OPENCL编程模型

汤善江 副教授
天津大学智能与计算学部
tashj@tju.edu.cn
http://cic.tju.edu.cn/faculty/tangshanjiang/

Outline

- OpenCL概述
- OpenCL抽象模型
  - 平台模型
  - 执行模型
  - 内存模型
  - 编程模型
- OpenCL程序设计
  - 程序设计流程
  - 编程实例

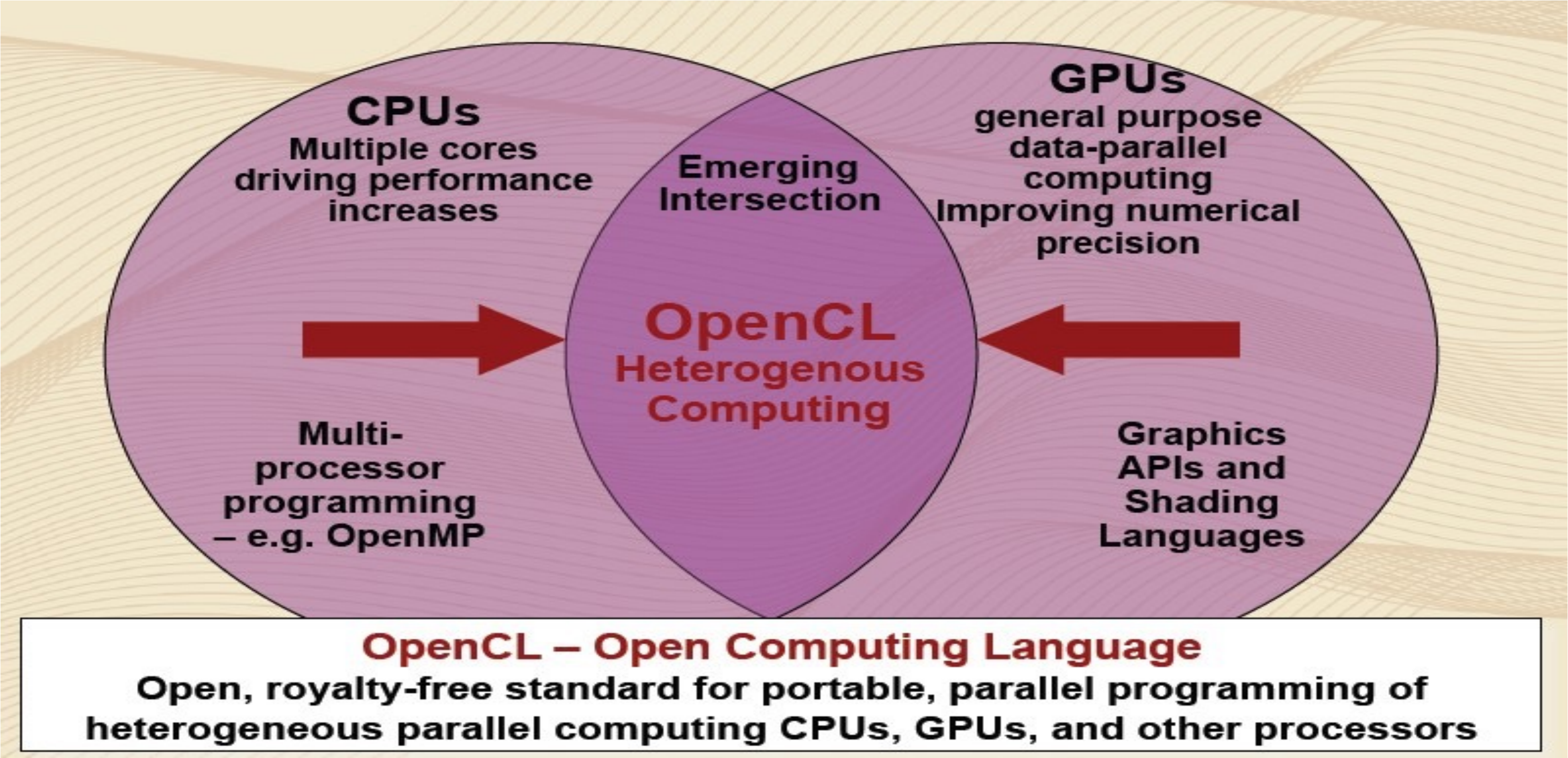
Outline

- **OpenCL概述**
- OpenCL抽象模型
  - 平台模型
  - 执行模型
  - 内存模型
  - 编程模型
- OpenCL程序设计
  - 程序设计流程
  - 编程实例

异构计算系统

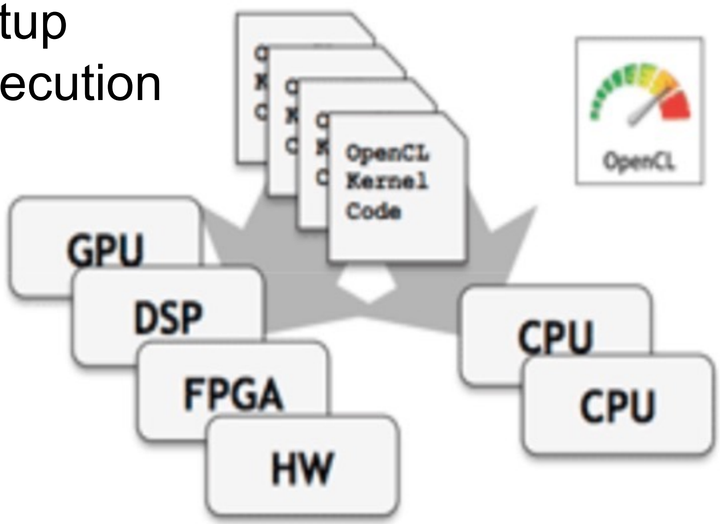
- A modern computer system has:
  - CPU(s)
  - GPU(s)
  - DSP processors
  - … other?
- Need to make the best use of **all** the available resources from within a **single** program:
  - One program that runs well (i.e., reasonably close to "hand-tuned" performance) on a heterogeneous mixture of processors.

异构计算系统编程


CPUs
Multiple cores driving performance increases
Emerging Intersection
GPUs
general purpose data-parallel computing
Improving numerical precision
OpenCL
Heterogenous Computing
Multi-processor programming – e.g. OpenMP
Graphics APIs and Shading Languages
OpenCL – Open Computing Language
Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

什么是OpenCL?

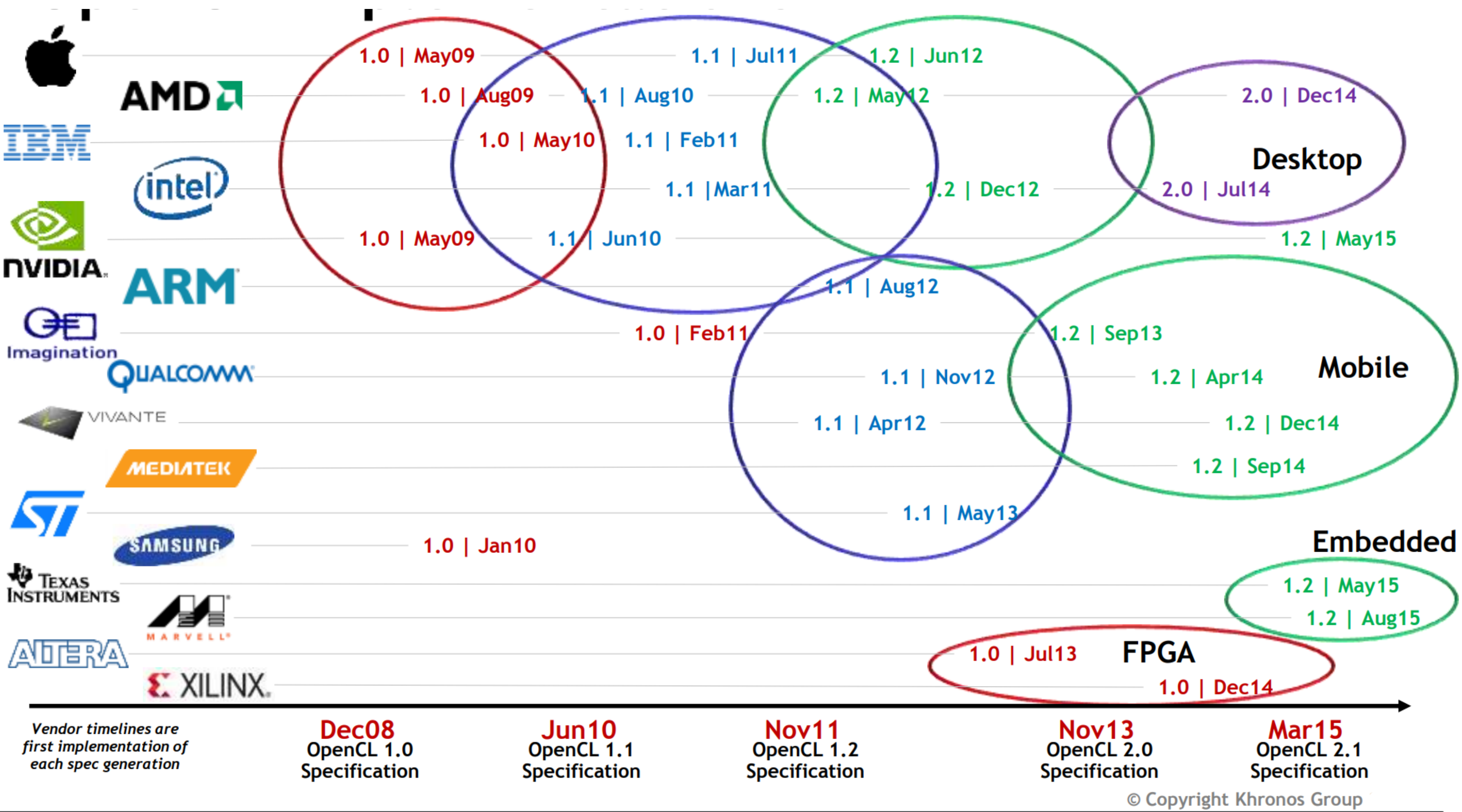
- OpenCL（Open Computing Language）是一个为异构平台编写程序的框架，此异构平台可由CPU，GPU或其他类型的处理器组成。
- Low-level programming API for data parallel computation
  - Platform API: device query and pipeline setup
  - Runtime API: resources management + execution
- **Cross-platform** API
  - Windows, MAC, Linux, Mobile, Web…
- **Portable** device targets
  - CPUs, GPUs, FPGAs, DSPs, etc…
  - **One code** tree can be executed on CPUs, GPUs, DSPs, FPGA and hardware
- Implementation based on C99
- Maintained by Kronos Group (www.khronos.org)
- Current version: 3.0 with C++ support (classes & templates)



