CPSC/ECE 4780/6780

General-Purpose Computation on Graphical Processing Units (GPGPU)

Lecture 13: (OpenCL) Introduction to OpenCL

What is OpenCL?

- OpenCL – Open Computing Language
- An open and royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors
  - First released on December 8, 2008, by the Khronos Group
  - Aim is for it to form the foundation layer of a parallel computing ecosystem of platform independent tools
  - Includes a language for writing kernels, plus APIs used to define and control platforms

Why OpenCL?

- A primary benefit of OpenCL is substantial acceleration in parallel processing
  - Takes all computational resources as peer computational units
  - Complements OpenGL by sharing data structures and memory locations
- A second benefit of OpenCL is cross-vendor software portability
  - NVIDIA's CUDA is a proprietary implementations
  - OpenCL separates hardware implementations from software infrastructure
- OpenCL is largely derived from CUDA
  - Same basic functioning: kernel is sent to the accelerator "compute device" composed of "compute units" of which "processing elements" work on "work items"
    - CUDA Thread -> OpenCL work-item
    - CUDA Block -> OpenCL work-group
  - But unique data structures and functions
    - OpenCL is not easy to learn!

Anatomy of OpenCL

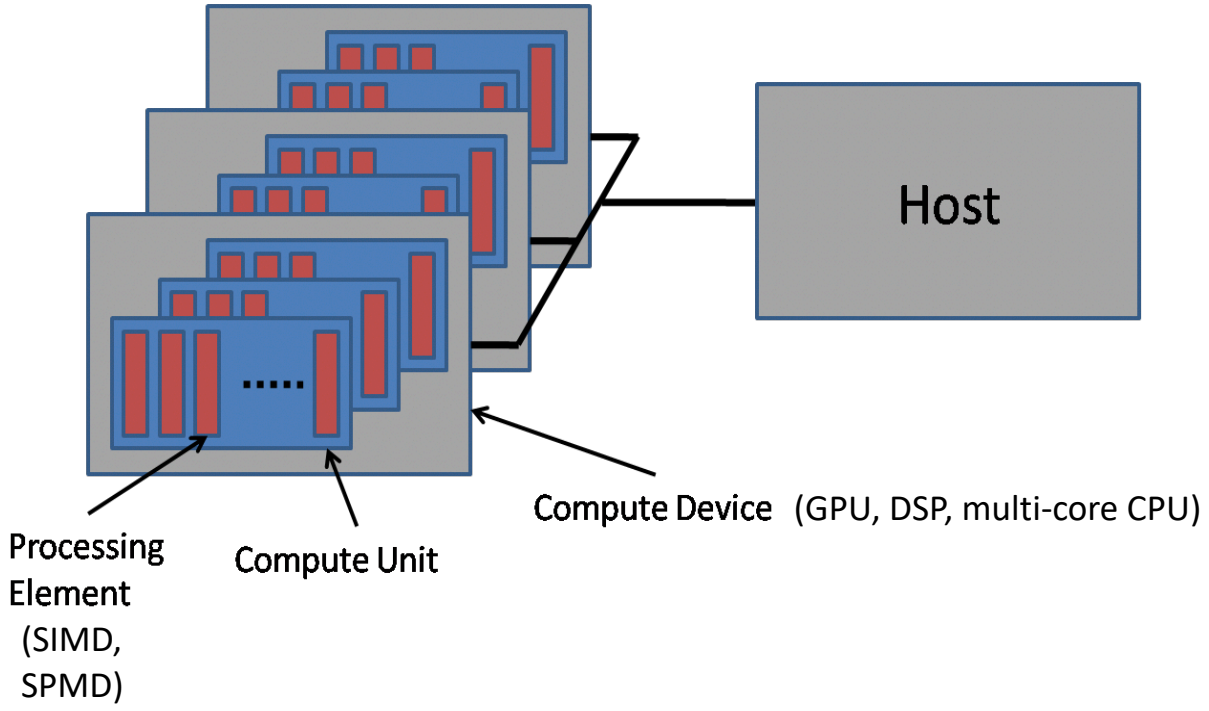
- The OpenCL development framework is made up of three main parts:
  - **Language specification**: describes the syntax and programming interface for writing kernel programs that run on the supported accelerator
    - Programming language is based on the ISO C99 specification with added extensions and restrictions
      - Additions: vector types and vector operations, optimized image access, address space qualifiers
      - Restrictions: absence of support for function pointers, bit-fields, and recursion
    - C language was selected as the first base for the OpenCL language
    - Well-defined IEEE 754 numerical accuracy for floating-point operations
  - **Platform API**: gives the developer access to software application routines that can query the system for the existence of OpenCL-supported devices
  - **Runtime API**: uses contexts to manage one or more OpenCL devices

