C-DAC Four Days Technology Workshop

ON

Hybrid Computing – Coprocessors/Accelerators
Power-Aware Computing – Performance of Applications Kernels

hyPACK-2013
(Mode-4 : GPUs)

Lecture Topic:
An Overview of OpenCL

Venue : CMSD, UoHYD ; Date : October 15-18, 2013

Heterogeneous Computing with OpenCL

Lecture Outline

Following topics will be discussed

- Part-I : An introduction to Heterogeneous comp. - OpenCL
- Part-II : The OpencL Specification - Kernels
- Part-III : OpenCL Device Architectures
- Part-IV : OpenCL Basic Examples
- Part-V : Understanding OpenCL's Concurrency and Execution Model

Source : NVIDIA, Khronos AMD, References

Source : References given in the presentation

Part-1

Introduction to Heterogeneous Computing

Why OpenCL ?

Software in Many-core world

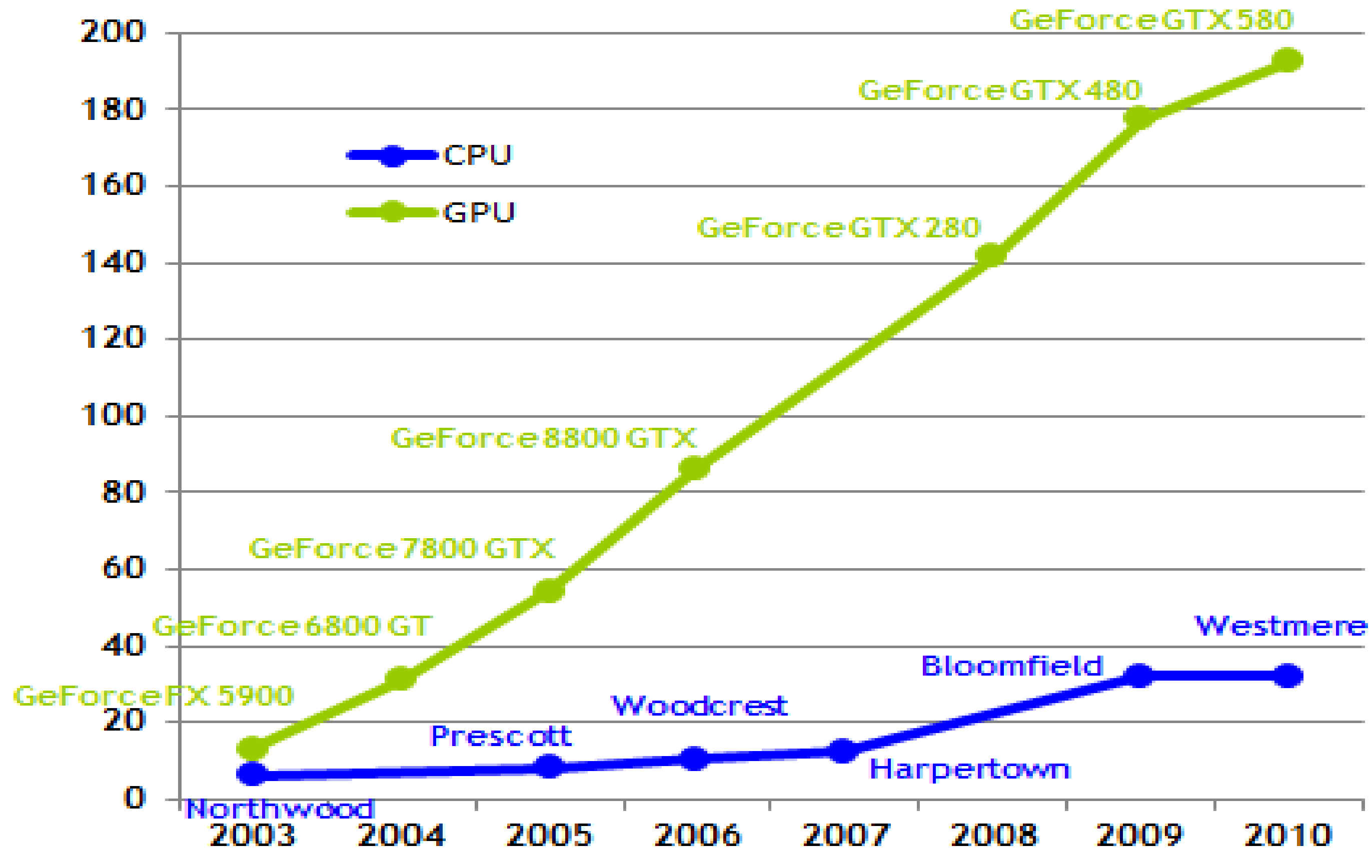


Figure Floating-Point Operations per Second and Memory Bandwidth for the CPU and GPU

Source : NVIDIA, Khronos, References

Software in Many-core world

Parallel Hardware delivers performance by running multiple operations at the same time

- ❖ Concurrency (Stream of operations - threads)
- ❖ Resource Utilization
- ❖ Data Parallel /Task Parallel / Load Balancing
- ❖ Manipulating low-level details of parallel computer is beyond our control – Performance
- ❖ Issues
  - Number of Operations & Data Movement

Source : Khronos, OpenCL Prog, Guide by Aaftab Munshi etc. & References

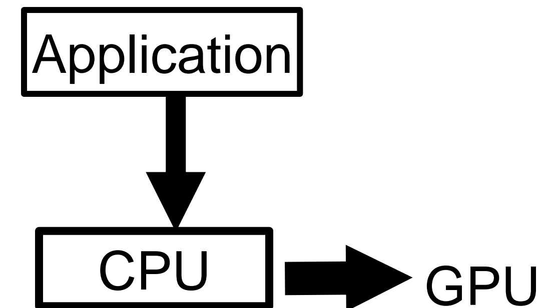
Software in Many-core world

GPU Computing : Think in Parallel - Some Design Goals

- ❖ Performance =

  parallel hardware + scalable parallel program

- ❖ GPU Computing drives new applications
  - Reducing “Time to Discovery”
  - 100 x Speedup changes science & research methods



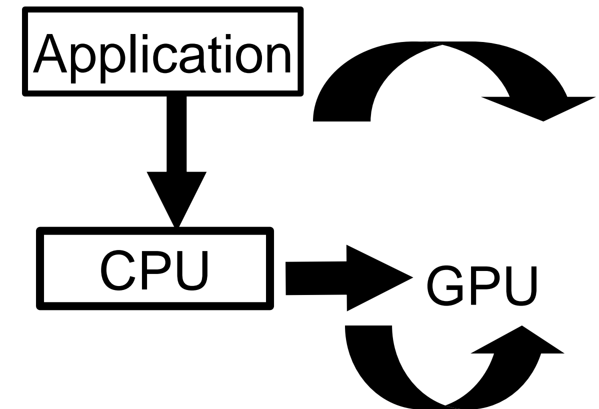
- ❖ New applications drive the future of GPUs
  - Drives new GPU capabilities
  - Drives hunger

Source : NVIDIA, Khronos, AMD, References

GPU Programming : Two Main Challenges

GPU Challenges with regard to Scientific Computing

Challenge : Programmability



- Example : Matrix Computations
  - To port an existing scientific application to a GPU
- GPU memory exists on the card itself
  - Must send matrix array over PCI-Express Bus
    - Send **A**, **B, C** to **GPU** over PCIe
    - Perform GPU-based computations on **A,B, C**
    - Read result **C** from **GPU** over PCIe
- The user must focus considerable effort on optimizing performance by manually orchestrating data movement and managing thread level parallelism on GPU.

Source : NVIDIA, Khronous, AMD, References

GPU Programming : Two Main Challenges

Challenge

- ❖ Example : Non-Scientific Computation - Video Games (Frames) (A single bit difference in a rendered pixel in a real-time graphics program may be discarded when generating subsequence frames)

- ❖ Scientific Computing : Single bit error - Propagates overall error

- ❖ **Past/Current History :** Most GPUs support single/double precision, 32/64 bit floating point operation, - all GPUs have necessarily implemented the full IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754)

Source : NVIDIA, Khronos AMD, References

Software in Many-core world

GPU Computing : Think in Parallel : Why Are GPUs So Fast?

- ❖ Optimized for structured parallel execution
  - Extensive ALU counts & Memory Bandwidth
  - Cooperative multi-threading hides latency
- ❖ Shared Instructions Resources
- ❖ Fixed function units for parallel workloads dispatch
- ❖ Extensive exploitations of Locality
- ❖ Performance /(Cost/Watt); Power for Core
- ❖ Structured Parallelism enables more flops less watts

Source : NVIDIA, Khronos AMD, References

GPU Computing: Use Parallelism Efficiently

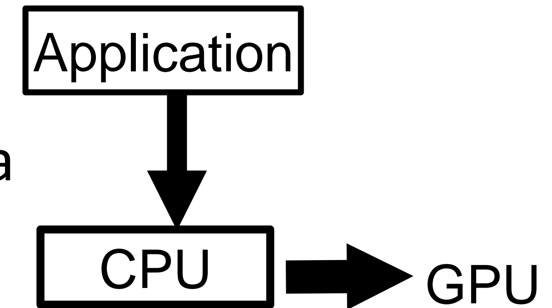
- ❖ Partition your computation to keep the GPU multiprocessors equally busy
  - Many threads, many thread blocks
- ❖ Keep resource usage low enough to support multiple active thread blocks per multiprocessor
  - Registers, shared memory

Source : NVIDIA, Khronos AMD, References

GPU Computing : Think in Parallel

GPU Computing: Take Advantage of Shared Memory

- ❖ Hundreds of times faster than global memory
- ❖ Threads can cooperate via shared memory
- ❖ Use one/ a few threads to load/computer data shared by all threads
- ❖ Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order non-coalesceable addressing
  - Matrix transpose example later



Source : NVIDIA, Khronos AMD, References

GPU Computing : Think in Parallel

GPU Computing: Optimise Algorithms for the GPU

- ❖ Maximize independent parallelism
- ❖ Maximize arithmetic intensity (math/bandwidth)
- ❖ Sometimes it's better to recompute than to cache
    - GPU spends its translators on ALUs, not memory
- ❖ Do more computation on the GPU to avoid costly data transfers
    - Even low parallelism computations can sometimes be faster than transferring back and forth to host

Source : NVIDIA, Khronos AMD, References

Software in Many-core world

- ❖ High Level Abstraction that hide complexity of hardware

- ❖ A heterogeneous programming language exposes heterogeneity
  - Trend towards increasing abstraction
  - **One language does'nt have to address the needs of every community of programmers**
  - High level frame works - High level languages and map to a low-level hardware abstraction layer for portability

- ❖ OpenCL is hardware-abstraction

Source : NVIDIA, Khronos AMD, References

Part-2 (a)

Introduction to OpenCL

Standardization

Source : NVIDIA, Khronos AMD, References

OpenCL tries to Standardize Parallel Programming

To standardize general purpose parallel programming for any application

Suitable for Heterogeneous systems – different Mircoprocessor Architectures (Ex : PCs - X86; PCs with discrete or integrated GPUs, Cell Phones, Embedded Systems

Khronos OpenCL working group making aggressive progress (www.khronos.org)

Source : NVIDIA, Khronos AMD, References

OpenCL tries to Standardize Parallel Programming

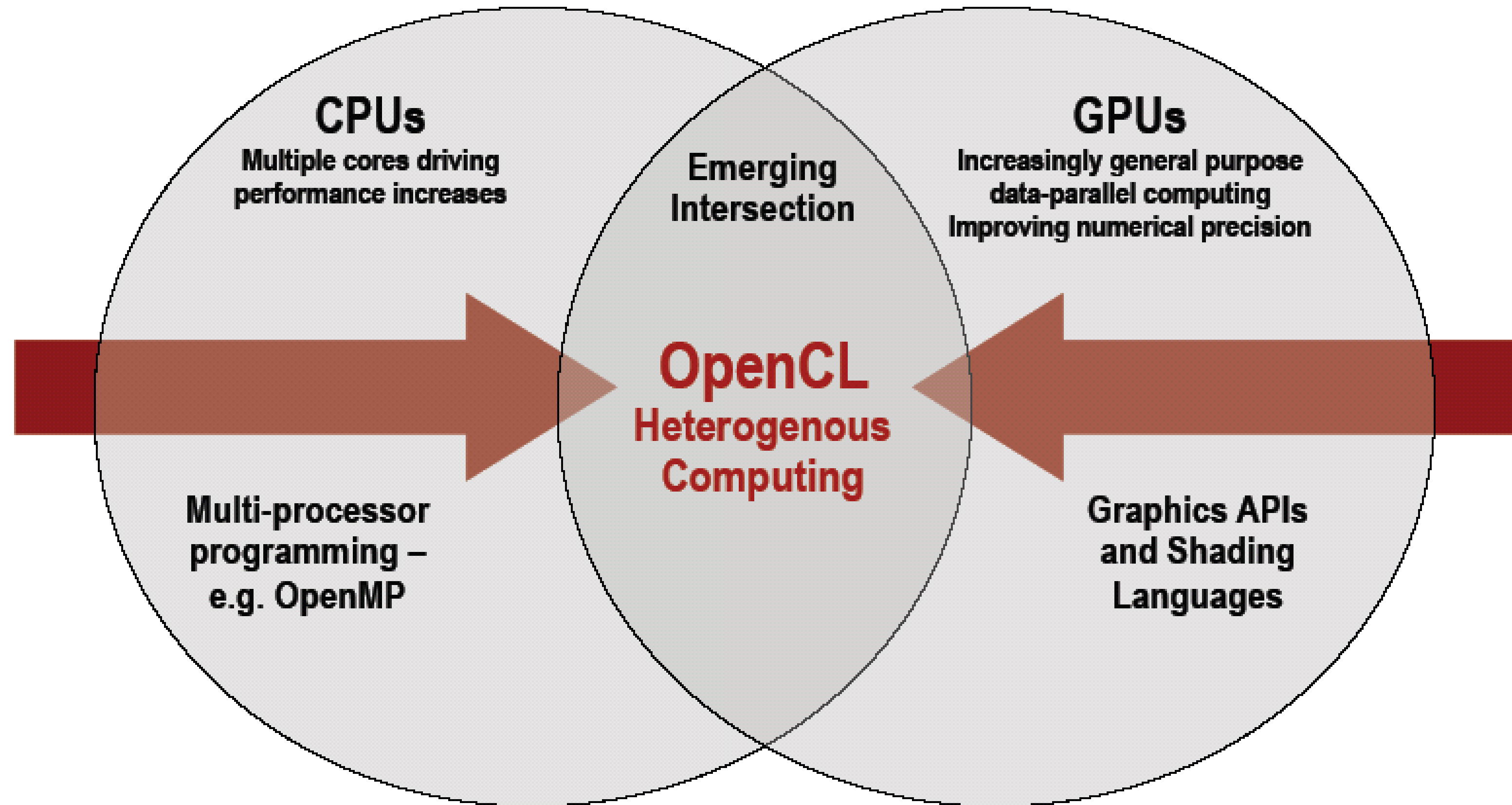
What Does OpenCL Mean ? : Challenging Objectives :

- Standardize framework and language for multiple heterogeneous processors
  - Developed in collaboration with industry leaders
- Software Developers
  - OpenCL enabled you to write parallel programs that will run portably on many devices
  - Royalty free – with no cost to use the API
- **End-User Benefits**
  - A wide range of innovative applications will be enabled and accelerated by unleashing the parallel computing capabilities of their systems and devices

