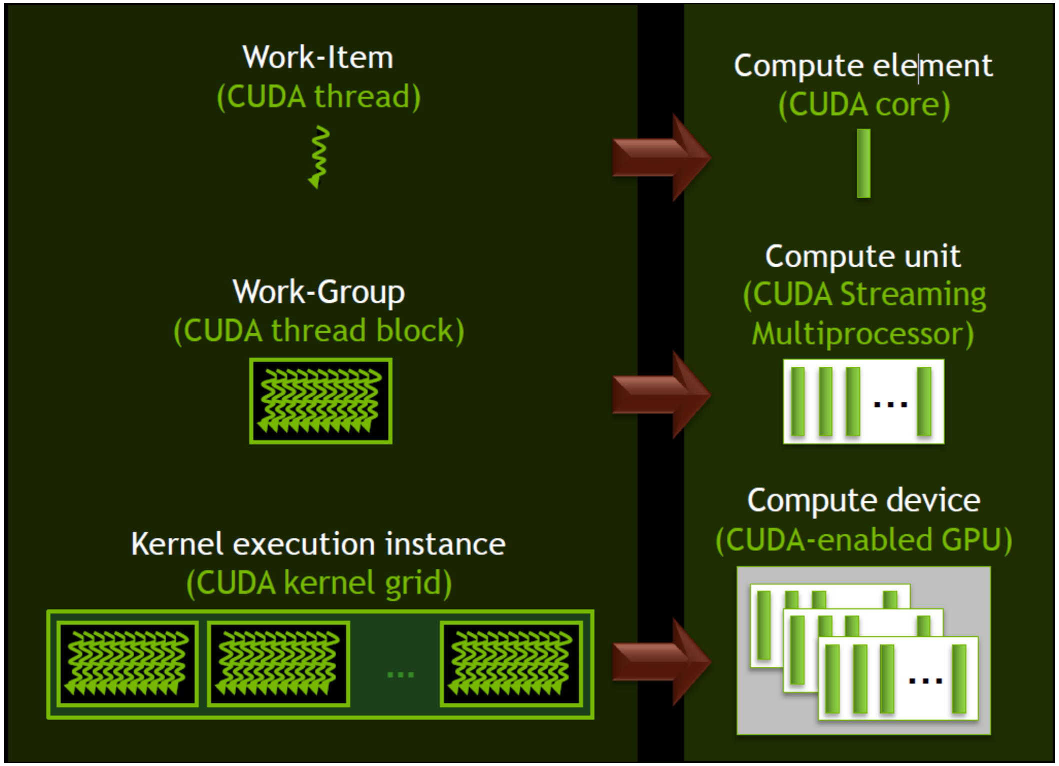
Kernel Programming

Work-item and Work-group

- **Work-item**
  - A work-item is a single implementation of the kernel on a specific set of data.
- **Work-group**
  - A work-group is a collection of work-items, and each has its own numeric identifier called the group ID.

Work-item

- **Definition:**
  The basic unit of work on an OpenCL device. Work-items are grouped into local work-groups



http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/06-intro_to_opencl.pdf

Work-group

- **Definition:**
  A work-group is a combination of work-items that access the same processing resources.

- **Advantages:**
  1. Work-items in a work-group can access the same block of high-speed memory called local memory.

  2. Work-items in a work-group can be synchronized using fences and barriers.

Kernel Programming

Work-item's Build-in Functions

- **Work-item function can be used to query the information relating to the data.**

```
// returns the unique global work-item ID for the specified dimension.
size_t get_global_id(dimidx);
// returns the unique local work-item ID in the work-group for the specified dimension.
size_t get_local_id(dimidx);
// returns the unique ID of the work-group  being processed by the kernel.
size_t get_group_id(dimidx);
// returns the number of dimensions of the data problem space.
uint get_work_dim()
// returns the number total work-items for the specified dimension.
size_t get_global_size(dimidx);
// returns the number of local work-items in the work-group specified dimension.
size_t get_local_size(dimidx);
```