OpenCL Architecture

- **Platform model**: Specifies that there is one processor coordinating execution (the host) and one or more processors capable of executing OpenCL C code (the devices)
- **Execution model**: Defines how the OpenCL environment is configured on the host and how kernels are executed on the device
  - Setting up an OpenCL context on the host
  - Providing mechanisms for host–device interaction
  - Defining a concurrency model used for kernel execution on devices
- **Memory model**: Defines the abstract memory hierarchy that kernels use, regardless of the actual underlying memory architecture
- **Programming model**: Defines how the concurrency model is mapped to physical hardware

The Platform Model

- One **Host** connected to one or more **OpenCL Devices**
  - Each OpenCL Device is composed of one or more **Compute Units**
    - Each Compute Unit is divided into one or more **Processing Elements**

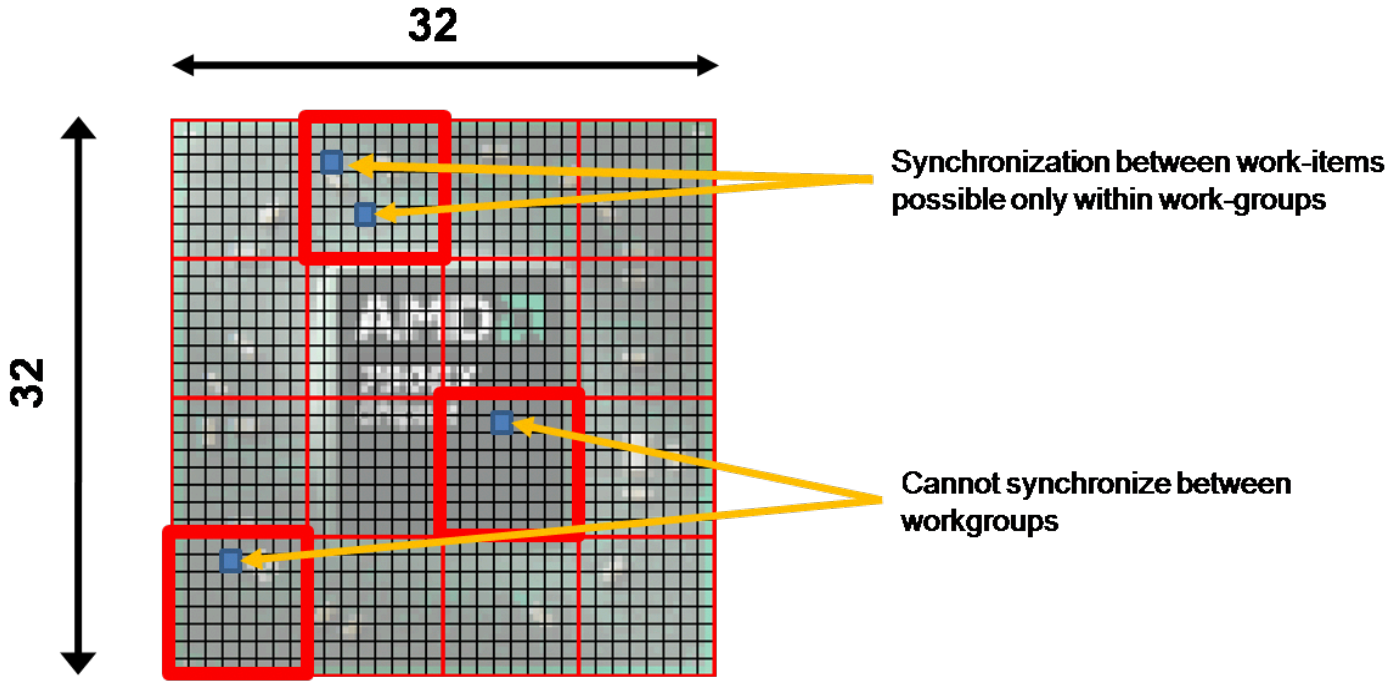


The Execution Model

- Kernels:
  - Basic unit of executable code runs on one or more OpenCL devices
  - Can be data parallel or task parallel
- Host programs:
  - Executes on the host system
  - Defines devices context
  - Queues kernel execution instances using command queues