Source : NVIDIA, Khronos AMD, References

OpenCL tries to Standardize Parallel Programming

Processor Parallelism : Processor Parallelism



OpenCL – Open Computing Language

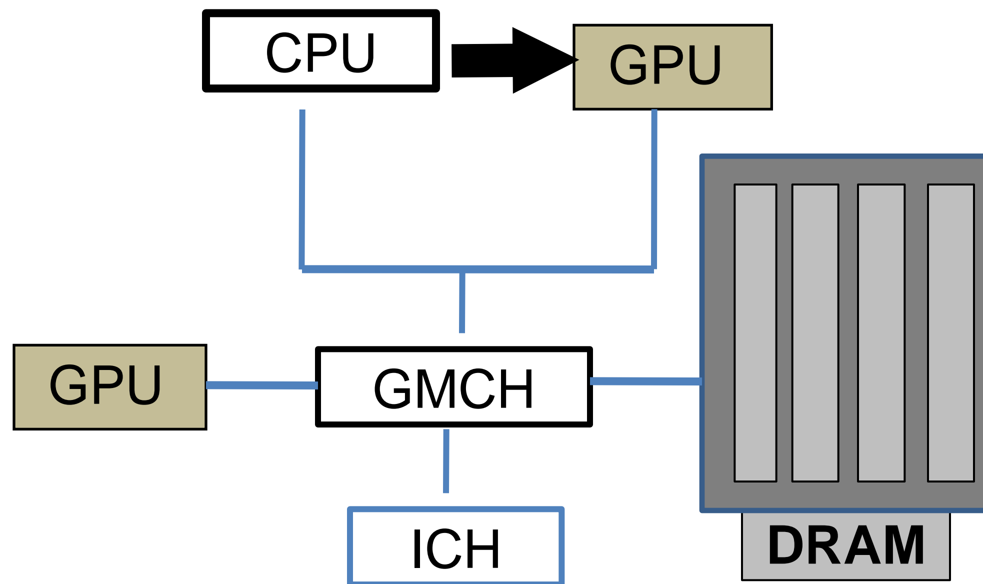
Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

Source : NVIDIA, Khronos AMD, References

OpenCL tries to Standardize Parallel Programming

Why OpenCL

Need Hybrid Programming on Heterogeneous Comp. Platforms



The future belongs to heterogeneous many-core platforms

Source : Khronos, OpenCL Prog, Guide by Aaftab Munshi etc. &References

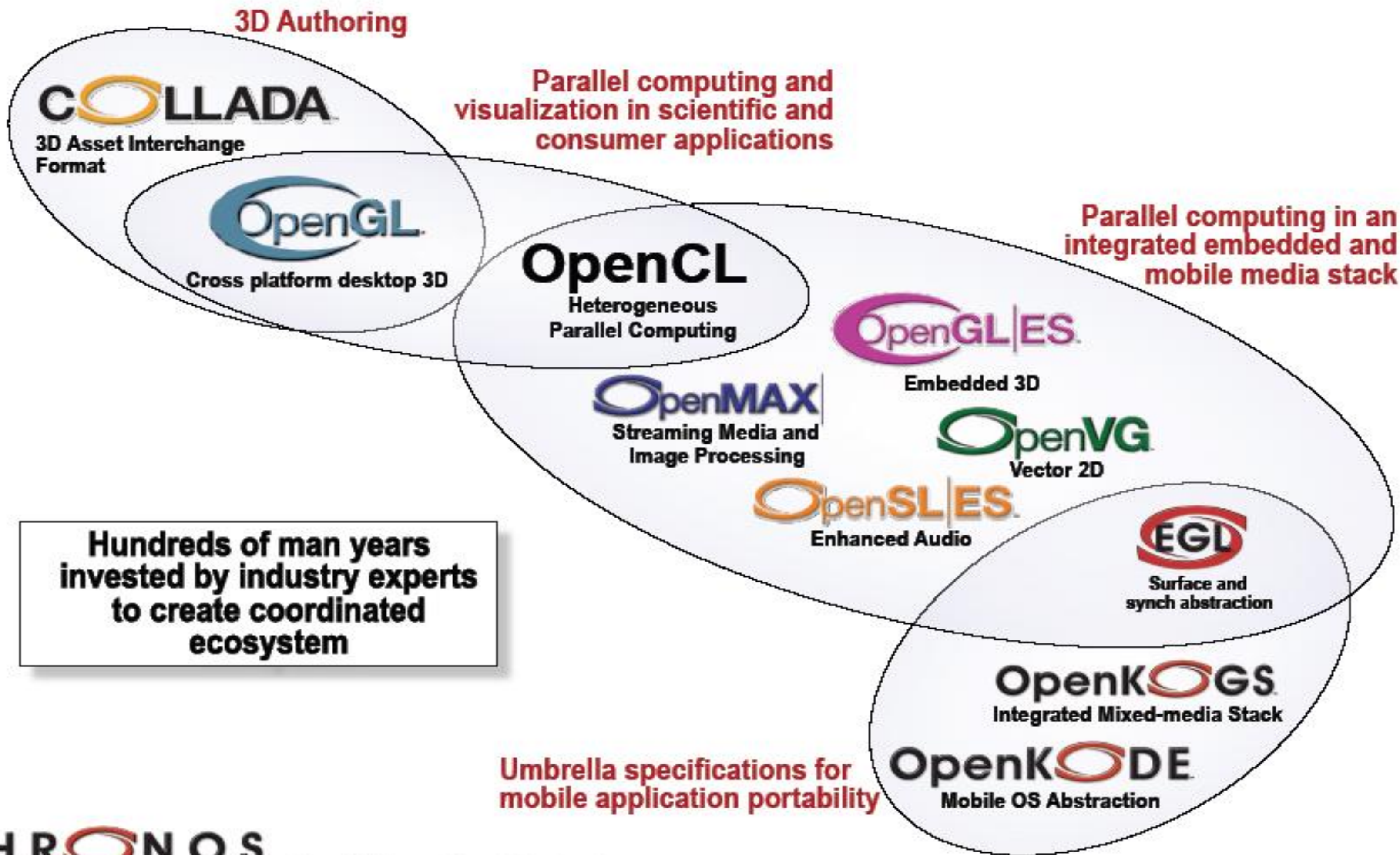
OpenCL tries to Standardize Parallel Programming

OpenCL Specification working group :

3DLabs, Activation Blizzard, AMD, Apples, ARM, Barco, Broadcom, Codeplay, Electronic Arts, Ericsson, Freescale, Hi, IBM, Intel, Imagine technologies, Motorla, Movid, Nokia, Nvidia, QNX, RapidMind Samsung, Seaweed,Takuni, Texas Instruments, University (Sweden), Microsoft

Source : NVIDIA, Khronos AMD, References

OpenCL and the Khronos EcoSystem



Source : NVIDIA, Khronos AMD, References

OpenCL tries to Standardize Parallel Programming

Why OpenCL

Hybrid Programming on Heterogeneous Comp. Platforms

Co-existence of Accelerators
Intel Xeon (Phi) RC-FPGA, & GPGPUs

Heterogeneous Comp. Platforms – Power & Energy Efficiency

Capacitance = 2.2 C
Voltage = 0.6V
Frequency = 0.5f
Power = 0.396 CV^2f

How our software should adapt to these platforms ?

Source : NVIDIA, Khronos AMD, References

OpenCL tries to Standardize Parallel Programming

Background & Challenging Objectives :

- ❖ OpenGL: Open Graphics Library
  - ➢ Widely supported application programming interface (API) for graphics ONLY
- ❖ OpenCL: "CL" Stands for Computing Language
  - ➢ providing an API library
  - ➢ Modifies C and C++ parallel programming
  - ➢ New Initiatives for other programming languages(Fortran)

Aim: to standardize general purpose parallel programming for any application

Source : NVIDIA, Khronos AMD, References

The OpenCL Standard

OpenCL Working Group : Challenging Objectives

- ❖ Diverse Industry Participation
  - Processor vendors, System OEMS, Middleware vendors, Application Developers
- ❖ Many Industry-leading experts involved in OpenCL's design
  - A healthy diversity of industry perspectives
- ❖ Apple initially proposed the working group
  - And served as specification editor

Source : NVIDIA, Khronos AMD, References

The OpenCL Standard

❖ **Challenging Objectives :**

➢ Arrive at a common set of programming standards that are acceptable to a range of competing needs and requirements

➢ ***The Khronos*** consortium – manages the OpenCL standard

- Developed an applications programming interface (API) that is general enough to run on significantly different architectures while being adaptable enough that each hardware platforms can still obtain high performance.

- Using the core language and correctly following the specification, any program designed for one-vendor can execute on another's hardware.

Source : NVIDIA, Khronos AMD, References

The OpenCL Standard

Challenging Objectives :

- ❖ Diverse Industry Participation
  - ➢ Processor vendors, System OEMS, Middleware vendors, Application Developers
- ❖ Many Industry-leading experts involved in OpenCL's design
  - ➢ A healthy diversity of industry perspectives
- ❖ Apple initially proposed the working group
  - ➢ And served as specification editor

Source : NVIDIA, Khronos AMD, References

The OpenCL Standard

❖ **Challenging Objectives** :

- OpenCL C is a restricted version of the C99 language with extension appropriate for executing data-parallel code on a variety of heterogeneous devices.

- Aimed for full support for the IEEE 754 formats

- Programming language, well suited to the capabilities of current heterogeneous platforms

Source : NVIDIA, Khronos AMD, References

The OpenCL Standard

❖ **Challenging Objectives :**

➢ The model set forth by OpenCL creates portable, vendor- and device-independent programs that are capable of being accelerated on many different platforms.

- The OpenCL API is C wit h a C++ Wrapper API that is defined in terms of the C-API.
- There are third-party bindings for many languages, including Java, Python, and .NET
- The code that executes on an OpenCL device, which in general is not the same device as the host-CPU, is written in the OpenCL C language.

Source : NVIDIA, Khronos AMD, References

OpenCL : Standardize Parallel Programming

- **Threading in Model for data level parallelism OpenCL**
  - Closely resembles the models in AMD-ATI Stream, CUDA & RapidMind
  - OpenCL threading is largely implicit
  - OpenCL allows programmers to manage threads more explicitly if programmers wish
- **Task-level parallelism**
  - Concurrently execute multiple kernels on multiple kernels on multiple CPUs, GPUs or systems with mixed architecture

Source : NVIDIA, Khronos AMD, References

OpenCL Design Requirements

- **Use all computational resources in system**
  - Program GPUs, CPUs and other processors as peers
  - Support both data- and task- parallel compute models
- **Efficient c-based parallel programming model**
  - Abstract the specified of underlying hardware
- **Abstraction is low-level, high-performance but device-portable**
  - Approachable –but primarily targeted at expert developers
  - Ecosystem foundation – no middleware or "convenience" functions
- **Implementation on a range of embedded, desktop, and server systems**
  - HPC desktop, and handheld profiles in on specification
- **Drive future hardware requirements**
  - Floating point precision requirements
  - Application to both consumer and HPC applications

Source : NVIDIA, Khronos AMD, References

OpenCL Design Requirements

- **Efficient c-based parallel programming model**
  - Abstract the specified of underlying hardware
- **Abstraction is low-level, high-performance but device-portable**
  - Approachable –but primarily targeted at expert developers
  - Ecosystem foundation – no middleware or “convenience” functions

Source : Khronous, References

OpenCL Design Requirements

- **Implementation on a range of embedded, desktop, and server systems**
  - HPC desktop, and handheld profiles in on specification
- **Drive future hardware requirements**
  - Floating point precision requirements
  - Application to both consumer and HPC applications

Source : NVIDIA, Khronos AMD, References

Design Goals of OpenCL

- Use all computational resources in system
  - GPUs and CPUs as peers
  - Data- and task- parallel compute model
- Efficient parallel programming model
  - Based on C
  - Abstract the specifics of underlying hardware
- Specify accuracy of floating-point computations
  - IEEE 754 compliant rounding behaviour
  - Define maximum allowable error of math functions

Source : NVIDIA, Khronos AMD, References

OpenCL Task Parallel Execution Model

- Data-parallel execution model must be implemented by all OpenCL compute devices
- Some computer devices such as CPUs can also execute task parallel compute kernels
  - Executes as a single work-item
  - A compute kernel written in OpenCL
  - A native C / C++ function

Source : NVIDIA, Khronos AMD, References

Part-2 (b)

OpenCL – Models

Source : NVIDIA, Khronos AMD, References

Conceptual Foundations of OpenCL

An Application for a heterogeneous platform must carry out the following steps.

- ❖ Discover the completion that make-up the heterogeneous system
- ❖ Probe the characteristics of these components, so that the software can adapt to specific features of different hardware elements
- ❖ Create the blocks of instructions (Kernels) that will run on the platform

Source : NVIDIA, Khronos AMD, References

Conceptual Foundations of OpenCL

An Application for a heterogeneous platform must carry out the following steps.

- ❖ Set up and manipulate memory objects involved in the computation.
- ❖ Execute the kernels in the right order and on the right components of the system
- ❖ Collect the final results
  - Above steps are accomplished through a series of APIs inside OpenCL plus a programming environment for the kernels

Source : NVIDIA, Khronos AMD, References

Conceptual Foundations of OpenCL

An Application for a heterogeneous platform must carry out the following steps.

- ❖ Discover the completion that make-up the heterogeneous system
- ❖ Probe the characteristics of these components, so that the software can adapt to specific features of different hardware elements
- ❖ Create the blocks of instructions (Kernels) that will run on the platform

Source : NVIDIA, Khronos AMD, References

Conceptual Foundations of OpenCL

An Application for a heterogeneous platform must carry out the following steps.

- ❖ Set up and manipulate memory objects involved in the computation.
- ❖ Execute the kernels in the right order and on the right components of the system
- ❖ Collect the final results
  - Above steps are accomplished through a series of APIs inside OpenCL plus a programming environment for the kernels

Source : NVIDIA, Khronos AMD, References

The OpenCL Specification – Models

❖ The OpenCL specification is defined in four parts, called models, that can be summarized as follows.

- Platform Model
- Execution Model
- Memory Model
- Programming Model

Source : NVIDIA, Khronos AMD, References

The OpenCL Specification – Models

❖ OpenCL Software Stack

- **Platform Layer**
  - ➢ Query and select computer devices in the system
  - ➢ Initialize a compute device(s)
  - ➢ Create compute contexts and work-queues
- **Runtime**
  - ➢ Resource management
  - ➢ Execute compute kernels
- **Compiler**
  - ➢ A subset of ISO C99 with appropriate language additions
  - ➢ Compile and build compute program executable
  - ➢ Online or offline

Source : NVIDIA, Khronos AMD, References

The OpenCL Specification – Models

- The OpenCL specification is defined in four parts, called models, that can be summarized as follows.

  - Platform Model
    - High Level description of the heterogeneous system

  - Execution Model
    - An abstract representation of how stream of instructions execute on the heterogeneous system

Source : NVIDIA, Khronos AMD, References

The OpenCL Specification – Models

- The OpenCL specification is defined in four parts, called models, that can be summarized as follows.
  - Memory Models
    - The Collection of memory regions within OpenCL and how they interact during at OpenCL computation
  - Programming Model
    - The high-level abstractions a programmer uses when designing algorithms to implement an application

Source : NVIDIA, Khronos AMD, References

The OpenCL Specification

- The OpenCL specification is defined in four parts, called models, that can be summarized as follows.

  - Platform Model
  - Execution Model
  - Memory Model
  - Programming Model

Source : NVIDIA, Khronos AMD, References

Part-2 (c)

OpenCL Specification

Platform Model

(In brief)

The OpenCL Specification

❖ **Platform model :**

- Specifies that there is one processor coordinating the execution (***the host***) and one or more processors capable of executing OpenCL C Code (***the devices***).
- It defines an abstract hardware model that is used by programmers when writing OpenCL functions (Called ***Kernels***) that execute on the devices.
- The platform model defines the relation between the host an device.
  - i.e., OpenCL implementation executing on a host x86 GPU, which is using a GPU device as an accelerator

Source : NVIDIA, Khronos AMD, References

The OpenCL Specification

❖ **Platform model :**

- Platforms can be thought of a vendor – specific implementations of the OpenCL API.
- The platform model also presents an abstract device architecture that programmers target writing OpenCL C code.
- Vendors map this abstraction architecture to the physical hardware.