Kernel Programming

Example: Vector addition

get_work_dim() = 1
get_global_size(0) = 8
get_global_id(0) = 5

	0	1	2	3	4	5	6	7
a	1	4	6	2	0	7	6	8
b	8	9	7	2	3	4	0	1
c	9	13	13	4	3	11	6	9

- **OpenCL C code**

```
__kernel void
dp_mul( __global const float *a,
        __global const float *b,
        __global float *c)
{
   int id = get_global_id(0);
   c[id] = a[id] * b[id];
}  // execute over n ”work items”
```

Kernel Program

Exa

a

b

c

9	13	13				

```
cl_int clEnqueueNDRangeKernel (
        cl_command_queue command_queue,
        cl_kernel kernel,
        cl_uint work_dim,
        const size_t *global_work_offset,
        const size_t *global_work_size,
        const size_t *local_work_size,
        cl_uint num_events_in_wait_list,
        const cl_event *event_wait_list,
        cl_event *event)
```

And now, If you set the local size = {4, 0, 0}.

```
a,
*b,
)
d = get_global_id(0);
c[id] = a[id] * b[id];
} // execute over n "work items"
```

Kernel Programming

Example: Vector addition

get_work_dim() = 1
get_global_size(0) = 8
get_global_id(0) = 5

get_local_size(0) = 4
get_local_id(0) = 1
get_group_id(0) = 1

	0	1	2	3	4	5	6	7
a	1	4	6	2	0	7	6	8
b	8	9	7	2	3	4	0	1
c	9	13	13	4	3	11	6	9

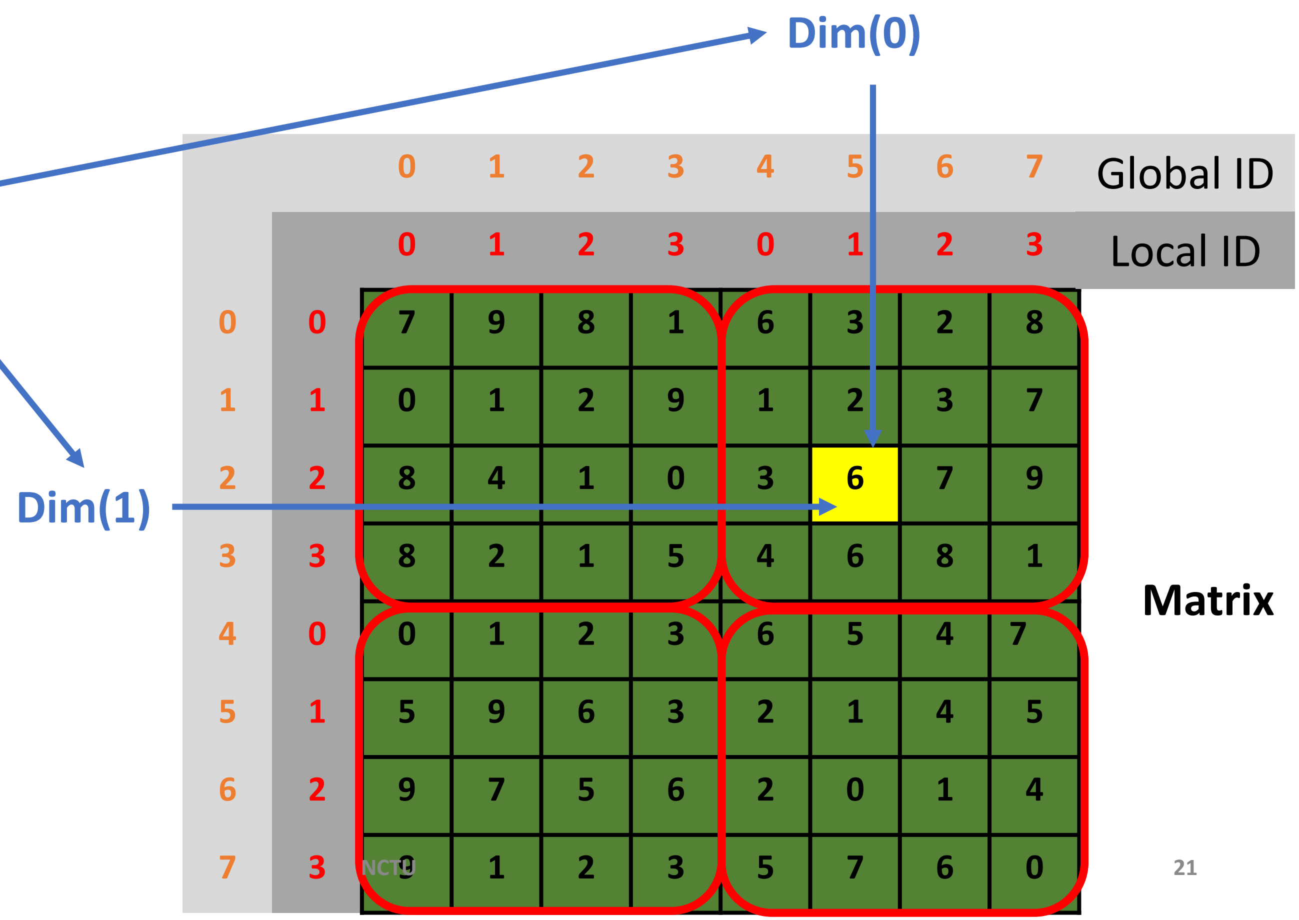
- OpenCL C code

```
__kernel void
dp_mul( __global const float *a,
        __global const float *b,
        __global float *c)
{
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
} // execute over n ”work items”
```