Kernels

- OpenCL exploits parallel computation on compute devices by defining the problem into an N-dimensional index space
  - Each work-item in this index space executes the same kernel function but on different data
  - An N-dimensional index space can be N = 1, 2, or 3

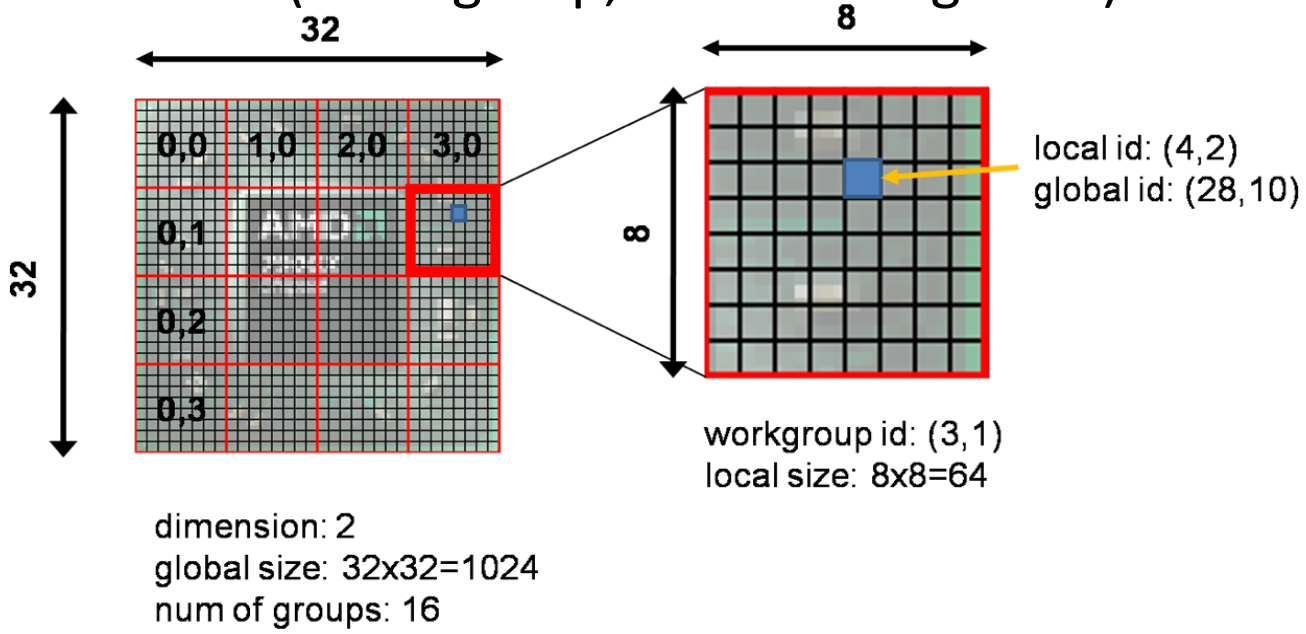


Grouping work-items into work-groups

Kernels are executed by one or more work-items. Work-items are collected into work-groups and each work-group executes on a compute unit

Work-group Example

- Global dimension:
  - 32 x 32 (whole problem space)
- Local dimensions:
  - 8 x 8 (work-group, executes together)



Kernel Implementation Example

- Replace loops with functions (a kernel) executing at each point in a problem domain
  - E.g., processing a 1024x1024 image with one kernel invocation per pixel (1024 x 1024 = 1,048,576 total kernel executions)