Source : NVIDIA, Khronos AMD, References

OpenCL PLATFROM AND DEVICES

Host-Device Interaction

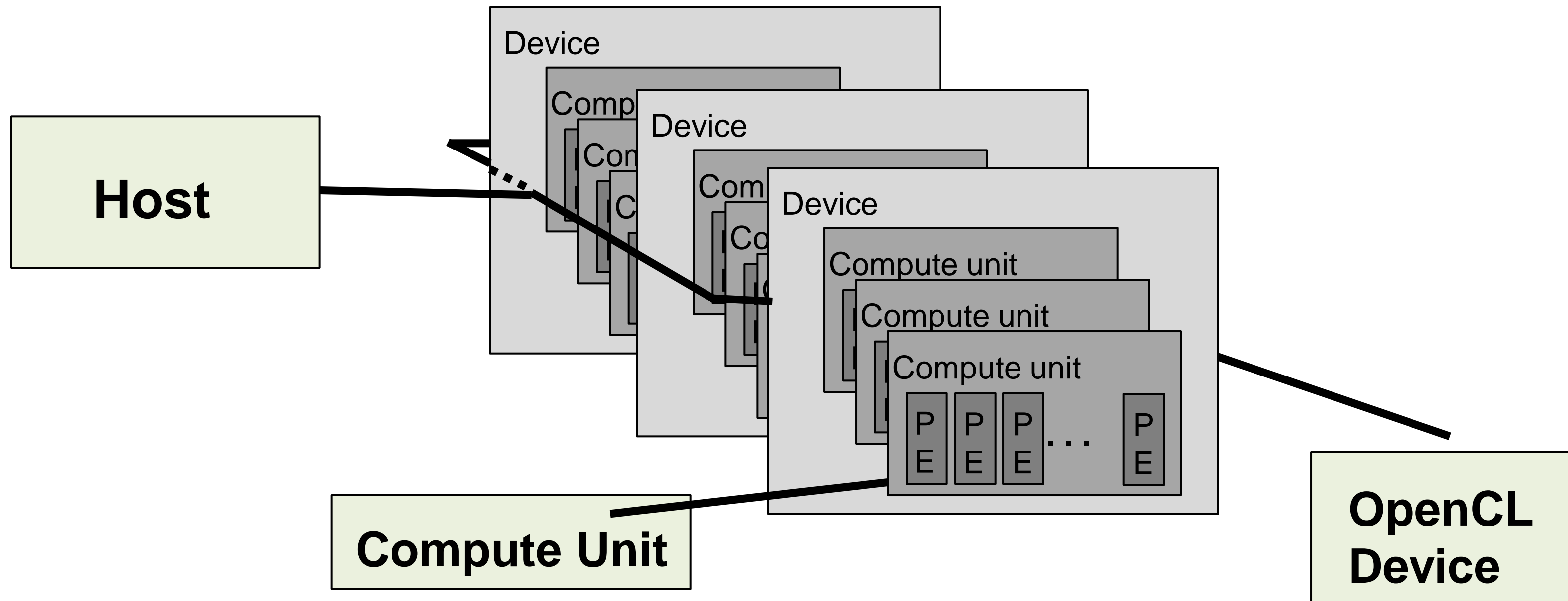
❖ Platform Model

- Provides an abstract hardware model for devices
- Present an abstract device architecture that programmers target when writing OpenCL C code.
- Vendor-specific implementation of the OpenCL API.

❖ Platform Model

- Defines a device as an array of compute units
    - Compute units are further divided into processing elements
    - OpenCL device schedule execution of instructions.

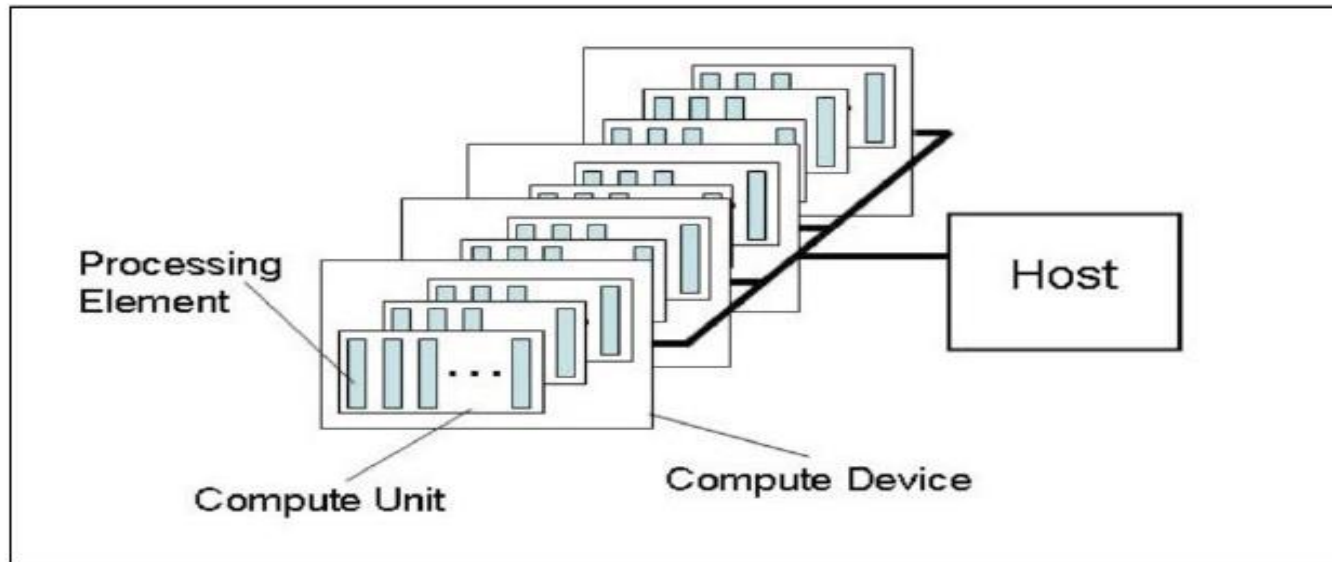
OpenCL Platform Model



The platform model defines an abstract architecture for devices.

- The host is connected to one or more devices
- Device is where the stream of instructions (or kernels) execute (an OpenCL device is often referred to as a **compute device**
- A device can be a CPU, GPU, DSP, or any other processor provided by Hardware and supported by the OpenCL Vendor

OpenCL Platform Model



- One Host + one or more compute Devices
  - Each compute Device is connected to one or more Compute Units.
    - Each compute Unit is further divided into one or more Processing Elements

Source : NVIDIA, Khronos AMD, References

OpenCL PLATFROM Model

How to discover available platforms for a given system ?

```
cl_int
ClGetPlatformIds(cl_unit num_entries,
                 cl_platform_Id *platforms,
                 cl_unit *num_platforms)
```

❖ Platform Model

- Defines a device as an array of compute units
  - Compute units are further divided into processing elements
  - OpenCL device schedule execution of instructions.

Source : NVIDIA, Khronos   AMD, References

OpenCL PLATFORM Model

How to discover available platforms for a given system.

❖ Application calls **`ClGetPlatformIds() twice`**

- The **first** call passes an **unsigned int** pointer as the **`num_platforms`** argument and NULL is passes as the **platform** argument.
  - The programmer can then allocate space to hold the platform information.
- The **second** call, a **`cl_platform_id`** pointer is passed to the implementation with enough space allocated for **`num_entries`** platforms.

Source : NVIDIA, Khronos AMD, References

OpenCL PLATFROM AND DEVICES

After platforms have been discovered, How to determine which implementation (vendor) the platform was defined by ?

The **`ClGetPlatformInfo()`** call determines implementation

The **`clGetDeviceIDs()`** call works very similar to **`ClGetPlatformId()`**

How to use `device_type` argument ?

GPUs : **`cl_DEVICE_TYPE_GPU`**

CPUs : **`cl_DEVICE_TYPE_CPU`**

All devices : **`cl_DEVICE_TYPE_ALL`** & other options

`Cl_GetDeviceinfo()` is called to retrieve information such as name, type, and vendor from each device.

Source : NVIDIA, Khronos AMD, References

OpenCL PLATFROM Model

After platforms have been discovered, How to determine which implementation (vendor) the platform was defined by ?

The **`clGetDeviceIDs()`**

```
cl_int
clGetDeviceIDs(cl_platform_id platform,
                   cl_DEVICE_TYPE_GPU device_type,
                   cl_unit num_entries,
                   cl_device_id *devices,
                   cl_uint *num_devices)
```

OpenCL PLATFORM Model

How to get printed information about the OpenCL, supported platforms and devices in a system ?

`CLinfo` prorgam in the AMD APP SDK

Uses **`clGetplatforminfo()`** and **`clGetDeviceInfo()`**

Hardware details such as memory size and bas widths are available using the commands

$ **`./CLinfo`** program gives complete information

OpenCL PLATFROM AND DEVICES

$ **./CLinfo**

Number of platforms :	1
Platform Profiles :	FULL_PROFILE
Platform Version :	OpenCL 1.1 AMD SDK –v2.4
Platform Name :	AMD Accelerated Parallel Processing
Platform Vendor :	Advanced Micro Devices, Inc.
Number of Devices :	2
Device Type :	CL_DEVICE_TYPE_GPU
Name :	Cypress
Max Compute Units :	20
Address bits	32

OpenCL PLATFROM AND DEVICES

$ **./CLinfo**

Max Memory Allocation:	268435456
Global Memory size :	1073741824
Constant buffer size :	65536
Local Memory type :	Scratchpad
Local Memory size :	32768
Device endianess :	little
Device Type :	CL_DEVICE_TYPE_CPU
Max Compute units :	16
Name :	AMD Phenom™ 11 X4 945 Processor

Source : NVIDIA, Khronos AMD, References

Part-2 (d)

OpenCL Specification

Execution Model

(In brief)

The OpenCL Specification

❖ **Execution model :**

- Defines
  - How the OpenCL environment is configured on the host
  - How kernels are executed on device
- This includes
  - Setting up an OpenCL context on the host,
  - Providing mechanism for host-device interaction, &
  - defining a concurrency model used for kernel execution on device
  - The host sets up a kernel for the GPU to run and instantiates it with some special degree of parallelism.

Source : NVIDIA, Khronos AMD, References

The OpenCL Execution Model

❖ Execution Model

- Application consists of **two** distinct parts
- **The host program**
  - Runs on the host
  - OpenCL does not define the details of how the host progrma works, only how it interacts with objects defined in OpenCL
- **A Collection of Kernels**
  - The Kernel execute on the OpenCL device

Source : NVIDIA, Khronos AMD, References

The OpenCL Execution Model

❖ **Execution Model - Kernels**

➢ **A Collection of Kernels**

- Execute on the OpenCL device
- Do the real work of an OpenCL application
- Simple functions transform **input** memory objects into **output** memory objects

Execution Model - Kernels

➢ **OpenCL defines two types of Kernels**

- **OpenCL** Kernels & **Native** Kernels

Source : Khronous, & References

The OpenCL Execution Model

❖ **Execution Model : Defines how the kernels execute**

➢ **Several Steps Exist.**

- **FIRST :** How an individual kernel runs on an OpenCL device ?
- **Second:** How the host defines the **context** for kenrel execution
- **THIRD:** How the kernels are enqueued for execution

Source : NVIDIA, Khronos AMD, References

The OpenCL Execution Model

❖ Execution Model - Kernels

➢ OpenCL Kernels

- Written in OpenCL C programming language and compiled with the OpenCL Compiler
- All OpenCL implementations must support OpenCL Kernels

➢ Native Kernels

- Functions created outside of **OpenCL** and accessed within **OpenCL** through a function pointer. (An Optional functionality within in **OpenCL** exist)

Source : NVIDIA, Khronos AMD, References

The OpenCL Execution Model

- The OpenCL Execution Environment defines the following how the kernel execute
  - Contexts
  - Command Queues
  - Events
  - Memory Objects (Buffers -large array /images
    - Buffers (allocate buffer & return memory object)
    - Image (2D & 3D)
  - Flush & Finish

Source : NVIDIA, Khronos AMD, References

Part-2 (e)

OpenCL Specification :Execution Model

How a Kernel Execute on an OpeCL Device

(In brief)

OpenCL Execution Model

❖ **OpenCL Program :**

- Kernels
  - Basic unit of executable – similar to a C function'
  - Data-parallel or task parallel
- Host Program
  - Collection of computer kernels and internal functions
  - Analogous to a dynamic library

Source : Khronous, & References

OpenCL Execution Model

- **Compute kernel**
  - Basic unit of executable code – similar to a C function
  - Data-parallel or task-parallel
- **Compute Program**
  - Collection of computer kernels and internal functions
  - Analogous to a dynamic library
- **Applications queue compute kernel execution instances**
  - Queued in-order
  - Executed in-order or out-of-order
  - Events are used to implement appropriate synchronization of execution instances

The OpenCL Execution Model

❖ **How a Kernel Execute on an OpeCL Device ?**

- **1.** A kernel defined on the Host
- **2. Issues a command :** The host program issues a command that submits the kernel for execution on an OpenCL device.
- **3. Creation of Integer index space :** The OpenCL runtime system creates an integer index space
- **4. Work-item :** An instance of the Kernel executes for each point in this index space and each such instance of an executing a kernel a **work-item**
- **Work-item** is identified by its coordinates in the index space & these coordinates are the global ID for the work-item.

Kernel Execution on an OpenCL Device

- OpenCL Approach :
  - The unit of concurrent execution in OpenCL is a ***work-item***
  - Map a single iteration of the loop to a ***work-item***
  - Tell the OpenCL runtime to generate as many ***work-items*** as elements in the input and output arrays
  - Allow the runtime to map those ***work-items*** to the underlying hardware i.e. CPU or GPU Cores in whatever way it views appropriate.