Kernel Programming

Example: 2D Matrix

get_work_dim() = 2
get_global_size(0) = 8
get_global_size(1) = 8
get_global_id(0) = 5
get_global_id(1) = 2
get_local_size(0) = 4
get_local_size(1) = 4
get_local_id(0) = 1
get_group_id(0) = 1



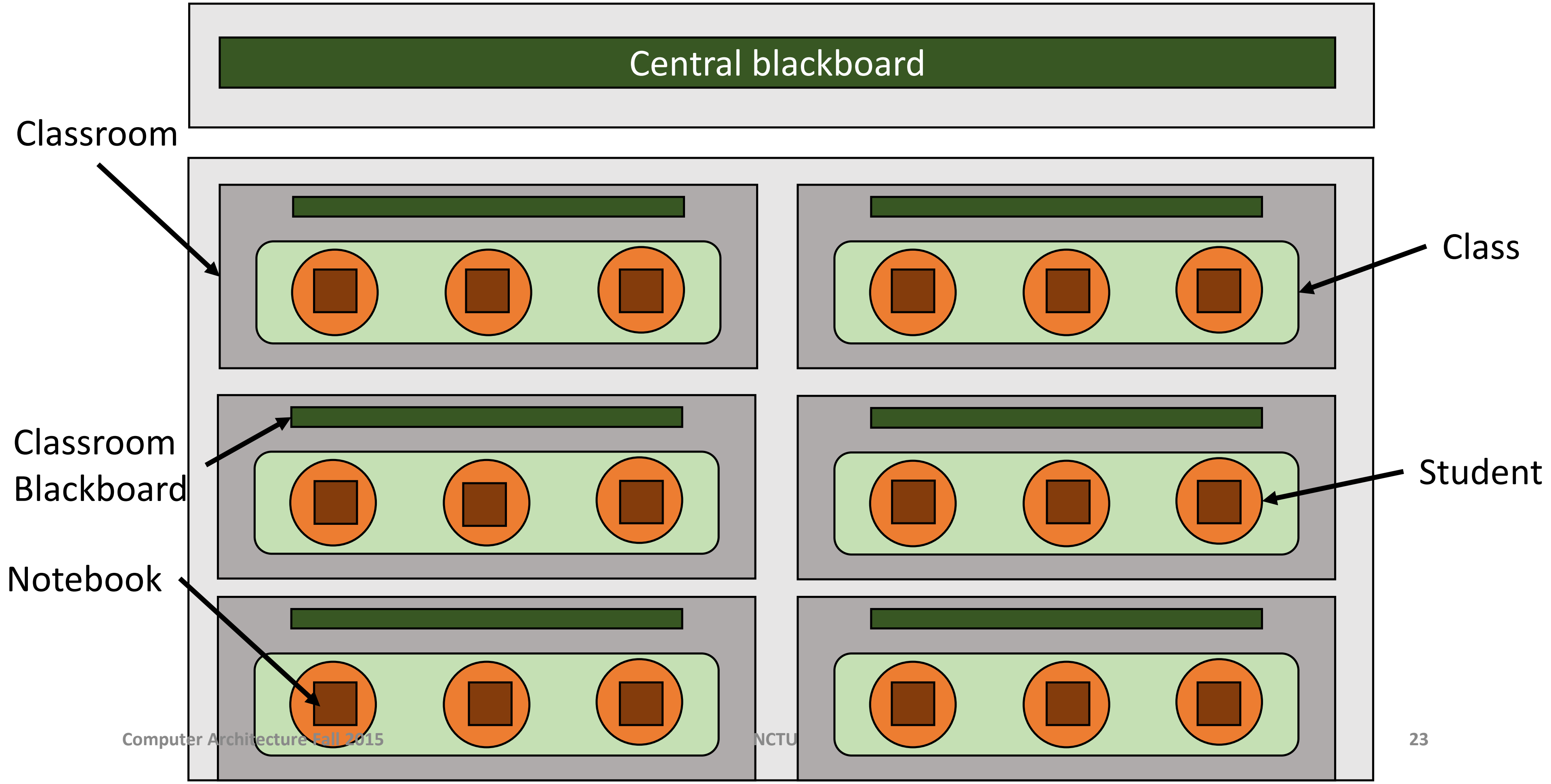
Brief summary

Work-item vs. Work-group

	Work-item	Work-group
Def.	Single implementation of the kernel	Combination of work-items
IDs	Global IDs and Local IDs	Group IDs
Memory model	Global/Constant/ Local/Private	Global/Constant/Local
Total number	The total number is called "Global size"	The number of work-items per group is called "Local size"