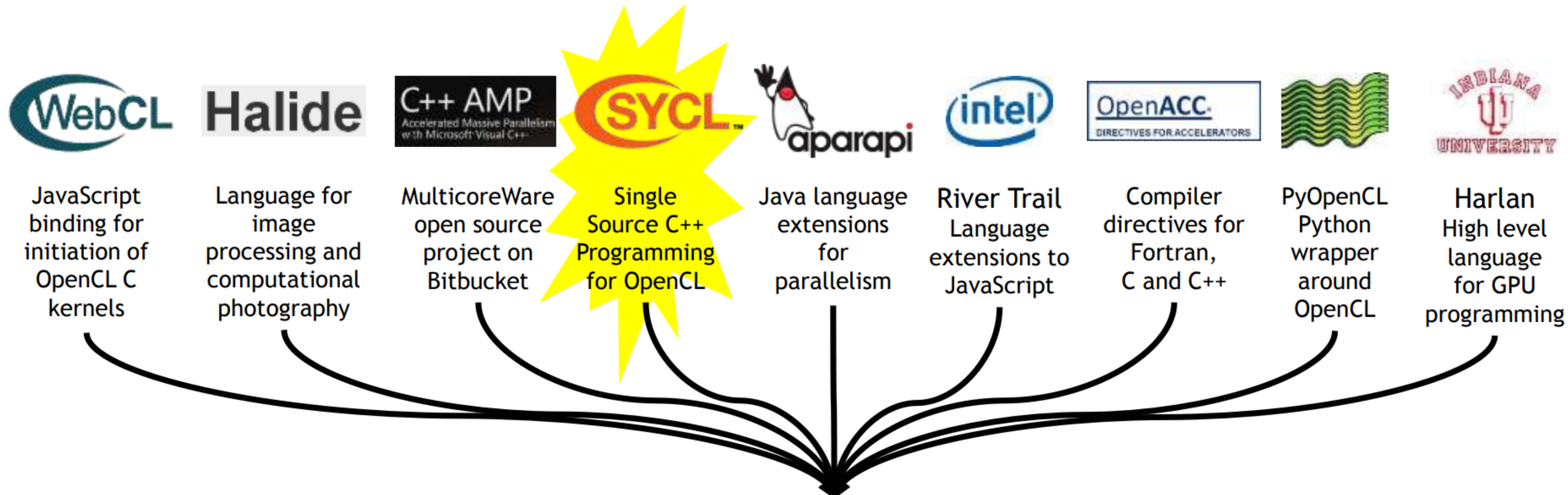
OpenCL的发展历史

- 2008年　Apple
  提出OpenCl规范
- 2008年　Khronos Group
  发布OpenCL 1.0标准
- 2013年　Khronos Group
  OpenCl 2.0标准发布
- 2013年　截至2013年11月，以AMD、Apple、ATI、Intel、Nvidia等为代表的公司，已经发布了多款支持OpenCl的产品。
- 2017年　Khronos Group
  OpenCl 2.2标准发布
- 2020年　Khronos Group
  OpenCl 3.0标准发布

OpenCL Implementations



OpenCL Front-End APIs



Approaching 200 languages, frameworks and projects using OpenCL as a compiler target to access vendor optimized, heterogeneous compute runtimes

OpenCL

This trend will be significantly accelerated by the availability of SPIR-V which is specifically designed to be a compiler target

OpenCL: A high-level view

- OpenCL applications:
  - A host program running on the PC
  - One or more Kernels that are queued up to run on CPUs, GPUs, and “other processors”.
- OpenCL is understood in terms of these models
  - Platform model
  - Execution model
  - Memory model
  - Programming model

Outline

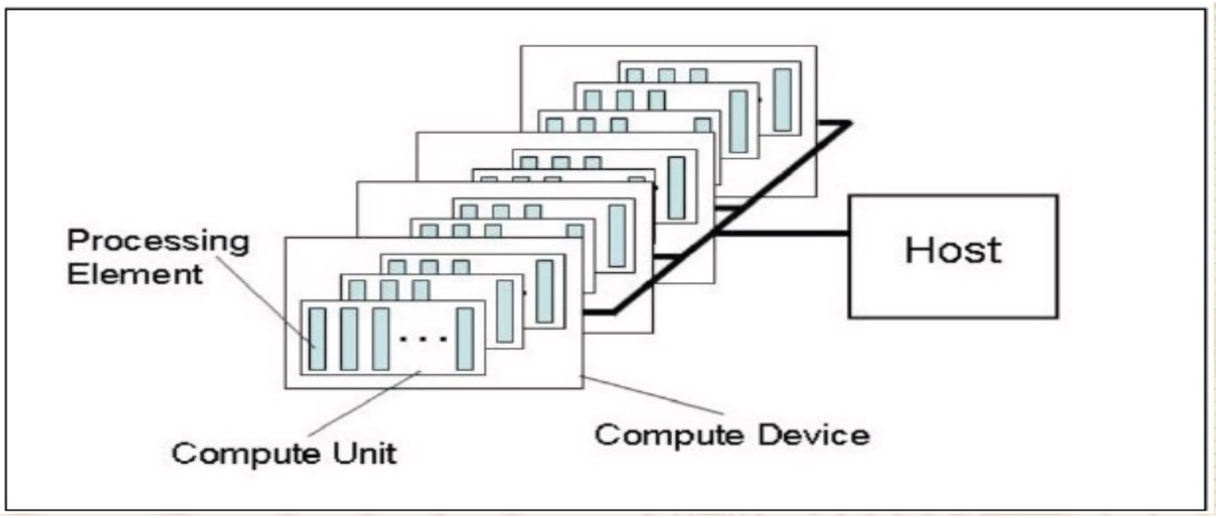
- OpenCL概述
- **OpenCL模型**
  - **平台模型**
  - 执行模型
  - 内存模型
  - 编程模型
- OpenCL程序设计
  - 程序设计流程
  - 编程实例

OpenCL平台模型

- Each OpenCL implementation (i.e. an OpenCL library from AMD, NVIDIA, etc.) defines *platforms* which enable the host system to interact with OpenCL-capable devices
  - Currently each vendor supplies only a single platform per implementation
- OpenCL uses an "Installable Client Driver" model
  - The goal is to allow platforms from different vendors to co-exist
  - Current systems' device driver model will not allow different vendors' GPUs to run at the same time

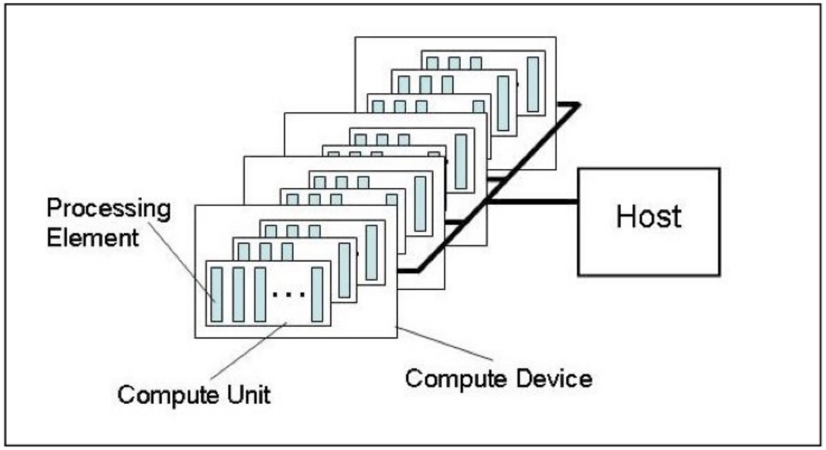
OpenCL平台模型

- One Host + one or more Compute Devices
  - Each Compute Device is composed of one or more Compute Units
    - Each Compute Unit is further divided into one or more Processing Elements
      - PE executes code as SIMD or SPMD