Source : NVIDIA, Khronos AMD, References

Kernel Execution on an OpenCL Device

- OpenCL implements hierarchy concurrent model
- OpenCL describes execution in fine-grained **work-items** and can dispatch vast number of **work-items** on architecture with hardware support for **fine-grained** threading
- When a kernel is executed, the programmer specifies the number of **work-items**
  - **Work-items** have unique global IDs from the index space
- **Work-items** are organized into **work-groups**. **Work-groups** have a unique work-group ID
- **Work-items** have a unique **local ID** within a **work-group**

Kernel Execution on an OpenCL Device

- Define N-Dimensional computation domain
  - **Work-items** should be created as an n-dimensional range (NDRange)
  - Each independent element of execution in N-D domain is called a **work-item**
  - The N-D domain defines the total number of **work-items** that execute in parallel – global work size.
  - The host program involves a kernel over an index space called an ***NDRange***
    - NDRange = “N-dimensional Range” & it can be a 1, 2 or 3-dimensional Range

Source : NVIDIA, Khronos AMD, References

Kernel Execution on an OpenCL Device

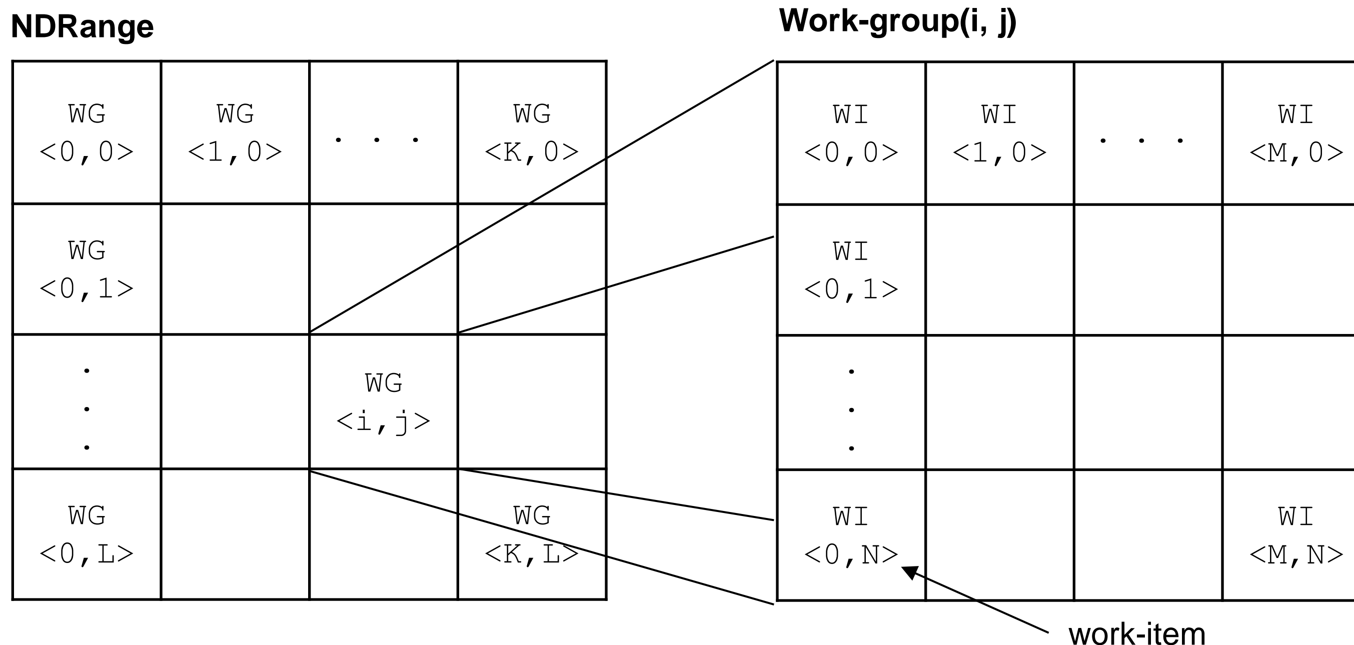
- Work-items can be grouped together – **work-group**
  - **Work-items** in **work-group** can communicate with each other
  - Can synchronize executing among **work-items** in group to coordinate memory access
- Execute multiple work-groups in parallel –
  - Provide more **coarse grained decomposition** of index space
- Mapping of global **work-size** to **work-groups**
  - Implicit or explicit

Source : NVIDIA, Khronos AMD, References

Kernel Execution on an OpenCL Device

Work-groups & Work-items

Scalability : Divide work-items of an **NDRange** into smaller, equally sized workgroups.



Work-items are created as an NDRange and grouped in workgroups.

An index space with **N** dimensions require work-groups to be specified using the same **N** dimensions : thus, a **three** dimensional index space requires **three**-dimensional work-groups.

Kernel Execution on an OpenCL Device

More about workgroups & work-items

- An **NDRange** is a *one-, two*-, or *three-* dimensional index space of **work-items** that will often map to the dimensions of either the input or the output data.
- The dimensions of the **NDRange** are specified as an **N**-element array of type **size_t** where **N** represents the number of dimensions used to described the **work-items** being created.

Kernel Execution on an OpenCL Device

- Kernels are the part of an OpenCL program that actually execute on a device. The OpenCL API
  - Enables an application to create a context for management of the execution of OpenCL commands, including those
    - describing the movement of data between and OpenCL memory structures and
    - the execution of kernel code that process this data to perform some meaningful task.
- The goal is often to represent parallelism programmability at the **finest granularity.**
- The generalization of the OpenCL interface and the lowest level kernel language **allows** efficient mapping to a wide range of hardware

Source : NVIDIA, Khronos AMD, References

Kernels and the OpenCL Execution Model

Work-groups & work-items

- ❖ Note that OpenCL requires that the index space sizes are evenly divisible by the work-group sizes in each dimension.
- ❖ For hardware efficiency, the work-group size is usually fixed to a favorable size
  - To satisfy the divisibility requirement, round-up the index space size in each dimension is required.
  - Specify the extra work-items in each dimension in such way that these extra items return immediately without outputting any data
  - Developer can pass "**NULL**" (implementation takes care-off)

Part-2 (f)

OpenCL Specification :Execution Model

Context

(In brief)

OpenCL Execution Model : Contexts

- ❖ Kernels are defined on the **host** and host the establishes the **context** for the kernels.
- ❖ Host defines the "**NDRange**"
- ❖ Host defines the "**queues** " that control the details of how and when the kernels execute

(Important functions are defined in APIs within OpenCL's definition.)

Task : Host Defines the Context for the OpenCL Application

- ➢ The **context** defines the environment within which the kernels are defined and execute

OpenCL Execution Model : Contexts

- ❖ How to co-ordinate the mechanisms for host-device interaction ?
- ❖ How to manages the memory objects that are available on the device ?
- ❖ How to keep track of the programs and kernels that are created for each device ?
- ❖ Support of APIs
  - ➢ Before a **host** can request that a kernel be executed on a device, **a context** must be configured on the **host.**
    - Enables it to pass commands and data to the **device**

OpenCL Execution Model : Contexts

- **The API function to create a context is clCreateContext()**
- The **context** is an abstract container that exists on the **host.**
- A **context**
  - Coordinates the mechanisms for host-device interaction,
  - Manages the memory objects that are available to the devices
  - Keeps track of the programs and kernels that are created for each device.
- The properties argument is used to restrict the scope of the **context**
  - **Context** may provide a specific platform ,enable graphics interoperability, or enable other parameters in the future.

OpenCL Execution Model : Contexts

❖ A **context**

- The number and IDs of the devices that the programmer wants to associate with the context must be supplied.

Remark : In OpenCL, the process of discovering platforms and devices and setting up a context is tedious. However, after the code to perform these steps is written once, it can be reused or almost any project.

OpenCL Execution Model : Contexts

❖ **How context includes OpenCL Devices and a program object from which the kernels are pulled for execution ?**

❖ A **context** is defined in terms of the following resources **:**

- **Devices :** the collection of OpenCL devices to be used by the host
- **Kernels :** the OpenCL functions that run on the OpenCL device.
- **Program Objects :** the program source code and executable that implement the kernels
- **Memory Objects : :** a set of objects in memory that are visible to OpenCL devices and contain values that can be operated on by instances of a kernel.

OpenCL Execution Model : Contexts

❖ The context is created and manipulated by **host** using the functions from the OpenCL API

- **On Heterogeneous platform,** the host may choose the GPU, other cores on the CPU, or combination of these.
- Once the choice made, the choice defines the OpenCL devices within the current **context**
- **Program Objects :** One of more program objects that contain the code for the kernels.
  - These can be thought as a " Dynamic library from which the functions used by a kernel are **pulled**.

OpenCL Execution Model : Contexts

❖ **More about Program Objects :**

➢ The program object is built at **runtime** within the host program

- Which target platform will be standard to OpeCL Specification ?

- How de we specify this information in host program ?

➢ Built the program object from the **source** at runtime.

- Compile the program source code to create the code for kernel. (The host program defines devices within the context)

OpenCL Execution Model : Contexts

❖ **More about Program Objects :**

More about Source Code :

- Regular String either statistically defined in the host program
- Loaded from a file at runtime
- Dynamically generated inside the host program

❖ **Context includes OpenCL Devices and a program object from which the kernels are pulled for execution**

OpenCL Execution Model : Contexts

❖ **More about Program Objects :**

More about Source Code :

- Regular String either statistically defined in the host program
- Loaded from a file at runtime
- Dynamically generated inside the host program

❖ **Context includes OpenCL Devices and a program object from which the kernels are pulled for execution**

OpenCL Execution Model : Contexts

```
clCreateContext(
   const cl_context_properties *properties,
   cl_unit 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}
```

OpenCL Execution Model : Context

- **"Context"; How the OPenCL Kernels works with memory ?**
- What is needed for **Command queue** ?
- Detailed memory model needs to be understand and How the openCL memory works at higher level ?
  - About Heterogeneous Systems :
    - Multiple Address Spaces to manage
  - OpenCL introduced the concept of Memory Object
    - Explicitly defined on the host
    - Explicitly moved between the host and the OpenCL devices

OpenCL Execution Model : Contexts

- The OpenCL specification also provides an API call tat alleviates the need to build a list of devices.
  - **`clCreateContextFromType()`** allows a programmer to create a context that automatially includes all devices of the specified type (e.g., CPUS, GPUs, and all devices)
  - After a creating a context, the function **`clGetContextinfo()`** can be used to query information such as the number of devices present and device structures.
- In OpenCL, the process of discovering platforms and devices and setting up a **`context`** is tedious. However, after the code to perform these steps is written once, it can be reused or almost any project.

OpenCL Execution Model : Context

A brief summary of OpenCL Context

❖ Context is the

- OpenCL Devices
- Program Objects
- Kernels
- Memory Object

that a kernel uses when it executes

Command-Queues :

How the host program issues commands to the OpenCL devices ?

OpenCL Execution Model : Context

A brief summary of OpenCL Context

- ❖ Context is the heart of any OpenCL application
- ❖ Context provide a container for
  - ➢associating devices,
  - ➢Memory Objects (e.g., buffers and images),
  - ➢command-queue (providing interface between the context and an individual object)
- ❖ Context drives the communication with, and between, specific drives and OpenCL defines it memory model in terms of these

OpenCL Execution Model : Context

A brief summary of OpenCL Context

- Example : A memory object is allocated with a context but can be updated by a particular device, and OpenCL/memory guarantees that all devices, within the same context, will see these updates as well defined synchronizing points

- Context – update as the program progresses, allocating or deleting memory objects and so on.

  - associating devices,
  - Memory Objects (e.g., buffers and images),
  - command-queue (providing interface between the context and an individual object)

OpenCL Execution Model : Context

In general, an application's OpenCL Usage look similar to this Context

1. Query which platforms are present
2. Query the set of devices supported by each platform
   a. Choose the select devices, using **clGetDeviceInfo(),** on specific capabilities
3. Create contexts from a selection of devices (each context must be created with devices from a single platform), then with a context you can

OpenCL Execution Model : Context

In general, an application's OpenCL Usage look similar to this Context

3. Create contexts from a selection of devices (each context must be created with devices from a single platform), then with a context you can

   a. Create one or more command-queues

   b. Create programs to run on one or more associated devices

   c. Create a kernel from those programs

   d. Allocate memory buffer and images either on the host or on the device

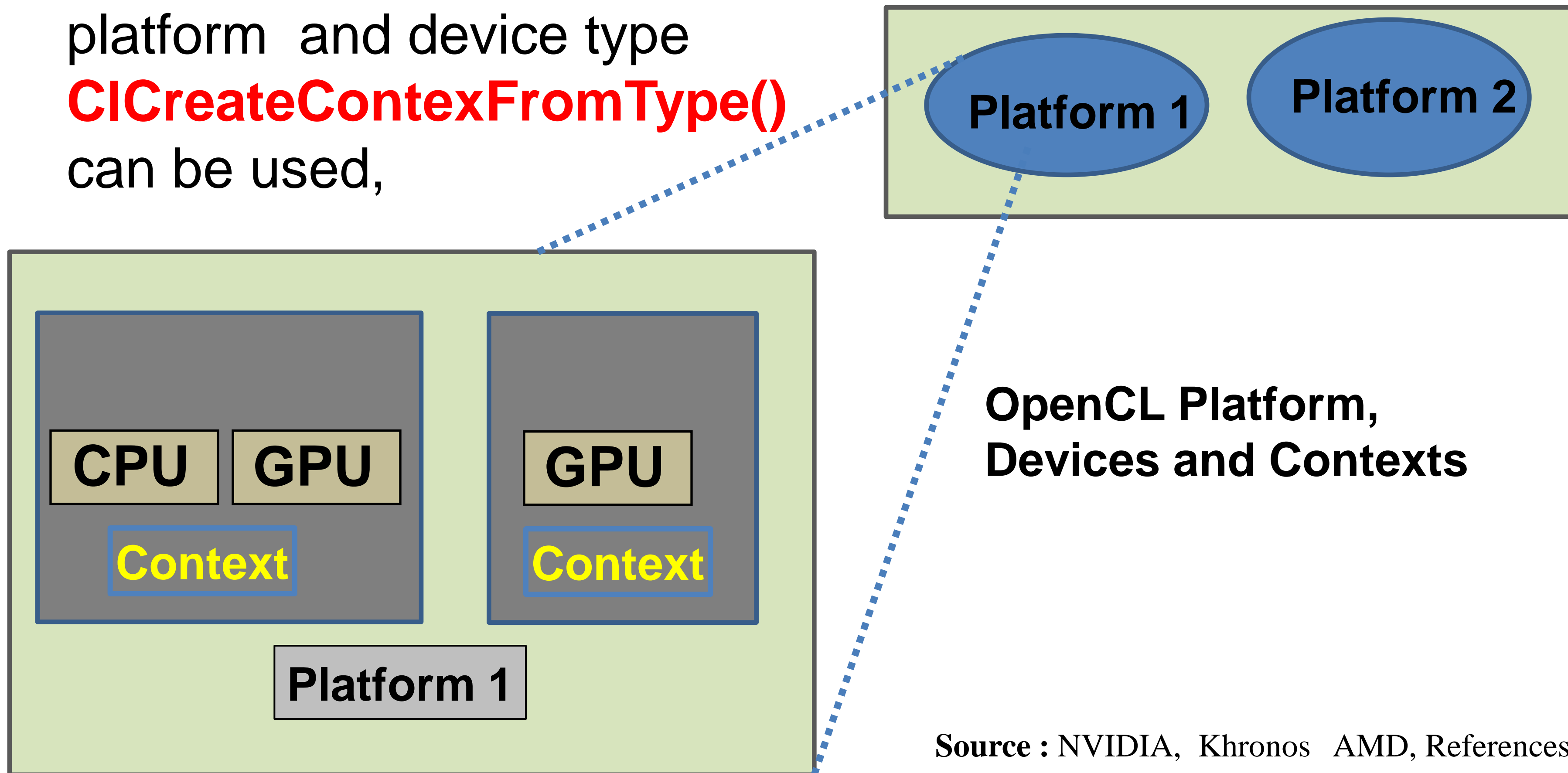
   e. Write or copy data to and from a particular device

   f. Submit kernels (setting the appropriate arguments to a command-queue for execution

OpenCL Execution Model : Context

❖ Given a platform and a list of associated devices, an OpenCL context is created with the command

ClCreateContext(), and with a platform and device type **ClCreateContexFromType()** can be used,



Source : NVIDIA, Khronos AMD, References

Part-2 (f)

OpenCL Specification :Execution Model Command-Queues

(In brief)

Source : NVIDIA, Khronos AMD, References

OpenCL Execution Model : Command-Queues

What is command-queues ?

- The interaction between the host and the **OpenCL** devices occurs through commands posted by a host to the **command-queue.**

- These commands wait in the command-queue until they execute on the OpenCL device

- Check for successful completion of "definition of the context" A command-queue is created by the host and attached to a **single** OpenCL device after the context has been defined.

OpenCL Execution Model : Command-Queues

About command-queues

- The host places commands into the command-queue, and commands are then scheduled for execution on the associated device**.** OpenCL supports three types of commands :

- **Kernel Execution commands :** executes a kernel on the processing elements of an OpenCL device

- **Memory commands :** transfer data between the host and different memory objects move data between memory objects, or map and unmap memory objects from the host address space.