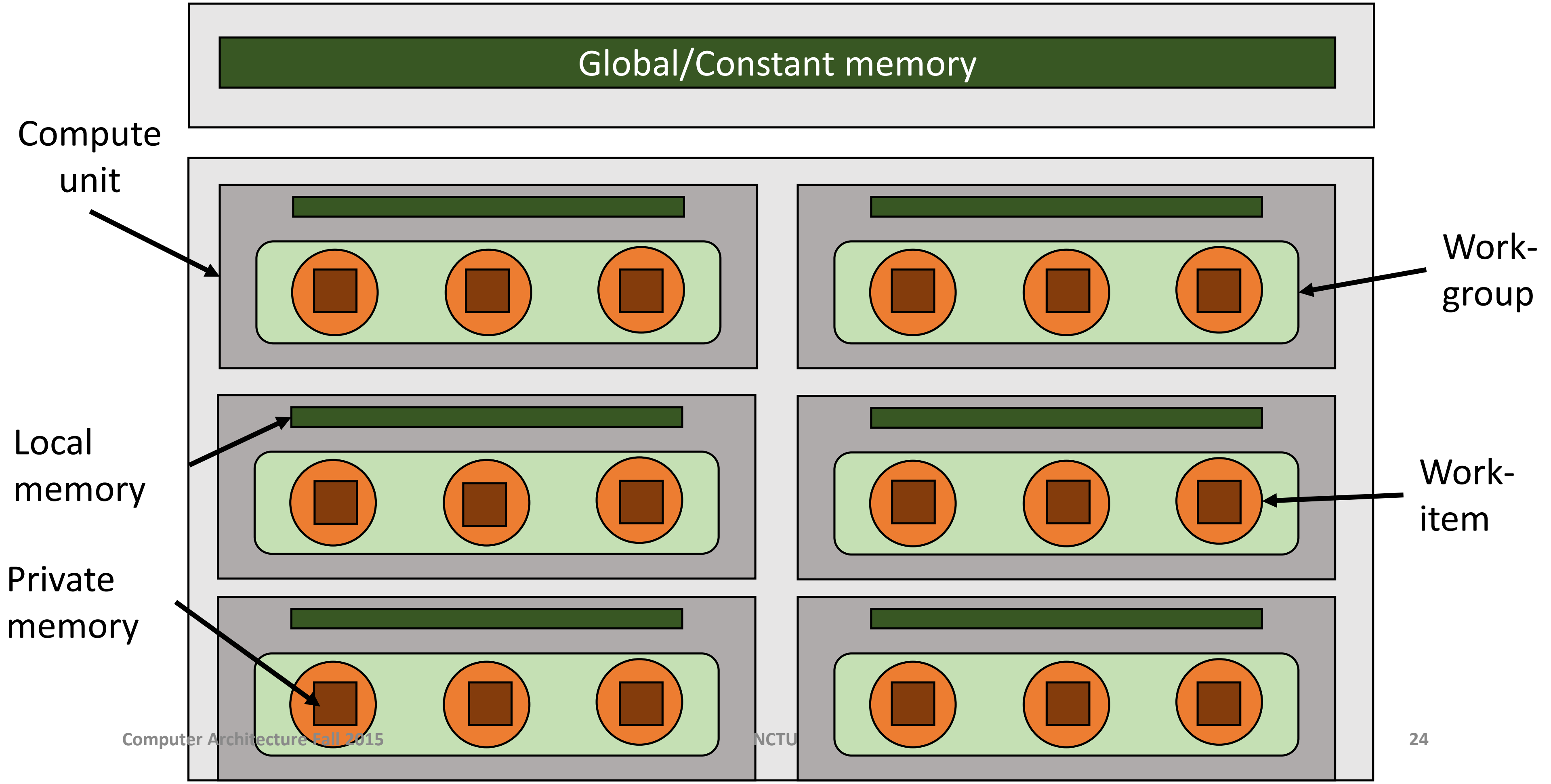
Work-item Analogy

Math Question in School



Work-item Analogy

Math Question in School



Outline

- OpenCL Review
- Data Transfer
- Kernel Programming
- Memory Model
- Synchronization
- Data Type

Memory Model of The Kernel

- **Global Memory**

  Can be read from all work items.
  It is physically the device's main memory.