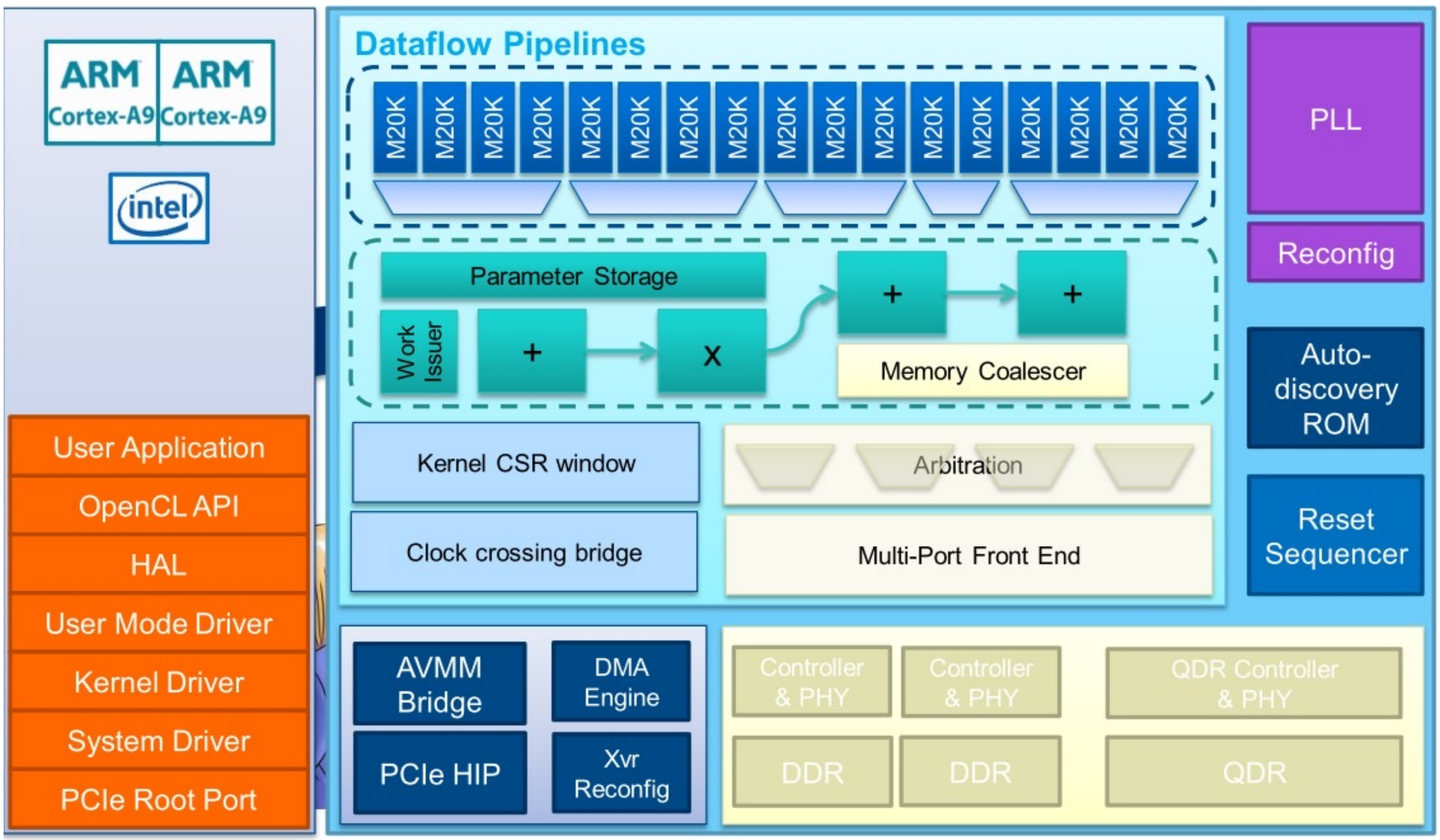
OpenCL平台模型

- The host is whatever the OpenCL library runs on
  - x86 CPUs for both NVIDIA and AMD
- Devices are processors that the library can talk to
  - CPUs, GPUs, and generic accelerators
- For AMD
  - All CPUs are combined into a single device (each core is a compute unit and processing element)
  - Each GPU is a separate device



Altera OpenCL平台模型

Developers mostly working happily at Application level in C for kernels and favourite application programming language



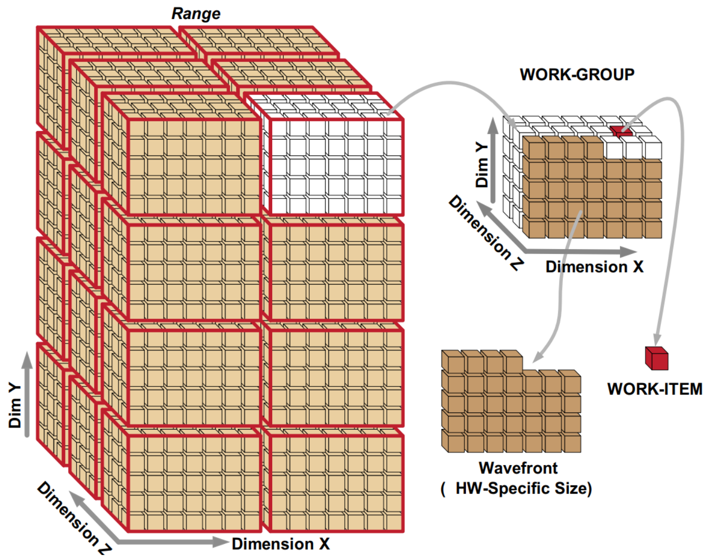
The OpenCL compiler takes care of much of the worries including machine-based profiling, tuning and optimization

Outline

- OpenCL概述
- **OpenCL模型**
  - 平台模型
  - **执行模型**
  - 内存模型
  - 编程模型
- OpenCL程序设计
  - 程序设计流程
  - 编程实例

OpenCL执行模型

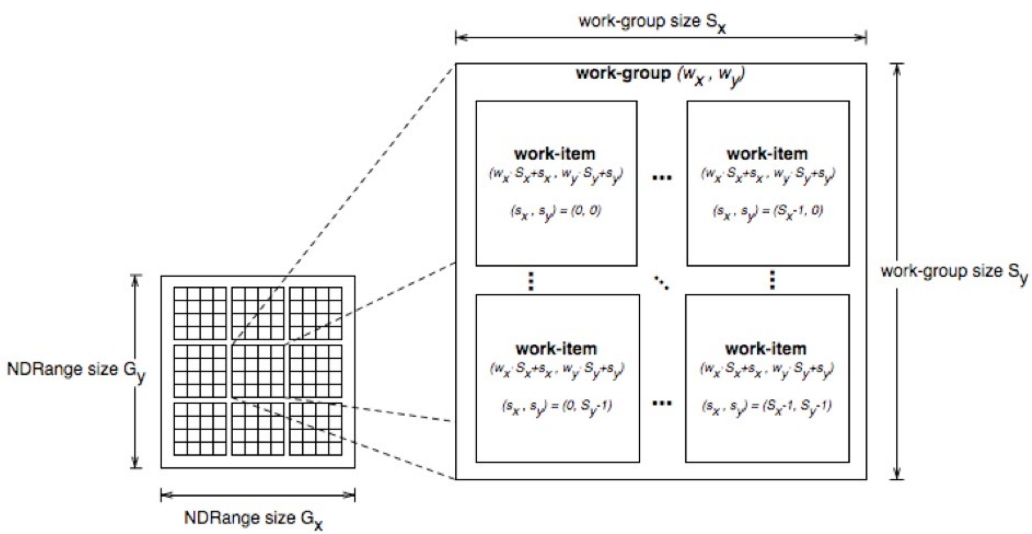
- A **kernel** is logical unit of instructions to be executed on a compute device.
- Kernels are executed in multi-dimensional **index space**: ***NDRange***
- For every element of the index space a **work-item** is executed
- The index space is tiled into **work-groups**
- Work items within a workgroup are synchronized using barriers or fences



AMD OpenCL User Guide 2015

OpenCL执行模型

- Host defines a **command queue** and associates it with a context (devices, kernels, memory, etc).
- Host enqueues commands to the command queue
- Work-items can uniquely identify themselves based on:
  - A global id (unique within the index space)
  - A work-group ID and a local ID within the work-group