- **Synchronization commands :** put constraints on the order in which commands execute.

OpenCL Execution Model : Command-Queues

❖ **About command-queues**

- Mechanism that the **host** uses to request action by the **devices**.
- Communication with a device occurs by submitting commands to a **command-queue**.
- Each **command-queue** is associated with only one device
  - **Step 1 : Host** decides which **device** to work with
  - **Step 2 :** A **context** is created
  - **Step 3 :** One **command-queue** needs to be created per device
- Whenever the host needs an action to be performance by a device, it will submit commands to the proper command queue.

OpenCL Execution Model : Command-Queues

About command-queues

The API **`clCreateCommandQueue()`** is used to create a command queue and associate it with a device.

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

OpenCL uses default **`in-order command queue`**

If **`out-of-order`** queues are used, it is up to the user to specify dependencies that enforce a correct execution order.

OpenCL Execution Model : Command-Queue

❖ **About command-queue**

- Any API that specifies **host-device** interaction will always begin with **clEnqueue** and require a command queue as a parameter.
- For ex :
  - the **ClEnqueueReadBuffer()**command requests that the device send data to the host and
  - **clEngueueNDRangeKernel()** requests that a kernel is executed on the device.

OpenCL Execution Model : Command-Queues

❖ **Remarks : context & command-queue**

- **First Step - Context :** The programmer defines the context and the command-queues, defines memory and the program objects
- The programmer builds any data structures needed on the host to support the application
- **Next Step - Command queue :**
  - Memory objects are moved from host onto the devices
  - Kernel arguments are attached to memory objects and then submitted the command-queue for execution

OpenCL Execution Model : Command-Queues

- **Remarks : context & command-queue**
  - **Next Step - Command queue :**
    - When the kernel has completed its work, memory objects produced in the computation may be copied back on the host.
- **Other Information : command-queue**
- What is the order in which the commands execute ?
- How the commands execution relates to the execution of the host program. ?

OpenCL Execution Model : Command-Queue

Other Information : command-queue

❖The commands always execute asynchronously to the host program

❖The host program submits commands to the command-queue and then continue without waiting for a commands to finish

- ❖ If necessary, for the host to wait on a command, this can be explicitly established with a synchronization

❖Commands within a single queue execute relative to each other in one of the two modes :

- ❖ In-order execution & Out-or-order execution

OpenCL Execution Model : Command-Queues

Other Information : command-queue

- ❖ Errors : Multiple executions occurring in-side an application may lead to potential disaster i.e. abnormal exist with error messages
  - Data may be accidently used before it has been written or kernels may be execute in an order that leads to wrong answers.
- ❖ The programmer needs some way to manager any constraints on the commands.
- ❖ Synchronization commands can be used to tell set of kernels to wait until an earlier set finishes.

OpenCL Execution Model : Command-Queue

Other Information : command-queue

- To support custom synchronization protocols, commands submitted the **command-queue** generate event objects.
- A command can be told to wait until certain conditions on the event object exists.
- It is possible to associate multiple queues with a single queues with a single context for any of the OpenCL devices within that context,
  - These two queues rub concurrently and independently with no explicit mechanism within OpenCL to synchronize between them.

Source : NVIDIA, Khronos AMD, References

OpenCL PLATFORM AND DEVICES: Events

❖ What is an event ?

- Any operation that **`enqueues`** a command into a command queue – that is any API call the begins with **`clEnqueue`** – produces an ***event***. Events have two main roles to OpenCL

  1. Representing dependencies

  2. Providing a mechanism for profiling

- API Calls the begin with **`clEnqueue`** also take a "**wait list**" of events as a parameter.

- By generating an event for one API call and passing it as an argument to a successive call, OpenCL allows us to represent dependencies.

- A **`ClEnqueue`** call will block until all events in its wait list have completed.

OpenCL : Specification : Heterogeneous Prog.

The Execution Environment

- Contexts
- Command Queues
- Events
- Memory Objects (Buffers -large array /images
  - Buffers (allocate buffer & return memory object)
  - Image (2D & 3D)
- Flush & Finish

Part-2(f)

OpenCL Specification : Memory Model

The OpenCL Specification

❖ **Memory model :**

- Defines the abstract memory hierarchy that kernels use, regardless of the actual underlying memory architecture
- The memory model closely resembles current GPU memory hierarchies. Other accelerators has no limited adoptability.
- To support code portability, OpenCL's approach is to define an abstract memory model that programmers can target when writing code and vendors can map to their actual memory hardware
- The memory spaces (*global memory, constant memory, local memory, private memory*) defined by OpenCL are used and are relevant within OpenCL programs.
- The memory spaces of OpenCL closley model those of modern GPUs

Source : NVIDIA, Khronos AMD, References

OpenCL : Specification : Heterogeneous Prog.

- **OpenCL Memory Model defines five distinct memory-regions**
  - Host memory
  - Global memory
  - Constant Memory
  - Local Memory
  - Private Memory
- **OpenCL Writing kernels**
  - Kernels begin with the keyword **_kernel** and must have a return type of **void**.

Source : NVIDIA, Khronos AMD, References

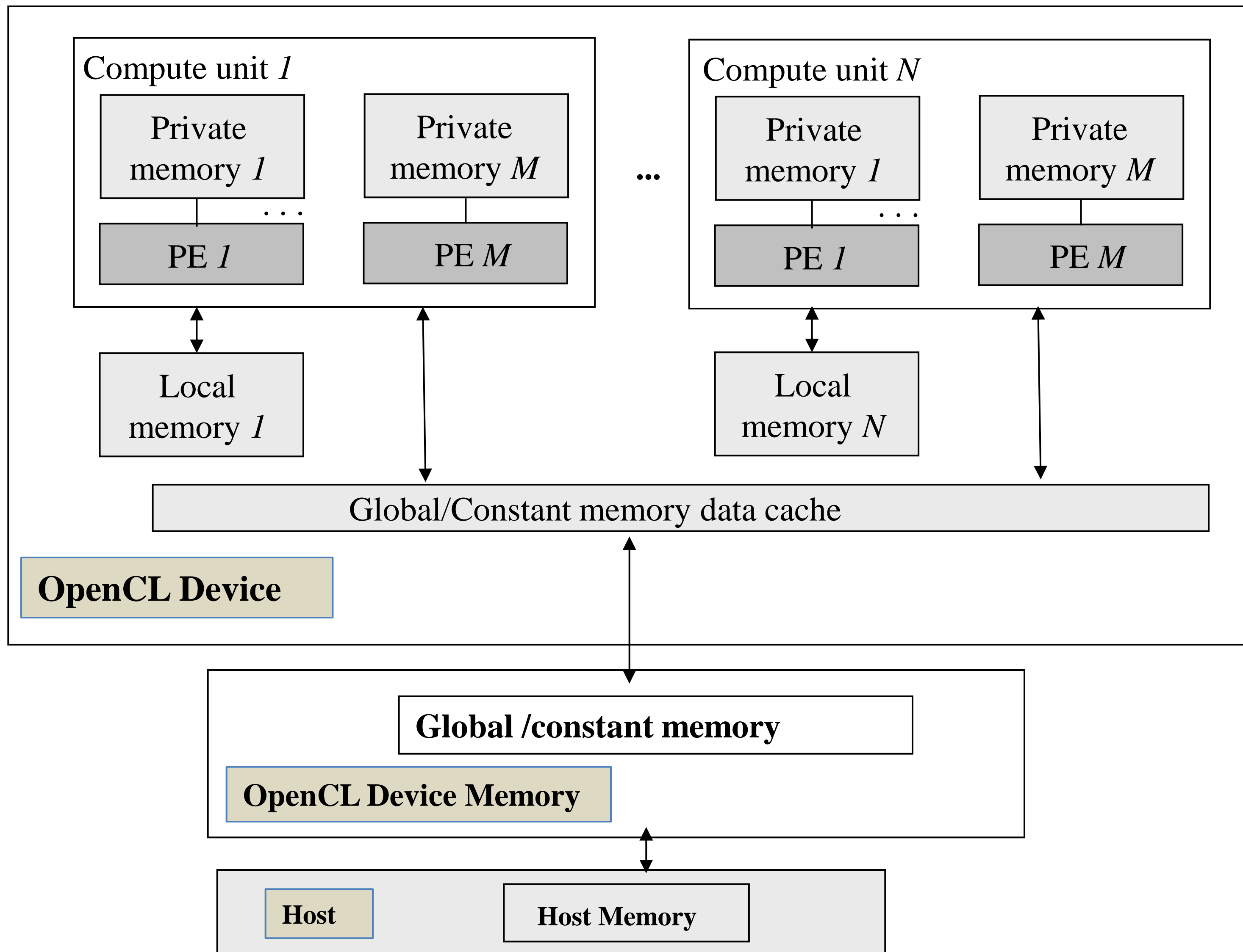
OpenCL : Specification : Execution Model

- **The Execution model tells**
  - How the kernel executes ?
  - How they interact with other kernels ?
- **Used "Memory Objects" for an associated command-queue**
  - How safe these memory objects can be used ?
- OpenCL defines two types of memory objects
  - Buffer Object
  - Image Object
- OpenCL – specify sub regions of memory objects as distinct memory objects

OpenCL : Specification : Heterogeneous Prog.

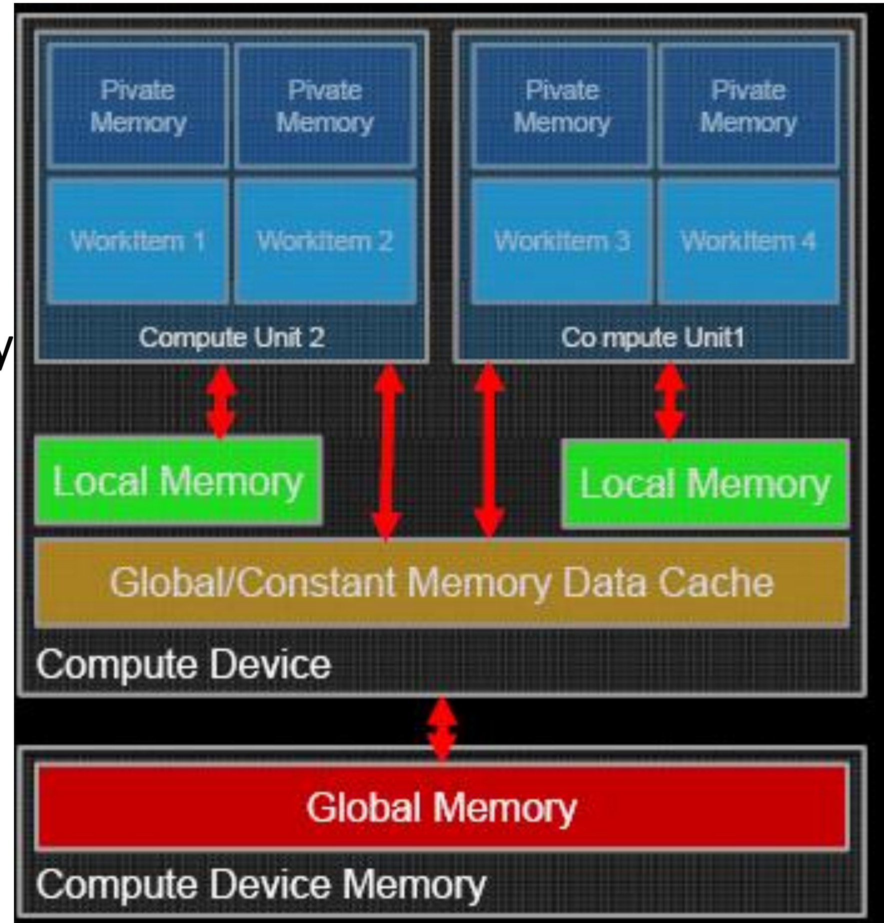
- **OpenCL Memory Model defines five distinct memory-regions**
  - Host memory
  - Global memory
  - Constant Memory
  - Local Memory
  - Private Memory
- **OpenCL Writing kernels**
  - Kernels begin with the **keyword _kernel** and must have a return type of **void**.

A summary of memory model to OpenCL



OpenCL Memory Model

- ❖ Implements a relaxed consistency, shared memory model
- ❖ Multiple distinct address spaces
  - ➢ Address spaces can be collapsed depending on the device's memory subsystem
  - ➢ Address qualifiers
  - ➢ _private
  - ➢ _local
  - ➢ _constant and_global
  - ➢ Example:
    - _global float4 *p;



Source : Khronos, References

OpenCL : Memory Model

The OpenCL : Abstract Memory Model Defined

- OpenCL'S approach is to define an abstract memory model

  - Programmers can target when writing code
  - Vendors can map to their actual memory hardware
  - The memory spaces defined by OpenCL :

    - Global Memory
    - Constant Memory
    - Local Memory
    - Private Memory

  - The key words associated with each space can be used to specify where a variable should be created or where the data that it points to resides.

Source : NVIDIA, Khronos AMD, References

The OpenCL : `Abstract Memory Model Defined`

❖ **Global Memory :**

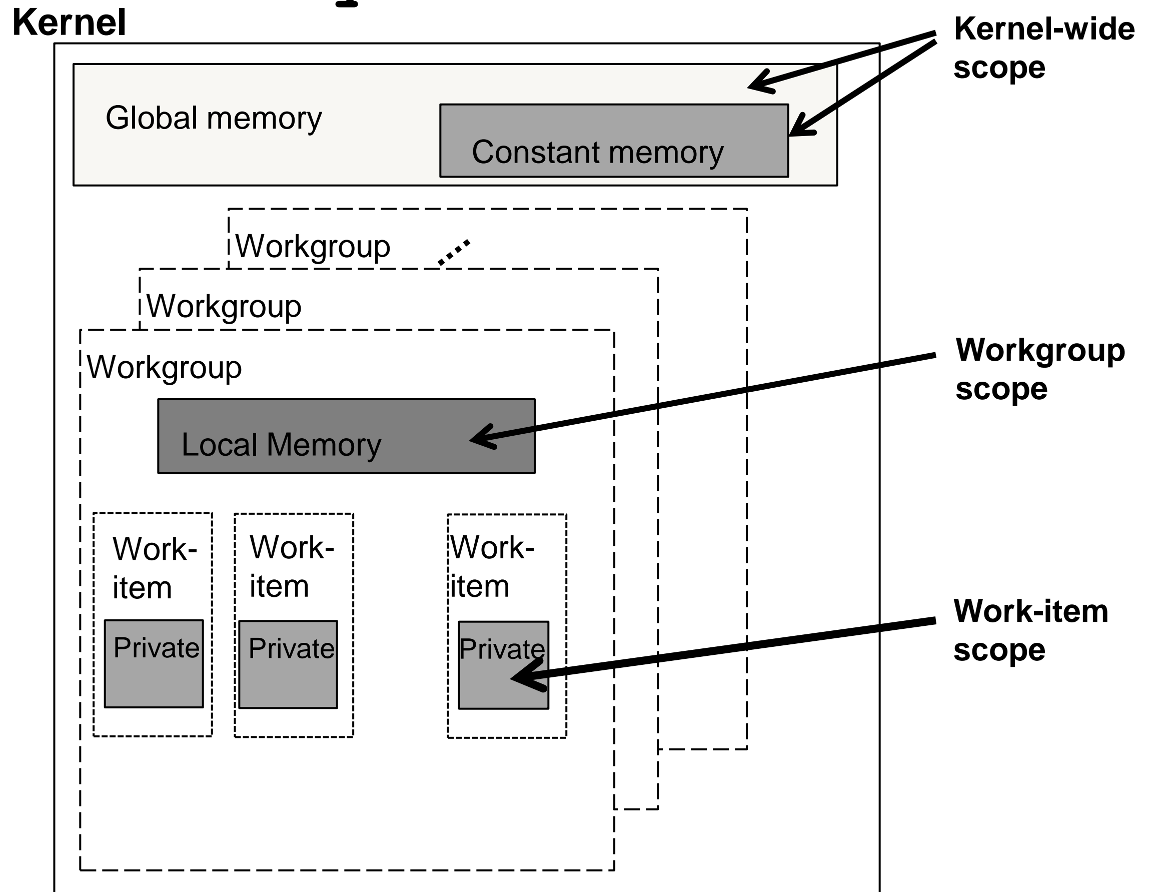
- Visible to all compute units on the device.
- Whenever the data is transferred from the host to device, the data will resides in global memory.
- And data transfer from the device to host must also reside in global memory :
  - The key-word `__global` is added to a pointer declaration to specify that data retrenched by the pointer, resides in global memory,

OpenCL : Memory Model

The OpenCL : Abstract Memory Model Defined

- Global Memory
- Constant Memory
- Local Memory
- Private Memory

Usually, the memory spaces of openCL closely model those of modern GPUs.