Scalar C Function	Data-Parallel Function

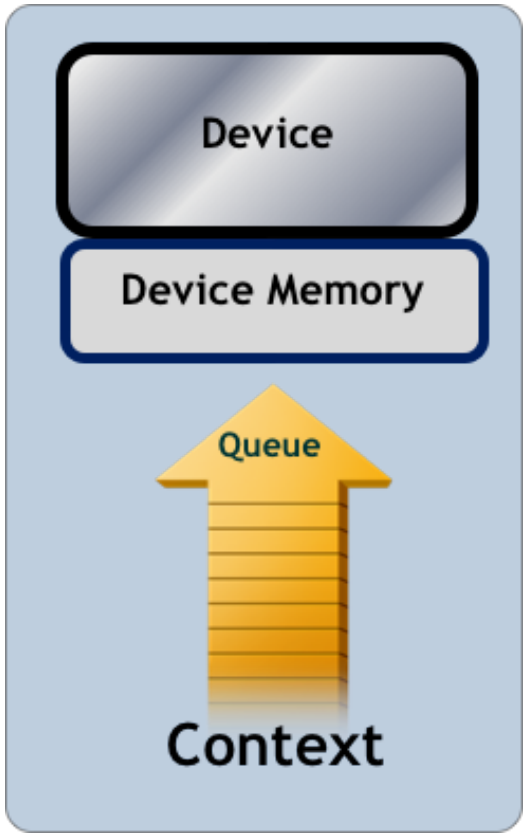
`void square(int n, const float *a, float *result)`
`{`
`  int i;`
`  for (i=0; i<n; i++)`
`  result[i] = a[i]*a[i];`
`}`
	`kernel void dp_square`
	`  (global const float *a, global float *result)`
	`{`
	`  int id=get_global_id(0);`
	`  result[id] = a[id]*a[id];`
	`}`
	`// dp_square execute over "n" work-items`

Host Program

- Responsible for setting up and managing the execution of kernels on the OpenCL device through the use of **context**
- Host can create and manipulate the context by including resources:
  - **Devices** — A set of OpenCL devices used by the host to execute kernels
  - **Program Objects** — The program source or program object that implements a kernel or collection of kernels
  - **Kernels** — The specific OpenCL functions that execute on the OpenCL device
  - **Memory Objects** — A set of memory buffers or memory maps common to the host and OpenCL devices

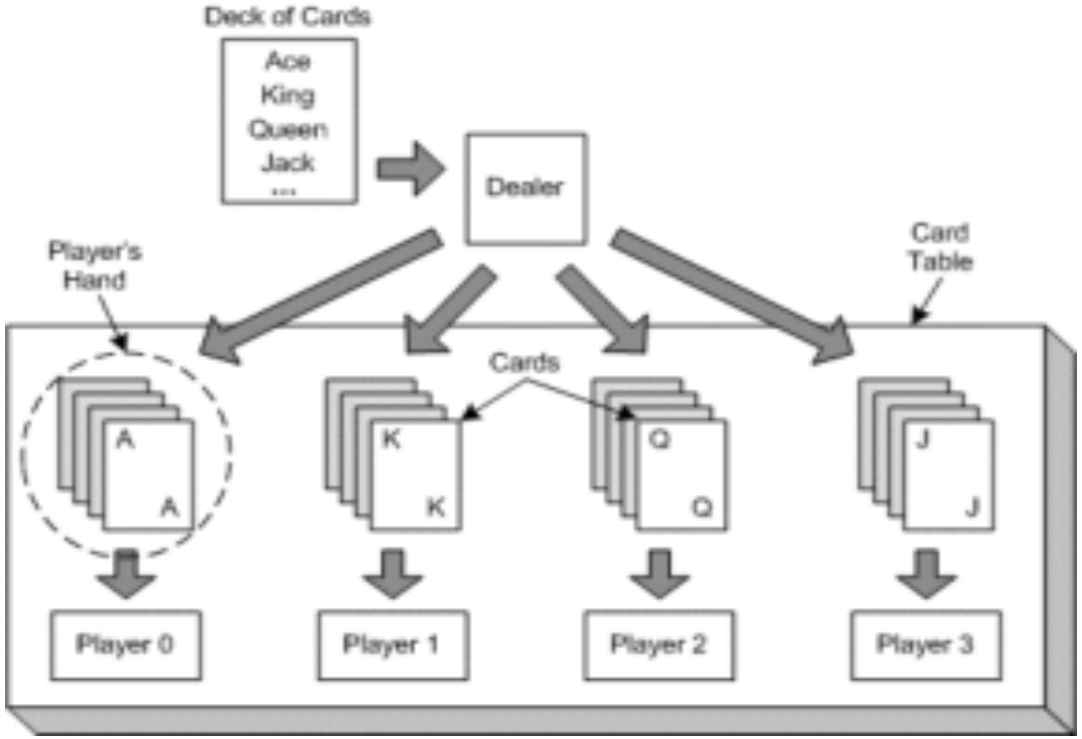
Context and Command-Queues

- **Context**: The environment within which kernels execute and in which synchronization and memory management is defined, including:
  - One or more devices
  - Device memory
  - One or more command-queues
- All commands for a device are submitted through a **command-queue**
  - Kernel execution commands: run kernel command
  - Memory commands: transfer memory objects
  - Synchronization commands: define the execution order of commands
- Commands are placed into the command queue in-order and execute either in-order or out-of-order