- **Constant Memory**

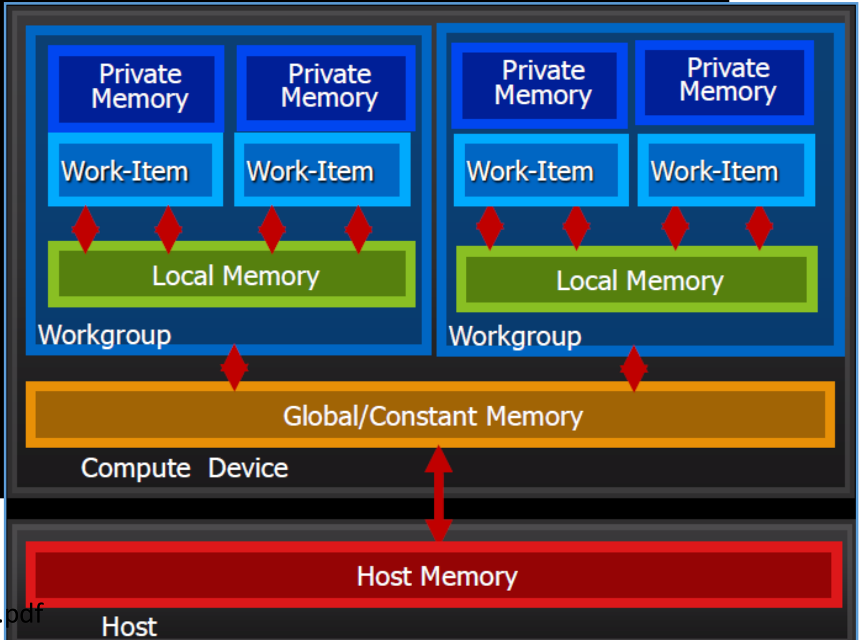
  Can be read from all work items. but can be used efficiently if the compute units contain hardware to support constant memory cache.

- **Local Memory**

  Can be read from work items within a work group.

- **Private Memory**

  Can only be used within each work item.



http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/06-intro_to_opencl.pdf

Memory Model of The Kernel

Global Memory

- The memory type visible to all work-items
- Usually the memory with largest capacity for a device but slowest memory system
- Mostly implemented with off-chip memory

Memory Model of The Kernel

Constant Memory

- **The read-only memory type visible to all work-items**

- **Advantages:**
  1. Used to broadcast data need no change.
  2. A little bit slower than local memory, but faster than global memory
  3. It helps to reduce the pressure of local memory

Memory Model of The Kernel

Local Memory

- **The memory type visible to the work-item in the same work-group**
- **Faster access time in comparison to global memory**
- **Mostly implemented with on-chip memory**