The abstract memory model defined by OpenCL.

Source : NVIDIA, Khronos AMD, References

The OpenCL : `Abstract Memory Model Defined`

❖ **Constant Memory :**

- Not specifically designed for every type of read-only data but, rather, for data where each element is accessed simultaneously by all **`work-items`**.
- Variables whose values never change also fall in the category.
- Constant memory is modeled as apart of global memory, so memory objects that are transferred to global memory can be specified as constant.
  - Data is mapped to constant memory by using the key-word **`__constant`**.

Source : NVIDIA, Khronos AMD, References

OpenCL : Memory Model

The OpenCL : `Abstract Memory Model Defined`

❖ **Local Memory :**

- Scratchpad memory whose address space is unique to each compute device :
- Local memory is modeled as being shared by a workgroup.
- Variables whose values never change also fall in the category.
  - Calling **`clSetKernelArg() with a size,`** but no argument allows local memory to be allocated at runtime, where a kernel parameter is defined as a **`__local pointer.`**
  - Data is mapped to constant memory by using the key-word **`__constant`**.
- Arrays can also be declared statically in local memory by appending the keyword **`_local`**, although this require specifying that array size at compile time.

OpenCL : Memory Model

The OpenCL : `Abstract Memory Model Defined`

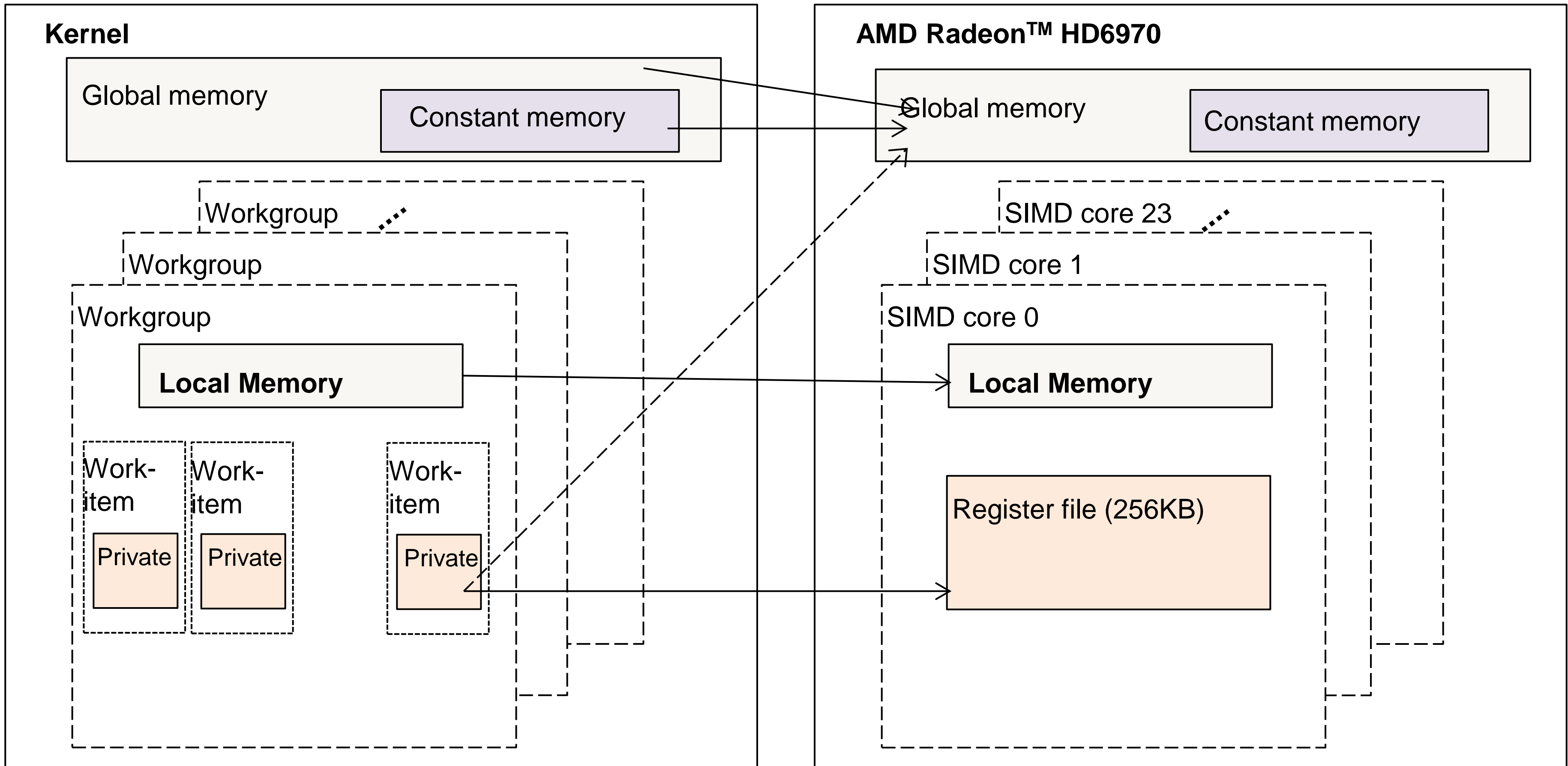
❖ **Private Memory :**

- Memory unique to an individual **`work-item.`**
- Local variables and non-pointer kernel arguments are private by default.
  - These variable are mapped to registers.

Source : NVIDIA, Khronos AMD, References

OpenCL : Memory Model

The OpenCL : Abstract Memory Model Defined



Mapping from the memory model defined by OpenCL to the architecture of an AMD Radeon 6970 GPU. Simple private memory will be stored in registers; complex addressing or excessive use will be stored in DRAM.

OpenCL : Writing Kernels

❖ OpenCL C kernels are similar to C functions and will be executed once for every work-item that is created. :

- Buffers can be declared in global memory (**`_global`**) or constant memory **`(_constant`**) memory.
- Images are assigned to global memory. Access qualifiers (_**`read_only`**, **`_write_only`**, and _**`read_write`**) can also be optimally specified
- The **`__local`** qualifier is used to declare memory that is shared between all **`work-items`** in a **`workgroup`**.
- Declare local memory allocations can be done differently using kernel-scope level..

OpenCL : Writing Kernels

- ❖ OpenCL devices, particularly GPUs, performance vary increase by using local memory to cache data that will be used multiple times by a **`work-item`** or by multiple **`work-items`** in the same workgroup.

- ❖ When developing a kernel, we can achieve this with an explicit assignment from a global memory pointer to a local memory pointer.

- ❖ Once **`work-item`** completes its execution, none of its state information or local memory storage is persistent.

- ❖ Any results that need to be kept must be transferred to global memory.

Part-2(g)

OpenCL Specification : Memory Objects

Source : NVIDIA, Khronos AMD, References

OpenCL PLATFORM AND DEVICES: Memory Objects

❖ Memory Objects

- OpenCL applications often work with large arrays on multi-dimensional matrices. This data needs to be physically present on a device before execution can begin

1. First Step : Data must be encapsulated as a ***memory object***
2. Second Step : transfer the data to a device

- OpenCL define two types of memory objects
- `clEnqueue` also take a "**wait list**" of events as a parameter.

1. **Buffers** : equivalent to arrays in C, created using **malloc(),** where data elements are stored contiguously in memory.
2. **Images** : Designed as opaque objects, allowing data for padding and other optimizations that may improve performance on devices.

OpenCL PLATFORM AND DEVICES: Memory Objects

❖ **Memory Objects :**

- Memory object is valid only within a simple context, after creation of memory object.

  1. To satisfy, the data Dependencies, **OpenCL** runtime manages movement to and from specific devices.

❖ **Memory Objects : Buffers**

- Buffers may help to visualize a memory object as a pointer that is valid on a device. (similar to call to malloc, in C or C++'s a new pointer

- The function **`clCreateBuffer()`** allocates the buffer and returns a memory object

OpenCL PLATFORM AND DEVICES: Memory Objects

❖ Memory Objects : Buffers

- Buffers may help to visualize a memory object as a pointer that is valid on a device. (similar to call to malloc, in C or C++'s a new pointer
- The function `clCreateBuffer()` allocates the buffer and returns a memory object
- Creating a buffer requires supplying the size of the **buffer** and a **context** in which the **buffer** will be allocated
- Buffer is visible for all devices associated with the context.
- Supply flags : Optionally, the caller can supply flags that specify that the data is read-only, write-only or read-write.

OpenCL PLATFORM AND DEVICES: Memory Objects

Memory Objects : Buffers

```
Cl_memclCreateBuffer(
    cl_context context,
    cl_mem_flags flags,
    Size_t size,
    void *host_ptr,
    cl_int *errcode_ret)
```

- **Memory Objects : Buffers**
  - Supply flags : Creating and initializing a buffer with other flags (simple option is to supply a host pointer with data used to initialize the buffer)

OpenCL PLATFORM AND DEVICES: Memory Objects

- **Memory Objects : Buffers**
  - Data contained in host-memory is transferred to and from an OpenCL buffer using the command
    - **`ClEnqueueWriteBuffer()` and**
    - **`ClEnqueueReadBuffer()`**

- Run-time determines the precise time the data is moved.
  - The buffer is linked to a context, not a device
  - If a kernel that is dependent on such a buffer is executed on a discrete accelerator device such as a GPU, the buffer may be transferred to the device.

OpenCL PLATFORM AND DEVICES: Memory Objects

Memory Objects : Buffers

```
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_unit num_events_in_wait_list,
    const cl_event *event_wait_list,
    cl_event *event)
```

OpenCL PLATFORM AND DEVICES: Memory Objects

❖ **Memory Objects : Buffers**

- Similar to other **enqueue** operations, reading or writing a buffer requires a command queue to manage the execution schedule.
- The **enqueue** function requires the buffer, the number of bytes to transfer, and an offset within the buffer.
- The **block_write** option should be set to **CL_TRUE** if the transfer into an openCL buffer until the operation has completed.
- Setting the **block_write** option to CL_FALSE allows **clEnqueue-WriteBuffer** to return before the write to **CL_FALSE** allows **clEnqueueWriteBuffer()** to return before the write operation has completed.

❖ Memory Objects : Images

- Images are type of OpenCL memory object that abstract the storage of physical data to allow for devices-specific optimization
- Use `clGetDeviceInfo()` to check the support of all OpenCL Devices.
- **Purpose of using Images** : to allow the hardware to take advantage of spatial locality and to utilize the hardware acceleration available on many devices.
  - Unlike buffers, images cannot be directly referenced as if they were arrays.

Source : NVIDIA, Khronos AMD, References

OpenCL PLATFORM AND DEVICES: Memory Objects

❖ Memory Objects : Images

- Images are type of OpenCL memory object that abstract the Images are an example of the OpenCL standard being dependent on the underlying hardware of a particular device.
- **Th**e elements of an image are represented by a format descriptor **`(cl_image_format)`**.
- The format descriptor specifies how the image elements are stored in memory based on the concepts of **channels**
  - The channels order specifies the number of elements that make up an image element (up to four elements, based on the traditional use of RGBA pixels), and the channel type specifies the size of each element.
  - These elements can be sized from 1 to 4 bytes and in various different formats (e.g., integer or floating point)

OpenCL PLATFORM AND DEVICES: Memory Objects

❖ Memory Objects : Images

- Creating an OpenCL image is done using the command (`clCreateImage2D()` or `clCreateImage3D()`
- Additional arguments are required when creating an image object versus those specified for creating a buffer.
  - First, the height and the width of the image must be given (and a depth for the three-dimensional case)
  - Image pitch (number of bytes between the start of one image and the start of the next.) may be specified if initialization data is provided.
- Additional parameters are required when reading or writing an image.
- Within a kernel, images are accessed with built-in functions specific to data type.

OpenCL PLATFORM AND DEVICES: Memory Objects

Memory Objects : Images

```
Cl_mem

clCreateImage2D(

   cl_context context,

   cl_mem_flags flags,

   const cl_image_format *image_format

   size_t  image_width,

   Size_t image_height,

   const image_row_pitch,

   void *host_ptr

   cl_int *errcode_ret,
```

OpenCL : Specification : Heterogeneous Prog.

- **Creating an OpenCL Program Object**
  - Process of creating a kernel (Character string, Character array, Program object
  - Intermediate OpenCL –ICD; NVIDIA –PTX, AMD-IL
  - Final and Intermediate representations

OpenCL : Specification : Heterogeneous Prog.

- **OpenCL Kernel**
  - Get kernel object
  - Execute kernels on a device
  - Extract a kernel from a program
    - To request from the compiled program object

Source : NVIDIA, Khronos AMD, References

Part-2 (h)

OpenCL Specification :

Details on…. on Programming Model

The OpenCL Specification

- The OpenCL specification is defined in four parts, called models, that can be summarized as follows.

  - Platform Model
  - Execution Model
  - Memory Model
  - Programming Model

Source : Khronous, & References

The OpenCL Specification

❖ Programming model :

- Defines how the concurrency model is mapped to physical hardware. The hardware thread contexts that execute the kernel must be created and mapped to actual GPU hardware units.
- OpenCL C code (Written to run on an OpenCL device) called a ***program.*** A program is a collection of functions called ***kernels***, where kernels are units of execution that can be scheduled to run on a device
- OpenCL software links only to a common runtime layer (called the ICD); & uses dynamic library interface at runtime
- Compiled at runtime through a series of API calls (The source code is turned into a program object (**OpenCL program object**) & then compiled to the generate the **OpenCL Kernel object** that can be used to execute kernels on a device.

Source : Khronous, & References

The OpenCL Specification

❖ Programming model :

- ➢ The data within the **kernel** is allocated by the programmer to specific parts of an abstract memory hierarchy.
- ➢ The runtime and driver will **map** these abstract memory space to the physical memory.
- ➢ The hardware threads contexts that execute the kernel must be created and mapped to actual GPU hardware units
- ➢ Executing a kernel requires dispatching it through an **enqueue** function.

Source : Khronous, & References

The OpenCL Specification

❖ Programming model :

- The process of creating kernel involves three steps.
  - **Step 1 :** The OpenCL source code is stored in a character string. If the source code is stored in a file on a disk, it must be read into the memory and stored as a character array.
  - **Step 2 :** The source code is turned into a object, **`cl_program`**, by calling **`clCreateProgramWithSource()`**.
  - **Step 3 :** The program object is then compiled, for one or more OpenCL devices, with **`clBuildPorgram()`**, If there are compile errors, they will be reported here.
- OpenCL provides APIs which takes a list of binaries that matches the device list.