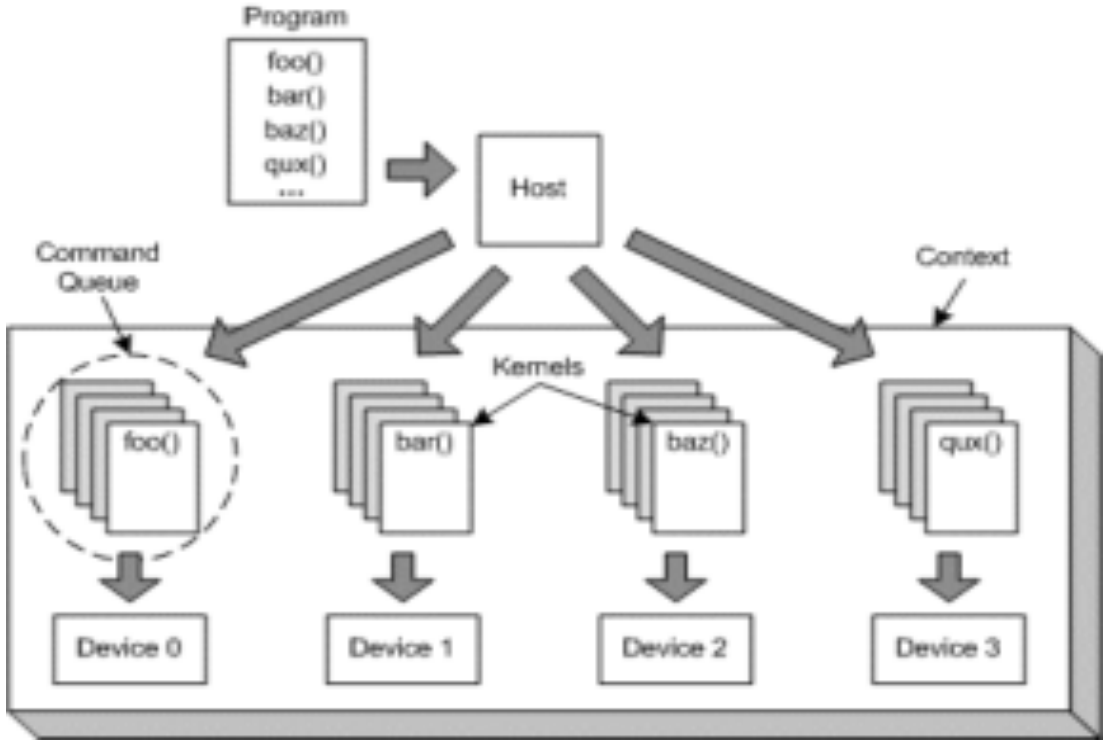
Understand Host Data Structure with a Card Game Analogy

- A dealer
  - Distributes cards to one or more players
  - Handles player's requests
- Players
  - Receives cards and play
  - Can't interact with other players
  - Can make requests to dealer