- **Advantages:**
  1. Share information among work items in the same work-groups
  2. Can be synchronized at the same time

Memory Model of The Kernel

Private Memory

- The memory type visible only to a work-item
- Fastest memory on a device
- No communication, so no synchronization
- Mostly implemented with register file

Outline

- OpenCL Review
- Data Transfer
- Kernel Programming
- Memory Model
- Synchronization
- Data Type

Synchronization Primitives

Work-group's Build-in Functions

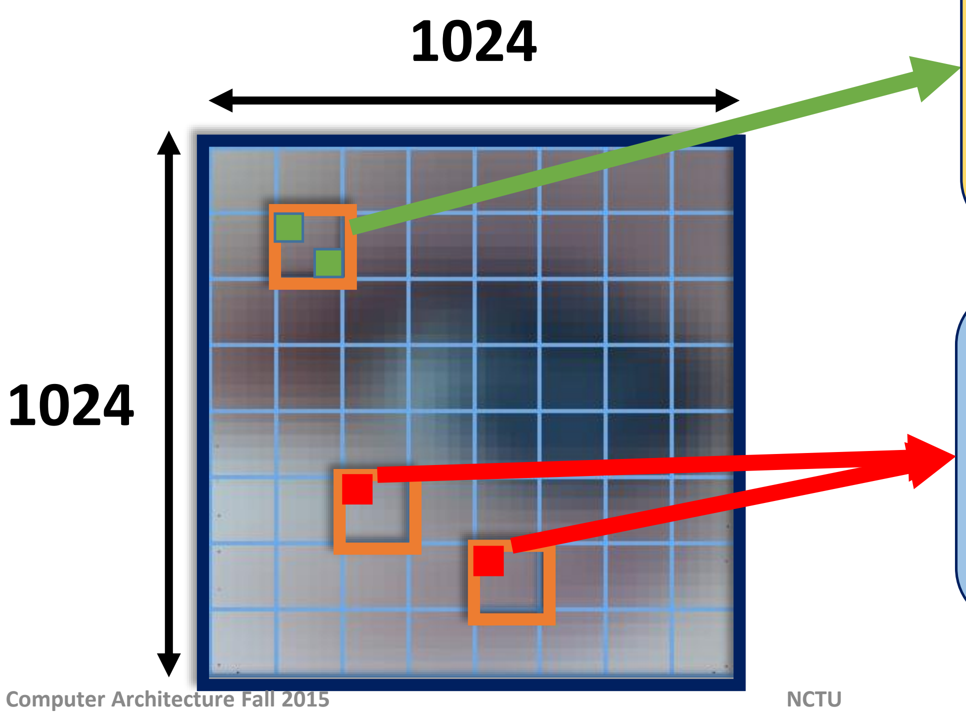
- **Built-in functions to order memory operations and synchronize execution.**
- **Memory fence flag can be *CLK_LOCAL_MEM_FENCE* or *CLK_GLOBAL_MEM_FENCE.***

```
// creates a barrier that blocks the current work-item until all others in the same group
// has executed.
void barrier(mem_fence_flag)
// ensure all reads and writes before the memory fence have committed to memory
void mem_fence(mem_fence_flag)
// ensures all reads before memory fence have completed
void read_mem_fence(mem_fence_flag)
// ensures all writes before memory fence have completed
void write_mem_fence(mem_fence_flag)
```