Source : Khronous, & References

Part-3

Important Steps in OpenCL Implementation

Source : NVIDIA, Khronos AMD, References

OpenCL Implementation Steps

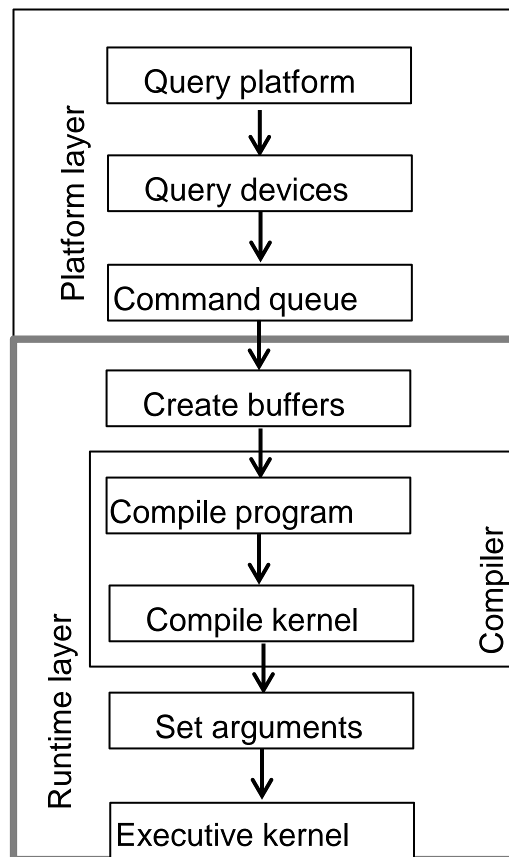


Figure 4.2 Programming steps to writing a complete OpenCL applications

OpenCL Important Steps – Implementation

Step 1 : Discover and initialize the platforms
Step 2 : Discover and initialize the devices
Step 3 : Create context
Step 4 : Create a command queue
Step 5 : Create device buffers
Step 6 : Write host data device buffers
Step 7 : Create and compile the program
Step 8 : Create the kernel
Step 9 : Set the kernel arguments
Step 10 : Configure the work -items structure
Step 11 : Enqueue the kernel for execution
Step 12 : Read the output buffer back to the host
Step 13 : Release OpenCL resources

OpenCL Important Steps – Implementation

Step 1 *: Discover and initialize the platforms*

Step 2 *: Discover and initialize the devices*

Step 3 *: Create context*

Step 4 *: Create a command queue*

Step 5 *: Create device buffers*

Step 6 *: Write host data device buffers*

The OpenCL specification in four parts, called models.

- **Platform Model**
- **Execution Model**
- **Memory Model**
- **Programming Model**

OpenCL Important Steps – Implementation

Step 7 ***:*** *Create and compile the program*

Step 8 ***:*** *Create the kernel*

Step 9 ***:*** *Set the kernel arguments*

Step 10 ***:*** *Configure the work -items structure*

Step 11 ***:*** *Enqueue the kernel for execution*

Step 12 ***:*** *Read the output buffer back to the host*

Step 13 ***:*** *Release OpenCL resources*

The OpenCL specification in four parts, called models.

- **Platform Model**
- **Execution Model**
- **Memory Model**
- **Programming Model**

OpenCL Important Steps – Implementation

- Create an OpenCL context on the first available device
- Create a command –queue on the first available device
- Load a kernel file (hello-world.cl) and build it into a program object
- Create a kernel object for the kernel function hello_world()
- Query the kernel for execution
- Read the results of the kernel back into the result buffer

OpenCL Important Steps – Implementation

```
_kernel void hello_kernel(_global *, *, )
{
     int gid = get_global_id(0);
     .........
 }

int main (int argc,  char** argv)
{
// Create an OpencL context on first available platform

// Create an command-queue on the first device
//  available on the created context
```

OpenCL Important Steps – Implementation

// Create OpenCL kernel

// Create memory objects that will be used as
// arguments to kernel.
// First create Host memory arrays that will be used to
// store the arguments to the kernel

// Set the kernel arguments

//Queue the kernel up for execution across the array
//Read the output buffer back to the Host
//Output the result buffer

OpenCL PLATFORM AND DEVICES: Flush & Finish

- The flush and finish commands are two different types of barrier operations for a command queue.
- The **`clFinish()`** function blocks until all of the commands in a command queue have completed.
- The **`clFlush()`** function blocks until all of the commands in a command queue have been removed from the queue.

```
cl_int clFlush(cl_command_queue command_queue)
cl_int clFinish(cl_command_queue command_queue)
```

Creating an OpenCL Program Object

❖ **What is an OpenCL C Code ?**

- OpenCL C Code (Written to run on an OpenCL device) is called a *program*.
- A program is a collection of functions called *kernels*, where kernels are units of execution that can be scheduled to run a device.
- There is no need for an OpenCL application to have been *prebuilt* against the AMD, NVIDIA, or Intel runtime.
- OpenCL software links to a command runtime layer (called the **ICD**); all platform-specific SDK activity is delegated to a vendor runtime through a dynamic library interface.
  - ICD: Installable Client Driver for OpenCL

Creating an OpenCL Program Object

What is an OpenCL^{TM} ICD ?

- The OpenCL ICD (Installable Client Driver) is a means of allowing multiple OpenCL implementations to co-exist and applications to select between them at runtime.
- User application is responsible for selecting which of the OpenCL platforms present on a system it wishes to use, instead of just requesting system default.
- Using

  `clGetPlatfromIDs()` & **`ClGetPlatfromInfo()`**

  functions to examine the list of available OpenCL implementations and selecting the one which best suites user requirements.

Creating an OpenCL Program Object

❖ About OpenCL™ ICD - Vendor Platform

- At this point, OpenCL Studio selects either the NVIDIA or AMD platform.
- There is **no support** for multiple platforms yet, but that will likely be another abstraction to manage multiple platforms and devices.
- The AMD driver lets you choose between the CPU and the GPU
- NVIDIA however only supports the “CUDA enabled NVIDIA GPU”

Creating an OpenCL Program Object

- **About OpenCL™ ICD - Vendor Platform**

  - At this point, OpenCL Studio selects either the NVIDIA or AMD platform.
  - There is no support for multiple platforms yet, but that will likely be another abstraction to manage multiple platforms and devices
  - The AMD driver lets you choose between the CPU and the GPU
  - NVIDIA however only supports the "CUDA enabled NVIDIA GPU"

Source : NVIDIA, Khronos AMD, References

OpenCL : The Execution Environment

Creating an OpenCL Program Object

- **How to create OpenCL Kernel ?**
- **What is the process of creating a kernel ?**

1. The OpenCL C source code is stored in a character string. If the source code is stored in a file on a disk, it must be read into memory and stored as a character array.
2. The source code is turned into a program object **`cl_program`**, by calling **`clCreateProgramWithSource()`**.
3. The program object is then compiled, for one or more OpenCL devices, with **`clBuildProgram()`**, If there are compile errors, they will be reported here.

Creating an OpenCL Program Object

❖ Is "Binary Representation " very vendor specific ?

- **AMD**: In the AMD runtime, there are two main classes of devices : x86 CPUs and GPUs
  - X86 CPUs `clBuildProgram()` generates x86 instructions that can be directly executed on the device.
  - For the GPUs, it will create AMD's GPU intermediate language (IL), a high-level intermediate language that represents a single **`work-item`** & compiled for a specific GPU's architecture later. (Generating ISA -code specific instruction set architecture)
- The advantage of using such an IL is to allow the GPU ISA itself to change from one device or generation to another in what is still a very rapidly developing architectural space

Creating an OpenCL Program Object

- **Is "Binary Representation " very vendor specific ?**
  - **Additional Feature :** Build process is the ability to generate both the final binary format and various intermediate representations
  - Serialize these binaries (Write them to out to disk)
  - **OpenCL** provides a function to return information about program objects, `clGetPrograminfo()`
    - Flags to this function : `cl_PROGRAM_BINARIES,` which returns a vendor-specific set of binary objects generated by `clBuildProgram()`
  - OpenCL provides `clCreateProgramWithBinary(),` which takes a list of binaries that matches its device list.

Binary Representation on GPUs

- **Is "Binary Representation " very vendor specific ?**
  - **NVIDIA:** calling its intermediate representation PTX (PTX is an intermediate assembly language for NVIDIA GPUs) NVCC is the NVIDIA compiler driver
  - **PTX:** a low-level parallel thread execution virtual machine and instruction set architecture (ISA). PTX exposes the GPU as a data-parallel computing device.
  - PTX defines a virtual machine and ISA for general purpose parallel thread execution. . (ISA - code specific instruction set architecture)

Binary Representation on GPUs

❖ **Is "Binary Representation " very vendor specific ?**

NVIDIA :

- PTX programs are translated at install time to the target hardware instruction set.
- PTX-to-GPU translator and driver enable NVIDIA GPUs to be used as programmable parallel computers.
- Provide a stable ISA that spans multiple GPU generations.
- Achieve performance in compiled applications comparable to native GPU performance.

OpenCL : The Execution Environment

The OpenCL Kernel

- ❖ **How to obtain "`cl_kernel`" object that can be used to execute kernels on a device ?**
  - ➢ Extract kernel from the **`cl_program`**
  - ➢ Similar to obtaining an exported function from a dynamic Lib.
    - The name of the kernel that the program exports is used to request it from the compiled program object.
    - The name of the kernel is passed to **`clCreateKernel()`**, along with the program object, and the kernel object will be returned if the program object was valid and the particular kernel is found.
  - ➢ A few more steps are required before the kernel can actually be executed.

OpenCL : The Execution Environment

The OpenCL Kernel

- **What are the steps required before the kernel can actually be executed ?**

  - Executing a kernel requires dispatching it through an **enqueue** function

  - Specify each kernel argument individually using the function **clSetKernelArg()**

    - This function takes kernel object, an index specifying the argument number, the size of the argument, and a pointer to the argument..

OpenCL : The Execution Environment

The OpenCL Kernel

- **What are the steps required before the kernel can actually be executed ?**
  - When a kernel is executed, this information is used to transfer arguments to the device
  - After any required memory objects are transferred to the device and the kernel arguments are set, the kernel is ready to be executed.
  - Requesting that a device begin executing a kernel is done with a call to **clEngueueNDRangeKernel()**

OpenCL : The Execution Environment

```
cl_int
clEngueueNDRangeKernel(
    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_unit num_events_in_wait_list,
    const cl_event *event_wait_list,
    cl_event *event)
```

The OpenCL Kernel : `clEngueueNDRangeKernel()`

- A command queue should be specified so that the target device is known
- The **`clEngueueNDRangeKernel()`** call is asynchronous
  - It will return immediately after the command is enqueued in the command queue and likely before the kernel has even started exeuction.
  - Either **`clWaitForEvent()`** or **`clFinfish()`** can be used to block execution on the host until the kernel completes.

The OpenCL Kernel : `clEngueueNDRangeKernel()`

- At this point, we have presented all the required host API commands needed to enable the reader to run a complete OpenCL Program

Part-4 (a)

Example Program -1

Addition of two vectors : How to define workgroups & work-items

- ❖ work-items within a workgroup can perform "barrier synchronization"
- ❖ work-items within a workgroup can access to a shared memory address space. (Does not affect the scalability of a large concurrent dispatch)

Example Program : Addition of two vectors of size 1024

- The workgroup size might be specified as

```
size_t workGroupSize(3) = (64,1,1);
```

- Total number of work-items for array : 1024
- Total number of workgroups : 1024/64 = 64 workgroups

Kernels and the OpenCL Execution Model

- **Example Program** : Addition of two vectors (**Sequential**)
  - Algorithm executes a loop with as many iterations as there are elements to compute.
  - Each loop iterations adds the corresponding locations in the **input** arrays together and stores the result into the **output** array.

```
//Perform element-wise addition of A & B and
//Stores in C - There are N elements per array
void vecadd(int *C, int *A, int *B, int N)
{
  for(int =0; i < n; i++)
  {
     C[i] = A[i] + B[i];
  }
}
```

Source : Khronous, & References

Kernels and the OpenCL Execution Model

- **Example Program** : Addition of two vectors (**Multi-Core Device**)
  - Use low level coarse-grained threading API (POSIX threads) (One can use Data Parallel model such as OpenMP).
    - Divide the work (i.e., loop iterations) between the threads
    - Work per iteration is (loop counter) may be small or large. Use Strip mining to chunk the loop iterations into a large granularity.

Source : NVIDIA, Khronos AMD, References

Kernels and the OpenCL Execution Model

❖ **Example Program** : Addition of two vectors (**Multi-Core Device**)

```
//Perform element-wise addition of A & B and
//Stores in C – There are N elements per array
//and NP CPU Cores

void vecadd(int *C, int *A, int *B, int N,int NP,int tid)
{
  int Elept = N/NP;  // elements per thread
  for(int = tid*Elept; i < (tid+1)*Elept; i++)
  {
    C[i] = A[i] + B[i];
  }
}
```

Source : Khronous, & References

Kernels and the OpenCL Execution Model

Example Program : Addition of two vectors (**OpenCL**)

❖ When an OpenCL device begins executing a kernel, it provides intrinsic function that allow a ***work-item*** to identify itself

- Current work-item position is given by OpenCL Intrinsic function **get_global_id(0)**

```
// Perform element-wise addition of A & B & Stores in C
// N work items will be created to execute this kernel.
__kernel
void vecadd(_global int *C,
                 _global int *A,
                 _global int *B)
{
 int tid = get_global_id(0);
  C[tid] = A[tid] + B[tid];
}
```

Source : NVIDIA, Khronos AMD, References

Part-4 (a)

Example Program -2

OpenCL : to write data-parallel programs

❖ Simple Matrix Multiplication Example:

- OpenCL host programs can be written in either C or using the OpenCL, C++ Wrapper API.

- Serial implementation : C or C++

  - The code iterates over three nested for loops, multiplying Matrix **A** by Matrix **B** and storing the result in Matrix **C**.
  - The **two** outer loops are used to iterative over each element of the output matrix
  - The **innermost** loop will iterate over the individual elements of the input elements of the input matrices to calculate the result of each output location.

OpenCL : to write data-parallel programs

❖ **Simple Matrix Multiplication Example:**

- OpenCL host programs can be written in either C or using the OpenCL, C++ Wrapper API.

- Serial implementation : C or C++

  - The code iterates over three nested for loops, multiplying Matrix **A** by Matrix **B** and storing the result in Matrix **C**.
  - The **two** outer loops are used to iterative over each element of the output matrix
  - The **innermost** loop will iterate over the individual elements of the input elements of the input matrices to calculate the result of each output location.