OpenCL线程组织结构

- Threads can determine their global ID in each dimension
  - get_global_id(dim)
  - get_global_size(dim)
- Or they can determine their work-group ID and ID within the workgroup
  - get_group_id(dim)
  - get_num_groups(dim)
  - get_local_id(dim)
  - get_local_size(dim)
- get_global_id(0) = column, get_global_id(1) = row
- get_num_groups(0) * get_local_size(0) == get_global_size(0)

OpenCL执行模型——与CUDA映射

- OpenCL terminology aims for generality

OpenCL Terminology	CUDA Terminology
Compute Unit	Streaming Processor (SM)
Processing Element	Processor Core
Kernel	Kernel
Host program	Host program
Wavefront (AMD)	Warp
Work-item	Thread
Work-group	Thread Block
NDRange	Grid

OpenCL执行模型——与CUDA映射

- Work Items Indexing

OpenCL Terminology	CUDA Terminology
get_num_groups()	gridDim
get_local_size()	blockDim
get_group_id()	blockIdx
get_local_id()	threadIdx
get_global_id()	blockIdx * blockDim + threadIdx
get_global_size()	gridDim * blockDim

OpenCL执行模型——与CUDA映射

- Threads Synchronization

OpenCL Terminology	CUDA Terminology
barrier()	__syncthreads()
No direct equivalent*	__threadfence()
mem_fence()	__threadfence_block()
No direct equivalent*	__threadfence_system()
No direct equivalent*	__syncwarp()
Read_mem_fence()	No direct equivalent*
Write_mem_fence()	No direct equivalent*

Outline

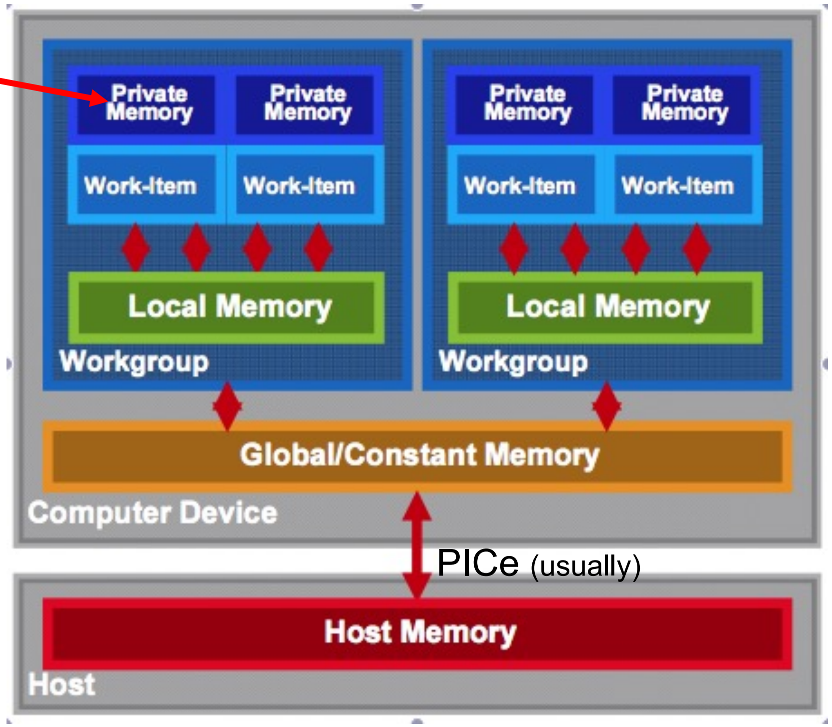
- OpenCL概述
- **OpenCL模型**
  - 平台模型
  - 执行模型
  - **内存模型**
  - 编程模型
- OpenCL程序设计
  - 程序设计流程
  - 编程实例

Outline

- OpenCL概述
- **OpenCL模型**
  - 平台模型
  - 执行模型
  - **内存模型**
  - 编程模型
- OpenCL程序设计
  - 程序设计流程
  - 编程实例

OpenCL 内存模型

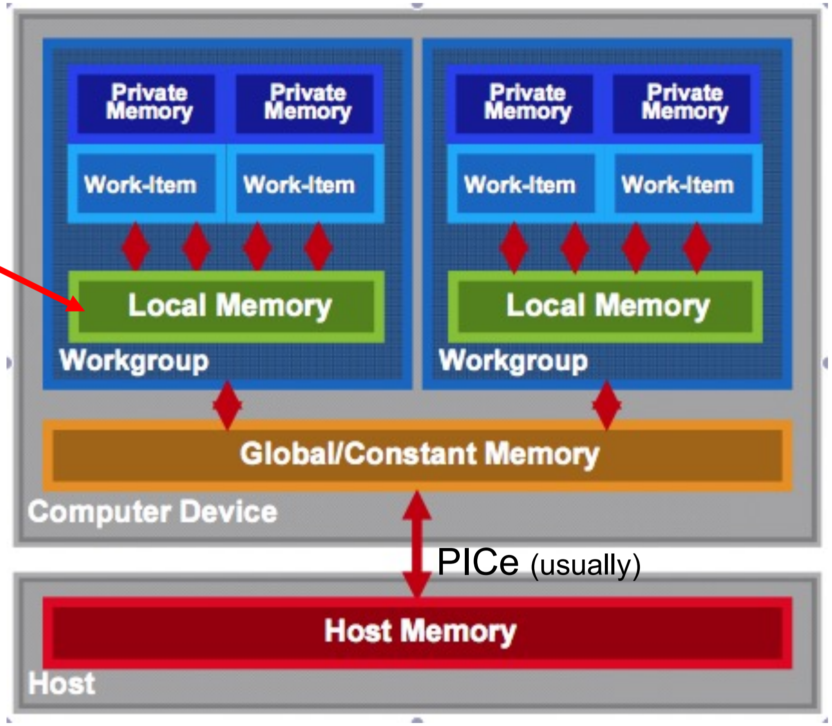
- Private memory: available per work item
- Local memory: shared in workgroup
- NB: No synchronization between workgroups
- Synchronization possible between work items in a common workgroup
- Global/constant memory accessible by any work-items (no guarantee to be synchronized)
- Host memory: access through the CPU



- Memory management is explicit…
- Data moved from: host->global->local and back

OpenCL 内存模型

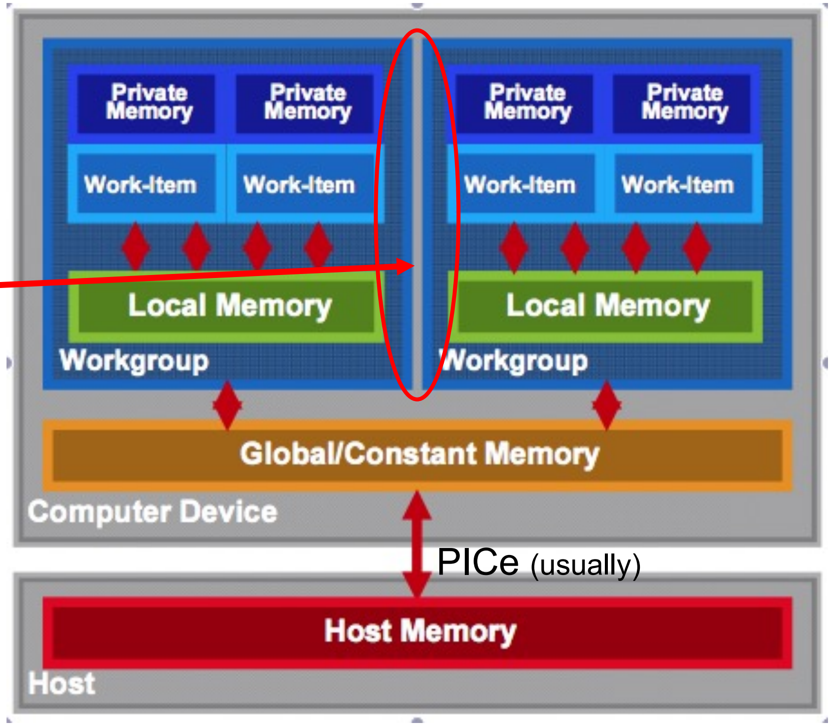
- Private memory: available per work item
- Local memory: shared in workgroup
- NB: No synchronization between workgroups
- Synchronization possible between work items in a common workgroup
- Global/constant memory accessible by any work-items (no guarantee to be synchronized)
- Host memory: access through the CPU