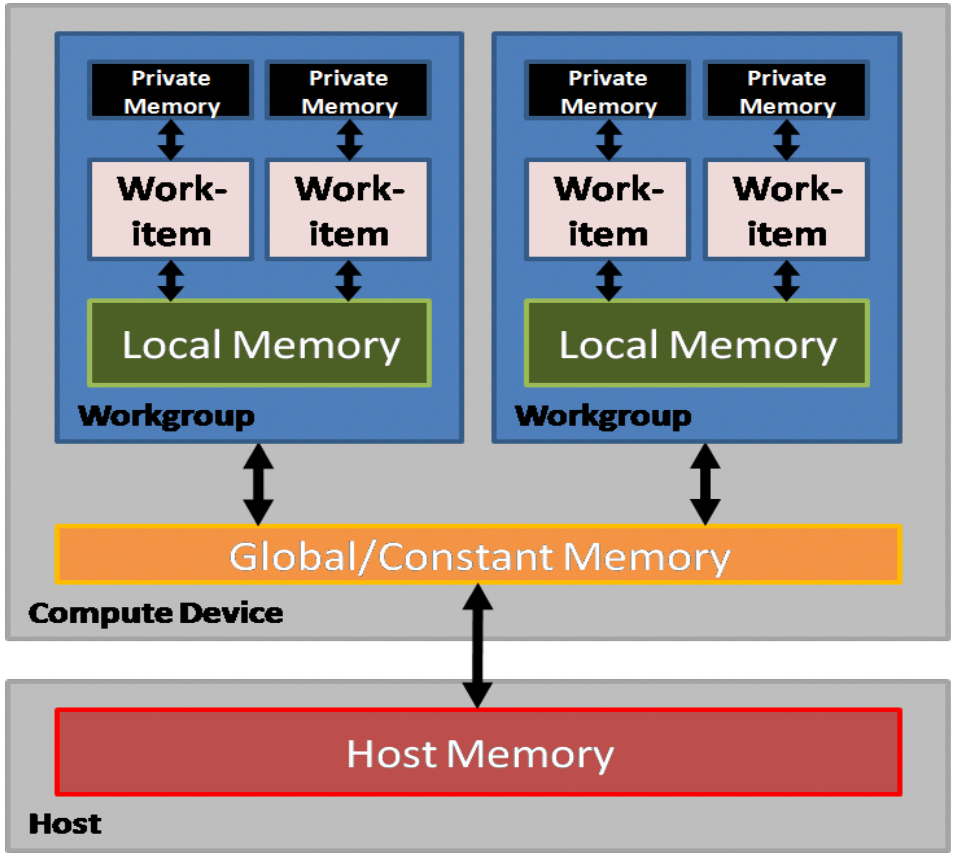
The Corresponding Data Structures

- **Host**: the card dealer
- **Device** (cl_device_id): players
- **Kernel** (cl_kernel): cards
- **Program** (cl_program): a deck of cards
- **Command queue** (cl_command_queue): a player's hand
- **Context** (cl_context): card tables



The Memory Model

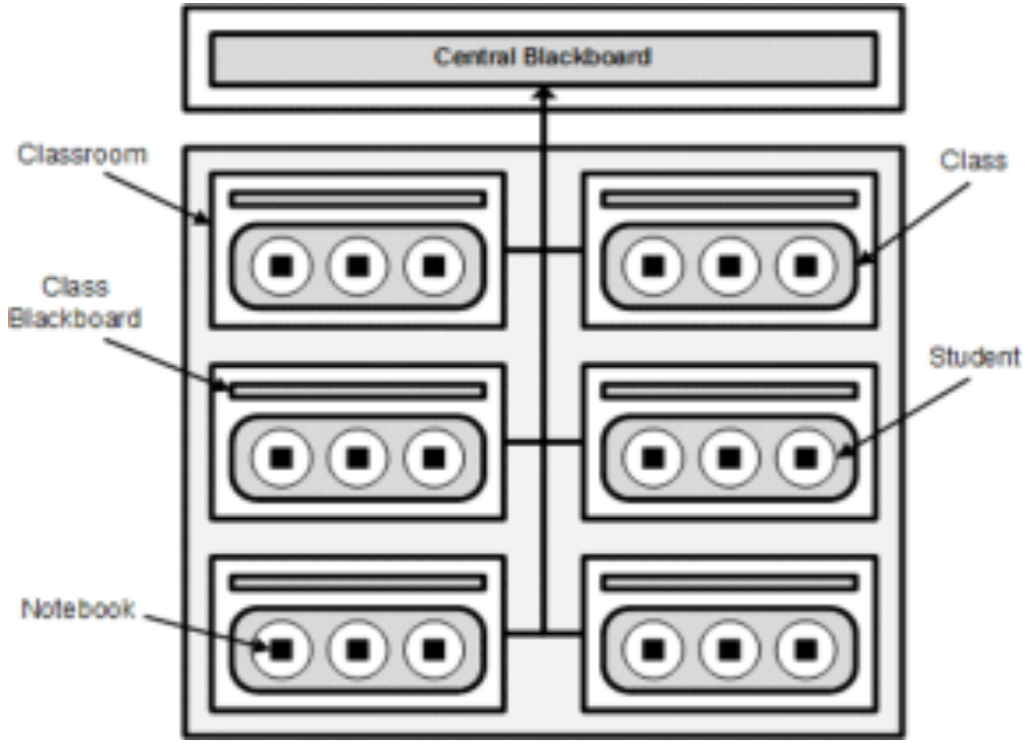
- Global memory and Constant memory
  - Visible to all work-groups
- Local memory
  - Shared within a work-group
- Private memory
  - Per work-item
- Host memory
  - On the CPU



Memory management is explicit: You are responsible for moving data from host->global->local and back. This process works by enqueuing read/write commands in the command queue.