Synchronization Primitives

Example: 2-dimension

- **Global dimensions: 1024 x 1024**
- **Local dimensions: 128 x 128**



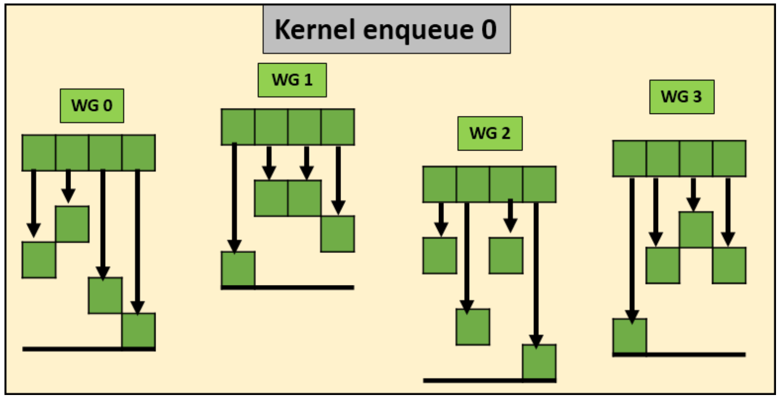
Synchronization between work-items possible only within workgroups: barriers and memory fences

Can not synchronize outside of a workgroup

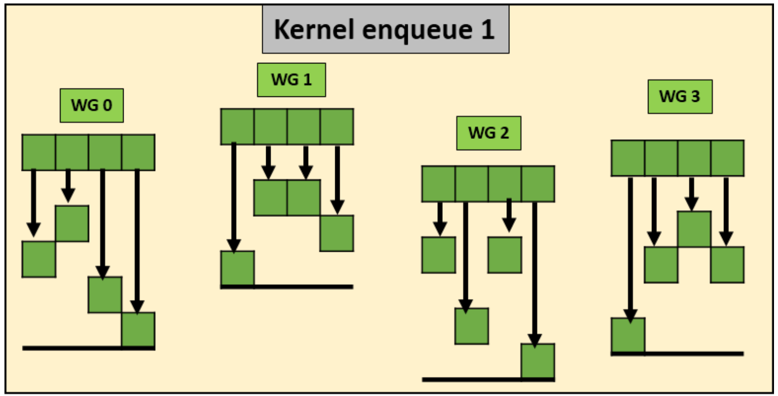
Synchronization Primitives

Barrier

- **Work-items in the same work-group can be synchronized with calls to the barrier function.**
- **OpenCL doesn't provide any functions that synchronize work-items in different WG.**