- Memory management is explicit…
- Data moved from: host->global->local and back

OpenCL内存模型

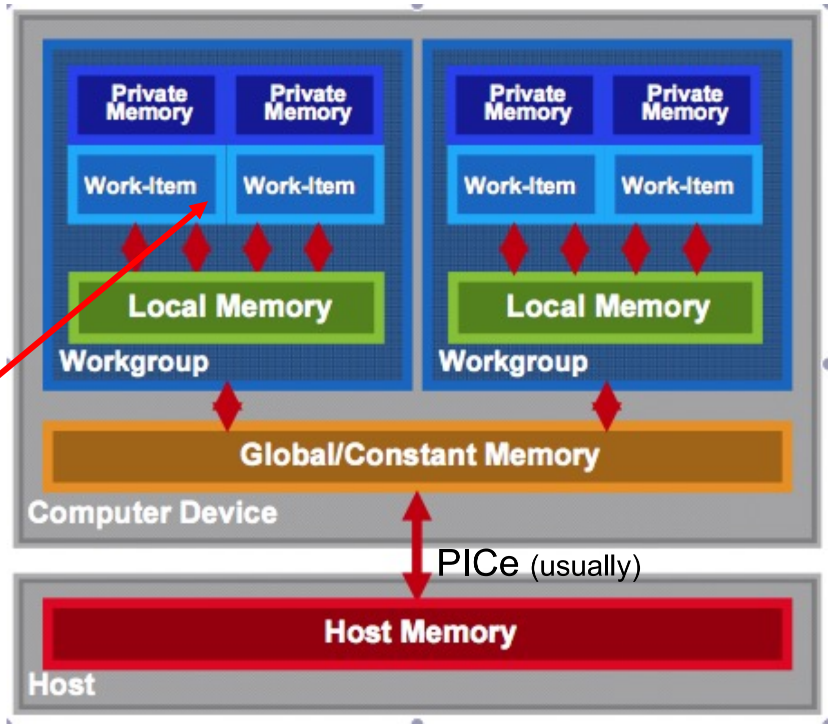
- Private memory:
  available per work item
- Local memory:
  shared in workgroup
- NB: No synchronization
  between workgroups
- Synchronization possible
  between work items in a
  common workgroup
- Global/constant memory
  accessible by any work-
  items (no guarantee to be
  synchronized)
- Host memory: access
  through the CPU



- Memory management is explicit…
- Data moved from:
  host->global->local and back

OpenCL 内存模型

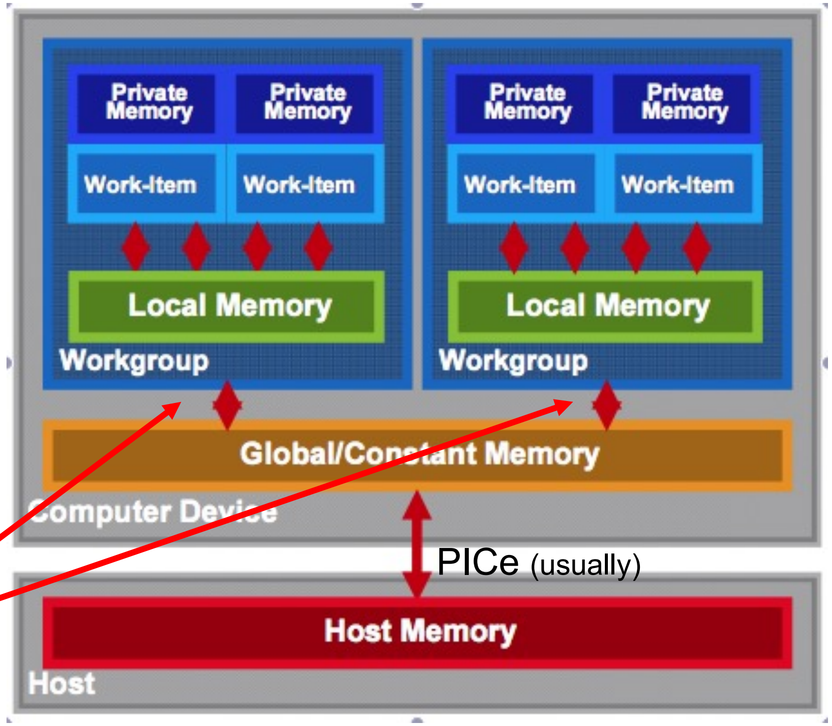
- Private memory:
  available per work item
- Local memory:
  shared in workgroup
- NB: No synchronization between workgroups
- Synchronization possible between work items in a common workgroup
- Global/constant memory accessible by any work-items (no guarantee to be synchronized)
- Host memory: access through the CPU



- Memory management is explicit…
- Data moved from:
  host->global->local and back

OpenCL 内存模型

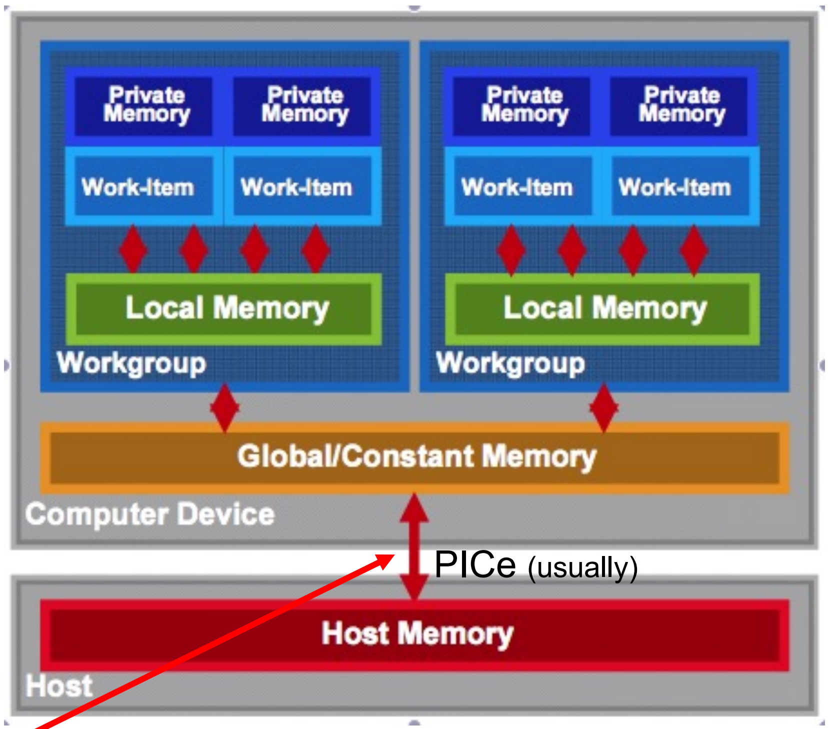
- Private memory: available per work item
- Local memory: shared in workgroup
- NB: No synchronization between workgroups
- Synchronization possible between work items in a common workgroup
- Global/constant memory accessible by any work-items (no guarantee to be synchronized)
- Host memory: access through the CPU



- Memory management is explicit…
- Data moved from: host->global->local and back

OpenCL 内存模型

- Private memory: available per work item
- Local memory: shared in workgroup
- NB: No synchronization between workgroups
- Synchronization possible between work items in a common workgroup
- Global/constant memory accessible by any work-items (no guarantee to be synchronized)
- Host memory: access through the CPU



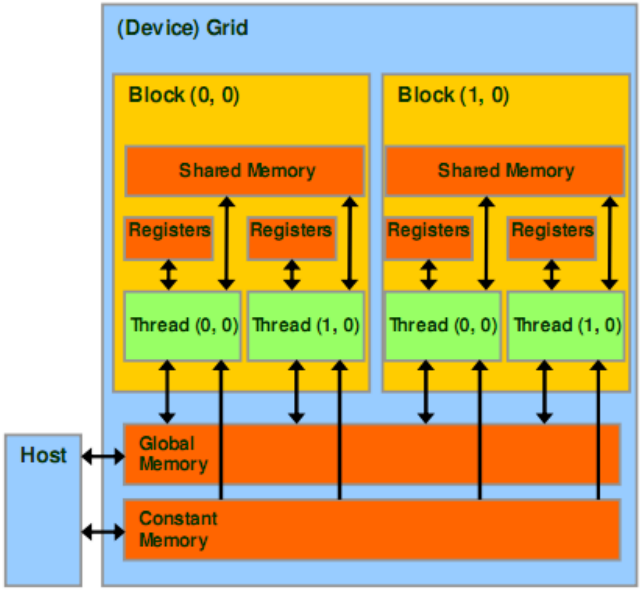
- Memory management is explicit…
- Data moved from: host->global->local and back

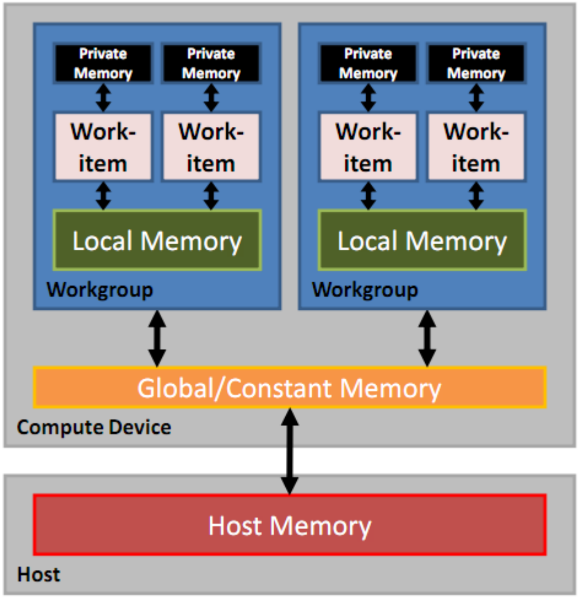
OpenCL 内存模型

- Memory management is explicit
  - Must move data from host memory to device global memory, from global memory to local memory, and back
- Work-groups are assigned to execute on compute-units
  - No guaranteed communication/coherency between different work-groups (no software mechanism in the OpenCL specification)