Understand Kernel Executing with a School Day Analogy

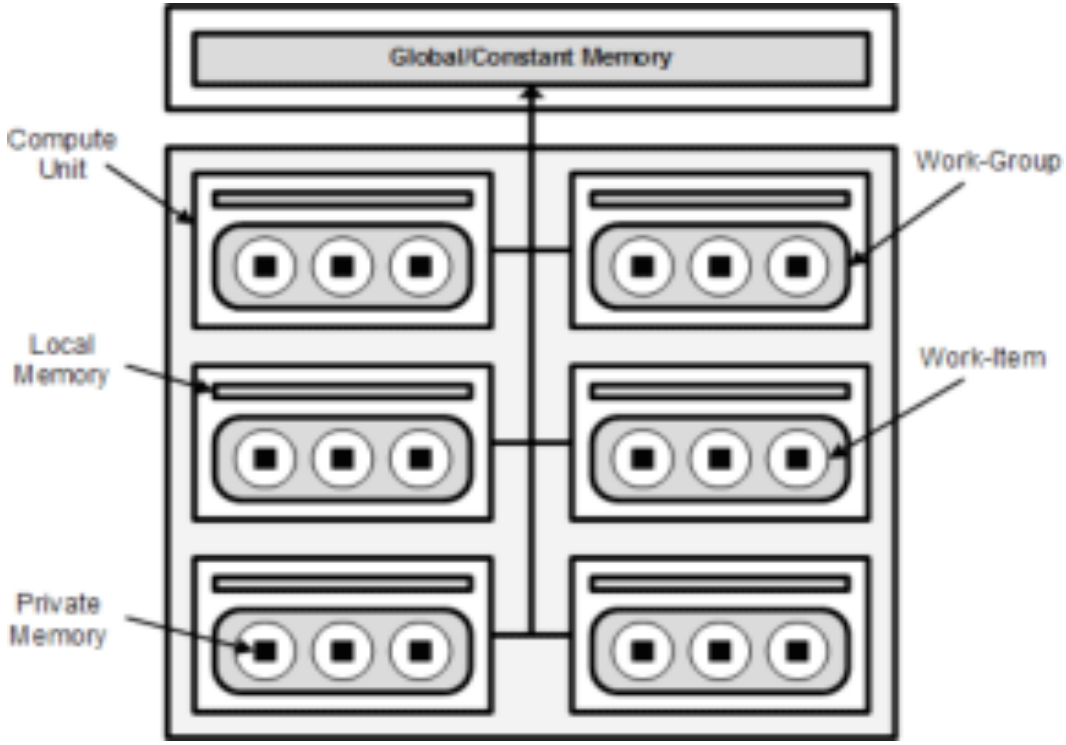
- School
- Classroom
- Students
  - Class ID
  - School ID
- Blackboard
- Notebook
- Class



Students solving math problems

The Corresponding Kernel Execution

- Device: the school
- Kernel: the math problem
- Work-item: student
- Work-group: class
- Compute unit: classroom
- Student class ID: local ID
- Student school ID: global ID
- Central blackboard: global memory
- Classroom blackboard: local memory
- Notebook: private memory



OpenCL Program Flow

clGetPlatformIDs clGetDeviceIDs clCreateContext clCreateCommandQueue	Organize resources, create command queue
clCreateProgramWithSource clBuildProgram clCreateKernel	Compile kernel
clCreateBuffer clEnqueueWriteBuffer	Transfer data from host to GPU memory
clSetKernelArg clGetKernelWorkGroupInfo clEnqueueNDRangeKernel clFinish	Lauch threads running kernels on GPU, perform main computation
clEnqueueReadBuffer	Transfer data from GPU to host memory
clRelease...	Free all allocated memory