Global synchronization.
End of kernel 0,
start of kernel1



Outline

- OpenCL Review
- Data Transfer
- Kernel Programming
- Memory Model
- Synchronization
- Data Type

Data Types

Supported Features

- Scalar data types
  - char , uchar, short, ushort, int, uint, long, ulong,
  - bool, intptr_t, ptrdiff_t, size_t, uintptr_t, void, half (storage)
- Image types
  - image2d_t, image3d_t, sampler_t
- Vector data types
  - Vector length of 2, 4, 8, and 16
  - Aligned at vector length
  - Vector operations and built-in

Vector Data Types

- **Vectors resemble arrays in that they contain multiple components of the same type.**

- **But there are two important differences**
  1. A vector of a given type can only contain a specific number of elements.
  2. when a vector is operated upon, every element is operated upon at the same time.

Vector Data Types

Example

- Traditional C code

```
float a[4], b[4], c[4];
for(int i=0; i<4; i++) {
    c[i] = a[i] + b[i];
}
```

- OpenCL vector type

```
float4 a, b, c;
c = a + b;
```

Simpler and faster!!

Vector Data Types

- **xxxn is the vector length that could be 2, 3, 4, 8, 16.**
- **For example, a 4 component floating point vector would be** *float4*

Vector data type	Description
charn	**Vector containing n 8-bit signed two's complement integers**
ucharn	**Vector containing n 8-bit unsigned two's complement integers**
shortn	**Vector containing n 16-bit signed two's complement integers**
ushortn	**Vector containing n 16-bit unsigned two's complement integers**
intn	**Vector containing n 32-bit signed two's complement integers**
uintn	**Vector containing n 32-bit unsigned two's complement integers**
longn	**Vector containing n 64-bit signed two's complement integers**
ulongn	**Vector containing n 64-bit unsigned two's complement integers**
floatn	**Vector containing n 32-bit single-precision floating-point values**

Vector Data Types

Preferred Vector Widths

- **However, not every device can process large vectors.**
- **For example, a *float16* is a 16 * 32 = 512 bit vector containing 16 floats.**

- **Solution: *clGetDeviceInfo()* introduced in Lecture 2**

```
// example: get device information.
cl_uint char_width;
clGetDeviceInfo(device, CL_DEVICE_PREFERRED_VECTOR_WIDTH_CHAR,
sizeof(char_width), &char_width, NULL);
```

Vector Data Types

Access Vector

- **There are several ways to access the components of a vector data type depending on how many components are in the vector.**

- **Vector2/3/4**

```
float2 pos;
pos.x = 1.0f;
pos.y = 2.0f;

// illegal since vector
// only has 2 components
pos.z = 3.0f;
```

- **Vector2/3/4/8/16**

```
float8 vec_8;
vec_8.s0 = 1.0f;        // the 1st component
vec_8.s7 = 2.0f;        // the 8th component

float16 vec_16;
vec_16.sa = 1.0f;      // or .sA, is the 10th component
vec_16.sF = 2.0f;      // or .sf, is the 16th component
```

Next Lecture Will Discuss

- Event structure
- Profiling
- Optimization

References

- Introduction to OpenCL.
  Cliff Woolley, NVIDIA
  http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/06-intro_to_opencl.pdf
- CUDA C Programming Guide
  http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz3g8riFfwE
- CUDA Overview.
  Cliff Woolley, NVIDIA
  http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/02-cuda-overview.pdf
- Introduction to CUDA
  James Balfour, NVIDIA Research
  http://mc.stanford.edu/cgi-bin/images/f/f7/Darve_cme343_cuda_1.pdf
- The OpenCL Programming Book - Free HTML version
  Publisher: Fixstars
  Author: Ryoji Tsuchiyama, Takashi Nakamura, Takuro Iizuka, Akihiro Asahara, Jeongdo Son, Satoshi Miki
- OpenCL in action
  Author: Matthew Scarpino