OpenCL内存模型——与CUDA映射

OpenCL Terminology	CUDA Terminology
Global Memory	Global Memory
Constant Memory	Constant Memory
Local Memory	Shared Memory
Private Memory	Local Memory





OpenCL内存模型——与CUDA映射

- Resources Qualifiers

Description	OpenCL Terminology	CUDA Terminology
Kernel global function	__kernel	__global__
Kernel local function	nothing*	__device__
Readonly memory	__constant	__device__
Global memory	__global	__device__
Private memory	__local	__shared__

CUDA与OpenCL关键指标对应汇总

CUDA		OpenCL
Thread	⟷	Work-item
Thread-block	⟷	Work-group
Global memory	⟷	Global memory
Constant memory	⟷	Constant memory
Shared memory	⟷	Local memory
Local memory	⟷	Private memory
__global__ function	⟷	__kernel function
__device__ function	⟷	no qualification needed
__constant__ variable	⟷	__constant variable
__device__ variable	⟷	__global variable
__shared__ variable	⟷	__local variable

Outline

- OpenCL概述
- **OpenCL模型**
  - 平台模型
  - 执行模型
  - 内存模型
  - **编程模型**
- OpenCL程序设计
  - 程序设计流程
  - 编程实例

OpenCL编程模型

- 数据并行化
  - Work-items in a work-group run the same program
  - Update data structures in parallel using the work-item ID to select
  - data and guide execution.
- 任务并行化
  - One work-item per work group … for coarse grained task-level parallelism.
  - Native function interface: trap-door to run arbitrary code from an OpenCL command-queue.
  - Multiple different kernels can be executed in parallel

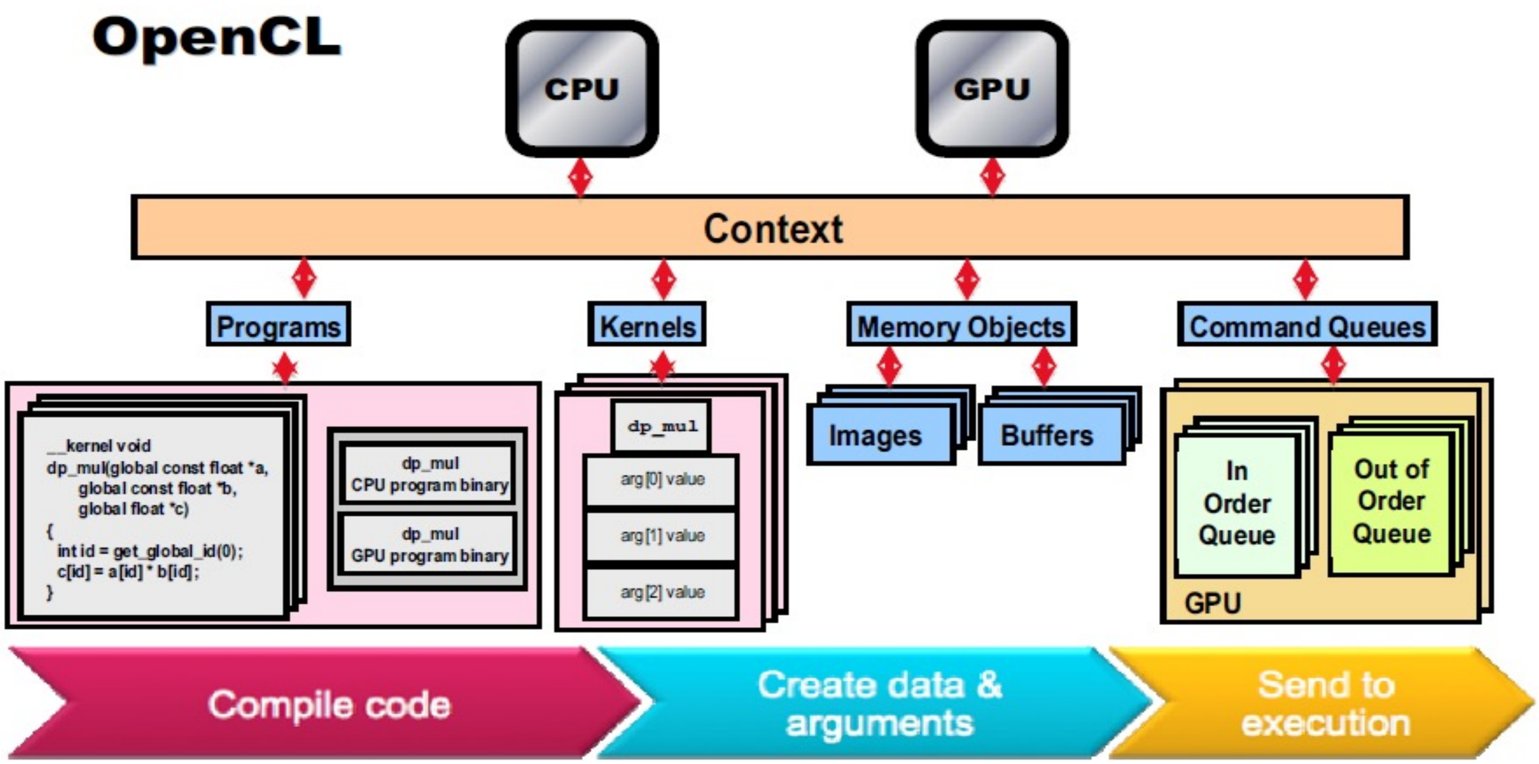
OpenCL 数据并行化

- Kernels executed across a global domain of **work-items**
  - **Global dimensions** define the range of computation
  - One work-item per computation, executed in parallel
  - Work-items are grouped in local **work-groups**
    - **Local dimensions** define the size of the work-groups
    - Executed together on one device
    - Share local memory and synchronization
  - Caveats
    - Global work-items must be independent: **No global synchronization**
    - Synchronization can be done within a work-group

OpenCL 任务并行化

- Expressing Task-Parallelism in OpenCL
  - Execute as a single work-item
  - A kernel written in OpenCL C
- clEnqueueTask
  - Imagine “sea of different tasks” executing concurrently
  - A task “owns the core” (i.e., a workgroup size of 1)

OpenCL Program Flow



OpenCL vs CUDA

- API Terminology

OpenCL Terminology	CUDA Terminology
clGetContextInfo()	cuDeviceGet()
clCreateCommandQueue()	No direct equivalent*
clBuildProgram()	No direct equivalent*
clCreateKernel()	No direct equivalent*
clCreateBuffer()	cuMemAlloc()
clEnqueueWriteBuffer()	cuMemcpyHtoD()
clEnqueueReadBuffer()	cuMemcpyDtoH()
clSetKernelArg()	No direct equivalent*
clEnqueueNDRangeKernel()	kernel<<<...>>>()
clReleaseMemObj()	cuMemFree()

Outline

- OpenCL概述
- **OpenCL模型**
  - 平台模型
  - 执行模型
  - 内存模型
  - 编程模型
- **OpenCL程序设计**
  - **程序设计流程**
  - 编程实例

OpenCL程序设计流程

1 查询平台	clGetPlatformIDs()
2 查询设备	clGetDeviceIDs()
3 创建上下文：将平台设备与上下文关联起来	clCreateContext() 或 clCreateContextFromType()
4 创建命令队列	clCreateCommandQueue()
5 读取、编译内核	clCreateProgramWithSource()或 clCreateProgramWithBinary() clBuildProgram()
6 打包生成内核	clCreateKernel()
7 创建缓存对象或图像对象，为内核参数分配内存	clCreateBuffer()、clCreateSubBuffer() clCreateImage2D()、clCreateImage3D()
8 设置内核参数，将上面分配的内存发送到设备上	clSetKernelArg()
9 执行内核	clEnqueueTask()或 clEnqueueNDRangeKernel()
10 读取设备上的处理结果	clEnqueueReadBuffer()
11 释放创建的资源： 创建的内存、 命令队列、 内核、 打包的程序、 上下文	clReleaseMemObject()、 clReleaseCommandQueue()、 clReleaseKernel()、 clReleaseProgram()、 clReleaseContext()

Selecting a Platform

```
cl_int  clGetPlatformIDs (cl_uint num_entries,
                          cl_platform_id *platforms,
                          cl_uint *num_platforms)
```

- This function is usually called twice
  - The first call is used to get the number of platforms available to the implementation
  - Space is then allocated for the platform objects
  - The second call is used to retrieve the platform objects