OpenCL PLATFROM AND DEVICES

Serial Implementation

```
// Iteration over the rows of Matrix A
for ( int i = 0; i< heightA;  i++)
{
    // Iteration over the columns of MatrixB
    for ( int j = 0; j< widthB;  j++) {
          C[i][j] = 0;

          // Multiply and accumulate  over values in the current row
          // of A and column of B
          for ( int k = 0; k< widthA;  k++) {
          C[i][j] += A[i][k] * B[k][j];
          }
   }
}
```

OpenCL : to write data-parallel programs

❖ OpenCL Simple Implementation : Matrix Multiplication

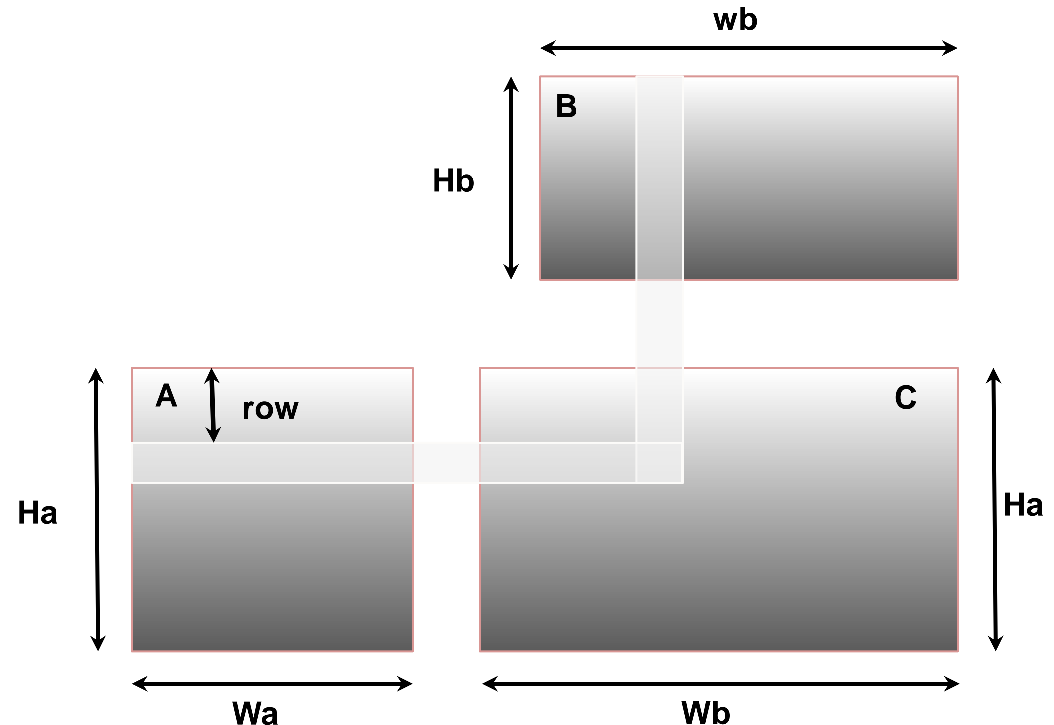
- **Two** outer loops work independently of each other
  - Separate **work-item** can be created for **each** output element of the matrix
  - The **two** outer for-loops are mapped to the **two** dimensional range of **work-item** for the kernel.

- Serial implementation : C or C++
  - The code iterates over three nested for loops, multiplying Matrix **A** by Matrix **B** and storing the result in Matrix **C**.
  - The **two** outer loops are used to iterative over each element of the output matrix
  - The **innermost** loop will iterate over the individual elements of the input elements of the input matrices to calculate the result of each output location.

OpenCL : to write data-parallel programs

OpenCL Simple Implementation : Matrix Multiplication

- Each **work-item** reads in its own row of **Matrix A** and its column of **Matrix B**.
- The data being read is multiplied and written at the appropriate location of the output **Matrix C**



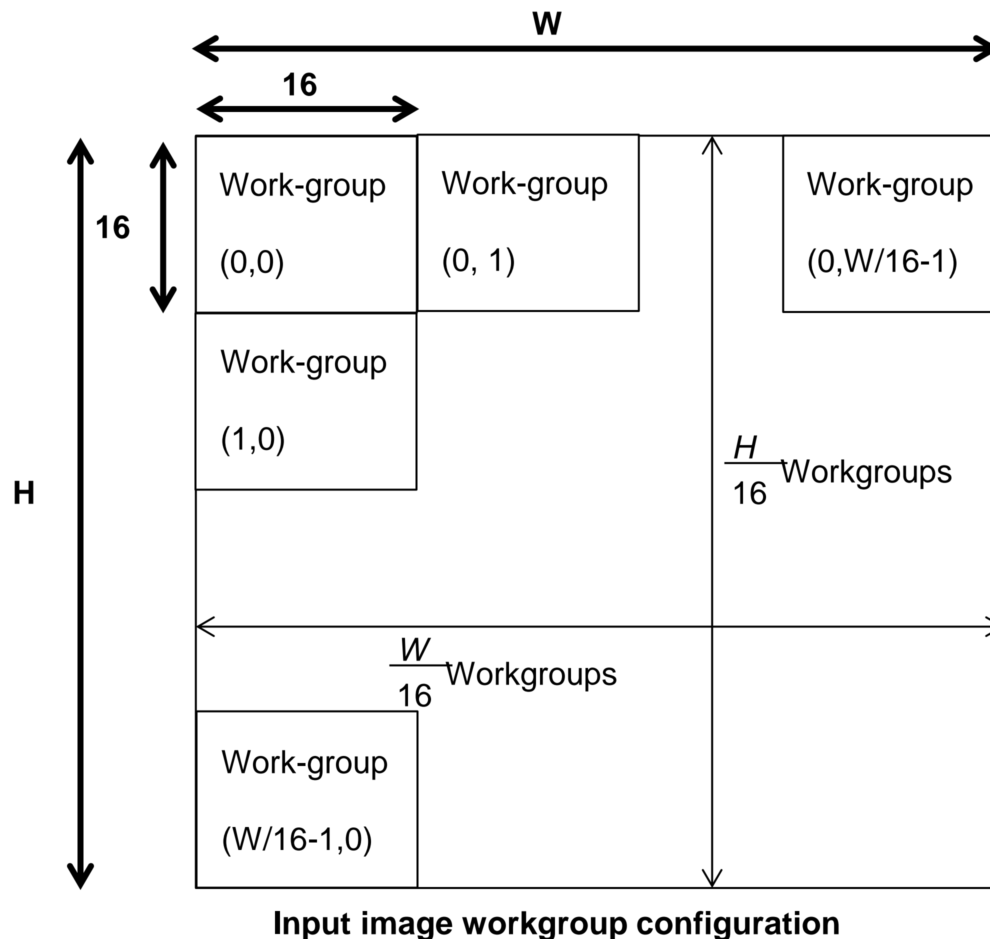
Each output value in a matrix multiplication is generated independently of all others.

Data Parallel Kernel Implementation

```
// Iteration over the rows of Matrix A
for ( int i = 0; i< heightA;  i++)
{
    // Iteration over the columns of MatrixB
    for ( int j = 0; j< widthB;  j++) {
            C[i][j] = 0;

            // Multiply and accumulate  over values in the current row
            // of A and column of B
            for ( int k = 0; k< widthA;  k++) {
            C[i][j] += A[i][k] * B[k][j];
            }
    }
}
```

OpenCL : Work-items & work-Groups



Input image workgroup configuration

Each element of the input image is handled by one work-item. Each work-item calculates its data's coordinates and writes image out.

Simple Matrix Multiplication Example

Step 1: Set Up Environment

In this step, we declare a context, choose a device type, and create the context and a command queue. Throughout this example, the ci ErrNum variable should always be checked to see if an error code is returned by the implementation.

```
cl_int ciErrNum;

//Use the first platform
cl_platform_i d platform;
ci ErrNum = clGet PlatformIDs (1, &platform, NULL);

//Use the first device
cl_device_id device;
ciErrNum = clGetDevice IDs(
   piatform,
   CL_DEVICE_TYPE_ALL,
   1,
   &device,
   NULL);
```

Source : NVIDIA, Khronos AMD, References

Simple Matrix Multiplication Example

```
context_properties cps[3]={;
   CL_CONTEXT_PLATFORM, (cl_context_properties)platform, 0};

//Create the context
cl_context ctx = clCreateContext(
   cps,
   1,
   &device,
   NULL,
   NULL,
   &ciErrNum);

//Create the command queue
cl_command_queue myqueue = clCreateCommandQueue{
   ctx,
   device,
   0,
   &ciErrNum0;
```

Source : NVIDIA, Khronos AMD, References

Simple Matrix Multiplication Example

Step 2: Declare Buffers and Move Data

Declare buffers on the device and enqueue copies of input matrices to the device. Also declare the output buffer.

```
 // We assume that A, B, C are float arrays which
 // have been declared and initialized

 // Allocate space for Matrix A on the device
  cl_mem buf ferA = cl CreateBuf fer(
   ctx,
   CL_MEM_READ_ONLY,
   wA*hA*si zeof(float),
   NULL,
   &ci ErrNum);

// Copy Matrix A to the device
   ci ErrNum = clEnqueueWriteBuffe-(
   myqueue,
   bufferA,
   CL_TRUE, 0,
  wA*hA*sizeof(float), (void *)A, 0.
  NULL, NULL);
```

Simple Matrix Multiplication Example

```
// Copy Matrix A to the device
   ci ErrNum = clEnqueueWriteBuffe-(
   myqueue,
   bufferA,
   CL_TRUE,
   0,
   wA*hA*sizeof(float),
  (void *)A,
   0.
   NULL,
   NULL);

// Allocate space for Matrix B on the device
   cl_mem bufferB = clCreateBuffer(
   ctx,
   CL_MEM_READ_ONLY,
   wB*hB*si zeof(float),
   NULL,
   &ci ErrNum);
```

Simple Matrix Multiplication Example

```
// Copy Matrix B to the device
cl ErrNum = clEnqueueWri teBuffer(
   myqueue,
   bufferB,
   CL_TRUE,
   0,
   wB*hB*si zeof(float),
   (void *)B,
   0,
   NULL,
   NULL);

// A1 locate space for Matrix C on the device
 cl_mem bufferC = cl CreateBuffer(
   ctx,
   CL_MEM_READ_ONLY,
   hA*wB*sizeof(float),
   NULL.
   &ci ErrNum);
```

Simple Matrix Multiplication Example

Step 3: Runtime Kernel Compilation

Compile the program from the kernel array, build the program, and define the kernel.

```
 // We assume that the program source is stored in the variable
 // 'source' and is NULL terminated
cl_programiriyprog = clCreateProgramWithSource (
   ctx,
   1,
   (const char**)&source,
   NULL,
   &ci ErrNum);

// Compi 1 p the program. Passing NULL for the 'devi ce_]1st'
// argume.it targets all devices in the context ciErrNum=clBuildProgram(myprog, 0,
NULL, NULL, NULL, NULL);

// Create, the kernel
cl_kernel mykernel = clCreateKernel(
   myprog,
   "simpleMulti ply",
   &ci ErrNum);
```

Simple Matrix Multiplication Example

Step 4: Run the Program

Set kernel arguments and the workgroup size. We can then enqueue kernel onto the command queue to execute on the device.

```
//Set the kernel arguments
clSetKernelArg(my kernel, 0, si zeof(cl_mem), (void *)&d_C); clSetKernelArg(mykernel, 1,
sizeof(cl_int), (void *)&wA);

cl Set Kernel Arg( my kernel , 2 , sizeof(cl_int), (void *)&hA); clSetKerne1Arg(my kernel, 3,
sizeof(cl_int), (void * )&wB); clSetKernelArg(my kernel, 4. si zeof(cl_i nt), (void*)&hB);

cl SetKernel Arg( mykernel , 5, sizeof (cl__mem), (void *)&d_A);

cl Set Kernel Arg( my kerne 1 , 6, sizeof( cl__mem) , (void *)&d_B );

// Set local and global workgroup sizes
//We assume the matrix dimensions are divisible by 16

size_t1ocalws[2] = 116 ,161 ;
size„t globalws[2] = iwC, hC};
```

Simple Matrix Multiplication Example

```
// Execute the kernel
ciErrNum = clEnqueueNDRangeKernel(
   myqueue,
   mykernel ,
   2,
   NULL,
   globalws,
   1ocalws,
   0,
   NULL,
   NULL);
```

Simple Matrix Multiplication Example

Step 5: Obtain Results to Host

After the program has run, we enqueue a read back of the result matrix from the device buffer to host memory.

```
// Read the output data back to the host
ciErrNum = cl EnqueueReadBuffer(
   myqueue,
   d_C,
   CL_TRUE,
   0,
   wC*hC*si zeof(float),
   (void *)C, 0,
   NULL,
   NULL);
```

The steps outlined here show an OpenCL implementation of matrix multiplication that can be used as a baseline. In later chapters, we use our understanding of data-parallel architectures to improve the performance of particular data-parallel algorithms.

OpenCL Summary

- History of OpenCL
- Easing cross-platform development with major enhancements for stream software strategy
- GPU Programming – OpenGL, DirectX, NVIDIA (CUDA), AMD (Brook+)
- Aggressively expanding stream strategy to consumer segment

OpenCL Summary

- A new computer language that works across GPUs and CPUs
  - C /C++ with extensions
  - Familiar to developers
  - Includes a rich set of built-in functions
  - Makes it easy to develop data- and task- parallel compute programs
- Defines hardware and numerical precision requirements
- Open standard for heterogeneous parallel computing

References

1. Randi J. Rost, OpenGL – shading Language, Second Edition, Addison Wesley 2006
2. GPGPU Reference http://www.gpgpu.org
3. NVIDIA http://www.nvidia.com
4. NVIDIA tesla http://www.nvidia.com/object/tesla_computing_solutions.html
5. RAPIDMIND http://www.rapidmind.net
6. Peak Stream - Parallel Processing (Acquired by Google in 2007) http:/www.google.com
7. guru3d.com http://www.guru3d.com/news/sandra-2009-gets-gpgpu-support/
   ATI & AMD http://ati.amd.com/products/radeon9600/radeon9600pro/index.html
8. AMD http:www.amd.com
9. AMD Stream Processors http://ati.amd.com/products/streamprocessor/specs.html
10. RAPIDMIND & AMD http://www.rapidmind.net/News-Aug4-08-SIGGRAPH.php
11. General-purpose computing on graphics processing units (GPGPU) http://en.wikipedia.org/wiki/GPGPU
12. Khronous Group, OpenGL 3, December 2008 URL : http://www.khronos.org/opencl
13. *OpenCL - The open standard for parallel programming of heterogeneous systems* URL : http://www.khronos.org/opencl
14. Programming the GPU and a brief intro to the OPENGL shading language – Marcel Cohan & VVR Talk
15. David B Kirk, Wen-mei W. Hwu nvidia corporation, 2010, Elsevier, Morgan Kaufmann Publishers, 2011
16. Benedict R Gaster, Lee Howes, David R Kaeli, Perhadd Mistry Dana Schaa, Heterogeneous Computing with OpenCL, Elsevier, Moran Kaufmann Publishers, 2011
17. The OpenCL 1.2 Specification (Document Revision 15) Last Released November 15, 2011 Editor : Aaftab Munshi Khronos OpenCL Working Group
18. The OpenCL 1.1 Quick Reference card

References

19. http://developer.amd.com/sdks/AMDAPPSDK/downloads/Pages/default.aspx AMD APP SDK with OpenCL 1.2 Support
20. http://developer.amd.com/sdks/AMDAPPSDK/downloads/Pages/default.aspx#oneAMD-APP-SDKv2.7 (Linux) with OpenCL 1.2 Support
21. http://icl.cs.utk.edu/magma/software/ MAGMA OpenCL
22. http://developer.amd.com/zones/OpenCLZone/pages/GettingStarted.aspx Getting Started with OpenCL
23. http://developer.amd.com/openclforum AMD Developer OpenCL FORUM
24. http://developer.amd.com/zones/OpenCLZone/programming/pages/benchmarkingopencl performance.aspx AMD Developer Central - Programming in OpenCL - Benchmarks performance
25. http://developer.amd.com/sdks/AMDAPPSDK/assets/opencl-1.2.pdf OpenCL 1.2 (pdf file)
26. http://developer.amd.com/zones/opensource/pages/ocl-emu.aspx AMD OpenCL Emulator-Debugger
27. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf The OpenCL 1.2 Specification (Document Revision 15) Last Released November 15, 201 Editor : Aaftab Munshi <I> Khronos OpenCL Working Group
28. http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/ OpenCL1.1 Reference Pages

Source : NVIDIA, Khronos AMD, References

Thank you

Any Questions ?

Source : NVIDIA, Khronos AMD, References