Selecting Devices

- Once a platform is selected, we can then query for the devices that it knows how to interact with

clGetDeviceIDs[4] (cl_platform_id *platform*,
cl_device_type *device_type*,
cl_uint *num_entries*,
cl_device_id **devices*,
cl_uint **num_devices*)

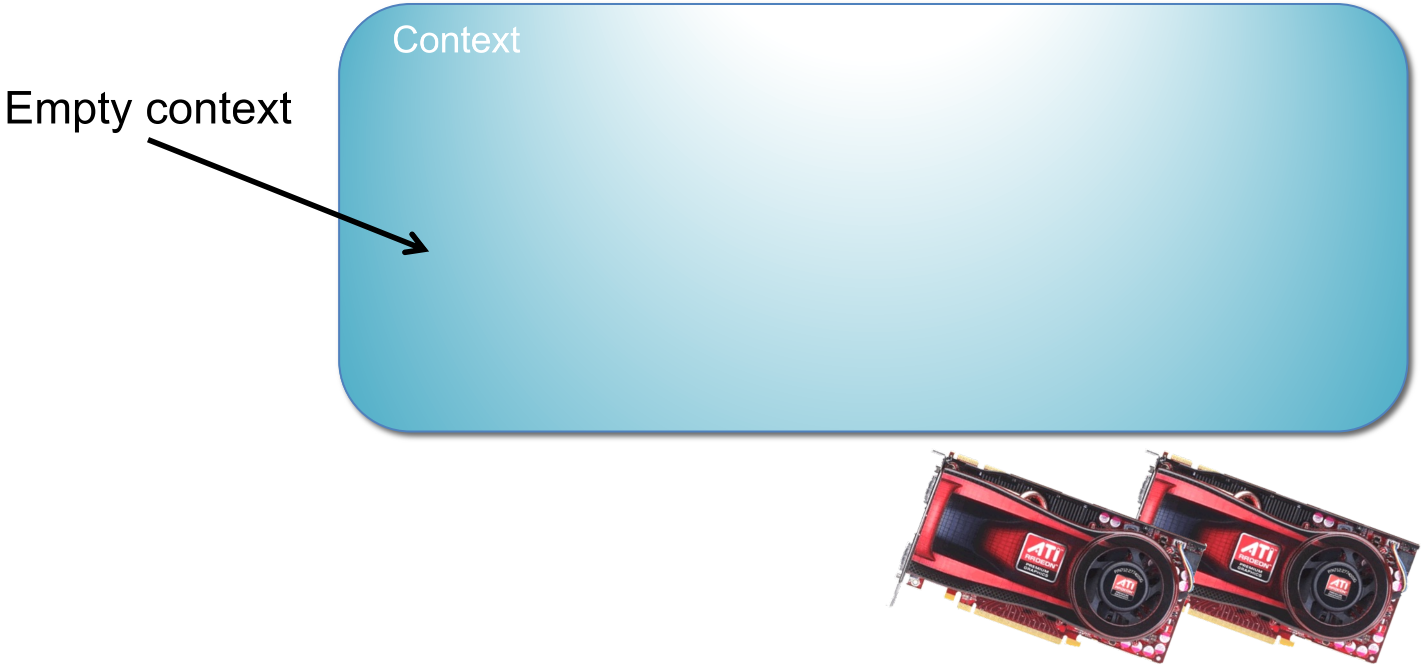
- We can specify which types of devices we are interested in (e.g. all devices, CPUs only, GPUs only)
- This call is performed twice as with clGetPlatformIDs
  - The first call is to determine the number of devices, the second retrieves the device objects

Contexts

- A context refers to the environment for managing OpenCL objects and resources
- To manage OpenCL programs, the following are associated with a context
  - Devices: the things doing the execution
  - Program objects: the program source that implements the kernels
  - Kernels: functions that run on OpenCL devices
  - Memory objects: data that are operated on by the device
  - Command queues: mechanisms for interaction with the devices
    - Memory commands (data transfers)
    - Kernel execution
    - Synchronization

Contexts

- When you create a context, you will provide a list of devices to associate with it
  - For the rest of the OpenCL resources, you will associate them with the context as they are created



Contexts

```
cl_context    clCreateContext (const cl_context_properties *properties,
                               cl_uint num_devices,
                               const cl_device_id *devices,
                               void (CL_CALLBACK *pfn_notify)(const char *errinfo,
                                         const void *private_info, size_t cb,
                                         void *user_data),
                               void *user_data,
                               cl_int *errcode_ret)
```

- This function creates a context given a list of devices
- The properties argument specifies which platform to use (if NULL, the default chosen by the vendor will be used)
- The function also provides a callback mechanism for reporting errors to the user

Command Queues

- A *command queue* is the mechanism for the host to request that an action be performed by the device
  - Perform a memory transfer, begin executing, etc.
- A separate command queue is required for each device
- Commands within the queue can be synchronous or asynchronous
- Commands can execute in-order or out-of-order

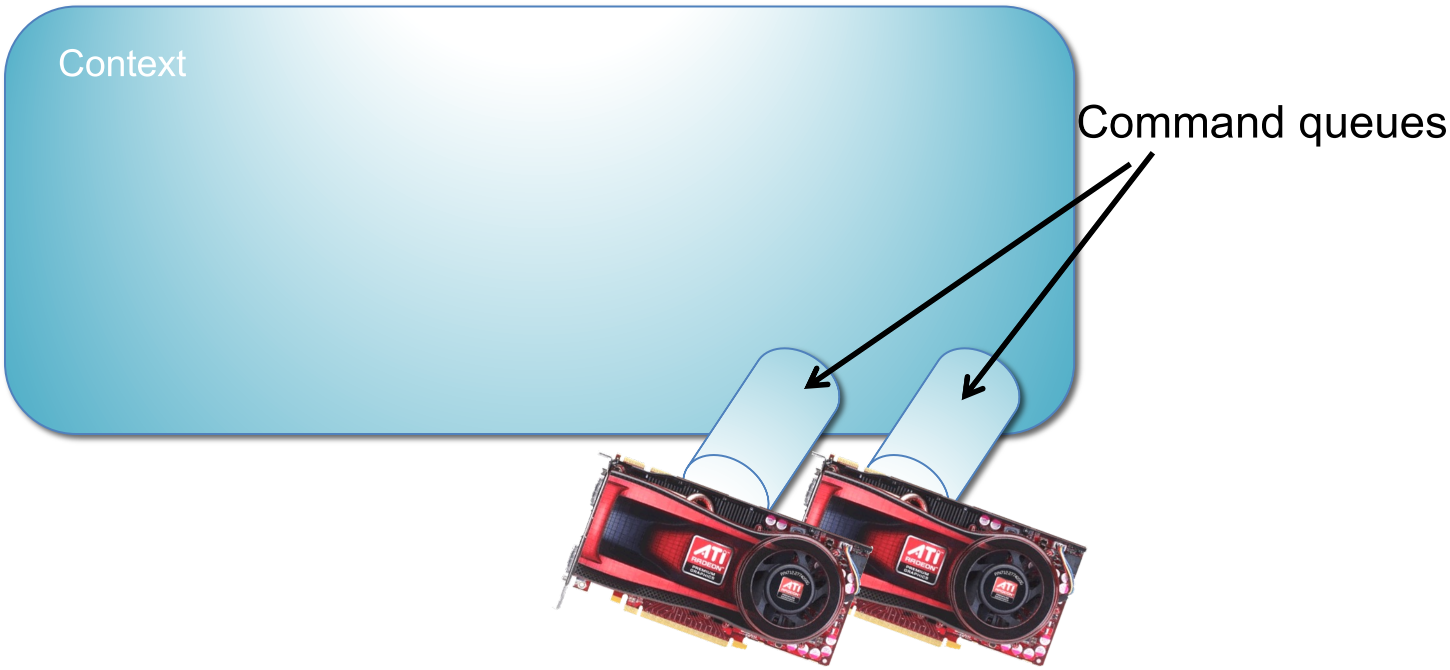
Command Queues

```
cl_command_queue    clCreateCommandQueue (cl_context context,
                                          cl_device_id device,
                                          cl_command_queue_properties properties,
                                          cl_int *errcode_ret)
```

- A command queue establishes a relationship between a context and a device
- The command queue properties specify:
  - If out-of-order execution of commands is allowed
  - If profiling is enabled

Command Queues

- Command queues associate a context with a device
  - Despite the figure below, they are not a physical connection



Memory Objects

- Memory objects are OpenCL data that can be moved on and off devices
  - Objects are classified as either buffers or images
- Buffers
  - Contiguous chunks of memory – stored sequentially and can be accessed directly (arrays, pointers, structs)
  - Read/write capable
- Images
  - Opaque objects (2D or 3D)
  - Can only be accessed via read_image() and write_image()
  - Can either be read or written in a kernel, but not both

Creating buffers

```
cl_mem    clCreateBuffer (cl_context context,
                          cl_mem_flags flags,
                          size_t size,
                          void *host_ptr,
                          cl_int *errcode_ret)
```

- This function creates a buffer (cl_mem object) for the given context
  - Images are more complex and will be covered in a later lecture
- The flags specify:
  - the combination of reading and writing allowed on the data
  - if the host pointer itself should be used to store the data
  - if the data should be copied from the host pointer

Transferring Data

- OpenCL provides commands to transfer data to and from devices
  - clEnqueue{Read|Write}{Buffer|Image}
  - Copying from the host to a device is considered *writing*
  - Copying from a device to the host is *reading*
- The write command both initializes the memory object with data and places it on a device
  - The validity of memory objects that are present on multiple devices is undefined by the OpenCL spec (i.e. are vendor specific)
- OpenCL calls also exist to directly map part of a memory object to a host pointer

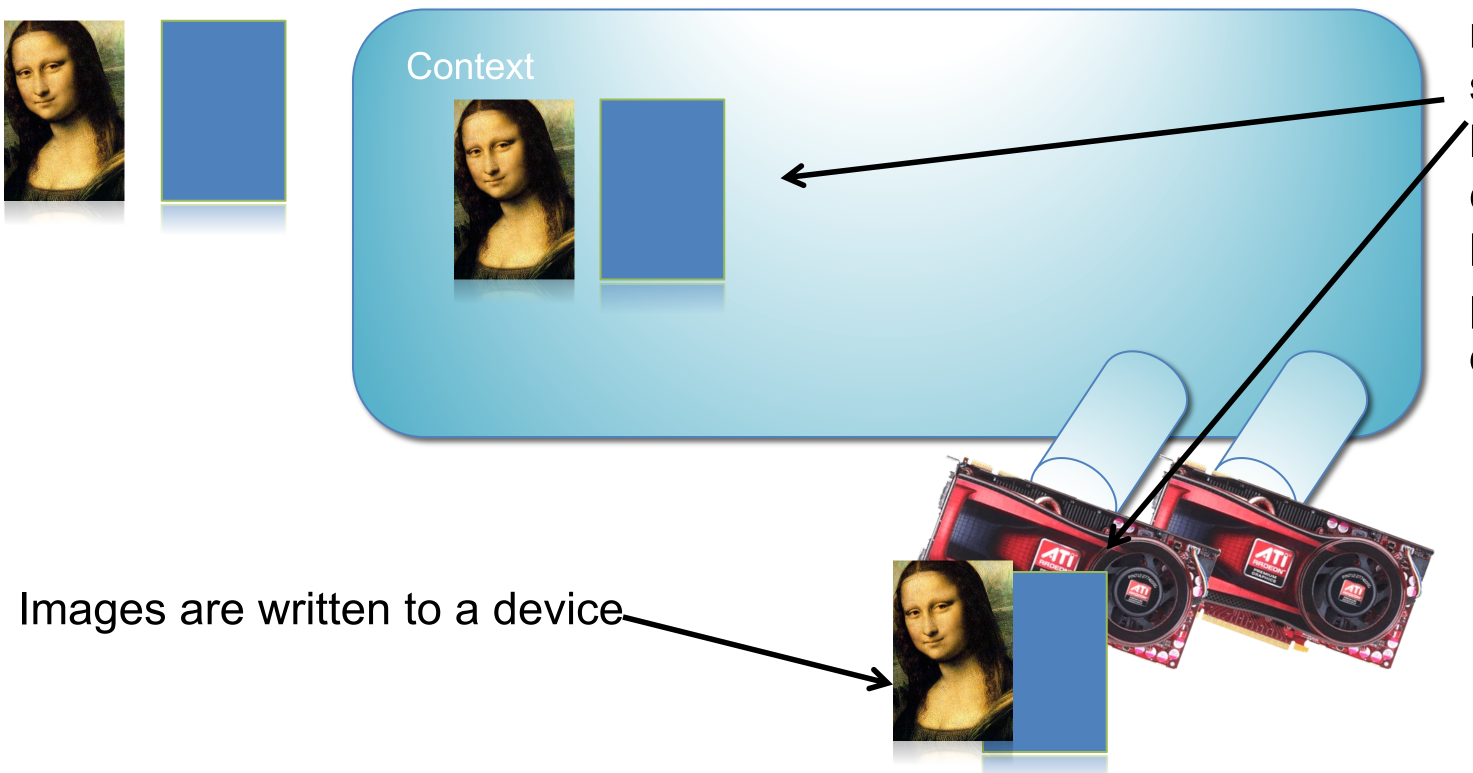
Transferring Data

```
cl_int clEnqueueWriteBuffer (cl_command_queue command_queue,
                             cl_mem buffer,
                             cl_bool blocking_write,
                             size_t offset,
                             size_t cb,
                             const void *ptr,
                             cl_uint num_events_in_wait_list,
                             const cl_event *event_wait_list,
                             cl_event *event)
```

- This command initializes the OpenCL memory object and writes data to the device associated with the command queue
  - The command will write data from a host pointer (*ptr*) to the device
- The *blocking_write* parameter specifies whether or not the command should return before the data transfer is complete
- Events can specify which commands should be completed before this one runs

Transferring Data

- Memory objects are transferred to devices by specifying an action (read or write) and a command queue
  - The validity of memory objects that are present on multiple devices is undefined by the OpenCL spec (i.e. is vendor specific)



Outline

- OpenCL概述
- **OpenCL模型**
  - 平台模型
  - 执行模型
  - 内存模型
  - 编程模型
- **OpenCL程序设计**
  - 程序设计流程
  - **编程实例**

OpenCL实例：向量加法

- The "hello world" program of data parallel programming is a program to add two vectors

  C[i] = A[i] + B[i] for i=1 to N

- For the OpenCl solution, there are two parts
  - Host code
  - Kernel code

OpenCL实例：向量加法

Address space qualifier

kernel qualifier

Global thread index

Vector addition

```c
// OpenCL Kernel Function for element by element vector addition
__kernel void VectorAdd(__global const float* a,
                        __global const float* b,
                        __global float* c,
                        int iNumElements)
{
    // get index into global data array
    int iGID = get_global_id(0);

    // bound check (equivalent to the limit on a 'for' loop for standard/serial C code
    if (iGID >= iNumElements)
    {
        return;
    }

    // add the vector elements
    c[iGID] = a[iGID] + b[iGID];
}
```

OpenCL实例：向量加法

Setup kernel grid

Allocate host resources

Create device context

Allocate device resources

Populate device memory

```
int main(int argc, char **argv) {
    // set and log Global and Local work size dimensions
    localWorkSize = 256;
    globalWorkSize = RoundUp(localWorkSize, iNumElements);

    // Allocate and initialize host arrays
    srcA = malloc(sizeof(cl_float) * globalWorkSize);
    srcB = malloc(sizeof(cl_float) * globalWorkSize);
    dst  = malloc(sizeof(cl_float) * globalWorkSize);
    FillArray(srcA, iNumElements);
    FillArray(srcB, iNumElements);

    // Create the context
    clGetPlatformIDs(1, &platform, NULL);
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
    context = clCreateContext(0, 1, &device, NULL, NULL, &ciErr1);

    // Create a command-queue
    cqCommandQueue = clCreateCommandQueue(context, device, 0, &ciErr1);

    // Allocate the OpenCL buffer memory objects
    dSrcA = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(cl_float) * globalWorkSize, ...);
    dSrcB = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(cl_float) * globalWorkSize, ...);
    dDst  = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(cl_float) * globalWorkSize, ...);

    // Copy data to GPU device
    clEnqueueWriteBuffer(cqCommandQueue, dSrcA, CL_FALSE, 0, sizeof(cl_float) * globalWorkSize, srcA, ...);
    clEnqueueWriteBuffer(cqCommandQueue, dSrcB, CL_FALSE, 0, sizeof(cl_float) * globalWorkSize, srcB, ...);
```

OpenCL实例：向量加法

- Build kernel program
- Set kernel arguments
- Launch kernel execution
- Read destination buffer

```
// Build the program
progSrc = oclLoadProgSource("VectorAdd.cl", ...);
prog = clCreateProgramWithSource(context, 1, progSrc, ...);
clBuildProgram(prog, 0, NULL, NULL, NULL, NULL);

// Create the kernel
kernel = clCreateKernel(prog, "VectorAdd", &ciErr1);

// Set kernel the Arguments
clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&dSrcA);
clSetKernelArg(kernel, 1, sizeof(cl_mem), (void*)&dSrcB);
clSetKernelArg(kernel, 2, sizeof(cl_mem), (void*)&dDst);
clSetKernelArg(kernel, 3, sizeof(cl_int), (void*)&iNumElements);

// Launch kernel
clEnqueueNDRangeKernel(cqCommandQueue, kernel, 1, NULL, &globalWorkSize, &localWorkSize,...);

// Synchronous/blocking read of results, and check accumulated errors
clEnqueueReadBuffer(cqCommandQueue, dDst, ...);

// Cleanup
Cleanup();
return 0;
}
```

参考资料

- OpenCL异构并行计算：原理、机制与优化实践陈铁,吴长江,刘文志 著
- Khronos OpenCL Registr, https://www.khronos.org/registry/OpenCL/